An FPGA implementation of an investigative many-core processor; Fynbos.

In support of a Fortran autoparallelising software pipeline

J. Wyngaard

Department of Electrical Engineering

University of Cape Town

Thesis Presented for the Degree of

Doctor of Philosophy

June 2014

Abstract

An FPGA implementation of an investigative many-core processor; Fynbos, in support of a Fortran autoparallelising software pipeline

Janet Ruth Wyngaard

June 2014

In light of the power, memory, ILP, and utilisation walls facing the computing industry, this work examines the hypothetical many-core approach to finding greater compute performance and efficiency.

In order to achieve greater efficiency in an environment in which Moore's law continues but TDP has been capped, a means of deriving performance from dark and dim silicon is needed. The many-core hypothesis is one approach to exploiting these available transistors efficiently. As understood in this work, it involves trading in hardware control complexity for hundreds to thousands of parallel simple processing elements, and operating at a clock speed sufficiently low as to allow the efficiency gains of near threshold voltage operation. Performance is therefore dependant on exploiting a new degree of fine-grained parallelism such as is currently only found in GPGPUs, but in a manner that is not as restrictive in application domain range.

While removing the complex control hardware of traditional CPUs provides space for more arithmetic hardware, a basic level of control is still required. For a number of reasons this work chooses to replace this control largely with static scheduling. This pushes the burden of control primarily to the software and specifically the compiler, rather not to the programmer or to an application specific means of control simplification.

An existing legacy tool chain capable of autoparallelising sequential Fortran code to the degree of parallelism necessary for many-core exists. This work implements a many-core architecture to match it. Prototyping the design on an FPGA, it is possible to examine the real world performance of the compiler-architecture system to a greater degree than simulation only would allow.

Comparing theoretical peak performance and real performance in a case study application, the system is found to be more efficient than any other reviewed, but to also significantly under perform relative to current competing architectures. This failing is apportioned to taking the need for simple hardware too far, and an inability to implement static scheduling mitigating tactics due to lack of support for such in the compiler.

Acknowledgements

I wish to acknowledge J. Collins most particularly as this work would not have been possibly without his compiler, and his and my supervisor Prof M Inggs' initiation of the project. I would further like to thank them both for their supervision over the course of this thesis.

I would also like to sincerely thank my colleagues in ACE lab at CHPC over the years, and most recently my husband Sebastian for all the support and company.

Finally the financial assistance of the following organisations towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at are those of the author and are not necessarily to be attributed to these bodies:

- The Centre for High Performance Computing (CHPC) a division of the Meraka Institute, a national research centre of the Council for Scientific and Industrial Research (CSIR), and an initiative of the Department of Science and Technology.
- The National Research Foundation (NRF)

Contents

List of Figures

List of Tables

LIST OF TABLES

Chapter 1

Introduction

The last decade has seen the computing industry make a near universal switch to parallel processing. This is evidenced right through the commercial sector from dual core mobile chips through quad-core desktops and 12 core servers, each core is in turn offering higher thread counts, wider SIMD units, or specialised functional units. Finally the academic processor literature now shows an array of new parallel architectures. The early 2000's saw a gradual recognition of the fact that the near doubling in transistor density accompanied by a similar drop in power consumption, seen nearly every fabrication generation from the 70's through the 90's, was beginning to fail[Bohr, 2007; Borkar, 1999; Olukotun and Hammond, 2005; Stutter and Herb, 2005], and this switch to ubiquitous parallelism is the immediate response. While clock speeds have fallen due to power limits, multi-core architectures have also quickly reached their scaling limits due to the overheads of their parallel model.

In the face of a plateaued power budget and the end of multi-core gains, this thesis examines one of the proposed alternative approaches to the need for increased performance; that of many-core architectures - or the use of significantly more fine-grained parallel execution on a processor core made up of hundreds to thousands of very simple processing elements (PEs). To this end, this work centres around designing an architecture that embodies the principles of a many-core processor. Named *Fynbos*, this architecture is prototyped on an FPGA to enable evaluation. Given that one of the biggest hurdles for such a radical design change is software programmability, Fynbos was designed to match the capabilities of an existing legacy Fortran autoparallelising compiler stack, termed *APPRASE*.

Both the many-core model and the APPRASE-Fynbos system are considered in a context targeting high performance computing (HPC) applications. This is in light of

APPRASE's historical use, Fortran's prevalence in the domain and the consequential significant burden of legacy code, the collective need for massive steps of energy efficiency improvement in HPC, and the domain's existing familiarity with wide spread parallelism.

To motivate for the many-core architecture approach the reasons behind falling clock speeds and plateauing multi-core scaling are needed and therefore first briefly presented here. How the issue relates to HPC is further outlined as it pertains to both hardware and software systems. Finally the origins and justification for the many-core approach are given and this work's corresponding hypothesis, arguments to be proved, scope limitations, and summarised conclusions.

1.1 Context and Definitions: The Switch to Ubiquitous Parallel Computing

The doubling in compute performance enjoyed until recently was in part driven by both Dennard's scaling[Dennard et al., 1974] and Moore's law[Moore, 1965]. For a given fabrication process an optimum exists. Divergence from this optimum in either direction means either greater chip complexity drives down yield and therefore costs up, or lowered complexity fails to sufficiently amortise the manufacturing costs. Moore observed that this optimum was moving upwards at a rate of approximately 2x per year, that is, in terms of cost per transistor the size of the optimal chip was increasing. Dennard alternatively described the physics principles by which the semiconductor industry had already been and would continue to follow, that the shrinking transistor size was not only beneficial in terms of area but also permitted a scaling increase in operating frequencies accompanied by a decrease in power consumption. The result is that faster and denser chips would operate within the same power envelope as the slower and less dense versions created with the previous fabrication process.

In what became a self-fulfilling prophesy, Moore predicted his observed trend would continue, which it has and still continues to do. Industry used the increasing clock speeds, and abundance of transistors available to implement Instruction Level Parallelism (ILP) and sophisticated hardware caching schemes, doubling compute performance every 18 months.

Around 2005 and 90nm fabrication Dennardian scaling began to falter as a result of reaching threshold voltages beyond which leakage current, rather than switching

current, dominated power consumption. Therefore, while transistor size continues to scale, it is with little to no increase in switching frequency due to power constraints. For the same power budget chips must now therefore either hold transistor density constant and increase clock speed, or accept a static or even drop in clock speeds and increase transistor density. Thus ended the trend of free performance gains for programmers and users.

With clock speeds thus essentially capped and even lowered by power constraints, ILP close to an engineering end, and with memory technologies lagging behind processors; industry has switched to rather finding performance in parallelism. Berekeley's 2006 seminal paper[Asanovic et al., 2006] summarised these problems as walls (memory, power, and ILP) which cumulatively create a brick wall for processor performance improvement. Therefore, while in the past power was cheap and transistors expensive, the reverse is now true and the last eight years has seen the move to lowered clock speeds mitigated by parallel execution primarily in the form of multi-core processors (see Figure 1.1).

The same paper additionally foresaw the end of multi-core processors, predicting what the last decade has borne out; that they were unlikely to scale beyond 16 cores. The reasons behind this lies in both the approach taken of duplicating complex out-of-order, pipelined, superscalar cores with swelling on-chip coherent cache sizes, and in the exponentially rising overheads of multi-core parallel execution (Amdahl's law [Amdahl, 1967], memory management, interconnect scaling). This model has been made possible by the continuation of Moore's law (see Figure 1.1(d)), but in order to remain within a practical power envelope clock frequencies have been predominantly lowered (see Figure 1.1(b)).

Adding experimental results to the core cap, a 2012 simulation study [Pinckney et al., 2012] examined the multi-core model across six fabrication generations using the SPLASH2 benchmark and taking into account the three primary parallel execution overheads. The overheads included Amdahl's law, the increasing proportion of power use going to leakage current, and the impact of memory access hierarchy and interconnect. Considering the above, they concluded that using a 32nm process technology and a median of 12 cores running slowly at near-threshold voltage (NTV), was the optimal trade-off point. This yielded an approximate 4x gain in energy efficiency over faster operation at the nominal voltage. Such a gain in turn leads to an approximate 4x increase in throughput. Unfortunately these values are a low point across process technologies. Efficiency gains, and median number of cores, have both decreased

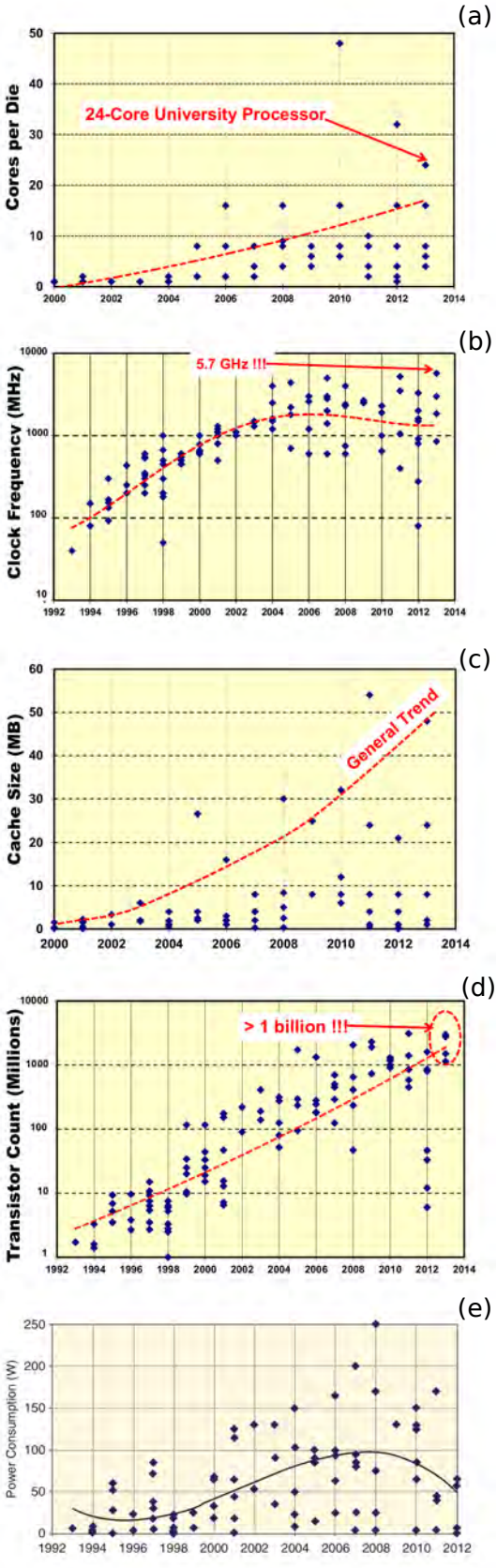


Figure 1.1 – Trend graphs taken from the ISSCC 2013 [ISSCC, 2013] (a-c) and [Smith et al., 2012] (e) technology trends reports. Note (c) Reflects total cache on die.

with each process generation. Effectively this confirmed the core number ceiling and drew a limit on the potential throughput advantage of the multi-core model, baring new innovation.

Continuing from this point, without further drastic drops in power availability per transistor, there is yet another wall quickly looming. The so called *utilisation wall* [Venkatesh et al., 2010] refers to the limited fraction of a chip that can be simultaneously operated at full speed while remaining within a practical power budget. Subsequent to this work beginning, industry discussion of the issue [Esmaeilzadeh et al., 2011; Hardavellas et al., 2011; Kaul et al., 2012; Taylor, 2012, 2013] has describes the wall in terms of *dark* or *dim silicon* referring to transistors available in an IC but which (due to insufficient power budget) can only be used at a lower clock rate, or infrequently, or simply not at all. Dark silicon is already used (caches relative to a FPU, SSE or other speciality logic). Designs are now needed, however, where instead of directly adding performance dark silicon increases energy efficiency across all application domains, thereby freeing up valuable power budget percentages for computation.

[Taylor, 2012] forecasts four possible outcomes of dark silicon, shrinking die sizes, dim silicon, specialised hardware, or an unpredicted breakthrough in semiconductor technology. Of the four, specialised hardware (application and operation dedicated hardware), and dim silicon (heterogeneous dynamically scaling clock frequencies, larger caches (Figure 1.1(c), reconfigurable logic, or NTV highly parallel architectures) are the most practically realistic. The economics of shrinking die sizes make it unattractive, and the unforeseen technological breakthrough is obviously unpredictable.

Whatever avenue is taken, the architectural model will be in part driven by what the Berkeley paper [Asanovic et al., 2006] listed as seven (later extended to 13) key computational domains. They expected these to be the basis of the scientific and engineering computational industry for at least the next decade. In brief, these are characterised as being numerically complex and data movement intensive. Further, the model must use dark silicon to increase efficiency, and find a way to decrease the required memory bandwidth as data movement is power expensive. The later can be partially achieved by exploiting temporal and spacial locality to a far greater degree. Equally, the software model adopted for these new architectures will (as always) be driven foremost by ease of use for satisfactory performance, but also will see significant weight placed on the practical issues of porting the enormous quantity of legacy code that will have to make the jump to any new architecture.

1.2 Problem as it relates to HPC

The HPC industry has been talking about achieving exascale performance for some time now. While there has been considerable progress, with the above industry status there has been no clear path as to how such might be achieved. In a 2011 report by Lawrence Berkeley National Laboratory on the challenges involved, targets for an exa-FLOP machine were set as, costing under $200M, using less than 20MW, and available by 2018 [Shalf et al., 2011]. For perspective, the latest Top500 [Top500, 2013] number 1 offers 54PFLOPs peak performance using just under 18MW, exa-FLOP is therefore another 100x greater compute performance for the same power cost. Looking at a 100x increase across the board, this would intuitively mean that for the same power and physical footprint as the current equivalent consumes, the exa-FLOP era would contain 20W TFLOP capable general purpose desktops and 20KW peta-FLOP size single servers. Using these numbers, such a machine would need to achieve a performance efficiency of around 20pJ per operation. Based on projected scaling, however, the Lawrence Berkeley report [Shalf et al., 2011] showed that while by 2018 10TFLOPs may well be possible with only 100W, a conservative estimate on the cost of data movement to supply the ALUs of such would be around 2000W because, while the cost of operations is likely to decrease, the cost of data movement, on- and off- chip, is unlikely to diminish by much. Locality and data management are therefore paramount to achieving exascale computing.

1.2.1 Architecture

With the above goals in mind, examining the current HPC architectures Table 1.1 lists the processor currently found in the top ten of the Top500. These are all broadly based on the high speed, deeply pipelined, multi-core, multi-threaded, and x86 based commodity architectures. The deep pipelines and sophisticated control logic mean a complex data path can handle out-of-order, predicated, and predictive instruction issue, while being fed by multiple cache levels. Together with support for historical programming models these processors achieve significant sequential and parallel performance for applications scaling to within the region of 16 threads per node. However, as simply increasing the core count of these architectures is not a practical means of further scaling, some systems have added accelerators or picked more novel architectures.

Table 1.1 – A brief overview of the architectures used in the top 10 of the November 2013 Top500 list [Top500, 2013]

Primary Processor Architecture	Cores	Threads	Clock (GHz)	TDP* (W)	Fabrication process (nm)	Peak theoretical GFLOPS	Accelerator	Top500 ranks
Intel E5-2692/ 2680/ 2670	12/8	24/16	2.2/2.7	115/130	22/32	371/ 346/ 333	NVIDIA K20 (2012), Xeon-Phi (2013), none	1, 6,7 10 **
AMD Opteron 6274	16	16	2.2	115	32	282	NVIDIA K20	2***
IBM PowerPC A2 (Blue-Gene/Q (2011))	16+1	64	1.6	55	45	204.8	none	3, 5, 8, 9^
SPARC64 VIIIfx (2009)	8	16	2.0	58	45	128	none	4 ^^

[*]Thermal Design Power

[**]Tianhe-2, Piz Daint E5-2670 8C, Stampede E5-2680 8C, 10 SuperMUC (E5-2680 8C)

[***] Titan

[^] Sequoia, Mira, JUQUEEN, Vulcan

[^^] K computer

One means of increasing processor efficiency, is to decrease the generality of functionality, considering only power the most efficient use of transistors is application specific logic. There is therefore a growing trend to include some application specific functionality in a system. In the portable and embedded computing industries especially, the advantages of application specific hardware are obvious, and with a relatively narrow application range, targeting a processor at an application is a tangible avenue. GreenDroid [Goulding-Hotta et al., 2012] for instance, is a mobile architecture targeting the android market using multiple tiles each dedicated to a specific common mobile functionality domain. Or, while not represented in the top 10 shown, an application specific co-processor currently finding use in HPC, are the Convey

blades [Convey, 2013]. Integrating a Xeon and multiple FPGAs, which may be dynamically reconfigured into near application specific processors, Convey blades provide high speed sequential performance alongside highly parallelised application specific architectures for those domains that their compiler contains FPGA bit streams for. Running on an FPGA, the clock speed is significantly below that of a conventional CPU, yet because of the customisation and parallelism that can be achieved, they achieve performance and power efficiency for target applications.

For more general purpose HPC use, however, other than including a few special function units for networking, virtualisation or cryptography in a CPU, supporting application specific units across the full range of applications targeted is impractical. Considering also the complexities of software management and backwards compatibility this would require, having dedicated units for each domain would result in a "tower of babel"[Taylor, 2012] crisis in both hardware and software.

Taking the middle ground therefore, the most recent model of acceleration are general purpose GPUs (GPGPUs) and the related XeonPhi Intel [Intel, 2013b] (2013) co-processors, which are generalised versions of application specific hardware models. In the latest top500 [Top500, 2013], these accelerators contributed 35% to the performance share, primarily through a finer-granularity of parallel processing in a variant on vector processing. As shown in table 1.1, ranks 1 and 2 of the top 10 are accelerated with GPGPUs, and 6 and 7 with Xeon-Phis, indicating that the processing power is usable, at least with the top500 benchmark suite. And while it is not as simple a comparison to make (due to the influence of many parameters; CPU selection, fabrication size, memory capacity and arrangement, interconnect infrastructure), ranks 1-11 in the green500 [Green500, 2013] which ranks supercomputers according performance per watt, are accelerated with GPGPUs indicating that at least a measure of the expected power efficiency gains are also realisable.

Alternatively, ranks 3,5, 8 and 9, are BlueGene/Q systems, which make use of a very different approach to performance. While multi-core is already known to not scale energy efficiently, IBM attempted to adapt it in a manner that increased efficiency yet still supported a familiar programming environment. A goal they have at least partially succeeded at; filling ranks 12-34 bar 1, in the green500. Because it is specifically targeting the HPC application domains, BlueGene/Q makes use of massive parallel integration with processor chips containing 16 cores (plus 1 for management and 1 spare) each of which is comparatively; simple, slow and low power (55W vs >100W) while still supporting 64 threads and 4 SIMD units. This is twice the average socket

density (8 cores) for the rest of the top500 list, and only possible due to the lower TDP and hierarchical SMP arrangement, the latter again negatively affects programmability to a degree.

Finally, there are the SPARC64 VIIIfx and non-GPU accelerated Xeon based systems (ranks 4 and 10 respectively), which are conventional clusters that achieve the worst FLOPS/W ratings of the top 10. Both of these are older systems which have not taken any novel steps towards higher efficiencies.

Returning to the search for another means of increasing compute efficiency, if application specific hardware across the board is impractical, and middle ground solutions remain limited in how broad a range of applications they can benefit [Lee et al., 2010; Vuduc et al., 2010], a further solution is needed. One proposal for such, the subject of this thesis, advocates for the use of hundreds to thousands of parallel compute units (ALUs) closely integrated on a single chip. The rational for this lies primarily in the above discussion of fabrication technology trends, in order to avoid the utilisation wall, dim silicon must be exploited. As such, NTV operation is needed which means a significant drop in clock speeds. If done in a programmable and application amenable manner, the much higher degree of parallel execution that thousands of simple cores offer may be able to compensate for the raw speed loss. Further, the idea is also in line with Pollack's rule [Pollack, 1999] which noted that under the complex sequential core architectural model, performance is roughly proportional to the square root of transistor count or chip complexity. Power, however, increases linearly with chip complexity. The result is the evidenced decreasing performance for power trade-off, where the optimal core complexity for power cost point is below the current degree of sequential core complexity. In other words, considering that control logic carries out no mathematics, if the logic real-estate used in complex ILP and caching control logic is instead used on many simple cores it may be possible to carry out more real processing work for the same power.

1.2.2 Programmability

If fine-grained parallelism is the most efficient and viable approach to further performance gains, the practicalities of such in terms of application mapping need to be examined. Concerning application parallelism, Amdahl's law indicates that there is insufficient parallelism in current applications to compensate for the many-core proposed drop in clock speeds. Due to the size of the problems and data sets involved HPC is possibly the one sector most suited to taking advantage of significantly greater

degrees of parallelism for a wide spread domain of generic applications. While Amdahl's law stands in valid opposition to the approach of scaling up to greater parallelism, it neglects to take into account the possibility of increasing the parallelisable problem size. Gustafson's law [Gustafson, 1988] formalises how speed-up should, in contradiction to Amdahl, be calculated by scaling the problem to the number of available processors rather than fixing the problem size. In the case of HPC applications, enlarging the problem size is often not just possible but desirable, finer meshes in fluid dynamics, large numbers of bodies in n-body molecular dynamics simulations, more threads in monte carlo astrophysics and financial computations, or smaller time steps and grid sizes in climate modelling. All of these would benefit if the computational power existed to handle such in a practical time span and power budget.

Concerning application expression and mapping to the hardware, for HPC currently the defacto standard programming model is distributed computing through task parallelism and data distribution. Both of these are implemented with message passing and threading compiler pragmas and parallel language constructs (for instance, OpenMP, MPI, Pthreads). If the fine-grained parallelism of a many-core architecture were to be utilised in HPC it is hard to see why the gains provided by these should be discarded. It is more likely that the many-core architectures would simply replace the multi-threaded, multi-core, and multi-chip nodes within a cluster. As such any many-core programming model would need to integrate with the current standards. As an example, a primary factor in the adoption of GPGPU's was the as-of-yet still evolving but novel programming model created for them. The model manages to abstract some of the multiple levels of concurrency while supporting a C-like programming environment (OpenCL, CUDA or FCUDA, OpenAcc, and others), and enabling integration of CPU and GPGPU code into a single binary. Alternatively, taking an even easier route the Xeon Phi is specifically constructed to enable easy porting of HPC codes to itself. Rather than requiring the level of re-write GPGPUs do, OpenMP and MPI are both supported for the Phi albeit alongside Phi-specific vector instructions and with the need for tuning. A new programming model that integrates with the current standards but is capable of expressing the degree of parallelism under consideration here is needed.

In terms of languages, for reasons of both appropriate form of expressions and historical momentum Fortran is arguably the most commonly used in HPC [Loh, 2010]. For instance, within academia as a whole Fortran remains second only to MATLAB in prevalence [Prabhu et al., 2011]. This is due primarily to the fact that the codes

in use are well established open-source codes written in the 90's and can therefore be relied on to have already been thoroughly verified, debugged, tested, and optimised. Interestingly, the [Prabhu et al., 2011] survey also found that while the vast majority of respondents traded accuracy for time, because of the complexity involved few tuned their applications to new hardware platforms or tweaked the parallel sections of code as they found the learning curve too steep. Lastly, in contrast to the Fortran dominance of established fields, emerging interdisciplinary fields are developing their models in C/C++.

1.3 The many-core hypothesis

The above co-processors have stepped into the immediate performance gap in HPC, and demonstrated finer-grained parallelism's potential within select domains. In doing so some of the principles and gaps to be avoided have been exposed in hardware design and programmability. The proposal on which this work is based takes the concept of fine-grained parallel hardware in a slightly different direction (and to an extreme) advocating for thousands of simple cores operating in a MIMD manner on one chip.

1.3.1 Architecture definition

Going into more details on what a many-core or thin-core based architecture consists of, the concept existed in vague terms earlier but was formally, if briefly, set out by Intel Fellow S. Borkar in a 2007 paper [Bohr, 2007]. Subsequently, it has been discussed and refined further by himself and others (for instance [Borkar, 2010; Borkar and Chien, 2011; Catanzaro et al., 2010; Marowka, 2011]). In its briefest form it is proposed that the non-scaling multi-core model of tens of complex high speed cores should be replaced with hundreds to thousands of more simple slow cores. That is given the current and imminent power and technological limits, chips containing thousands of simple cores exchanging control complexity for parallel width in ALUs may offer a way to achieving sufficient efficiency for exascale computing.

While Borkar's 2007 paper describes some of the challenges of such a many-core architecture, he sets out no real details on practical means regarding memory organisation, PE capabilities and hierarchy, instruction and data paths, system control, or

even communication infrastructure. Borkar proposes only the use of two operating frequencies and voltages to be dynamically selectable on each PE as a compromise on fine-grained power management, and cautions that particular attention will be needed as regards the interconnect as a potentially major power draw. Similarly, the high memory bandwidth required to supply such an array of processing elements (PEs) is also noted as another point of possibly significant power draw. In later works some further refinements included incorporating a large high-speed sequential core amidst the sea of simple PEs, suggesting the sophisticated compiler placement of data as one means of power saving regarding data movement, and the use of heterogeneous PEs in conjunction with the fine-grained power management.

In consequence to this undefined nature of the term many-core has been applied to a range of architectures in the literature. Many of these do not easily fit what Borkar described often due to limits in their scalability. But in contrast, as already discussed, GPGPUs easily scale to multiple thousands of cores (NVIDIA's latest K40 single IC GPGPU supports 2880 GPU cores) and are certainly a model of extracting energy efficiency with extreme parallelism. However, if Borkar's proposal is meant to serve a wider range of applications than the underlying SIMD nature of GPGPUs allow GPGPUs and other application specific architectures are also not representative of the above proposal.

Therefore, given that there is thus far no formal definition of many-core, that [Bohr, 2007] is the first to set out the concept (but only in high level terms) on which this work is basing its hypothesis, that this concept is backed up by others, and that other works exist which use the term in manners that imply a different definition, for the purposes of specifying the goals of this work the term many-core is defined to mean an architecture for which the follow is true:

1. The potential exists for scaling to a size where there are multiple hundreds of PEs on a single IC.

2. To ensure NTV operation is possible, the primary source of performance is parallel execution rather than clock speed.

3. The data and instruction paths are intentionally kept simple to redeem logic real-estate for more PEs.

4. A PE is capable of executing an instruction and fits within the following boundaries:

   (a) Is in-order.
   (b) Is single threaded although multiple pipelines may exist.
   (c) Supports an instruction set and register capacity too limited to operate an OS.
   (d) Is capable of simultaneously executing a different instruction to any other PE.
   (e) Is able to select operands from and store results in uniquely designated registers.
   (f) Is voltage and frequency tunable.

5. On-chip sharing of data between PEs is possible, and while it's extent is not defined its should not be optimised for a particular application domain.
6. The quantity of PEs should be exploitable in terms of affording redundancy against fabrication and use created defects.
7. The following are not required but also not excluded:
   (a) The PEs need not be homogeneous.
   (b) The sea of simple PEs need not be capable of operating an OS.
   (c) The sea of simple PEs need not be capable of operating independent of a host OS supporting core.
   (d) Multiple operations may be embedded within a single instructions such as in VLIW architectures.
   (e) The PEs need not be capable of dynamically issuing instructions.
   (f) A high-speed sequential core may or may not be integrated on chip with the sea of simple PEs.
8. No structure for how the simple PEs are arranged is given.
9. No operating control model is defined.
10. In the context of this work only, due to the focus on HPC floating-point support is required.[1]

[1] It is recognised that there are techniques available for running HPC applications without floating point support but considering practicalities around adoption of new technology this is considered necessary for the immediate future.

1.3.2 Programmability challenges

The question of how to program any radically new, more efficient, and parallel architecture, is an entire domain of research. Parallel programming has always been difficult and this new era only adds an additional suite of challenges.

1. Any new programming model for many-cores will, as ever, need to take into account:
   (a) Portability between architectures.
   (b) Ease of use considering the skill level of target users.
   (c) Ways of avoiding common parallel pitfalls.
   (d) The burden of legacy code.
   (e) Mechanisms for parallel debugging and verification in a possibly non-deterministic setting.
2. Many-core will only add further to these with the new challenges of:
   (a) Integration with current cluster models.
   (b) Multi-level concurrency.
   (c) The need for new algorithms.
   (d) Architecture heterogeneity at multiple system levels from PEs through cores to ICs.
   (e) Low-level power and interconnect management access.
   (f) New forms of memory management (locality, access ordering, immutability, sharing).

The last of these, the issue of memory management in particular is fundamentally an execution control issue which as discussed lies at the crux of achieving energy efficiency, making it of paramount importance in examining programmability. In a discussion around implicitly parallel programming models in 2007 [Hwu et al., 2007] advocated that hardware architects cannot simply pass this control burden to the software because of the burden it places on the programmer. A number of the older architectural models reviewed in chapter 2 do take it on in hardware in innovative manners, however, in a context where transistor operation comes at a power premium, software control offers significant advantage not just in energy efficiency but also in

range and flexibility of application domain support and the problems of portability and heterogeneity.

Determining how to handle the above control issue means that the line defining hardware and software responsibilities needs to be drawn in order for either to be developed. In the case of this work the location of that line is most strongly influenced by the compiler stack used which pre-dates the hardware. While the move to many-core architectures is revolutionary it exhibits elements of a return to techniques used before clock speed and ILP were a sufficient easy source of performance gains. The first generations of supercomputers used massive, for the time, parallelism on a level that was more fine-grained than the now classic cluster structure. This thesis uses a Fortran autoparallelising software pipeline that was originally created to improve parallel performance on such a machine.

The attractions of autoparallelisation are obvious and all the greater when considering porting hundreds of thousands of lines of legacy codes. It addresses all the problems traditionally associated with parallel programming the complexity of managing code dependencies, the possibility for non-deterministic behaviour and other pitfalls introduced by manual optimisations, the common need for and difficulty in scheduling synchronisation between processes, and the abstraction level of modern languages. And instead offers parallel operation via traditional sequential coding. This makes for a very low barrier to use, which is a significant need considering the apparent low parallelisation skill level within the science user community of HPC [Prabhu et al., 2011].

Further in favour of using autoparallelisation in this particular case is the fine-grained nature of parallelisation under consideration. Historically the majority of autoparallelisers have been concerned with high-level data and task parallelisation, where the number of parameters to be taken into account when making a decision is very high, as is the cost of the compiler making a poor decision. The fine-grained parallelisation under consideration in this work, however, is concerned with the parallelisation of single instructions rather than application tasks, and the data placement of single words rather than large blocks of contiguous memory. In this case therefore, because the sphere of influence of tasks (operations) and data (operands) is so much lower, the costs of a poor decision are more likely to have less of an impact on the whole application. Also, with a lower number of parameters of influence on a single operation decision making is simpler. Finally where human insights into an application can play a significant role on high level task and data parallelism, the level of par-

allelism invoked on a many-core parallelising single operations (ADD, MUL, MOVE and similar) for anything other than the most trivial of codes, is beyond human capability to analyse and schedule. The processes is rather a matter of book keepingS for which computers are already ideally suited.

Some of the issues which an autoparallelisation based programming model does not immediately cater for, however, include the problems of non-parallelisable algorithms, memory management as an execution control mechanic, portability and heterogeneous systems, and low-level fine-grained control of power and interconnect hardware. How this work intends to cater for such will be argued in the later chapters, predominantly by judiciously dividing these between hardware and software.

1.4 Thesis Hypothesis and scope limitations

For all of the above reasons it appears worth while to investigate what would be required in a many-core architecture to match the APPRASE legacy Fortran autoparallelising pipeline, with the goal of increasing compute efficiency when executing HPC applications. The following hypothesis is therefore put forward:

That a many-core architecture designed to match the legacy APPRASE Fortran autoparallelising pipeline, can achieve improved compute efficiency over a traditional superscalar sequential core, while still providing reasonable performance when executing HPC codes, making the APPRASE-Fynbos system approach a viable option in tackling the walls currently facing the computing industry.

In arguing for and against this hypothesis, the following research questions will be addressed:

1. Is Fynbos representative of the proposed theoretical many-core architecture, how does it compare to other similar architectures, and how well does it facilitate optimal MIMD operation for HPC applications?

2. Do Fynbos and APPRASE together meet the needs of a programming and architectural model for HPC applications, and in doing so are the practicalities of industry adoption addressed?

3. What compute performance and efficiency metric values does Fynbos theoretically achieve and how do these compare to current metrics and the requirements for exascale computing?

4. What compute performance and efficiency metric values does the APPRASE-Fynbos system together achieve, relative to a traditional superscalar sequential x86 core, and how well would it therefore serve as a multi-core replacement in an HPC multi-node cluster?

5. Considering the answers to the above, would the use of the APPRASE-Fynbos system facilitate creating a exascale compute system, and in either case what insights does this thesis's work in creating the system provide regarding the many-core hypothesis?

The high-level novelty contributions of this work will include:

1. Design of a novel many-core processor for general purpose HPC and consideration of its use as a replacement for current multi-core chips in a cluster.

2. Use of a Fortran autoparallelising compiler to port sequential Fortran to a many-core architecture.

3. Examination of the performance and energy costs of using an FPGA as a many-core substrate.

4. A review of many-core like architectures.

The scope of this work is subject to the following limitations:

1. As an initial investigative prototyping exercise, the hardware and software layer needed to cluster many-cores is not considered.

2. How languages other than simple sequential Fortran might be ported to many-cores is not addressed.

3. Other than mention in the literature review, how many-core architectures and applications other than HPC might relate to one another is not considered.

4. While infrastructure is developed to move data and instructions in and out of Fynbos, due to the prototype nature of the system and development kit style hardware available real world usable IO issues could not be addressed.

1.5 Chapter Summary

To summarise therefore, with the end of Dennardian scaling chip power budgets are at their maximum yet Moore's law continues and so transistor densities will continue to grow. In the face of this power wall and adding the ILP, memory, and utilisation walls, an architectural model significantly more energy efficient than current approaches is required if industry is to continue increasing available performance. To circumvent the problem industry has turned to pausing and even reversing slightly the last decades rapidly increasing clock speeds, and instead has increased core counts per chip. The multi-core model has served as a temporary solution but will not continue to scale beyond 16 cores, as beyond this point the overheads of the model overrun any performance gains. In another more recent approach GPGPUs and other application specific accelerators are a further response to the walls, and one which has found prominence in the HPC market demonstrating the performance and efficiency advantages possible with a finer granularity of parallelism, than clustering, multi-cores, or multi-threading. They are, however, fundamentally limited in application domain range by their SIMT model.

Focusing on HPC specifically, and its push towards exascale, a 100x increase in compute performance is needed for a near zero increase in power cost. Industry is therefore looking into new, ways of using dark silicon, fabrication process technologies, power management methods, and the programming models needed to accompany each. The new architectural model will be driven by demands for support of numerically complex data movement intensive computations, and the need for energy efficient computation. Equally, the new programming model needed to program such a new architecture will, apart from ease of use, see significant pressure to provide backwards compatibility or a means of porting all the legacy sequential code currently in use.

Within this substantial scope for new work, this thesis is focusing on a proposal for chips containing hundreds to thousands of cores operating at NTV and achieving performance through unprecedented MIMD fine-grained parallelism. Based on the problem set size and scalability of scientific HPC codes, it has been argued that HPC is a domain likely to benefit from such an architecture. Considering that a dominant language in HPC is Fortran, and to ensure such an architecture is programmable, it has been argued that a legacy Fortran autoparallelising software pipeline is the best means of exposing the extreme nature of the parallelism under consideration. This

approach is additionally advantageous considering the quantity of legacy codes in use in HPC, which based on the above technological conundrum present an unavoidable and immense barrier to further performance gains in HPC if a radically different architecture requires they be recoded. An admitted caveat to this, however, is that there will be codes for which the underlying algorithm needs to be changed.

In the domain of many-core literature there exists a range of architectures claiming the title yet differing widely. For the purposes of this work a definition that is appropriate for the defined target of HPC has therefore been given, focusing primarily on floating point support, generality of applicability, and scalability in cores, memory and interconnect.

The existing literature while focused more on hardware for target applications and primarily created prior to 2007, does provide insights into designing a many-core. More recent architectures have also appeared subsequent to this work beginning which will add further to the discussion but which will be shown to be targeting different hypotheses.

The remaining chapters of this thesis will therefore argue for the above as follows:

Chapter 2 examines: architecture designs with aspects relevant to a HPC many-core processor design, future facilitating technologies, concurrent programming models that offer potential means of programming a many-core architecture.

Chapter 3 gives the historical background to the APPRASE project, and examines how the APPRASE compiler pipeline maps HPC applications to the fine-grained parallelism of Fynbos and is distinct from other autoparallelising software stacks.

Chapter 4 describes the Fynbos architecture, how it implements the goals of the many-core architectural model, and with what success and failure.

Chapter 5 reports on performance and efficiency metrics achieved by the Fynbos RTL design on 2 different FPGAs, as well as inferring what such a system might scale to if implemented as an ASIC, and evaluates the APPRASE-Fynbos system according to standard metrics and using a case study comparison exercise to a Xeon processor.

Finally Chapter 6 draws together the conclusions of each prior chapter to make final arguments around the hypothesis ultimately showing, Fynbos to be representative of a many-core architecture and to be capable of greater efficiency than any of the other architectures reviewed (if ported to ASIC), and APPRASE to be capable of the claimed capacities in autoparallelising Fortran and effective at utilising the Fynbos Hardware. However, it is also found that even this efficiency is short of that

needed for exascale by an order of magnitude. Further, due to a range of deficiencies in both hardware and software revealed in testing and beyond this work's scope, the joint system fails to come near required application performance. Recommendation are made, however, for how this might be relatively easily rectified.

Chapter 2

Literature Review

2.1 Introduction

Chapter 1 showed that fundamentally the barrier to further performance is power consumption, and that in order to use the abundant transistors available more efficiently, dark and dim silicon need to be exploited to provide performance. Among the few approaches to this that are being explored is the many-core hypothesis. As described earlier this aims to trade reduced complexity for significant parallelism operating at a lower clocks speed in a manner that might gain performance and computational efficiency.

In the HPC industry the need for innovation in parallel computing is now accepted standard even if many-core is not the approach ultimately adopted. This is visible in the rapid escalation in use of GPGPU and other innovative parallel architectures, and the wide range of tools and languages appearing to facilitate parallel programming. As already shown HPC has always used parallel computing and interestingly appears to be returning to some of the early techniques used. Broad architecture models the industry has tested in the past include VLIW and its subsidiary EPIC, EDGE, vector, DataFlow, Streaming, reconfigurable, and MPPAs.

Within the above list of architectural approaches and the course of computer architecture history, there is a huge body of work with some degree of relevance. The selection of work reviewed here is therefore a selection of those most relevant where relevance is based on; shared many-core like features, use in HPC, use of software to gain compute efficiency and use of alternative hardware techniques to gain compute efficiency. As a result a significant number of the works reviewed are older but include one or more of the above factors. Obviously beyond these are the works created more

recently which have developed within the same industrial context as this work. These again were narrowed to a manageable number based on similarity in architectural features or design goals and approach. Despite these restrictions the field remained large and the following is therefore an extensive subset.

The majority of the following therefore at least partially embrace the notion of replacing complex dynamic hardware for compiler complexity, gaining chip real estate and TDP proportion for computation. These therefore also share in the costs and gains of replacing dynamic hardware with an alternative means of control, providing insights into the relative costs of the different means of making this trade-off.

This leads to the equal challenge of programmability. Most recently industry has sought ways of explicitly exposing the inherent parallelism available in hardware and algorithms to the programmer. Extensions for existing sequential languages, new parallel languages, and new analysis tools have done this thereby enabling the performance gains of multi-core processors. These approaches are not necessarily appropriate to many-cores however. The domain of software architecture is no smaller than that of hardware and cannot be ignored as the two are intrinsically linked, but it is also not a primary focus of this work. Therefore given this work's hypothesis, Section 2.3 firstly examines what an appropriate many-core programming model would ideally exhibit, before reviewing currently available programming models, and finally briefly examining available autoparallelising code and the tool chains.

2.2 Many-core-like Architectures

As outlined in Chapter 1, a wide spread definition of many-core has yet to emerge within the literature. This work uses the term as it is used by S. Borkar[Borkar, 2007] and others [Catanzaro et al., 2010; Marowka, 2011; Verdoscia and Vaccaro, 2012] in reference to a hypothetical processor containing hundreds to thousands of simple PEs. Having defined the term for the purposes of this thesis, the following predominantly examines architectures containing drastically simplified cores which are closer to sophisticated ALUs than traditional cores, or which display some of the envisaged characteristics of a many-core processor.

In a search for many-core amenable design tactics, and to ensure novelty, the range of architectures that could be reviewed is large but is narrowed with the following reasoning. Prior to Borkar's proposal designers were already running into cross-chip delay problems and as always looking into new avenues for greater performance.

As a result some of the following were developed prior to industry collectively turning to greater parallelism for performance gains but are relevant for their otherwise motivated approach to scalability. Similarly, some architectures while designed for a specific application domain, are included for their approach to another relevant characteristic. Finally, still further architectures are included that contain more sophisticated PEs than sought after but which are included because they give greater context to the current search for greater performance and illuminate the trade-offs involved in taking the particular many-core avenue this work has selected. Equally, many others are excluded for being too far removed from the goals of this work, and still others simply for tractability.

Within such a definition and scope therefore the following architectural models are used as categories for discussion, dataflow, MPPA, streaming, and alternative. To ensure the review is not unwieldy, each architecture description is limited to a discussion of, operating model, memory and data movement model, and programming model, as far as the literature allows. Comparisons are made throughout the descriptions, but drawn together within each category in a comparative sub-subsection analysis.

It should lastly be noted that given the range of applications these are each targeted at and the time span over which they were developed it is impossible to draw direct performance comparisons. Therefore instead, where possible a minimal descriptive analysis is alternatively given.

2.2.1 Dataflow ISAs

A dataflow architecture executes statements on the arrival of all required operands. This automates a significant proportion of control making the expression of a program's DAG in hardware far more direct than is the case in more traditional general purpose processors. Where multiple PEs are used, dataflow, spacial, and temporal locality, are each inherently exploited using compile-time control saving on memory access delays. For amenable applications the model lends itself to a wide execution path of multiple parallel units in a manner that uses less chip-wide control logic than would otherwise be required, easily exploiting fine-grained instruction and data parallelism. While operand arrival is responsible for statement firing, the compiler stack is therefore responsible for the correct arrangement of data and statements. Scheduling is consequently static, which baring any mitigating factors is a major disadvantage but is in-line with the proposed ideal of passing more control to the compiler.

Due to their asynchronous nature, from the very first dataflow processors developed in the mid 70's all have needed to handle memory access ordering. These initial processors relied predominately on custom languages with built-in constructs to enforce correct order. The approach minimised the memory control logic needed, but contributed to limiting their use to a few application domains. More recently the model has been re-explored in the form of a number of hybrid processors. These later versions have instead adopted predominantly hardware solutions to the memory access ordering problem and have sought to execute imperative languages (C particularly) efficiently. This has opened the door wider to more application domains.

TRIPS[Gebhart et al., 2009; Sankaralingam et al., 2006; Smith et al., 2006], WaveCache[Swanson et al., 2003, 2007], and ASH[Budiu et al., 2004] or its sequel Tartan[Mishra et al., 2006], were developed in the mid 2000's and are therefore not products of a many-core hypothesis. Rather they were created specifically in the face of lengthening wire delays, which an array like structure and distributed control was able to address. All four utilise some form of direct producer to consumer communication model, conserving control and data path bandwidth, and reducing the required register renaming logic and files.

2.2.1.1 TRIPS

TRIPS [Gebhart et al., 2009; Sankaralingam et al., 2006; Smith et al., 2006] is an implementation of the Explicit Data Graph Execution (EDGE) ISA, defined primarily by the joining of a block atomic execution model with a dataflow producer-consumer communication model. Of the dataflow architectures reviewed here TRIPS is the only design to have been taken all the way to silicon on a motherboard, and therefore to have fully investigated practical scalability factors.

In overview, a TRIPS system is defined to minimally consists of, four TRIPS chips each containing 2GB of shared L2 cache and two processors. By design, up to eight such minimal subsystems may be combined for 256 PEs in 16 processors. Each chip contains a square array of 16 PEs. Within an array, five instruction and four data caches are shared by four rows, and a minimal register file is associated with each column.

Within an atomic block, LOAD and STORE operations are annotated with an ID, to ensure correct memory access ordering within that block. The register files serve as a means of data transfer between atomic blocks, and are only written to on the completion of an entire block ensuring the easy roll back of miss-speculated blocks.

Predicting the direction of program flow, a global control tile issues up to eight such instruction blocks at a time; one non-speculative and seven speculatively. This effects a form of dynamic instruction issue and sees 1024 simultaneous in-flight instructions (128 per block). Further, the issue of blocks amortises the per-instruction costs of instruction issue and adds memory latency tolerance. While each block is a modified form of hyperblock,[1] and is limited to one exiting branch, multiple predicated branches are possible within a block, further mitigating the limits of static instruction scheduling. While block issue is controlled by a global PC, within a block dataflow controls execution.

Examining the PEs each contains, instruction and operand buffers for up to 64 instructions, a standard single issue pipeline, integer unit, and FP unit. All operations may be predicated and, apart from divide, are pipelined.

In line with the goal of scalability four separate point-to-point interconnects provide, instruction dispatch, control, status, and operand paths. Using these compile-time static instruction placement optimises for shortest data path distances between PEs within a block.

The TRIPS compiler supports both C and Fortran and carries out a range of standards optimisations such as loop unrolling, and function in-lining. When scheduling it also makes use of optimisations such as, loop optimisation, or tail duplication. Finally the creation of hyperblocks is carried out iteratively, only after which register allocation is done.

2.2.1.2 WaveCache

WaveCache [Swanson et al., 2003, 2007] is a simulated implementation of the proposed WaveScalar ISA. Similar to TRIPS, WaveCache uses a modified version of hyperblocks, *waves*, to group code segments. Waves allow for more than one entry point, are larger, and may contain multiple control joins excepting the branch stage of loops (a side effect of WaveCache's memory ordering methods). The principle of distributing memory amongst ALU units is also shared, however, there is a 1:1 ratio of instructions to functional units. That is the PEs in WaveCache create a sea of instruction registers each with a dedicated functional unit. A working subset of instructions is held within this "cache". This leads to a more fundamental difference between the two in their execution and ordering control models. Where TRIPS uses a global PC to

[1] A sequence of predicated program segments with a single control entry point and potentially multiple exit points. [Mahlke et al., 1992]

move between blocks, program execution in WaveCache is entirely data-driven, new instructions are fetched on the arrival of data causing an instruction cache miss.

Such an execution model is appropriate for a system optimised for applications with high instruction re-use. The expectation is for an instruction to remain in the "cache" for many cycles executing repeatedly on different data points. To enable the execution of multiple waves of the same instructions simultaneously, a tag is added to data points, whenever all operands of matching tags have arrived that instance of the instruction will execute.

These tags are also used in the approach to correct memory access order, which is similar to that used by TRIPS. In the case of TRIPS, the ID tag is to distinguish between atomic blocks, while WaveCache's more complex tag enables correct ordering without waiting for wave completion. The tags used are in fact wave numbers which can be incremented using a WAVE_ADVANCE instruction. The compiler therefore inserts such instructions at the end of each wave distributing the task of data tagging instead of centralising it at the memory. Using the tag, the data memory is able to determine the correct access ordering.

In terms of dynamic control, while TRIPS makes use of aggressive speculation, WaveCache maps all control paths leaving a mix of predication and steering instructions to ensure correct execution order. Mapping all control paths and using the tag matching to effect a form of simultaneous instruction issue comes at a logic cost. The matching tables in each PE represents around 60% of its total logic.

To ensure scalability, PEs are arranged hierarchically beginning with two PEs making a *pod*. Within a pod two 5 stage pipelines are able to snoop each others' ALU bypass networks and instruction schedules for back-to-back execution purposes. Up to eight instructions can be queued in a PE at a time. Further up four pods make a domain (8PEs) which shares one FPU, and four domains make a cluster (32PEs) which shares a coherent L1 data cache. Each hierarchical level has corresponding internal and external interconnects with associated increasing latencies. Scaling from the snooping network, point-to-point lines connect pods in a domain, a bus connects domains in a cluster, and a NOC connects the clusters.

With wave-ordered memory, the WaveCache may be programmed in C. The code dataflow graph is first decomposed into waves, after which control instructions such as the WAVE_ADVANCE are inserted, and finally the predication and steering operations are added to direct data dynamically. On initialisation, every operation invokes an instruction cache miss until the array is filled.

2.2.1.3 Tartan

Tartan [Mishra et al., 2006] and its predecessor ASH [Budiu et al., 2004], were envisioned to be on-chip integrations of, asynchronous reconfigurable fabric (RF), an appropriate conventional RISC processor, and mutually shared L2 and data L1 caches. The envisioned RF is closer to application specific hardware than a general purpose processor. Where ASH connected application specific PEs constructed from LUTs. Tartan instead used the more coarse grained building block of 8 bit wide ALUs in a move towards better performance and more practical synthesis times. These ALUs are arranged according to the application, but in a regular structured array that executes as an asynchronous dataflow architecture. Up to 16 ALUs make up a configurable 128b wide functional unit, a *stripe*. Stacked vertically, 16 stripes create a *page*, and 16 pages arranged in a 4x4 array, a *cluster*.

The page grouping is designed to be the processing element of a hyperblock with its single entry point. Data therefore enters at the top of a page and moves down through the computation according to dataflow control. Communication within the page is via a partial crossbar, with no means of data feedback. This flow naturally facilitates pipelined operations where multiple stripes are needed, such as multiplication. Within a cluster, between pages the compiler statically arranges communication channels and switch boxes according to the application's DAG. Finally between clusters, communication is via a dynamically routed NOC capable of handling the less predictable nature of procedure calls and memory accesses.

As with the others, within the RF extensive predication is enabled by a number of single bit ALU elements. Like WaveCache all control paths are executed with customised MUXs (like WaveCache's steering instructions) using the predicates to determine which results are passed on where. Instead of block atomic execution, or tokens, however, Tartan uses predication on loads and stores to prevent or allow their execution. Where the compiler cannot determine dependence order, additional edges are added to the DAG creating dependence.

It has not been mentioned but TRIPS and WaveCache are designed to operate in conjunction with a traditional hosting CPU. Tartan stands out from the other two in that the RF is significantly more integrated with the CPU than in the case of TRIPS or WaveCache. This integration goes so far as to include enabling dynamic procedure calls between both the CPU and RF via a bus and FIFO interface. Historically dataflow architectures, with no registers to save values in, do not support function calls but

rather in-line code. WaveCache manages a limited manner of such via compile time inserted functions that return destination addresses rather than passing values. In the case of Tartan, however, due to shared L1 and L2 caches, the system is able to dynamically choose the best platform, CPU or RF, and invoke it on one or the other. The compiler also has more freedom in placing operations on the most appropriate architecture as data sharing is inherent. This comes with various costs, however, not least of which is the limit on scalability. The previous architectures achieved scalability by distributing the L1 caches. Tartan, however, is constricted and suffers performance loss due to long memory access times as functional units are placed further away from the L1 caches and from each other (40-80% of RF computation time is communication).

Regarding software, Tartan also executes C using two separate tool flows after the compiler has determined what is to be compiled for the RF and CPU (a division made along procedure boundaries). Compilation for the CPU is standard but the RF involves first the transformation of C into a predicated SSA form that uses dataflow as its control mechanism. DAG analysis and the insertion of additional dependence edges then allow the application to be divided into hyperblocks. Using the hyperblocks as functional units, each is finally synthesised into the data-triggered clock-less pages of above.

As is to be expected Tartan achieves significantly higher energy efficiencies, where the model suits an application, but generally under-performs relative to a superscalar CPU. The lack of performance is primarily due to the above communication overheads, but the authors also note their compiler could exploit further optimisations. For the most part the optimisation they do make, such as the use of speculation and predication, can also be exploited by a super scaler CPU, leaving Tartan only with the issue width advantage.

2.2.1.4 D^3AS

Currently only published as a high level concept with a simple prototype implementation, the Demand Data Driven Architecture System (D^3AS) [Verdoscia and Vaccaro, 2012], is a far more recent dataflow architecture directly inspired by the many-core hypothesis. Having concluded that many-core processors are the most efficient approach, this work examines the problem of many-core programmability and develops the architecture from there. Interestingly their conclusion is to revert to the 70's approach and adopt a custom functional language, CHIARA [Verdoscia et al., 2004].

As with the others D^{3}AS also has an asynchronous hierarchical arrangement of PEs includes groups of multiple simple cores with different groups taking on different subsections of a program flow graph. Direct connections link PEs within groups and a message passing NOC connects groups. Details on the actual contents of the simple cores, (beyond that each is identical) or the memory hierarchy used, are not presented. The distinction in hardware therefore, is the use of a demand driven model on top of the data driven dataflow model. Along with any data value, a valid bit is transmitted and firing will only occur when two valid operands are registered, removing any need for data acknowledgement communication.

The CHIARA language defines a functionally complete set of basic operations that include data routing. These are directly supported in hardware. While higher-level functions may be created, they will be compiled down to this basic set making the language both the high-level and assembly level language of the processor. The goal of this is to make for an easy mapping of code to dataflow graph to hardware.

To demonstrate viability, a FPGA demonstrator is used to solve a simple linear matrix equation iteratively using two methods. The results show communication dominating execution time but up to a 11x speedup due to the parallelisation on a 1024 PE system.

2.2.1.5 Comparisons

To conclude, it is worth considering how the above might perform in an HPC environment. Excluding D^{3}AS, all of the above performed extensive evaluations using standard benchmarks including SPEC2000. WaveCache also used Splash2 for its multi-threading performance and, along with Tartan, Mediabench [Lee et al., 1997]. However, despite this commonality and appropriate benchmark, each uses a different metric making comparisons difficult.

With hand optimisation, TRIPS showed an average 3x cycle count speedup over a Core 2 processor. However, using compiled only codes, it achieved only 60% of the Core 2 performance. While 1024 in flight instructions are possible, the compiler is not able to always fill all blocks, and pipelines stall due to instruction cache misses, branch mispredictions, and load dependence mispredictions. The authors concluded that the TRIPS implementation of EDGE was simply too poor a match for certain program characteristics, naming specifically programs containing many small functions or indirect jumps.

Over a range of applications, for single threaded codes, WaveCache appears to average out as equivalent to an out-of-order Alpha EV7 in IPC. On multi-threaded codes from SPLASH2, however, performance ranges from 0.4 IPC to 168 IPC, depending on thread numbers and application, with a similar range (<1x to 85x) in speedup relative to a dual-core version of the Alpha.

Finally Tartan, with its reconfigurable application specific hardware, focuses more on energy saving than performance in its evaluations, achieving up to a 3x improvement in efficiency over an out-of-order core. Tartan achieved speedups of up to 4x on media applications, but on more general applications saw significant slowdowns. The authors accredited this primarily to a lack of compiler speculating ability.

Based on this and their support of C, these dataflow architectures offer a more viable option than their 70's predecessors but have adopted complex hardware to handle levels of control beyond the dataflow execution model. In terms of gleaning from their (authors) experiences, the following points can be drawn out:

On operating model, while broadly grouped under the use of dataflow control, TRIPS retains the von Neuman PC. WaveCache and Tartan implement more decentralised control. The former gains easier ordering at the cost of scalability, while the later gains scalability at the cost of hardware complexity. All have made use of the dataflow model so as to exploit less restrictive asynchronous timing.

On memory and data movement, Tartan and D^3AS both list communication costs as significant factors in resulting execution times. Distributed memories are shown to facilitate scalability but require a control ordering mechanism in the asynchronous environment. Where Tartan's reconfigurable one-use hardware allowed more simple static interconnects, TRIPS and WaveCache required more complex dynamically configurable interconnects at the lowest level. All three relied on a dynamically configurable high level chip-wide interconnect.

On programmability, supporting C or Fortran obviously goes a long way towards programmability. Excluding D^3AS, in order to extract efficient operation by minimising data movement the remaining three architectures use their own forms of hyperblocks, traditional compiler codes optimisations, and various degrees of static scheduling. Each achieves a level of energy efficiency greater than the benchmarked sequential CPUs but for the most part at the cost of performance. The degree of integration

with a host processor is a further important factor here, often appearing to degrade performance, but is not one which is not examined closely in these papers beyond noting its advantages and costs.

2.2.2 MPPAs

Massively Parallel Processor Arrays (MPPAs) form another more generic category of many-core like architectures. Most commonly applied to architectures directed at data-level parallel applications, particularly multimedia. They fill some gaps in the embedded market between, embedded microprocessors, DSPs, and reconfigurable architectures. While generally constructed of PEs containing more sophisticated microprocessors than a simple ALU, some articles have even included GPGPUs under the term.

The higher degree of sophistication in PE cores facilitates programmability but also enables run-time configurability in operation and interconnect. The works reviewed here are therefore architectures that have selected to apply the term to themselves, and which are considered still worth noting in terms of this work's definition of many-core architectures.

2.2.2.1 Proposed event-driven programmable array

[Lee et al., 2012] describes a MPPA, thus far only in conceptual simulation, which they call "A Programmable Processing Array Architecture Supporting Dynamic Task Scheduling and Module-Level Pre-fetching". Similar to the bulk of work so far presented their system contains an array of identical PEs, a host CPU, and a large system memory, but offers some different approaches to counter familiar parallel processor challenges.

Putting the proposal in the domain of MPPAs, each PE contains a small but complete microprocessor. Rather than just an ALU, a small scratchpad memory, and facilitating I/O and control logic units. The microprocessor is used exclusively to execute program segments (*modules*) allocated to the PE, with all administrative control handled by dedicated functional units.

Acknowledging that a coherent global shared memory is not scalable, yet seeking to avoid the data sharing complexities of distributed memories, an event-driven asynchronous execution model is adopted. Using a coarser granularity of operation than the dataflow architectures, program segment modules execute atomically on the

arrival of predefined inputs. Together, the event driven model and atomic execution make a globally shared and multi-ported memory possible without requiring a coherence protocol. The approach also simplifies the programming task by removing the need for synchronisation measures such as locks or barriers, and eases debugging as problems are isolated to a module and its inputs.

While a global memory with correct access ordering makes data sharing easy, access latencies are a motivation for the use of prefetching. As module execution is triggered by input arrival, and a module's inputs are predefined, prefetching of data and instructions can be done accurately. Practically, each PE's scratchpad memory is divided in two, with one half servicing the current module and the other storing prefetched data and instructions for the next module to follow.

Prefetching also serves to hide the overhead incurred by dynamic scheduling. A further variant on the traditional dataflow architecture, dynamic scheduling is proposed as it offers better parallel resource utilisation.

To minimise communication latencies the task of dynamic module issue is administrated by a designated PE. This PE runs the execution engine; a software scheduling algorithm that tracks the generation of results relative to modules input sensitivity lists. Produced at compile time, modules move between three queues, waiting (for inputs), ready (inputs have all arrived), or run. The execution engine moves modules accordingly between the queues, and on request for more work by a PE, issues the next modules from the FIFO to it. Multiple execution engines might be implemented to prevent congestion from too many PEs requesting input.

Requests for new modules, output data value statuses, and requests for system memory accesses, are all delivered to the execution engine via a point-to-point NOC using message queue FIFOs in each PE. The requests for system memory access are made via requests to the execution engine which then accesses system memory with the help of the host CPU.

As the system is only a simulated proposal no programming tool chain is discussed. Using task-level parallel benchmarks with a high rate of short tasks and heavy dependencies between tasks, i.e tasks that would perform very poorly on a GPGPU, they find the system to be efficient scaling up to 56 PEs. The authors conclude that with a more sophisticated scheduling scheme (currently first come first serve) and the use of more execution engines the system might scale to 100's or 1000's of PEs.

2.2.2.2 Kalray

A recent commercial endeavour to use the MPPA term, the Kalray processor [Kalray, 2013] is very different to the above and is aimed primarily at embedded systems but the company also advertised its suitability for use in, signal processing, cryptography and networking, control, multimedia and HPC. The use of GCC and C/C++, hardware support for full IEEE 754 FP arithmetic (through precision configurable ALUs), and support for OpenMP and POSIX parallel libraries, make its use in HPC look plausible.

As a commercial endeavour the published details are minimal, however two particular novelties in the hardware include, the use of a 5-wide issue capable VLIW architecture as the processor in the PEs, and the use of a partitioned global address space (PGAS). The array described consists of 16 clusters of, 16 VLIW PEs, sharing a memory unit, and an additional dedicated I/O core. The dedicated I/O core runs a custom OS handling PCIe, ethernet, GPIO, and DDR interfaces. The use of a full core in support of simpler cores in clusters is not novel. But as the first instance of such in this review it is worth noting as another way of approaching many-core architectures in a manner divergent to this works. Details on the interconnect between cores are not given other than that there is a propriety NOC.

Two programming interfaces are offered, each with their own tool chain. In the first a program is modelled as a dataflow graph consisting of *agents* passing data through *channels*. Once programmed so as to represent a dataflow graph with concurrent portions, the dataflow compiler is able to map these agents to clusters, assign memory resources, and arrange the I/O. A dataflow simulator, profiler and Trace and Debug platform facilitate the programmer in extracting performance. In the second, POSIX or OpenMP threads are mapped to cluster cores under the management of the I/O core OS.

2.2.2.3 AMBRIC

An earlier commercial endeavour, AMBRIC[Halfhill, 2006], ultimately failed and is unsuitable for HPC but has been used often in the literature as a comparison architecture. It sits as one of a plethora of MPPAs to emerge in the mid 2000's, designed primarily for image processing (integer only arithmetic). Compared to its competitors which used HDL (MathStar[1]), custom languages (SeaForth[Seaforth, 2013]), ex-

[1]The website is no longer available but [Halfhill, 2006; Lanuzza et al., 2007; Top and Gokhale, 2009] references it.

tensions to C (SiliconHive[1]), or a combination of such (D-Fabrix[Halfhill, 2005], PicoChip[2]), AMBRIC developers used a subset of Java to make for easy programming. Sequential Java routines were parallelised using a graphical scripting language, exploiting predominantly only data-level parallelism.

The PEs themselves consisted of two types of custom RISC processors. Depending on selected word size both could offer multiple instruction issue providing a limited form of ILP exploitation. Of significance, the hierarchy of PEs adopted a globally asynchronous locally synchronous approach. Based on programmer selection varying sizes of clusters of heterogeneous PEs operated synchronously within themselves, but at a clock speed that depended on load and not necessarily in synchronisation with other clusters.

To provide synchronisation the simple channel interconnect offers and receives data from and into designated registers accessible to the internal ALUs when desired. Either end can inhibit an offered transfer.

In various applications (image processing and others) AMBRIC achieves more than real-time performance but is not insignificantly beaten by custom FPGA implementations [Hutchings et al., 2009; Top and Gokhale, 2009]. The obvious trade-off being ease of programming.

While clearly not suited for HPC AMBRIC's most notable characteristics lie in the chosen means of synchronisation. Globally asynchronous while local synchronous operation advantageously enables fine-grained power and performance controls. Within such an environment, the interconnect used, as in the case of the above proposed event-driven array, contributes the imposed synchronisation and correct ordering advantage of dataflow architectures. By giving control to both ends of a channel an operation will not occur before the correct data has arrived and nor will the data be overwritten until used. Using point to point channels of interconnect guarantees scalability but suffers from high overheads when broadcasting data.

2.2.2.4 Comparisons

To conclude, while MPPAs do not generally fit within this works definition of many-core processor as the PEs are too complex, and neither are they conventionally used in HPC or benchmarked in a comparable manner, of the plethora in existence, the above in particular make contributions to the discussion worth noting.

[1]The website is no longer available but [Blake et al., 2009] references it.

[2]Was aquired by www.mindspeed.com in 2012 but was also referenced most recently by [Choi, 2011]

The previously reviewed Dataflow architectures with their use of static compilation achieved, simple control, asynchronous and therefore scaling operation, avoided expensive coherence protocols, and cumulatively achieve very fine-grained parallelism in line with the goals of the hypothetical many-core. The event-driven model and globally synchronous locally asynchronous natures of the first MPPA proposed event-driven array and AMBRIC, together with the prefetching of the former, demonstrate alternative means of achieving similar gains. But this is at the costs of a coarser granularity of parallelism and possibly poor performance on applications only demonstrating radical parallelism on small units of work. The coarser granularity also improves programmability, but this is necessary as where the dataflow architectures relied more on their compilers and static scheduling these architectures rely more heavily on the programmer for parallelism and control.

Kalray further suggests a method of integrating OpenMP and POSIX support to a many-core environment through the grouping of full Linux cores with clusters of PEs. While not described in detail Kalray also demonstrates the use of a PGAS memory model in this environment.

In terms of gleaning from their experiences, the following points can be drawn out:

On operating model, the proposal's joint use of event-driven and atomic execution, and AMBRICs channel and FIFO interconnect, both achieve the same gain as the dataflow architectures; asynchronous operation. As their primary finding, the proposal suggests that rather than static scheduling the overheads of dynamic runtime scheduling, in addition to memory access latencies, can be hidden by prefetching. This is while still improving programmability and resource utilization.

On memory and data movement, the proposals of matching the asynchronous control system that already guarantees correct order, with a multi-bank multi-ported shared memory rather than a coherence protocol is appealing. However, the cost is a more sophisticated core in each PE capable of running larger tasks/operations to mitigate the access latencies incurred, and more control logic in each core. From the limited results it is difficult to separate the memory access and dynamic scheduling latencies in order to determine a minimal operation size. More testing on the memory model is needed. Kalray proposes, along with others in this review such as Rigel, the use of PGAS which theoretically results in less memory traffic, this will be discussed in more detail in a later section.

On programmability, in the case of the proposed event driven programmable array their approach further gains a simplification in expressing parallel operation as no manual barriers or locks are required. AMBRIC's use of Java and their debugging environment appears promising for multimedia type applications but would not be suitable for the size of HPC applications. Kalray's support for C/C++ and parallelising pragmas offers promise but as of yet no results for comparison.

2.2.3 Vector, VLIW, and Streaming Processors

Historically the term streaming processors has primarily referred to the embedded realm of accelerators for multimedia applications. But it has also been applied to a range of other architectures which, in addition to DSP chips also includes GPUs and IBM's cell broadband engine [IBM, 2012] (both used for gaming and HPC), and clusters arranged as streaming systems designed for HPC [Dally et al., 2003]. The current prominence of GPGPUs in HPC has already been noted, but various other studies have looked further into the potential for alternative streaming type architecture in HPC [Ahn et al., 2004; Yang et al., 2007; Zhang et al., 2008].

The domain is relevant to this work partly because it contains GPGPUs (the closest architecture to the hypothesised many-cores in main stream HPC use), but also because these architectures exploit a level and granularity of parallelism lower than multi-cores. Common to the various implementations of the model are two unifying characteristics, the use of high bandwidth memory hierarchies, and wide parallel processing structures, traditionally vector processing units (VPUs), VLIWs and more recently GPUs. The first exploits data locality with LOAD and STORE operations that occur in vectors or streams worth of data. The second aims to exploit data parallelism through simultaneous processing of multiple operands. Both factors are indicative of the type of application most suited to these designs, but a many-core architecture is also going to need to negotiate supplying high data rates to each PE and clearly is also a parallel processing structure albeit on a much larger scale. How these two characteristics are practically implemented (deep pipelines, multiple processing channels, or both in various forms) varies, as does the programming abstraction used. The range of intrinsics supported in each also varies significantly.

Vector processors While the term "streaming processor" has only recently been applied to HPC systems, vector processing units at least have long been used since the

early supercomputers. First in 1976 with the Cray-1[Russell, 1978] and its later versions, the NEC SX[Watanabe, 1987], or later still the VPP500[Utsumi et al., 1994] to name just three examples. Subsequently, as transistor counts have increased, commodity CPUs have also added support for vector instructions. Intel for instance first introduced MMX instructions capable of concurrently processing 64-bits either as a single integer or as two 32-bit integers, four 16-bit integers, or eight 8-bit integers. Its successor, AVX, expanded to offer 256 bits, and as will be discussed later the Phi adds further extensions to reach 512-bits. AMD and others have implemented their own similar equivalents.

A vector register file and data path feed a vectors worth of deeply pipelined functional units, collectively a vector processing unit (VPU). To ensure performance on real world codes, generally additional scalar pipelines and scalar register files for scalar code sections accompany the vector functional units. Conforming to the streaming model, SIMD operation and a high bandwidth memory hierarchy reduce the latency cost of LOADs and STOREs, and the required fetch-decode bandwidth. Vector chaining offers a means of operating on intermediate results directly without returning them to memory first. Software strip mining ensures as optimal as possible use of the VPU width when mapping loops to the hardware width. Independence between operations within a vector is guaranteed by the programming model. Collectively these features allow for deep pipelining and efficient functioning.

While such a vector pipeline does make use of a complex logic consuming data and control path, control is all on a vector rather than element basis resulting in power efficiency gains. The SIMD nature also means reduced program sizes. Due to the streaming form of operation on bulk data, when advantageous the fetch-decode hardware can be powered down while a vector is processed. Parallelising loops reduces the number of branches to be taken. And finally vector memory access is more efficient than scalar. Most commonly, however, the programmer is responsible for finding the parallelism in an application for execution on these architectures.

Very Long Instruction Word Processors Alternatively, VLIW (already mentioned in reference to Kalray's smaller units above) is most particularly relevant to the many-core discussion as while still exploiting ILP, the onus is placed on a sophisticated compiler for finding it, in line with the argument for less complex hardware. By grouping multiple different functional units (FUs) together into a single processor for simultaneous use, a single instruction word (bundle) containing operating directions for

each FU is used to issue all instructions in one fetch-decode cycle, achieving a similar saving as a VPU but without the limits of true SIMD operation. While multiple FUs remain (as in superscalar), and instructions may be executed out-of-order, and each FU makes use of a deep pipeline, as far as possible the complex infrastructure involved in determining and correctly circumventing or adhering to dependencies is passed on to the compiler.

While not strictly a streaming architecture and designed for general purpose use, Intel's explicitly-parallel instruction-set computing (EPIC) based Itanium architecture, is the only VLIW based architecture to find significant use in HPC and therefore used illustratively here. The latest Itanium revision, Poulson [Intel, 2012b], was released in 2012 supporting 8 cores of 12-wide issue VLIW operation, multi-threading, and 54MB on die cache.

Many concepts of superscalar architectures are incorporated into Itanium, branch prediction, dynamically scheduled pipelines, register renaming, and scoreboarding. The complexities are, however, not equal to say on a Xeon due to the compiler capacity. For instance to improve performance by filling empty spots software speculation schedules safety net check instructions where independence is unconfirmed. Similarly in implementing branch prediction, trace scheduling is used to schedule a whole program trace based on a predicted outcome. Rather than implementing complex recovery hardware the alternative trace and a set of roll back clean up instructions is instead also simply provided for when the prediction was incorrect. For branches that are difficult to predict or short, predication can be more beneficial. Both branches are scheduled to execute but the incorrect one is disabled on condition evaluation.

Despite VLIWs potential advantages a number of significant limitations in the classic model have precluded VLIW from becoming dominant in main stream domains, although adaptations have found niches. Firstly, depending on the application, compilers have struggled to find sufficient operations to keep the full width of FUs sustainably occupied. Secondly, as alluded to and as with all statically scheduled systems, dynamic interruptions such as interrupts, cache misses and unpredictable latencies, or branches hurt performance. While cache misses can be predicted with a high probability, recovery from a miss must still be handled which can be expensive in register storage across a multiple pipelined functional units. Thirdly, a compiler is tied to a specific revision of implementation of an architecture as it requires foreknowledge of the latencies of all operations, and the number and range of FUs available. To handle the latter, Itanium was designed to process groups of very long

instructions. All instructions within a group are guaranteed to be independent making the width of the processor in use irrelevant provided it does not exceed the group width. Fourthly and contradictorily to the notion of reducing required silicon, the architecture is wasteful in terms of instruction memory, potentially storing many NOPS, and repeated operations where loops have been unrolled, ultimately being generally predisposed to significant instruction bloat. To address such, later designs formed variable length instructions so as to have a shorter instruction when only executing a few instructions, others made use of compressed formats although such requires decompression hardware, and others such as Tilera [Tilera, 2013] created special case instructions for instances such as when only one operation was to be called. Fifthly and finally, due to the centralised fetch-decode module and register file, scalability is limited. A body of work exists addressing the scalability problems of the centralised instruction decode and fetch modules and register files. [Zhong et al., 2005] presents work on decentralising the control path for VLIW processors and summarises others work on distributing the register file and FUs into clusters. Clustered VLIWs (multiple VLIWs together to make a "wider" VLIW) encounter communication delays between VLIW processors but have found use in printers predominantly.

VLIW based architectures have failed to find a dominant position in any domain but have nevertheless been thoroughly investigated and repeatedly created in various forms. Embedded media applications have at times made use of the concept as more programmable than a single application ASIC, yet with greater efficiency than a more general purpose processor (examples include the TI C6x series, Lx/ST200, and Philips TM1300).

Streaming processors The streaming model structures applications into a series of sequential compute kernels which operate on streams of data elements by applying the same instructions on all items. Kernels are isolated in terms of memory access to only the head of an input stream and tail of the output stream, minimising the memory accessibility requirements. Software pipelining of these kernels together with spatial and temporal locality within the streams, mean that the data path, from off-chip memory down to ALU, may be pipelined and bandwidth optimised. Extending vector processors, the overhead and cost of high bandwidth and memory access latencies are therefore staggered and amortised through a deeper memory hierarchy customised for bulk stream transfers. While software pipelining of kernels in this manner

may be useful to many-cores, the similar limited data accessibility is not sufficient in a MIMD environment.

In a streaming processor therefore, the same set of instructions, or micro program, is applied to all the elements within a stream of data. Exploiting this hardware local stream buffers and direct producer-consumer pathways save on memory bandwidth. In vector processors a single common instruction is applied to each element within a vector and chaining provides for a degree of direct producer-consumer data passage. But the architectures containing VPUs generally also supply a scalar unit and do not generally support or expect as extensive a degree of streaming.

The argument could be made that GPGPUs are vector processors, in the form of multi-core versions where each streaming multiprocessor (SM) is a 32-way hyper-threaded 32-wide vector processor with support for predication. But these also include extensive support for streaming data as their performance gains rely on simultaneously overlaying both operations and memory accesses.

These underlying model properties allow for a number of hardware short cuts and optimisations. It is implied that the programmer or compiler is responsible for grouping and ordering tasks into kernels and assigning the input and output data streams to these. Consequently minimal global signalling and control logic is needed which facilitates a scalable design with potential for many PEs. The independence of kernels means that where additional ILP or DLP is exploited within a kernel, any resulting communication is local and therefore main memory does not need to be accessible to the PEs. Further with the producer-consumer pipeline of kernels, using an output stream as the next input stream, intermediate results need not be written back to off-chip memory providing efficient local-only data transfers. Finally, SIMD execution, whether fine or coarse grained, reduces the memory, fetch, issue and decode logic required for instructions, and the data path control logic.

Considering the wide array of domains streaming, VLIW, and vector processors that are used, the following section examines only a limited selection of those that have been used in HPC. A further subset of streaming processors includes systolic arrays which also make use of the characteristic many small PEs, point-to-point communication systems, and static scheduling. Generally operating in lockstep each PE performs either a compute or communication service each cycle, ensuring data is passed from one PE to the next, each performing a specific step. The linear interconnect and pipeline processing mechanism used, however, is too limiting for general purpose computation.

2.2.3.1 Imagine

The Imagine processor [Kapasi et al., 2002, 2003] is widely cited as a comparator in the literature on many-cores, as a streaming based exploration of the model. Although designed specifically for multimedia processing following its release two studies additionally examined its potential for scientific computing [Che et al., 2008; Du et al., 2006; Zhang et al., 2008].

Imagine is operated as a co-processor, where one host is able to co-ordinate a number of Imagine chips. Each Imagine processor chip contains 48 compute units (in the form of 8 clusters, each containing effectively a 6-way VLIW PE), one shared SRF, a scalar microprocessor core, stream controller and interface logic. All 8 clusters receive their instructions collectively in a SIMD manner, and operate in lockstep. A SDRAM is assigned per processor chip, these are not shared coherently with the host or other processors.

The six pipelines of each VLIW are: three ALUs, two multipliers, and notably a division and square root unit. Going beyond many other media processors, all arithmetic units support both integer and floating point operations. Supporting these within each cluster are, a communication unit, local scratch pad memory (1KB), and local register file (LRF). To facilitate scalability, the LRF is partitioned to give each functional unit its own dual ported register file.

Program flow is carried out according to a schedule jointly coordinated by the host processor, the stream controllers, and the micro-controller. The stream controller serves to direct instructions from the host to the correct module, these include commands to transfer data and programs between the different units, synchronise the clusters, and initiate kernel execution on the micro-controller. The stream controller uses a scoreboard to determine what hardware resources are available dynamically. Differing from some of the direct data path links discussed elsewhere, intra-cluster communication takes place via a switch, coordinating movement between the ALU outputs, SRF outputs and LRF inputs. Inter-cluster communication is also possible via the network interface which includes off-chip communication.

One particular hurdle to using Imagine is that it is programmed in two custom streaming languages, KernelC and StreamC, in addition to allowing a programmer to embed arbitrary C/C++ in the StreamC codes. StreamC provides a means of ordering kernels and organising data into streams. Its compiler checks for dependencies

between kernels and stream load/stores, looks for opportunities for strip mining and determines the optimal allocation and scheduling of the SRF.

At a lower level, KernelC is used to enforce the rules of streaming memory access and is the context in which applications are fitted to stream data access and cyclic execution mode. Kernel instructions are VLIW instructions issued to the clusters in a SIMD manner, each cluster accesses the same LRF location but in its dedicated LRF. These instructions operate with all latencies known making it possible to statically issue a complete instruction schedule. Data dependencies are identified at compile time, as in other architectures reviewed, removing the need for register renaming, dependence checking, or any dynamic instruction issue hardware. The KernelC compiler applies high level optimisations including loop unrolling, and software pipelining, and performs a data dependency analysis. Included in the kernel instructions, are specification for data movement between ALUs and LRFs.

Three mitigating approaches are used in addressing the performance costs of static scheduling. In the first a select operation makes it possible to predicate sections of code. In the second, along with standard loop termination conditions, the end of an input stream can be considered a termination signal. Finally in the third, conditional streams are supported where in effect items may be accessed conditionally on a per PE basis. Classically a SIMD streaming architecture has no control over which stream data item is retrieved next. Generally the next available value in its LRF is simply delivered to an ALU. If movement of data between LRFs is required it is done so at the cost of a memory operation. Equally there is no data dependant control over how values are placed into an output stream. Conditional streams, however, make it possible to ignore some elements moving on to the ones following, or alternatively to select an element from a different LRF. In effect this makes it possible to convert conditional control decisions into conditional data routing decisions resulting in more efficient use of PEs. For instance where a case statement is highly unbalanced or element processing latency is value dependant, elements can be conditionally separated into shorter single case or common latency streams.

Although programmability was a point of concern, [Ahn et al., 2004] concludes Imagine is an efficient architecture capable of providing performance in the multimedia domain. Those examining Imagine for scientific use found that given compute rather than memory bound applications and provided the problem set fit entirely on-chip, Imagine significantly outperformed x86 cores in floating point operations per cycle (but not raw performance as it runs at 400MHz). Unfortunately Imagine also

performed badly regarding efficiency primarily due to idle ALUs, a result of limited ILP and load imbalances.

The authors do not identify or speculate on the root cause of these failings. It is therefore unknown it might be possible to rectify such given a better compiler or perhaps more FP units as appropriate for the domain.

2.2.3.2 Merrimac

Developed during the same time period as Imagine, Merrimac [Dally et al., 2003] is an investigation into what a streaming system designed for HPC would consist of. Preceding the focus on the power wall Merrimac was based on the apparent restriction expensive bandwidth was imposing on cheap arithmetic logic. As such the work describes a goal of creating a system with a higher ratio of arithmetic logic to bandwidth. More recently bandwidth remains expensive and tight power budgets means that only efficient use of dark silicon is really cheap. However, the many-core hypothesis could be restated as a focus on increasing the ratio of arithmetic logic to all other power costing infrastructure. The commonality suggesting this work may have relevant comments to contribute.

Taking its queue from traditional supercomputing systems, Merrimac scales up to a two petaflop system of 32 cabinets. Each cabinet contains 32 boards of 16 nodes (processor chip and memory). Each processor chip consist of 16 clusters each containing 4 FP MADD functional units and a scalar processor. These boards and processor chips are the point of interest where the memory hierarchy effects advantages in bandwidth savings. For each 2GB main memory DRAM an on-chip cache bank interfaces with all clusters. Within a cluster a SRF is shared with four LRFs each of which is allocated to a functional unit.

By moving streams worth of data between main memory and the cache and finally the SRF, the transfer overhead is reduced, and producer-consumer locality between kernels is exploited. By applying a kernel to an entire stream, memory operations can be overlapped with a kernel's execution. To do so Merrimac uses a similar stream instruction set as Imagine, consisting of scalar operations which are executed on a conventional scalar core, stream instructions that trigger kernels to operate on a stream in the SRF, and the suggested stream memory operations including scatter and gather.

Within the SRF data words are aligned for transfer to the appropriate LRF at minimal cost. The LRFs are sufficiently large so as to exploit locality within streams,

by storing both operands and intermediate results minimising the number of write-backs and fetches required. Balancing the size of LRFs versus the requirements of kernels is carried out by the compiler when assembling kernels, if a kernels data requirements are too large there will be excessive data swapping between layers somewhere in the hierarchy. Minimising the cost of these transfers, the LRFs are physically aligned with ALUs, requiring only short distance on-chip communication rather than chip wide. A cluster switch does make data sharing possible between LRFs.

Although only simulated, they found that for a given range of scientific applications the model proved cost effective (128MFLOP/$) and scalable up to a 2 PFLOP machine.

For a system placing 16 clusters, or 64 ALUs, on a chip, the architecture scales to 1M MADDs for 2PFLOPs. This puts it in the league of the BlueGene architectures discussed later. Depending on how a PE is classified, the Merrimac chips can claim to support 64 PEs (4*16MADDs), which fails to reach the many-core vision of hundreds, but offers a different model of many PEs that demonstrates an approach to the problems of control, I/O, and memory-hierarchy that enables performance for some HPC applications. With more current fabrication processes, more nodes may be fitted to the single chip making the system a scalable many-core supercomputer. Whether the same and other applications would perform on such, however, is unknown.

2.2.3.3 Rigel

Rigel [Kelm et al., 2009] is another example of many-core integration, which takes a different point of attack. To minimise scheduling complexity and on-chip communication load, a single program multiple data (SPMD) programming model is adopted. This limits the parallelisation scope to task and data parallelism and sets the context for a low overheard (low latency) interconnect. To further minimise non-compute hardware complexity, a significant quantity of work is passed on to the programmer. Due to its time of design (2007) general purpose use was not a consideration but rather it was designed specifically to be an accelerator for data and task parallel applications, almost as a competitor to GPGPUs. A more recent paper [Johnson et al., 2011], however, has reviewed it in the current context and makes suggestions for how it might be more general purpose. Given its original target the PEs used are RISC like dual-issue cores containing an in-order ALU and pipelined single precision FPU.

Rigel attempts to balance performance, programmability, and non-arithmetic hardware consumption, primarily via a unique memory model. A single address space

supports a measure of coherence in a balance between programmability and hardware costs. To do so the common mechanism of PEs clustered hierarchically is used, eight PEs share a 64KB cache to form a cluster, 128 clusters form a tile, eight tiles make up the processor. Alongside the tiles 8MB of on-chip global cache serve as a buffer and are the primary point of coherence. By using the local cluster caches for data sharing within a cluster, the need for message passing and the associated latency and logic overheads are removed. Instead the local cache is implicitly coherent at the cost of ordering hardware only. These cluster caches observe a lazy write-back protocol, observing no hardware managed coherence with the global cache except for on eviction. This again limits hardware and network traffic costs. Synchronisation, however, is therefore primarily the responsibility of the programmer and the compiler.

In line with the execution model adopted the compiler will force a global synchronisation following the execution of a set of parallel tasks. Additionally the programmer is given the ability to manually force a cluster cache flush. This will synchronise a line or the whole cache with the global cache, but will still not invalidate any other copies of data in other cluster caches. For full coherence further broadcast invalidation and update instructions also exist. Alternatively a programmer is also able to insert synchronisation barriers at points which force global memory coherence at the cost of global memory access and broadcasts.

Relative to the alternatives, Rigel relies heavily on the programmer to ensure optimal data locality, correct memory ordering, and task parallelisation. In order to facilitate such a low-level interface API is provided giving direct access to a range of atomic operations incorporating direct access of a single PE to the global cache, the creation, placement, and managements of task queues, the ability to insert synchronisation barriers, and the already mentioned flush and invalidate instructions. Through this a programmer is able to exploit fine-grained task parallelism at negligible overhead on the scale of a 1000 PEs, which requires the low latency global synchronisation means described above.

Execution occurs according to a single binary but PEs execute threads of independent control flows with all synchronisation taking place on barrier insertion, and atomic operations. Temporal and spacial locality are provided in the hardware as much as in any standard cache hierarchy, but is obviously influenced by a programmer's co-location of data sharing tasks. Included in the low-level API are also instructions for explicit prefetching and direct bypassing of local memory.

In reviewing the architecture in today's context a number of points stand out. The cluster shared cache model was considered sufficient for inter-PE communication due to an assumption that inter-cluster sharing would be uncommon and with most of such taking place in the form of writing back results to main memory after barriers. This does not necessarily hold for a more general purpose many-core. Due to the fabrication technology at the time Rigel was limited to single precision FP only, looking at current GPGPUs this is no longer necessarily a reasonable constraint.

Noting a constraint all of these architectures face, the review paper acknowledges the relative memory bandwidth bottleneck. The authors offer the possibility of using optical interconnects and 3D die stacking to increase bandwidth without exhausting the power budget, and notes how this makes data locality all the more important.

Rigel was made as an accelerator the same as GPGPUs. More recent GPGPUs, while remaining accelerators, do now support a form of virtual memory, and allow for a form of multiple concurrent kernels. Rigel holds an advantage over GPUs as it is free of all the graphics specific hardware.

2.2.3.4 RAW

The RAW [Taylor et al., 2002] microprocessor was an architecture designed to be general purpose in its use of an array of PEs and a NOC. Older than most reviewed here, it went on to become the commercialised Tilera [Tilera, 2013] chip set for which understandably there is less detailed information available. While lacking classic streaming characteristics, it is included in this section because it also preforms best when operating on streamed blocks of data. When a set of its PEs are commandeered for a streaming appropriate application, direct access to off-chip memory makes a streaming operation and sequence of kernels easy to map to the hardware.

RAW's most distinguishing feature is it's low latency interconnect. Without a hierarchy the processor consists of a single array of 16 fairly standard MIPS based 8-stage processors each accompanied by, a FPU, branch prediction unit, and separate 32KB data and instruction caches. These are connected by four (N,S,E,W) full duplex channels arranged as point-to-point connections for scalability. Two of these are programmable and operate according to static network instructions generated at compile time and stored in a dedicated network instruction cache. The remaining two provide a more dynamic service to handle interrupts, cache misses, and unpredictable messages. To reduce latency even further, and facilitate ILP specifically, both networks pick up their data directly from the ALU bypass connections.

The static network therefore acts as a bridge between PE by-pass networks, effectively providing register type latencies and thereby allowing for superscalar like ILP exploitation. To minimise the cost of dynamic messages the compiler may make conservative predictions of what messages might be sent and statically reserve channels in the static network for such. Alternatively the compiler will use conservative latency estimates and statically schedule around such. Finally, both networks are muxed directly to the chip pins making provision for DMA and direct inter-chip or alternative peripheral access. This off-chip memory access sits at the top of a memory hierarchy further consisting of shared L2, L3 and L4 caches and finally distributed L1 caches.

The dynamic network has a longer latency than static because each router needs to read the packet header. The static network instructions already know all movements that will occur and the router can therefore be pipelined such that data is transferred immediately (1clock cycle per hop, ~2 if its a corner but the compiler routes to optimise this). Therefore stream transmission rather than scalar words are more efficient on the dynamic network.

RAW set out to be capable of exploiting all forms of parallelism, TLP, ILP, DLP, and streams. To do so it aimed to expose all levels of hardware to the compiler for static scheduling, including PE registers and ALU data paths, while also supporting dynamic messaging. The C (and originally Fortran) compiler supports multiple processes and employs a gang scheduling policy for sub-threads in each.

Unlike all of the statically scheduled architectures or the superscalar dynamic dependency checking of traditional CPUs, the RAW compiler makes use of the parallel PEs and statically assigns the task of dependence checking to specific system nodes. The compiler does, however, take the step of finding fine-grained ILP and considering the communication and synchronisation overheads, before assigning tasks to PEs for static instruction issue.

[Taylor et al., 2004] made an extensive examination of RAW's performance, executing SPECfp, SPECint, a number of dense and sparse matrix scientific applications, and the STREAM benchmark. Relative to a common processor of the day (a Pentium III) emulated 16 and 32 PE RAWs achieved speedups on some applications. Where achieved the performance gains were attributed to the 16 way parallelism, and the significant bandwidth through the streaming data access direct from the network as opposed to fetching from caches. Where ILP in an application was limited the authors proposed a 2-way issue ALU as beneficial, and indicated that further work was needed on the RAWCC C and Fortran compilers in use.

Based on the available specification sheets, Tilera's top end 64b Tilera Tile-Gx8072 processor shows some fundamental adjustments to the original philosophy. The architecture has clearly moved to focus more on the embedded networking and multimedia markets with FP support present but more limited. The basic PE has fundamentally changed to a 3-way issue VLIW design with support for SIMD instructions. Each PE can still execute independent processes, or a collection can run SMP Linux. With fabrication advances the chip runs significantly faster at 1.2GHz (relative to 425MHz) and supports up to 72 64bit PEs. A patented NOC, termed iMesh, is in use but not detailed, and they have extended their distributed memory to the patented "Dynamic distributed cache" with a much deeper fully coherent hierarchy all the way up to a conglomerate L3, making it distributed but shared. L1 and L2 constitute the distributed on-chip portions of the hierarchy which remain directly accessible in their PE's from off-chip. Programmability is improved with new graphical multi-core debugging tools but remains using the standard GNU GCC tool chain and now focuses solely on C/C++.

A paper [Muddukrishna et al., 2013] on task scheduling uses Tilera's alternative processor flavour, the TilePro64 [TILEPro, 2011], as a demonstration processor and includes more details. The TILEPro64 lacks all floating point support and exhibits smaller cache sizes but use the same hierarchy. [Muddukrishna et al., 2013] describes how the cache hierarchy while hardware coherent is also programmable. The coherence protocol allocates every cache block sized chunk of main memory to a specific bank of the L2 cache known as the home cache. The location of this home is selectable in software. The home cache makes all load and store requests from all tiles. On a miss the home PE will supply the block requested and depending on the software configuration this will be delivered to either or both the requesting PE's L2 and L1 caches. All caches write through to the home cache. This system is similar to Rigel but in this case the main cache is one level down (L2 instead of L3), consequently distributed, and configurable.

The on-board cache exhibits non-uniform access times due to its distributed nature, and the optimal use of it can be compared to the extensive work on running OpenMP code on NUMA systems, although the scales are very different (10's of cycles vs 1000's). The [Muddukrishna et al., 2013] paper showed how important it was to take these effects into account and make use of the ability to configure the distributed cache.

2.2.3.5 GPGPUs

Of all the architectures reviewed here GPGPUs are the only ones in main stream HPC use, and further are the only architectures achieving the hypothesised thousands of simple ALU cores, albeit via SIMD operation simplifying the control units required. With 3584 GPU cores (scalar ALUs, also termed thread processors), AMD's FirePro S10000 [AMD, 2013], containing two Tahiti GPUs has the highest core count outright. NVIDIA's GK110B based K40 alternatively offers 2088 GPU cores [NVIDIA, 2013c]. Finally, while not strictly a GPU of any form but of a similar nature and programming model, Intel's Xeon Phi [Intel, 2013b] contains 976 GPU cores. In contrast to NVIDIA's propriety CUDA or AMD's use of OpenCL, the Phi offers full support for both the legacy x86 instruction set and OpenMP and MPI in traditional C and Fortran. The markedly lower core count is as a result of this support, core numbers have been traded for improved code portability and programmability.

In hardware, these GPU cores are grouped into 16 lane wide SIMD vector processor like units. Each VPU serves as a SIMD hardware thread, multiple of which are grouped to create the GPU equivalent of a traditional core or PE. In NVIDIA terminology these multi-threaded SIMD cores are streaming multiprocessors (SMXs), or in AMD nomenclature they are compute units (CUs). Phi cores, or PEs differ here, containing only one vector unit and an x86 pipeline. In each, an array of these PEs creates a SOC. To improve performance in code containing short conditional branches, register masks enables per lane predication and branching, albeit at an efficiency cost.

To overcome the practicalities of feeding, controlling, and programming these thousands of cores, a novel level of compromise in hardware flexibility and control is adopted via an entirely new programming model. Dubbed by NVIDIA as single instruction multiple thread (SIMT), SIMT operation provides a less limiting form of SIMD control by allowing multiple threads of SIMD instructions to be inter-weaved, thereby hiding memory latency. The Kepler, GCN, and Phi architectures are capable of supporting up to 64, 40, and 4 SIMD threads per PE respectively. The lower number of threads in the Phi is in part compensated for by larger caches relative to the others. However, in all three the caches are small relative to a traditional CPU core. The architectures are targeting applications that benefit less from large caches and rather, as is characteristic of a streaming architecture, provide high bandwidth memory hierarchies that supply streams and vectors worth of data. Hardware scatter and gather

instructions help keep code vectorised when making non-unit stride vector memory accesses.

To achieve performance using this model, however, an even greater degree of parallelism needs to exist in an application than would otherwise be required if the hardware supported only single threads. SIMT remains a departure from the many-core hypothesis of many PEs carrying out independent instructions. Under such a definition the GPGPU type architectures only support 180 (K40), 112 (S10k), and 61 (Phi) different threads, short of the hypothesised thousands. Admittedly, however, compared to the very simplified PEs discussed earlier, these PEs are capable of issuing 8, 4, and 1, respectively, SIMD instructions per cycle, and clearly scale to much higher core counts.

Due to their lower clock rates and, SIMD nature, the use of GPGPUs in HPC takes the form of node accelerators rather than standalone nodes. The resulting memory separation, coupled with the relatively slow PCIe interface, means memory management is critical to achieving performance. Intel's most recent announcement of Knights Landing [Anthony, 2013] potentially offers a better approach in which the the Phi is a socketed processor to be paired with a Xeon. AMD and NVIDIA appear to also be working on similar projects with AMD's Heterogeneous System Architecture project [AMD, n.d] (HSA) combining four x86 CPUs with a GPU on chip, and NVIDIA's Project Denver [Gastor, 2013], another SoC integrating 64bit ARMv8 cores with a GPU architecture. Current systems, however, make use of a global GDDR memory on board the card sharing data and instructions.

From the GDDR global memory an on-chip shared L2 cache leads down to L1 caches private to each PE but shared between the VPUs. Portions of these L1 caches are further allocated to individual VPUs or SIMD threads to act as private caches while the rest remains shared. The ratio of such is manufacturer dependant. Finally each GPU core, or thread processor, has an associate set of registers. As seen in the other streaming architectures, this hierarchy is accompanied by an equally scaling bandwidth hierarchy that exploits locality and makes use of stream and vector data operations.

Table 2.1 below gives a numerical comparison between the K40, S10k, and Phi, in terms of these hardware components.

NVIDIA Tesla K40 Differing from the S10k the K40 contains a single GK110 [NVIDIA, 2013b] GPU. In the NVIDIA hardware CUDA threads are executed on a GPU core, dou-

Table 2.1 – A comparison of the hardware specifications in NVIDIA's K40 (GK110B), AMD's FirePro S10000, and Intel's Xeon Phi. FLOPS. FLOPS/W values are not given due to the benchmark dependant nature of such.

	Tesla K40	FirePro S10000	Xeon Phi 7120
Multi-threaded SIMD Cores, per GPU	15 SMXs	28 Compute Units	61 extended Pentium CPU cores
SIMD SP VPUs	12	4	1
Additional functional units	64 DP units, 32 SFUs,	1 scalar integer unit	1 64b x86 hardware pipeline
Width of VPUs	16	16	16
GPU cores per PE	192	64	16
IPC	8 vector instructions	4 vector instructions, 1 scalar instructions	1 vector instruction, 1 scalar instruction*
Thread storage	64 warps (2048 CUDA threads)	40 wavefronts (2560 thread processes)	4 Threads
Data L1 Cache	64KB	16KB + 64KB	32KB
L2 Cache	1.5MB	768KB	30.5MB
ICs	1x GK110B	2x Tahiti PRO	1x Phi
GPU cores (Total : Per PE : Per VPU)	2880:192:16	3584 (2*1792):64:16	976:16:16
ALU hierarchy per higher level (GPUS : PEs : VPUs : GPU Cores)	1:15:12:16	2:28:4:16	1:61:1:16
TDP	235W	375W	300W
Clock			
Processor	745MHz	825MHz	1.1238GHz
Clock boosting	<875MHz (GPUBoost)	Dynamic (PowerTune)	1.33GHz (TurboBoost)
Performance			
Peak SP TFLOPS	4.29	5.91	2.3
Peak DP TFLOPS	1.43	1.48	1.2
Memory			
GDDR5	12GB	6GB**	16GB
Max Bandwidth	288GB/s	480GB/s (240GB/s per GPU)	352GB/s
ECC support	Yes	Yes	Yes
Process technology	28nm	28nm	22nm
PCI Express	3.0	3.0	2.0
Display ports	None	4x Mini DisplayPort, 1x DVI***	None

[*]Only true for those operations enabled for pair issue

[**] At SC2013 a 12GB version was announced for release in the first quarter of 2014

[***] Specific to the Active cooling card

ble precision unit, or special function unit (SFU), operating with a program counter, registers, and thread private memory in the L1 cache. Threads executing within the same SMX share data via a shared section of the L1 cache, while sharing between SMXs takes place in the L2 cache. This latest revision, the Kepler series, has significantly increased the on-card memory, on-chip cache sizes, number of cores, and flexibility in configuration. Hardware support for two new features of significance to GPGPU use in HPC has also been added, dynamic parallelism, and Hyper-Q.

Dynamic parallelism involves using one kernel to launch another. The hardware support enables threads to, independent of the CPU, synchronise on results and create and launch new threads. By removing the CPU from the loop CPU-GPU communication can be significantly reduced and the programming required simplified. By doing so it also increases the range of parallel applications that may perform well on the GPGPU.

Hyper-Q allows multiple (32) MPI processes or threads to concurrently issue work to a single GPU. In consequence some applications will avoid what would have otherwise been false serialisation as processes wait for others even though the GPU is only being partially utilised. Practically the maximum number of possible work queues or connections in the GPU logic, between the host and the CUDA Work Distributor, has been increased in hardware. This can improve the sustained GPGPU workload and therefore system efficiency.

AMD FirePro S10 000 AMD's Graphics Core Next (GCN) architecture [AMD, 2012] makes use of the same fundamental architectural principles as NVIDIA but naturally differs in implementation. Comparing the SK10 to the K40 in Table 2.1, the most obvious differences involve differences in memory capacities and the use of 2 GPUs in one card has implications for L2 cache data sharing. Due to the commercial nature of both most of the micro-architecture specifics are unavailable, however, the following differences appear most prominently:

- The GCN clusters fewer (4 vs 12) VPUs into a Compute Unit (CU), the SMX equivalent, and instead has more of them (28 vs 15). This has consequences in terms of control and data sharing between VPUs within the CUs, and has meant that instruction caches can be shared between four CUs rather than each SMX having its own. Similarly while both use the SIMT operating model, the S10k issues threads in bundles (wavefronts) of 64 threads compared to NVIDIA's bundles (warps) of 32 threads. This, together with the distribution of GPU cores, means

the GCN architecture can support up to 2560 threads in flight per CU compared to the K40's 2048 per SMX.

- Where NVIDIA distinguishes 64 DP units and 32 SFUs per SMX the GCN appears to handle DP and transcendental operations directly in the SP VPUs using microcode routines.

- The GCN includes full support for graphics rendering and, unlike the K40, includes display ports. This support includes new hardware implemented image processing instructions. A motivation given for such involves the graphics applications within HPC such as gesture recognition and video searches.

- While both offer dynamic clock and voltage scaling dependant on workload and environment AMD's *Power Tune*[AMD, 2011] offers a much finer granularity and higher switching speed than NVIDIA's *GPU boost* [NVIDIA, 2013c].

- AMD does not yet appear to offer an equivalent to NVIDIA's dynamic parallelism or HyperQ. The latest OpenCL release 2.0[KHRONOS Group, 2013], however, does include software support for dynamic parallelism suggesting that the hardware support may follow.

Intel Xeon Phi 7120P/7120X While also a streaming PCIe co-processor, the Xeon Phi [Intel, 2013b,c] is arguably closer to a 61 core SMP on a single chip than a GPGPU. The Phi compute unit equivalent is a Pentium P54C core extended to include a single 16-way VPU. Table 2.1 shows how adding the x86 pipelines is a logic cost trade-off. This use of a standard instruction set, however, means OpenMP and MPI annotated C and Fortran codes can simply be recompiled rather than re-coded. To derive real performance gains for the architecture, however, code does still need to be tuned for the memory hierarchy, interconnect, and most specifically to make use of the 512b-wide VPUs which are not accessed via standard SSE or AVX instructions.

With a full core associated with it, the VPUs in Phi receive instructions and data through the Pentium core's scheduler [rezaur rahman, 2013]. This does, however, make use of GPU like L1 instruction and data caches, including dedicated vector buses in the case of the data. As with the GPGPUs, while fully pipelined, both the scalar and vector pipelines are in-order. Depending on the combination up to two instructions per cycle may be issued, one to the VPU and one to the scalar ALU. While

storage and control support exists for up to four threads per PE, this results in a far lower maximum in-flight lane thread count of only 68.

Making the unit SMP like and differing with the single unified GPU L2 caches, a fully coherent 30.5MB L2 cache is distributed across the cores and linked into one via a bidirectional core ring interface (CRI). The CRI consists of multiple separate links from a heavy weight wide data path through to lower weight address, flow control, and L2 cache coherence control paths. Similar to the AMD VPUs„ double precision, transcendentals, reciprocal, square root, and log are supported in hardware using the single precision ALUs rather than separate modules.

2.2.3.6 ClearSpeed

The final architecture included here is analogous to a single GPU compute unit joined to a scalar processor. The ClearSpeed processor [Clearspeed, 2007] was first released in 2003, at the time as a relatively novel SIMD streaming PCI co-processor. Designed for HPC vector and task loop unrolling the CSX architecture supported a RISC instruction set with full double precision floating point support. Running at 250MHz it could achieve up to 50GFLOPS sustained DGEMM performance for 25W.

The instruction and data hierarchy was able to issue one instruction per cycle to either, a scalar PE providing scalar execution and control, an array of 96 similar PEs offering SIMD operation on 5-way VLIW like units, or an IO unit. These instructions could be overlapped. Similar to GPGPUs branching within the array is handled with enable bit masks. Context switching between threads offered a less sophisticated means of SIMT operation. Apart from clock speed and scale the architecture differed in its lack of the various memory sharing levels GPGPUs use and its more complex VLIW fundamental unit of each PE.

Distinguishing itself from the alternatives of its time the programming model was implicitly parallel. As will be elaborated on in Section 2.3, an implicit model attempts to free the programmer from the lower level details of parallelism requiring only guidelines and leaving the compiler to carry out much of the work. Using an extension to C an application could be written as a sequential program. Only where parallel operations were expected their variables were declared with special keywords designating them as such and the functions called to operate on such were replaced by the ClearSpeed parallel equivalent.

The architecture saw some early success. Yet despite its more familiar instruction set and multithreading capabilities it has not been widely adopted. This is perhaps

due its similarities with yet poorer performance against GPGPUs and the less intuitive programming model provided in the face of CUDA.

2.2.3.7 Comparisons

To conclude, the architectures in this section, despite their common category span a wide range of approaches. Each addresses the problem of wide-span parallelism differently with different trade-offs leaving none of them as clearly demonstrating the best approach. While RAW and Rigel were targeted at significantly smaller problem sets, Imagine and Merrimac are directed at HPC size problems but adopted a more traditional cluster approach into which RAW and Rigel for instance could theoretically be placed. Alternatively the GPGPUs, Phi and Clearspeed make the distinction of being accelerators.

The use of the streaming model in HPC has very mixed results. There are a range of studies done to examine the potential performance of HPC applications on streaming processors in general. The uptake of GPGPUs is testimony to the fact that in many cases they can provide significant efficiency and performance gains.

In terms of gleaning from their experiences the following points can be drawn out:

On operating model, SIMD operation (VPUs, AVX, GPGPUs) serves scientific matrix and multimedia applications with a high efficiency (due to the amortising of control) and in some cases an easier programming model than MIMD machines. Where sufficient SIMD parallelism does not exist in an application, however, the width in both processing units and memory hierarchy can result in significant inefficiencies that must be counteracted with ILP techniques that are specific to each of the range of architectures described here. Optimisation of a memory hierarchy for streaming data is also clearly greatly advantageous in performance and efficiency, however, again this is very limiting in application domain applicability regardless of the implementation used.

On memory and data movement, the limitations of a co-processor model on a streaming architecture have already been noted and moves are under way to integrate GPGPU like architectures more closely with their host for memory access purposes. The use of static scheduling to specifically improve data movement in RAW is also uniquely interesting. In an architecture with a fully distributed memory system (without a cache)

such an approach could be used, not to predict and pre-empt unknown memory access latencies, but instead to ensure data has arrived at the correct location timeously. This could be done perhaps sometimes by making multiple copies where advantageous and at others by using up empty operation slots to move available data ahead of its required time.

On programmability, in the above streaming architectures more than others the trade-offs between scalability, general purpose performance, and programmability are most clear. The GPGPUs achieve significant scalability and performance for a narrow domain at the cost of re-writing code. Rigel and RAW offer a means to greater scalability with novel distributed cache hierarchies that are still capable of emulating coherence. Unfortunately both of these indirectly tax the programmer for optimal data distribution, cache assignment, and synchronisation managements, albeit in manners that share this load with the compiler.

2.2.4 Alternative Approaches

Outside of the above categories there are a spectrum of particularly more recent architectures that exhibit a range of characteristics straddling different categories. Such broad terms could include an unmanageable number of designs but remains within the scope described at the start of relevance to the hypothesis at hand, the following newer works appear relevant.

2.2.4.1 Reconfigurable Architectures

As the most common reconfigurable platform, despite recent advances FPGAs will always compare poorly to ASICs in energy and area efficiency simply due to the additional infrastructure cost of enabling reconfiguration. Coarse Grained Reconfigurable Arrays (CGRAs), however, attempt to reduce this gap by reducing the granularity of reconfigurability (from bit to word level). Being in effect MPPAs with simple ALUs for PEs rather than complete CPUs CGRAs are very much in line with our definition of many-cores. Traditionally, however, they have been used with known and deterministic codes most often for multimedia applications. A 2011 survey of CGRAs [Choi, 2011] unfortunately agrees that they remain highly appropriate for certain applications where the relevant flexibility for performance trade-off is apt, most often in ap-

plications with limited dynamic control requirements, but that they are less appropriate elsewhere.

[Taylor, 2012] in their analysis of how to exploit dark silicon points to CGRAs as an example of dim silicon use considering that computations are generally laid out in space resulting in a low duty cycle use of elements and reduced multiplexing of data paths. Again while this use will be most appropriate for some domains it does not suggest that CGRA should be applied to domains outside of their design space. For instance while [Shrivastava et al., 2011] examines and succeeds at enabling multithreading on a CGRA, it is in a manner contradicting the advantageous low duty cycle [Taylor, 2012] points out. Rather by assuming a streaming model of computation and restricting the compiler to allocating kernels to subsets of the array to allow for multiple threads to operate simultaneously, memory bottlenecks are alleviated and the resource usage duty cycle is increased. And while succeeding the approach improves performance in multimedia applications specifically.

A further instance of reconfiguration being used to tackle the problem this work's many-core hypothesis is aimed at is the work of [Venkatesh et al., 2010], the authors of the aforementioned utilisation wall. Their "conservation cores" take an approach tangential to the many-core hypothesis in which customised cores aimed purely at reducing energy consumption rather than acceleration are dynamically loaded into a configurable fabric on a per task basis (using a C-to-silicon infrastructure). The reasoning being that if energy consumption of a task can be reduced, there is more TDP available for other parallel task to be executed in additional cores running simultaneously.

It is therefore reasonable and apparent that in addressing the efficiency barrier confronting industry, reconfigurability might be best used when seeking efficiency and performance through dynamic implementations of customised application specific hardware. This is as opposed to using reconfigurability in an avenue that seeks to create a general purpose architecture (Convey [Convey, 2013] application personalities being another example from within HPC). While an equally appropriate avenue of research, this is an alternative approach to that under consideration in this work. As a result, while this work makes use of an FPGA and will investigate the use of such for a many-core, exploiting reconfigurability on a per application basis is beyond the scope of this work. This remains an option but one for which the software capacity has not been created. For these reasons the significant array of reconfigurable architectures have not been reviewed in depth here.

2.2.4.2 Embedded Systems

There are a range of architectures targeted at the embedded systems domain which are cited in comparison to many-core designs in the literature. These are inappropriate for the target domain and lack the memory and arithmetic support particularly a HPC system will need. However, given that they are commonly referenced two of such are briefly reviewed here.

FlexTiles [Brillu et al., 2014] is another very recent approach to many-core (published 2014). Targeted at embedded systems their reasoning is more relevant than the actual implementation. Having noted industry's reluctance to adopt many-core due to lack of legacy code portability, the apparent unsustainable nature of the solution, and the unprofitably low product volumes of custom heterogeneous many-core architectures, they propose FlexTiles as a conversion medium. Using the term many-core differently to this work, they are attempting to provide an array of heterogeneous application specific accelerators (in common with the reasoning of [Homayoun et al., 2012]) rather than a sea of ALUs. Represented as nodes, general purpose processors, DSPs, DDR memory, I/O, and a reconfigurable domain are all connected via a NOC. The accelerator nodes are classified as micro-programmed (domain orientated processor, custom domain orientated processor), and Data-flow (configurable data-flow, pure data-flow). As one of the first architectures to take advantage of 3D fabrication a reconfigurable layer is connected to a layer containing multiple DSP and GPP nodes. Using a main controlling program written in C threads are ranked in priority to allow dynamic resource allocation at run-time. The threads are in turn written in the target core appropriate language (C or VHDL). Later work will show whether this serves its goals or not.

P2012 [Benini et al., 2012] is often cited in many-core comparisons and is proposed as a platform for architectural development through, fine-grained power control, the use of the extensible STxP70 processor (a dual-issue 32bit RISC processor including a FPU from STMicroelectronics) for its basic PE, and infrastructural support for additional hardware acceleration modules. Targeted at embedded data parallel applications with the goal of filling the area and power efficiency gap between GPPs and hardware accelerators such as GPGPUs, the software stack supports OpenCL and OpenMP along with a propriety lower level language. Hosted by an ARM host processor, a control core supports four clusters, each of which supports up to 16 PEs,

a cluster controller, and any optional additional hardware accelerator modules. In a manner resemblant of FlexTiles and Rigel, the system uses a PGAS that avoids the costs of cache coherency. Each cluster shares a L1 data memory but any processor may directly access a remote L1 memory or, under the control of the cluster controller, obtain DMA access to main memory. Side-stepping the SIMD problems of GPGPUs, each PE has it's own L1 instruction cache, and is free to execute an independent thread making the system MPMD. To facilitate synchronisation, hardware acceleration exists for semaphores, barriers, and joins. Considering each cluster is operating on an independent clock, the interconnect is asynchronous. When emulated only 8 PEs filled a Virtex6 LX550T, running at 66MHz, a 28nm ASIC implementation was expected in 2012 but is not yet visible in the literature for further performance examination.

Epiphany (Adapteva) [Adapteva, 2012, 2013; Gwennap, 2011] is unsuitable for HPC but makes a number of design choices that are worth noting. As with the majority of array many-core processors, the design is power efficient, scalable, and most easily applicable to naturally parallel applications. Epiphany has been designed to maximise single precision FLOPS/Watt performance with a future mobile speech recognition and visual computing market in mind. The 64 core SMP processor achieves 100 GFLOPS on 2Watts at 800MHz in 28nm and is programmable in ANSI C/C++ and OpenCL. The largest contributions to its power efficiency is a minimised custom RISC CPU in the PEs and a low-overhead NOC.

As a further divergent approach to many-core Epiphany is able to limit its PEs to minimalistic CPUs that are appropriately simplified in direct correlation to the target application. The custom CPU pipeline supports dual issue to a FPU, and ALU or LOAD/STORE operation. To save power the pipeline is kept short (minimising the need for bypass logic and latches) and supports a minimal instruction set excluding all complex integer operations such as division and multiplication, as well as maintaining no FP divide, square root, or de-normals. Further there is no branch prediction, speculative instruction issue, or dynamic instruction re-ordering. Instead optimal instruction order is a function of the compiler. Unlike other fully compiler static systems though, the pipeline will detect and stall on RAW and WAW dependencies. A 64b wide memory bus and accompanying instruction is able to supply the pipeline with two operands for a LOAD.

The low-overhead NOC involves using a combination of a point-to-point design (providing scalability and high speed, 1GHz, operation) and the memory architec-

ture. In a approach that shares aspects with Rigel, there is no cache coherence hardware and a single address space is shared, producing power and network load savings. Going beyond Rigel, however, Epiphany also removes all memory controllers in the PEs and does not replace the coherence hardware with programmer control. Instead a global unprotected shared memory space is physically distributed across the array allotting a quantity to each PE, with all banks accessible by all cores. PE instructions and memory may be stored inter-weaved, although best performance is obviously achieved when appropriate locality is maintained and they are placed in separate banks. Local access is guaranteed in order but remote access involves a custom weakly ordered model requiring programmers consideration.

The address space supports up to 4GB but portions of this may be in off-chip memory banks or in up to 4095 cores. To support such off-chip peripherals, multi-chip scaling and to minimise network congestion the NOC carries 3-channels, on-chip write, off-chip-write, read request. Each core's router supports five directions of data travel simultaneously, north, south, east, west, and local. A point-to-point hop costs 1.5 cycles per routing hop assuming no congestion. Together this NOC and memory model reduce message overheads and power consumption at the cost of programmability. While the compiler will take locality into account the programmer is required to ensure there are no bank conflicts, and when accessing remote memory must take precautions to ensure correct memory access order. To mitigate the latency of point-to-point loading the array is divided into quadrants each of which are loaded from distinct interfaces simultaneously.

A final point of power saving includes the extensive use of clock gating and their approach to clock distribution. While the former is not novel this is important in a many-core design where there will be significant proportions of idle core clock cycles. In the latter, as there are no other global signals, the design ignores clock skew saving on compensating hardware and simplifying routing.

To enable programmer control an additional library is supplied along with appropriate multi-core and parallel debugging tools. Modified versions of the GCC and GDB tools are also capable of making use of standard optimisations such as loop unrolling, although for now hand optimisation is also required. The burden of manual memory management, however, while removing the need for fully static compiler scheduled instructions, significantly raises the cost of both porting existing codes and programming new applications.

By so reducing the inter-task communication bottleneck the design is able to support parallel programming at both a large kernel level but also at a very fine grained level (although this requires programmer work). While clearly unsuitable for HPC adopting the simplicity of a shared but unprotected memory space and taking advantage of the reduced message overhead this brings are two design features worth exploring in a HPC many-core design. In conjunction with static scheduling to circumvent the programmability and porting problems, full double precision support, and a means of mitigating the lack of dynamic instruction issue such as TRIPS uses for instance, might result in a design capable of using a message passing NOC for thousands of cores. This without having to resort to other designs' retention of cache hierarchies or remaining global signals. This last point is important as it enables multi-chip designs with no glue logic required.

2.2.4.3 Fabrication Advances and Consequences

Regarding fabrication techniques, [Hanson et al., 2006][Jain et al., 2012; Seok et al., 2008] have shown that Near-Threshold Voltage (NTV) operation, sometimes termed Near-Threshold Computing (NTC), can improve transistor efficiency by orders of magnitude. However, the technology is still immature and problems remain in need of solving before it might be widely adopted [Chang and Haensch, 2012; Kaul et al., 2012]. A dominant issue is the matter of process variations and timing uncertainty which occur more frequently as the transistors are scaled down. Static faults will be located during testing and can, given the abundance of PEs, simply be flagged for the compiler as unusable. Dynamic errors occurring during operation, however, require the design to have a coping mechanism that will be model dependant. [Krimer et al., 2010] for instance (addressing such in a SIMD architecture) proposes adding lane-local decoupling FIFO queues to buffer decoded instructions. Provided failures are equally distributed through out the lanes (a statistical probability) this will mean the whole unit need not wait for a slow lane. These queues are accompanied by synchronisation logic to be invoked on scatter, gather, and shuffle operations, or should a buffer reach its maximum. To handle the same issues in other architectural models will clearly require equally customised circumventing adaptations.

3D stacking with through silicon vias (TSVs) offers great advantages in power and performance due to shortened signal lines and increased physical accessibility. Further the use of layers makes it possible to integrate different process fabrication technologies into the same chip, and to increase yield by through partial verification by

verifying functionality of layers prior to final merging. Being able to select the optimal substrate for each layer means substrates can be matched exactly to the requirements of that layer's functionality, saving power and manufacture costs where some layers can operate at lower performance levels relative to others. The technology brings with it, however, new levels of power density and the accompanying thermal dissipation requirements. Handling of this issue and the practicalities of stacking and TSVs are still active areas of research, as is the development of new software to handle all of these new chip design issues, thermal implications, application mapping, and hardware design exploration[Fabre et al., 2012]. The use of NTV operation is also an option for handling the thermal density issue although this does then bring the problems already discussed as associated with such.

Demonstrating some of the potential options TSVs with NTV can make available, researchers at Michigan University have prototyped 2 layers and 64 cores of a proposed 7 layer 128 core chip Centip3De[Dreslinski et al., 2013]. The 64 cores (ARM Cortex-M3) are grouped familiarly in 16 clusters of four, each of which share a data and instruction L1 cache. Due to the lower activity rates of the caches, however, they can be run at 4x the core clock speed. By operating each core on a clock 90 degrees out of phase with the others and using a 4-stage cache pipeline each core has apparent single cycle cache access. Fabricated in 130nm and using NTV operation the cores run at 10MHz and the caches therefore at 40MHz. As a further measure of optimisation the clusters can be run in a boost mode where three out of four cores are disabled temporarily and the core frequency raised to meet that of the caches improving single thread performance. While this work is not pursuing the same form of many-core (the ARM PEs being much more sophisticated), it does serve to highlight the potential opportunities available with TSVs.

3D stacking technology has also revived an old idea that approaches the performance issue from a different angle,, that of processor-in-memory (PIM) or intelligent RAM (IRAM). Originally seriously investigated in the early 90's the idea saw many small processing cells integrated within a DRAM so as to perform simple tasks on data values in place in memory and potentially to perform prefetches into the processors cache. At the time despite a number of academic investigations the principle was not taken up by industry, primarily due to its then unsettled state and the drive to create cheaper DRAM. More recently, however, [Kang et al., 2012] has argued that there is now more pressing motivation for PIMs to be taken up by industry. Motivation for this lies in, the memory bottleneck only becoming more prevalent, the need

for scalability in the face of increasing latencies as processing logic grows and lands further from memory, and the fact that the integration of different fabrication technologies is now established practice. Along with a concise summary of prior work on PIMs, they point to industry's very positive response to the "Hybrid memory cube" by Micron [Micron Technology, 2011], a 3-D stacked DRAM and logic control circuit connected with TSVs, effectively a form of a PIM, as evidence for its imminent arrival. The same authors have seen the republishing of their 1999 design for such a PIM, FlexRAM [Kang et al., 1999], at ICCD 2012[Kang et al., 2012]. While the approach has a limited scope for HPC applications (the need to keep the PEs so simple means they lack FP support), it does promise performance gains in other domains and demonstrates an alternative manner of using many PEs placed close to memory. As with all the other massively parallel architectures proposed, however, it does fare best when operating on a inherently data parallel application.

Intel's latest 22nm 3D Tri-gate or FinFET transistors are another more tangible innovation, already in use in their Ivy-Bridge processors, showing 10 fold reduction in leakage current or a 50% reduction in active power. On-chip power management has also seen new approaches, with Power-management Control Units (Intel's Ivy Bridge), and fine-grained frequency control that dynamically alters the clock speed of subunits of logic according to the current work load [Vangal et al., 2008]. This leads also to the need for more research into software management of hardware power control and for improvement in the actual hardware of voltage regulators doing the controlling [Perlmutter, 2012].

2.2.5 Summary

This review is incomplete, spanning the whole history of computing are many more processor designs containing aspects of relevance to an attempt at many-core parallelism. DSPs for instance address an entire application domain that also makes use of many PEs in a single coherent unit. They have been disregarded here for brevity considering the application domain but may well have techniques worth reviewing in light of the new challenges being addressed here. The exclusion criteria most generally applied involved the designs being, significantly older, not prominent in the literature, too far removed from the goal of general purpose HPC many-core computing (meaning most often a lack of floating-point support, too little memory or bandwidth), poor programmability or requiring a custom language, or a fundamental lack of scalability.

While the majority of processors reviewed attempted to serve as general purpose processors, it is only inevitable that some application domains will perform better and worse on a given design. Control model, memory infrastructure, data communication mechanism, PE arrangement, and programming model all influence each other along with the stated goals of the hypothesised many-core model. The challenge in all of these has been to balance the trade-offs involved which will again be the case in this work's architecture other than that some guidelines can first be drawn out of the above.

Summarising the above therefore, it is apparent that the memory infrastructure of a system has the greatest influence on scalability and the inter-PE communication design. A distributed memory model clearly enables scalability but costs either additional management hardware or programmer effort to ensure correct order of access. As seen this does not necessarily mean a full coherence protocol is required, some of the approaches reviewed included, matching hierarchies of PEs to a hierarchy of caches (with some levels implicitly coherent and others adopting formal coherence strategies), access operations ordered by ID tags, programmable memory accessing interconnects, exploitation of application specific bulk memory access (streaming), and manual or compiler memory management structures. Static scheduling particularly enables the last of these in many designs but it in turn comes at the cost of poor dynamic code handling requiring further mitigating measures. In these architectures therefore speculative execution (with prefetching), predication, branch prediction and hyperblocks, are amongst the most common means of handling dynamic code with which they achieve mixed results. Where branch prediction is used one notable advantage in these architectures regards their running at a lower clock speed, the lower speed means a shorter pipeline is possible and therefore miss-predictions are less expensive, or alternatively the abundance of cores can be used to execute both paths.

Where architectures were designed to operate in close conjunction with and in parallel with a host CPU (so as to optimally apportion code to the best processor) shared access to a common memory was needed to accommodate dynamic function calls and task distribution. In the dataflow architectures that accommodated this the shared memory in particular was found to degrade performance and limit scalability. Alternatively co-processors onto which tasks and memory are off loaded in isolation suffer from the latency cost of such data transfers and cannot use the host CPU as

optimally in covering fundamentally serial code sections. Moves toward greater integration between the co-processor and host CPU to reduce memory transfer latency are already visible in GPGPU domain, these will likely continue to use atomic portions of memory.

Given a memory model the control model adopted further influences scalability, efficiency, and the interconnect. Dataflow architectures hand off control predominantly to static scheduling gaining efficiency. Of the MPPA reviewed the more complex and hardware expensive PEs are able to handle much more control at run-time which in combination with the PE arrangement and interconnects split control between data-flow like operation on non-dynamic segments of code and programmer effort. The streaming model's exploitation of the application domain allows for high efficiency and scalability through simple control infrastructure and PE hierarchy.

The inter PE communication mechanism is affected by both the memory and control model adopted. In different designs reviewed the infrastructure selected was variously able to, access PE local caches, snoop PE data paths, and communicate seamlessly off-chip with peripherals and further processors. Equally the degree of interconnectivity ranged from being all-to-all chip-wide through to nearest-neighbour point-to-point only connections. In some instances it was flat but multi-layered and in others hierarchical. Finally functionally in some cases it served purely as a means of data movement and in others acted as a mechanism for synchronisation and event control. Each of these characteristics is a consequence of the higher level design within which it exists and therefore unique to the architecture. The only common point free to be drawn out involves the interconnect's effect on architecture scalability and application domain range.

As already noted, due to their target applications the streaming architectures are able to operate with minimal inter-PE communication requirements leaving memory scaling as their scalability limitation. Alternatively the dataflow architecture includes a degree of reconfigurability (hardware or compiler programmability) in their interconnects, and use a range of hierarchical combinations. None are infinitely scalable but due to the asynchronous operating model complexity around ordering data access appears to be the scaling limitation. With more sophisticated PEs the MPPA interconnects are closer to conventional designs (even offering message passing in some cases), but manage to avoid some of the conventional interconnect restrictions through their globally asynchronous operating model and majority use of PGAS memory model (unfortunately the complex PEs are more costly in hardware).

Aside from scalability and efficiency these interconnects all affect application suitability and programmability. Those that are the most configurable and which have the greatest capacity to share data across the entire processor are clearly the least limiting. However, given the other architectural factors that are linked to the interconnect this review has not revealed a means of determining which best meets these ideals in combination with the other many-core ideals.

While the majority of architectures reviewed made use of a bulk quantity of identical PEs (and one or two specialised units generally concerned with control and memory management) within PEs different operators were catered for to greater and lesser degrees (multiple integer pipelines alongside one FPU, or one special function unit alongside multiple identical general purpose ALUS).

The possible use of NTV operation for a many-core means mechanisms for faulty transistor operation detection and correction need to be designed for. Circumventing dynamic faults in particular is a point for consideration in design.

A table form summary comparison of many of the above hardware and compilation characteristics is given in Appendix A Table A.2. This should only be viewed, however, as a concise reference rather than a means of comparison. The fundamental design approach differences mean direct numerical comparisons of various hardware characteristics is largely misleading.

Regardless of the interconnect and memory infrastructure used all the architectures perform better if data and instructions are optimally allocated in space and time. Programmability is therefore of equal concern in analysing these approaches as the following section will detail more fully. From the above it can be noted that those architectures that offered conventional languages and parallelising tools met with more success or longevity. Traditional parallel programming approaches were only found in the MPPAs offering more complete CPU cores rather than ALUs as PEs. Instead those architectures that used the finest granularity of PEs adopted the highest degree of automated hardware control and static scheduling.

2.3 Concurrent Software Stacks and Languages

In addition to the hardware challenges of many-core, programmability is of equal importance and has its own set of challenges. As alluded to in Chapter 1, despite the difficulties involved the significant appeal of autoparallelisation lies in how much more challenging the alternative is. If a substantially new architecture is to be adopted the

enormous quantity of sequential legacy code still in use will require porting. The lack of expertise in parallel programming will exacerbate the already inherent time consuming difficulties of such [Hochstein et al., 2005], and as new applications will be developed or old algorithms replaced with new a usable parallel programming model for such is also needed. In the face of ubiquitous multi-cores a wide range of tools [1] and language extensions have appeared. But these and other traditional programming models (along with their associated languages and tool sets), simply do not have the required components to allow a programmer to make use of a many-core level of parallel hardware. To demonstrate this the following identifies the challenges of many-core programming and then reviews the currently available parallel programming models.

The following points were identified in Chapter 1 as in need of address by any new many-core targeting parallel programming paradigm. They are elaborated on in more detail as follows. Some are common for any programming model with subtle adjustments, others are unique to many-core. For each aspect where exactly responsibilities lie in the trade off between hardware and software control will be architecture dependant but those tasks handed to software need to be covered by an overarching model that makes programming such a system manageable.

- *Memory:* Memory management stands out as the largest hurdle in hardware and software. Compared to current memories many-core memory structures need to supply data at a much higher bandwidth in order to feed thousands of cores while still maintaining a form of coherency and correct ordering in some fashion, and all without placing too heavy a burden on the programmer or the control hardware to correctly manage such. Uniquely, however, a many-core memory system has in its favour a clock speed advantage where off-chip memory may be clocked multiple times faster than the processor cores. Ultimately each hardware model may require its own low-level memory software model with a uniform higher level abstract model allowing performance portability, but this is less than desirable considering the additional layer it will require. As much as any autoparallelising compiler might offers to relieve the programmer and hardware of the control burden, it can only do so given a mechanism for such. Whether that mechanism includes full latency foreknowledge

[1] Intel Parallel Studio XE suite [Intel, 2013e], Vector Fabrics [Vector Fabrics, 2013], or for embedded software CriticalBlue [CriticalBlue, n.d], are just a few

and static scheduling, or a explicit synchronisation and ordering tactic, or some other means the model must supply this, a compiler can only implement it.

- *Heterogeneity:* While specialised on-chip modules accelerating a CPU have been used successfully for a while now via compiler automation and the addition of new hardware specific instructions to languages, the higher level problem of different architectures within a compute node remains only partly solved. The GPGPU programming models in current use have taken steps towards easily integrated CPU+GPGPU programming by using standard languages as a base. But GPGPUs are only one alternative architecture. The push for computational efficiency suggests many more application specific hardware modules are likely to be added with the further possibility that some will also be reconfigurable. It is unrealistic to expect programmers to deal with the complexities of each different hardware architecture and their associated drivers and so the need for functional and performance portability is a significant challenge. One approach is to develop managing software that, given an environment, automatically selects and tunes for the most appropriate target, compiler, and routine [Benker et al., 2011; Chafi et al., 2011]. These approaches require greater initial effort but lead to greater performance portability (ability to achieve the same performance standards on multiple architecture platforms) than the hardware brand specific GPGPU models appearing. Using autoparallelisation to address this aspect of a model in its entirety borders on looking for a magic fix-all. Given the complexity of autoparallelisation addressing even one many-core model is a significant challenge, and one that requires an architecture specific approach without attempting to re-target the same scheduler to multiple architectures. But the above levels of automation in selecting compilers and architectures according to high-level diagnostics looks promising.

- *Multi-level concurrency:* Multiple levels and forms of hardware concurrency have to be exposed to either the programmer or the software stack, from vector units, through multi-threading, multiple cores and multiple chips, to multiple nodes and racks. Traditional HPC programming models have established predominantly manual ways of the dealing with the higher-levels of this stack, whether or not these can or should be integrated with a model that also handles the lower levels remains undetermined.

- *Programmability:* Part of accomplishing this lies in how appropriate the provided constructs are to the applications and hardware under consideration. Another part lies in what industry is already familiar with and skilled at. A new model must provide an easy manner of expressing and using parallelism in applications and hardware, but also do so in a manner that costs a realistic degree of effort. In HPC, C/C++ and Fortran particularly dominate, hence the voluminous number of different parallel extensions these languages have seen [Aldinucci, Marco Torquati; Intel, 2013d,f, 2014; Kennedy et al., 2011; Loveman, 1993; Schimbac, 2013; UPC, 2005; Wallcraft, 2005], while the more generally popular languages such as Python, Java, and Matlab, have also seen extensions [IBM, 2013; Miyoshi et al., 1997; Sharma and Martin, 2009; Vanovschi, 2014; Yelick et al., 1998] attempting to make the switch to parallel easy. Where the compiler cannot do the work the new OpenCL [KHRONOS Group, 2013]/ CUDA [NVIDIA, 2013a] models have exposed the many levels of hardware concurrency to the programmer using familiar syntax, in what appears to be a good balance between compiler and programmer effort, but this is also the reason the models are so hardware brand specific. A final domain of FPGAs in HPC has seen far more limited success with the likes of DIMEC, MitC, ImpulseC, and CUDA on FPGAs, having mostly failed to find a place due to how different, constricted, and far removed the new languages really are from the original. Autoparallelisation of sequential code would clearly be greatly advantageous as the process of developing sequential code is already very familiar and well supported.

- *Support of heritage codes:* The above point leads in to the need for a way in which legacy codes can be ported to any new model with as little effort as possible. With out a means of portin systems will be held back for years unable to provide upgraded performance and supporting two different domains of old and new code. If an industry standard language is used it will certainly ease the porting effort simply due to the prevalence of programmer skill, even where extensive re-writing is required to adapt applications to new hardware features or language constructs. All of the language and library extensions listed above help with this but not as much as autoparallelisation could. OpenCL and CUDA for instance, while C based still require significant code rewriting, compiler directives such as OpenACC offer instead an even easier conversion. Ultimately either autoparallelisation will solve this problem or industry will sit with many

codes at best operating in a handicapped manner until someone rewrites them. Meanwhile other codes will be rewritten sooner as a completely new algorithm justifies the effort.

- *Portability:* Already mentioned as part of the difficulties of heterogeneity, the ability to run the same code on multiple hardware architectures takes three forms, language, functional, and performance portability. Targeting language portability (ability to compile the same language on multiple architectures) improves application programmability simply by having greater numbers of programmers skilled in a single widespread language. Conversely a language independent model allows for flexibility in implementation leaving room for application or hardware specific optimisations that the model may not directly address. This potentially also opens the door to reuse old codes and languages getting around the issue of legacy code. A model that targets functional portability (ability to achieve the same net functionality on multiple architectures) facilitates incremental porting where old codes function on new architectures needing only tuning for performance rather than a complete re-write. This makes porting significantly easier and is already a familiar reality in HPC. Currently DSLs satisfy the needs of and for language independence and language portability. OpenACC in the future and OpenCL now are the closest to satisfying functional portability. Performance portability, however, is so far an unsolved problem for which auto parallelisation again perhaps has the best chance of providing a solution.

- *Effective verification, debugging, and algorithm analysis:* The model should implicitly foster an environment that, firstly minimises the probability of programmers inadvertently introducing the bugs most commonly associated with parallel programming, and secondly enables a means of easily debugging and verifying application functionality. Included in such a task is the further need for profiling to determine inappropriate algorithms or code portions in need of further tuning. Message passing auto synchronises threads, OpenMP requires manual synchronisation, and OpenCL/CUDA operates on the assumption of, and exploits the fact that, warps/wavefronts are independent but the programming model means the compiler will see if there is a conflict in say data access and will not allow the programmer make such a mistake. Autoparallelisation alternatively is based on the assumption that once a sequential code functions cor-

rectly the transformed code will perform in the same way and therefore does not need a specialised debugging tool chain.

- *Control granularity:* This is again very closely tied to the division of labour between software and hardware control and so will vary in implementation up and down the concurrency stack. Depending on the hardware being addressed access to fine-grained power and interconnect control features and reconfigurable components may be required. How memory coherency and access ordering, as well as PE synchronisation are handled, may also be controlled in hardware as already discussed. Or a programming model will need to cover such tasks possibly requiring a low-level and very fine-grained degree of control. This in turn may be automated or abstracted away by high-level constructs so as to free up the programmer as much as possible to work under the assurance that such is provided.

This section covers a very complex and large subject, which while equally important is subsidiary to the primary focus of this thesis. For both of these reasons the following scope limitations are imposed. Firstly a focus is kept on what is currently in mainstream HPC use or has emerged in the last few years since Berkeley's pivotal 2006 paper. Secondly, an overview of the subject is given only to give context and this is therefore not an exhaustive literature review. Showing APPRASE's novelty is not considered a requirement for this work as the focus is on the hardware and APPRASE is only an enabler, albeit an inseparable one. Consequently attention is not given to the many optimisation methodologies being proposed which improve one particular aspect of software parallelisation such as shrinking the search space in autoparallelisation for faster turn around times [Mustafa et al., 2011], or improving caching methodologies [Muddukrishna et al., 2013], or better networking protocols [Diaz et al., 2012], or quicker dependence graph generation [Kim et al., 2010], or adding commutative constructs to reduce implied serialisation [Prabhu et al., 2011], or leveraging commutativity of some functions to allow for reordering and a gain in performance [Aleen and Clark, 2009], or many others.

When addressing the issues of many-core programmability the adopted programming model, rather than parallel language, defines everything around how a programmer will interact with the parallelism in the hardware and match it to the same in the applications. A programming model will on some gradient hide and expose various hardware features trading programmer effort, or in the case of autoparallelisation

compiler effort, for optimisation opportunities and vice versa. For these reasons this section focuses on programming models rather than the details of their implementation so as to compare their suitability for many-cores. The general practical implementations of programming models are instead briefly summarized as follows.

- The use of libraries to extend a language and implement a model makes the model as functionally portable as the implementing language, and possibly offers low-level optimisation at the cost of only the library author's effort rather than through exposure of the hardware to the application programmer. However, libraries are limiting when the constructs they provide either only make use of hardware features present on some architectures, or only suit the nature of a limited number of applications.

- Alternatively languages may be extended to implement a model using compiler directives. By exposing a high level of hardware to the programmer the challenges of parallelism are divided between the programmer and compiler. Unlike libraries the model will only be as functionally portable as supporting compilers for different target architectures that have been developed. Once developed further architecture specific performance tuning is an optional step but possible, unlike with libraries and is a minor effort relative to code rewriting.

- Finally, as already alluded to in the hardware discussions, the option exists to develop new low-level and dedicated languages containing inherently parallel constructs (see the survey paper [Diaz et al., 2012] for many examples). For targeted applications a dedicated language should result in the highest performance gains possible. As such these languages are generally developed for application customised architectures. Re-writing a code in a new language obviously comes at the highest programmer effort cost, however, and is generally only used in special cases. If its advantages could be defended the current upset in the industry could arguably justify the effort but this has not yet been shown.

As proposed in Chapter 1 and briefly argued above, autoparallelisation offers to circumvent the problems of parallel programming, allowing a traditional sequential model to be used by the programmer and leaving the parallelising work to the compiler. So as to argue for this approach in the context of many-core cluster nodes, the remainder of this section examines existing and experimental parallel programming models in Subsection 2.3.1, and then looks at the state of autoparallelisation in Subsection 2.3.2, Subsection 2.3.3 summarises and concludes.

2.3.1 Current HPC Programming Models

Parallel computing is not a new field to HPC with a range of established parallel programming models in use as industry standard. While many-core looks to exploit very fine-grained parallelism, the coarse-grained task and data parallelism in cluster size codes remains more than relevant and appropriate for exploitation. Therefore while too coarse-grained for direct application to programming a many-core processor these models will continue to serve. Consequently they present a possible avenue to industry acceptance of new hardware if many-core capabilities can be added to them.

2.3.1.1 Distributed Memory Based Message Passing Model

The long time de facto standard for programming distributed memory systems is MPI (a library). Here there is no global address space and explicit messages are required to access remote data. The model executes codes predominantly using a single program multiple data (SPMD) form, for which the programmer is responsible for distributing the tasks and data. While care is needed to avoid deadlocks of nodes waiting on send or receive messages, the action of communicating is two-sided having an advantageous and implicit synchronising effect between processes. Remote access, however, obviously also has a higher latency than local and this eventually limits the scalability of an application. Most implementations provide libraries for C/C++ and Fortran, although there exist others also for Python, Ruby, Perl, R, Java and other languages.

2.3.1.2 Shared Memory Based Multi-threading Models

At an application level a shared memory system assumes all memory locations are equally accessible and in a single address space. With no need for explicit message passing application development is made much easier. With little visibility of what is remote or local, however, the model makes it difficult to take advantage of locality and therefore does not scale well across a cluster.

In software two flavours of implementation exist. In library form, Pthreads (C), Intel's Thread Building Blocks(C++), or Java Threads, offer a low-level fork-join appropriate threading option, however, the programmer is responsible for ensuring race conditions and out of order accesses do not occur. This makes it problematic for HPC where the number of PEs (not necessarily the same as the number of threads) scales

to very high numbers. Alternatively, the OpenMP specification, implemented as compiler directives and a managing runtime environment, offers a higher level of abstraction where synchronisation and memory access are managed. This makes OpenMP more appropriate for use in HPC and it is widely supported in a range of C/C++ and Fortran compilers. Attempts have also been made to add such support to Java but with less success.

2.3.1.3 Global Address Space Model and Variations Thereof

Subsequent to the above the partitioned global address space (PGAS) has emerged offering a compromise in which the whole address space remains unified and directly accessible as in a shared global address space. The affinity of a location to a process or thread as remote or local is made explicit through the use of global or local data structures. As with OpenMP, the programming burden of generating messages is handed off to a runtime manager leaving the programmer to simply access a global data structure. While communication can therefore occur solely through the data structures most implementations also offer facilities for bulk communication and synchronisation actions.

In comparison to MPI such remote accesses, once turned into messages, are also only one-sided meaning the implied synchronisation between processes needs to be replaced. This one sided communication protocol and greater control over data location, however, leads to a lower communication burden and therefore to improved application scalability. The SPMD model for a cluster is therefore better supported than in the pure shared memory models, while retaining the advantages available to shared memory systems.

Apart from the runtime libraries many of which are detailed in [Diaz et al., 2012], implementations of PGAS include language extensions such as, UPC (UnifiedParallel C)[UPC, 2005], Co-Array Fortran[Wallcraft, 2005], and Java based Titanium[Yelick et al., 1998]. New languages designed specifically as PGAS languages also exist such as Chapel[Chamberlain et al., 2007] and X10[IBM, 2013]. Lastly GASPI[GASPI, 2014] is an PGASAPI that is interoperable with MPI.

PGAS, however, also assumes all processes will run on similar hardware, and does not support the dynamic spawning of new processes. The former makes the model inappropriate for the more recent heterogeneous HPC environments, and the latter leaves no way of handling load imbalances. While not originally targeted at HPC a proposed Asynchronous Partitioned Global Address Space (APGAS) model[Saraswat

et al., 2010] has nodes execute multiple tasks from a task pool and capable of invoking work on other nodes. Operation on the various nodes is therefore asynchronous requiring additional constructs and management of data. APGAS has so far been implemented in X10.

2.3.1.4 Hybrid and Heterogeneous Models

Increasingly HPC systems are being created with hierarchies of concurrency created by clusters of multi-cores, even more recently these hierarchies are being further extended with additional co-processors. To handle such system architectures programmers are turning to mixed models using MPI and OpenMP, and if needed a further co-processor specific model. As alluded to earlier newer low-level but multi-architecture coding environments such as OpenCL [Kicherer et al., 2012] and PEPPHER [Benker et al., 2011] are also investigating means of adding architecture agnosticism in a heterogeneous environment. Their approaches include sophisticated compilers, runtime resource aware management systems, and architecture specific fine tuning in the form of pragmas and multiple variants of system critical code. Depending on the implementation of such a potential down side to this approach is the resulting enlarged binaries and their generation cost when containing the same function written multiple times in, for example, CUDA, OpenCL, and Intel Compiler directives.

All of these approaches continue to evolve as the hardware keeps changing. Most recently co-processors are being more closely integrated with their host CPU (for example Intel's stand alone Xeon-Phi chip Knight's Landing and AMD's series of single chip integrated GPU-CPU APUs). With this further hardware integration of such architectures into clusters, their programming models are also being extended to facilitate greater software integration. While the above described the traditional parallel programming models, GPGPUs and the Xeon Phi are the architecture in current use closest to the many-core hypothesis envisioned. As a result they make use of a programming model that is both far removed from those already reviewed, and particularly noteworthy in this work. The programming models for these types of co-processors provide very similar hardware abstractions and assume a common methodology, and so they are discussed as the same model here, but it should be noted that their implementations do differ and in some ways differ significantly.

Due to its novelty a range of new terminology and definitions are required, it is assumed the terminology is familiar but if not Table A.1 in appendix Subsection A.1.1 attempts to explain these terms by comparing the physical and abstract hardware

models with the programming models used by AMD and NVIDIA for their GPGPUs (terminology for both OpenCL and CUDA is given)[1]. Only the term *kernel*, taken from the classic streaming model, is familiar with its same meaning of a set of instructions that sequentially operate concurrently on a stream's worth of data. On GPGPUs this generally takes the form of mapping a sequential loop into a vectorised form so as to efficiently use the VPU arrangement of GPU cores.

For brevity NVIDIA GPGPUS and the CUDA programming model implementation particularly are used here illustratively, but the domain contains a range of other implementations. CUDA and OpenCL both extend C/C++ at a minimum through libraries, compiler directives, and extensions to the languages. These are accompanied by customised compilers predominantly based on Clang/LLVM[Lattner, 2013]. Most recently adding to these is OpenACC[OpenACC, 2013], an API of compiler directives extending C/C++ and Fortran in a manner that extends the OpenMP API. While CUDA is limited to the domain of NVIDIA GPUs, OpenCL and OpenACC are also used on the Xeon Phi. All of these aim to provide as seamless as possible support for the writing of codes that simultaneously make use of both CPUs and GPUs, and to enable compilation of such into a single binary.

The Portland Group (PGI) has extended compiler support via their FCUDA compiler[The Portland Group, 2013] which extends Fortran to support CUDA constructs, and in adding support for OpenACC in their PGI Accelerator [Wolfe, 2012]. Other APIs also available include C++AMP [Microsoft, 2013] (a DirectX API that compiles to OpenCL) and Intel offload compiler directives [Intel, 2013a] which add Fortran support for the Xeon Phi. This range of approaches is indicative of the challenges involved, the following examines the key approaches taken towards such.

Main memory access on these co-processor devices requires data and instructions to be transferred from system memory across an interface to the co-processor. Due to latency costs of all such transfers this step is critical to performance as an incorrect decision is costly and therefore for the most part the programmer is required to manage it. CUDA requires the programmer to explicitly allocate memory on the device and make the transfer. Abstracting a level of detail OpenCL allows data to rather be moved in and out of buffers, instead of known memory locations. Doing so improves

[1]The implementation for the Xeon Phi is sufficiently different that for clarity's sake it is not included in the table but the principles discussed for the most part are also true for it.

code portability but may result in unanticipated transfers as the instantaneous location of buffers is unknown to the programmer. Compared to these two extremes, C++AMP, Intel offload directives, and OpenACC, each offer spectrums of control from CUDA like precise manual control, through OpenCL buffer approaches, to completely automated compiler controlled memory management.

Multiple levels of concurrency must be exposed by a GPGPU programming model, from multiple chips, through multiple cores, to multi-threading, and vector processors hardware. At the highest level program functionality is expressed in kernels made up of *CUDAthreads*. A kernel describes program functionality as a set of statements in which the operands are vectors. Programmers are able to program on a per vector index basis, one index value in the vector operands making up a *CUDA thread* or a single lane instantiation of the kernel. Programming on such a basis, however, will more than likely cost efficiency and performance. Instead kernels are described and the number of threads in a kernel is dependant on the length of the vectors concerned. In hardware each lane of a VPU executes a thread, CUDA therefore manges this hierarchy of concurrency with a hierarchy of programming constructs. At the lowest level vector operations are effectively implicit with a sequence of such making up a kernel. At the highest level, where task level concurrency exist in the algorithm, multiple concurrent kernels may be described with independent numbers of threads.

Only once the functionality is thus described, as vector operations in multiple parallel and sequential kernels, is the programmer then required to consider the capacity of VPUs, SMX's and different memory levels as it pertains to the number of threads needed for each kernel. How each of the hardware components described earlier is optimally accessed is a complex discussion. Given that this work's approach to the same problem is entirely different the details of such may be found in Appendix A.1.1 if required.

OpenCL on the AMD hardware adopts the same approach with different terminology and size constraints dictated by the hardware. For instance wavefronts, rather than warps, are 64 rather than 32 threads wide. In both cases the possibilities for multi-threading and multi-core are exposed, while the VPU operation is mostly hidden apart from guidelines around performance cliffs in thread block and grid sizes. OpenACC offers to expose the same levels of parallelism to the programmer, but will also automate much of the allocation if left to it. C++AMP alternatively hides these

multiple levels of parallelism, leaving the programmer to trust the compiler. Intel's offload directives are using the OpenMP multi-core parallelism, the programmer must create enough threads to make use of the multiple levels of parallelism offered by the hardware, and Intel Phi vector instructions when using the VPUs.

Considering portability, each of the above frameworks fair as expected in function and performance portability based on their approaches. CUDA codes perform well on all NVIDIA GPGPUs but are limited to such, and older codes will require re-tuning to take advantage of newer hardware features. OpenCL is designed to provide functional portability, offering correctly functioning code which may be tuned per-target to achieve performance. As C++AMP hides most of the hardware it also succeeds at functional portability, but will rely on the target architecture's compiler to achieve performance. OpenACC also provides functional portability and PGI suggests it may provide greater performance portability in time, but this remains to be seen [,Brent Leback, Douglas Miles, 2013]. Intel's offload directives are solely for use on the Xeon Phi and are ignored in the face of other targets.

2.3.1.5 Implicit Programming Models

The phrase *implicitly parallel programming model*, has been applied to programming models that lie between the above explicitly parallel models, and full automatic parallelisation of entirely sequential codes. Where the above models require considerable effort to explicitly order and parallelise tasks, taking into account low-level concurrency and control concerns implicit models are attempting to provide an easier means of conveying valuable human insights to a compiler capable of carrying out the final complex and bug prone stages of parallelising. It is argued that this is more likely to achieve performance than a fully automated parallelisation of sequential code, which may only achieve moderate performance gains due to the sequential expression of an application introducing apparent but false constraints.

The primary means of implying parallelism is the use of additional custom language semantics. These enable the compiler to make useful inferences as to how an application might execute in parallel while retaining the easy readability and development process that a sequential language and code ordering brings. With the code description not solely responsible for exposing all hardware parallel features the model offers better performance portability and greater scalability. Due to the easier nature

of the parallelisation style it also offers easier maintainability. Whether such a system can achieve equal or better net performance results, however, is less clear.

The following three examples demonstrate three different approaches to implementing such a model.

1. [Harris and Singh, 2007] and [Ioannou and Cintra, 2011], amongst others, achieved some success in seeking to complement traditional coarse-grained parallelism by overlaying it with a finer granularity of speculative execution. The basis for both was to use existing resources that were going unused in SMP environments. Both achieve performance gains of between 10% and 80% on a range of applications. The techniques are exploiting implicit parallelism as the work done is speculative and therefore not available for explicit exploitation by a programmer but implicit to the algorithms. Where [Harris and Singh, 2007] required more manual intervention based on analysis tool outputs, [Ioannou and Cintra, 2011] expands on thread-level speculations (TLS) to offer automated TLS integrated alongside explicit threading.

2. [Hwu et al., 2007] suggests an implicit model where programmers express code in a canonical form that exhibits a sequential ordering of sections marked for parallel execution. These marked sections are in turn passed through an automated concurrency analysis tool, guided by the programmer's assertions and with provision for programmer's feedback, before final stages of a tool chain synthesise a correct parallel program. A traditional sequential language may therefore be used in describing functionality and sequential ordering, with the parallelism implied by programmer's assertions rather than low-level parallel constructs.

   The authors distinguish their proposal from current parallelising compiler pragmas as being compiler hints rather than directives, leaving the compiler to do more detailed analysis where it has been suggested. This analysis pertains to finding both parallelism in the application and considering the best match between this and the available parallel opportunities in different hardware substrates. They also distinguish the work from completely automated parallelisation as the programmer evaluates the trade-offs involved in selecting a more parallel but not necessarily equivalent algorithm and is hardware aware. The approach is therefore not intended as a fix-all where legacy codes might be automatically ported, as only manual algorithm evaluation and code re-working

will ensure results. Further they note that as with assembly code now, there will remain cases where either, manual low-level and explicit parallelisation, or domain specific languages, are necessary.

3. Stanford University's Pervasive Parallelism Laboratory (PPL) [Stanford University, 2013] is actively pursuing an implicitly parallel model based on domain specific languages (DSLs). They are doing so in a manner, however, that means that much of the effort involved in developing such is amortised across multiple DSLs sharing a common base of language constructs and compiler stack components. [Chafi et al., 2011] introduces their approach which will ultimately consist of many DSLs created in the same Scala programming language [de Lausanne, 2014]. Using the same base language improves interoperability between different DSLs within a single application, and the effort put into the compiler and testing and debugging infrastructure is shared.

   The nature of a DSL is to raise the level of abstraction through domain appropriate constructs making efficient programming easier. These DSLs are further sequential languages with all parallelism and use of architecture specific features implied or left to the compiler. Provided the compiler exists this leaves the codes portable across a range of architectures.

   Their proposed compiler additionally consists of an optimisation framework and runtime environment, *Delite*. While each DSL is intended for sequential coding, Delite specific operations defined in a DSL are packaged with their immediate dependencies. Based on these dependencies the Delite runtime is able to build a dynamic execution graph and schedule parallel operations accordingly, and the granularity of parallelism desired is therefore set according to the granularity of Delite operations defined. The execution graph further enables Delite to parallelise and map applications to multiple heterogeneous architectures optimally. Operations from a sequential thread are submitted to the environment for execution but return a proxy for the results immediately allowing the thread to continue by submitting the next operation. The application is oblivious to the fact that the operation may well not yet have finished, but runs ahead speculatively allowing more operations to be submitted.

   Because operation dependencies are also supplied the runtime environment is able to maintain a dynamic task graph. After taking into account communication costs, it schedules as many independent tasks as possible for parallel exe-

cution on multiple parallel architectures. With many operations taking place in parallel the speculation will often succeed and the correct results will be ready for use when needed. Where the hard result rather than the proxy is required before it has been generated, however, its source operation will be scheduled for immediate completion.

The Delite framework and runtime environment therefore expose both task and data parallelism without using explicit parallel code designators. Task level parallelism is exposed by enabling the dependence checked speculative run-ahead execution for each invocation of a Delite operation, and by providing parallel hardware capacity for such. Where domain knowledge suggests that some operations be executed on a GPGPU or other custom architecture, the DSL author must assert such in the DSL definition, and the compiler must be provided with a means of transforming the operation into the architecture appropriate equivalent. In the case of GPGPUs for instance this means, marking the operations for such, matching Delite operations to CUDA operations, and ensuring only straight forward memory mappings are required. This can be made easier by the restricted semantics of the DSLs which can allow for automatic transformations from DSL to CUDA. The exposure of data level parallelisation is also due to a DSL author's domain knowledge. In this case operations that the author knows contain data parallelism must be defined accordingly. To facilitate such a number of data parallel archetype classes already exist in Delite which may be easily expanded.

The Delite approach is similar to the use of libraries to extend general purpose languages to support parallelisation for certain applications. It is the use of DSLs, however, that make it possible to have implicit parallelisation as there is a domain from which such might be inferred about the data structures or algorithm. Further there is greater scope for application specific optimizations. Where a general purpose language compiler will err on the cautious side, algorithm robustness may allow for more relaxed dependencies for a lower precision result and fewer iterations in a sub function may be a sufficient best effort computation.

The above three approaches are all radically different, sharing only the trait of collaborating with the compiler but leaving it to find more parallelism than is the case in any of the previously examined models.

2.3.2 Autoparallelising Software Stacks

Historically the performance gains possible with autoparallelisation have generally fallen far short of what can be achieved through manual parallelisation, at least over any sort of range of applications rather than single instances. More recently targeting HPC specifically are predominantly high-level source-to-source tools which generally take in C/C++ or Fortran and either convert OpenMP annotated code to another parallel form for a new architecture (eg MPI or CUDA), or take sequential code and add OpenMP, MPI, or CUDA transformations. A limited list of these environments includes, Cetus [Dave et al., 2009] (C to OpenMPC, or OpenMP C to MPI C or CUDA), its much older predecessor Polaris [Blume et al., 1996] (Fortran to parallel Fortran Dialect), PLUTO [Bondhugula et al., 2008] (sequential C to OpenMP C), Par4All [Villalon et al., 2012] (sequential C and Fortran to OpenMP C, CUDA, or OpenCL), and the Rose compiler infrastructure [LLNL, 2014] (C/C++ and Fortran to OpenMP and UPC). In addition to serving in these roles as functional autoparallelisers, these tool suits provide the infrastructures within which much of autoparallelisation research takes place. The problem is too complex for a whole process pipeline to be developed by one investigator. Instead the majority of the significant body of published work in the field is focused on optimising or improving a single stage and is done so using tools such as these.

As already discussed, however, these tools target a granularity of parallelism far too coarse for the goals of a many-core architecture. To give context to the later description of APPRASE and before examining systems that are more similar to it a brief introduction to the different stages in an automated parallelisation is given here. Any autoparallelisation tool suit will go through the following three stages in some form or another.

1. *Conversion to IR:* The process of optimising and parallelising generally requires code first be converted into a tool dependent intermediate representation (IR) which serves to better expose dependencies and allows for greater flexibility in code re-ordering. In some cases this stage will be run after the following dependence analysis but generally before hand.

2. *Dependence Analysis:* In order to determine analyse and circumvent dependencies, a directed acyclic graph (DAG) of the operations to be done is first built. Based on such, some dependencies may be circumvented through code (IR) reorganisation, before a final DAG is again produced.

3. *Instruction Scheduling:* Using the DAG and taking into account the effect on net execution performance of various hardware components, instructions must be ordered and placed for execution on the parallel hardware. This schedule in the IR, will ultimately be converted back into either an appropriate form of parallel high-level code or a parallel binary.

Three sources of dependencies exist, application data, application control, and hardware functionality. Data dependencies are best represented in Directed Acyclic Graph (DAG) where vertices correspond to instructions/tasks, and directed edges connect vertices indicating dependence and its direction. The 'acyclic' in DAG simply indicates that no unending cycles exist, as should be the case in a correctly graphed functional application with no infinite loops. Depending on the nature of the data dependence simple steps can allow for their removal. To help in discussion of such the standard dependency classifications are briefly given:

- *True/flow dependence or Read After Write [RAW]* $S_1 \delta S_2$: Occurs when a later instruction depends on the result of a preceding instruction. Data computed in S_1 is used in S_2, so data flows from S_1 to S_2.

- *Anti-dependence or Write After Read [WAR]* $S_1 \overline{\delta} S_2$: Occurs when an instruction uses a variable which is later updated. A variable used in S_1 is updated in S_2. This dependence is a name dependence and can therefore be removed by creating a copy of the value under a new name such that the updating of the variable in S_2 doesn't change the value used in S_1.

- *Output dependence or Write After Write [WAW]* $S_1 \delta^\circ S_2$: Occurs when a variable is assigned different values at different points in an application, the final assignment must obviously come last. S_1 causes a variable to be updated and later S_2 updates the same variable with a new value. This is also a name dependency and can therefore also be removed through renaming.

- *Control dependence* $S_1 \delta^c S_2$: Occurs when the outcome of a preceding instruction determines whether or not a following instruction should even be executed. S_2 is control dependent on S_1 to determine if it executes.

As suggested it is possible to remove all anti- and output dependencies through variable renaming, leaving the program with only true data dependencies. A code transformed to be in this state is said to be in Single-static-assignment form [SSA]

[Cytron et al., 1989]. One of the limitations of such is that compile-time analysis can only reveal static dependencies. Run-time generated dynamic dependencies must be handled with some form of speculation. A significant quantity of work has gone into this domain, where loop iterations, subroutines, or branches may be speculatively executed either by the hardware or according to compile-time generated schedules in software. In either case a means of rolling back or side-stepping the invalid results is needed. This can be very expensive in hardware registers and control path. [Aleen and Clark, 2009] and [Saad et al., 2012] both include reviews of such speculative methods showing their success.

A second point of concern in using the removal of static dependencies through SSA form, is that it has the potential to result in a much larger DAG with a longer critical path than had existed previously. The goal, however, is to create the opportunity for greater parallel execution which will ultimately reduce execution time relative to the sequential original.

Finally, when making scheduling decisions architectural and control dependencies are quantified using various heuristics. Scheduling is carried out by examining a list of candidate ready-to-execute instructions every time step, and assigning value or rank to each based on a heuristic that indicates which it is 'best' to schedule next. Different compilers have adopted different approaches to this stage depending on the target architecture and granularity of parallelisation. It must be taken into consideration that in a particularly large application this may result in long compile times depending on the scheduling techniques used. Different scheduling techniques rely on different heuristics. The collection of these may require a separate parses of DAG. The following describes some of the more common ones.

- *Start time:* The earliest, latest, or difference between two instruction's start times. Instructions may be scheduled according to who has the earliest start time on the understanding that this may free up more instructions in the next steps. Updates to start times must be constantly made during scheduling as children's start times will be altered as parents are scheduled. The same is done in reverse when considering latest start time and instructions are packed from the bottom up. The difference between the earliest and latest start times adds further information as a slack value indicating effect on the critical path (the sequence of instructions that are the direct cause of the minimal total execution time as they must execute sequentially, defined as zero slack). In a pipelined data path this is distinct from *execution time* but the same methods can still largely be applied.

- *Number of uncovered children/parents:* Depending on the direction of traverse (forward from program start to finish or the reverse) this quantifies the number of children/parent nodes that will be immediately available as candidate nodes in the next time step. The more candidate nodes to be made available, the greater scheduling flexibility made available. This may be further weighted with the sum of the delay between a child and its parents.
- *Number of parents:* The more parents a node has the larger the number of instructions a node must wait for to be scheduled.
- *Execution duration:* In an environment where different instructions require different periods of time to complete execution, the execution duration will affect the quantity of available resources for the scheduling of new instructions in the next step. In general it is better to schedule those instructions with the longest execution duration first as the ones consuming the most resources (time).
- *Alternate type:* In an environment with multiple different functional units selecting instructions based on their being different to others scheduled in the same time step is clearly essential to ensuring efficient resource use.
- *Path length:* The number of arcs between a node and the DAG's root or the final end associated leaf can be used in attempting to balance the execution of all paths of a DAG.
- *Registers:* Again, as resource usage must be balanced, for fine grained instruction scheduling it can be advantageous to postpone an instruction that use many registers or to promote instructions that use less.
- *Length of critical path:* Initiating the longest critical path the earliest has the potential to minimise execution time.

In this work the scheduling alternatively is carried out effectively based on start time, but in contrast to other work found it uses a reverse traversal of the new DAG.

With the above process in mind those compilers that engage in the lower level parallelisation of single application instructions are discussed here. For research purposes, SUIF [Wilson et al., 1994], Open64 [CAPSL, 2014], and LLVM [Lattner and Adve, 2004] are tool suits each centred around their own custom IRs facilitating compilation research on multiple architectures and languages. In the same domain as the tools listed above these are only distinguished here as within the literature they appear to

be used primarily for this lower level approach rather than the source-to-source transformers.

For production work, in addition to being capable of auto-threading through OpenMP support, the standard gcc [GCC Wiki, 2009], IntelC/C++ [Intel, 2012c], and PGI compilers [PGI, 2013] used for compiling both sequential and parallel annotated codes, are all responsible for extracting any possible ILP opportunities (branch prediction, speculative execution, out-of-order execution, register renaming, instruction pipelining). Further than this, depending on the users flag selection, basic loop unrolling, and vectorisation may optionally be carried out although such may come with cautions on their guaranteed correctness. These approaches, however, despite their apparent low level only exploit either the most obvious forms of parallelism or rely solely on the complex control structures in place in the complex core being targeted.

Alternatively, the domain of *'Binary re-writers'* takes in sequential binaries generated by the likes of the above and similar compilers. Using the above described standard autoparallelising techniques these parallelise the machine code instructions rather than the source. At this low level, where original source language is indiscernible, significant portability, programmability (multilingual codes are no longer a problem), and legacy code difficulties are removed. However, while recent works [Kotha et al., 2010; Pradelle et al., 2012] have made advances in tackling the genre's traditional barrier (lack of high-level symbolic information from which to infer data structure and dependencies), re-targeting a binary to a new ISA as radically different to the x86 and similar ISAs as a many-core is, is impractical. With minimal 1:1 mappings of machine instructions, the resulting binary would be enormously bloated, indirect and fundamentally inefficient.

None of the above, however, are targeting architectures at all representative of a many-core. Selecting from the architectures reviewed in the previous section, the VLIW Itanium compiler and the original RAWCC compiler(Fortran was discarded with the redesign and move to TRIPS architecture) are the only ones to support Fortran. While the details of the original RAW compiler are not published, and it is beyond the scope of this work to go into a detailed review of Itaniums scheduling algorithm, the high level tactics of such were elaborated on in explaining the hardware. It remains only to list two broad steps taken by VLIW compilers in general. LOADS are generally pushed to execute as early as possible on the basis that such will give them as much time to execute as possible in the case of a cache miss. STORES are similarly pushed to execute as late as possible, all within the constraints of greater dependencies. Once

these stages have been arranged complete operations are bundled into groups and scheduled as VLIW instructions.

2.3.3 Summary

Having summarised existing parallel programming models and briefly reviewed the current state of autoparallelising software stacks, it is evident that there is minimal work being done on this thesis' specific domain. This is not surprising given the mutual development path that must be followed and the novelty present in creating a many-core architecture-compiler system that exploits static parallelism. It is also evident that current traditional parallel programming models are unsuitable for handling many-core granularities of parallelism.

None of the currently used parallel programming models meet the requirements for many-core entirely. Nevertheless, the manner in which GPGPU programming models are integrating with the more traditional shared and distributed cluster models, agrees with the earlier proposal that a similar avenue would be appropriate if many-core architectures were to be adopted as cluster nodes or node cores.

The development of a number of distinctly different implicitly parallel programming models and their middle ground approach also lends weight to the proposed tactic of using autoparallelisation in targeting the many-core architecture, but not the complete cluster system. The authors of these works recognised that in many low-level cases a parallel guiding but sequential development environment combined with appropriate automation can achieve more than manual efforts.

While autoparallelisation in the only directly relevant context of VLIW and Itanium architectures has not seen distinct success historically, a great deal of new work is being done using the same compiler development environments as the GPGPU models use (most notably LLVM/CLANG). This shows the basic infrastructure and methods for such are already known. The complexity and hardware specific nature of the GPGPU model also adds weight to the proposed hypothesis that the granularity of parallelism required for many-core architectures to succeed may only be achievable through automation that allows code development using a sequential model. The GPGPUs only achieve this level through a complex programming model and an averaging of control that costs either application domain range or performance. Use of the GPGPU model has also emphasised the apparent need for optimisations even on only subtly differing hardware platforms. This stands in opposition to the goal

of portability, although this work is not proposing that one autoparallelising compiler could solve this issue either and instead has taken this and other indicators to mean such multi-faceted portability will require a combination of multiple tactics. In presenting the ideal requirements for a many-core programming model, however, how autoparallelisation would facilitate providing each was argued showing extensive but not complete coverage of all points. Later chapters will elaborate on how the APPRASE-Fynbos system practically attempts to meet these ideals, along with acknowledging where it fails to do so and how it is positioned to let other infrastructures implement the rest.

Together with the already discussed reasons to support legacy code and autoparallelisation's potential capacity to do so, these factors indicate that despite its challenges an automated parallelising compiler offers many advantages the above models cannot contribute. This argument is made, however, while still also conceding the need for new many-core appropriate algorithms, regardless of how programming them is approached. Delite's use of DSLs in particular confirms this by indicating that others also see the expected need for new algorithms. Their work illustrates at least one other approach to the current revolution. Their approach is very different to that being explored in this work and is most applicable to new rather than legacy applications. This confirms that regardless of the advantages noted here autoparallelisation alone will not be a sufficient solution.

2.4 Summary and Conclusions

In summary therefore. There are many avenues of research needed to truly circumvent the walls facing an industry aiming to reach exa-scale computing. These challenges exist right through the entire chain from fabrication techniques and low level software control of such, through higher level hardware architectural models, to programming models and fundamental algorithm design to match the complete chain.

In Chapter 1 a definition of a many-core architecture was given based on the many-core hypothesis proposing papers. In the above a wide range of architectures have been reviewed, their only point of commonality being that they have something relevant to contribute to a discussion of exactly how a many-core architecture might be implemented/ Finally what the requirements of a many-core programming model are and how currently available models and autoparallelising tool sets compare to this

was discussed. Taking into account both the described definition, this thesis' hypothesis, and the literature reviewed therefore the following proposes what implementation strategies appear worth pursuing.

- The inherent trade-off between parallelism and latency offers a significant barrier to the many-core hypothesis, greater parallelism means data and control need to cover greater distances. Yet scalability underpins the many-core hypothesis.

  - As regards memory infrastructure, based on the literature reviewed the only application independent means of achieving extensive scalability it appears is with a distributed memory structure. Such a system comes with system control and data movement complexities addressed below, but also means the locality of data in space and time is as important if not more so than in any other architecture. Further this memory must be implemented in a manner that is not constrained to exploiting specific application domain characteristics.
  - As regards execution control and correct memory access ordering, scalability can only be achieved through operational models that exhibit decentralised control (dataflow operation, distributed instructions, asynchronous operation, or SIMD operation for instance). Any largely hardware based control mechanism will reduce scalability. However, of the works reviewed dataflow was too restrictive a control model for scaling and wide spread application domain applicability, streaming was too reliant on application domain specific exploits, and MPPA was too reliant on full CPU control capacity which places a greater burden on the programmer and more hardware is consumed in control (counter to the many-core hypothesis). An alternative approach is needed.
  - As regards data movement, in streaming architectures the problem of distance is masked by reducing data movement costs through bulk transfer and by parallelising and pipelining these transfers with execution. Static scheduling and a configurable or programmable interconnect can enable the same gains in a non-streaming environment by being in a position to pre-emptively initiate data transfers even of single words as soon as the data is available, and pipelining and parallelising them to occur alongside

execution too. It was seen in a number of the works reviewed that this is possible through the support of a compile time configurable or programmable interconnect which can further also be optimised for different application domains. Unfortunately an optimal similar interconnect strategy has not emerged clearly from the literature reviewed, primarily due to how closely linked it is to the remaining design choices.

- Static scheduling is able to address, facilitate and effectively match the implementation of all of the above points deriving from a distributed memory system (data locality, operational control, configurable interconnect). It will, however, cost performance when targeting a wide range of applications unless sufficient mitigating mechanisms that are primarily software and not hardware dependant are implemented. The literature suggests that the use of predication, branch prediction, and speculation in ways that exploit the advantages of the architecture (lower clock speeds with shorter pipelines, distributed memories, and an abundance of PEs), can succeed at this.

- Considering that the degree of parallelism required in codes running on a many-core is beyond the capabilities of manual coding in any case, the literature suggests that ideally, to ensure portability and adoption by industry, a sequential development environment should be used in combination with an autoparalleliser. Fundamentally this means both new programming models and new algorithms adapted to the new architectures are necessary. In order for any such changes to be adopted they will need to be introduced in a manner that facilitates porting of older codes, and exploits and expands rather than replaces existing expertise.

  - At the many-core autoparallelising level the literature also showed the advantages of exposing the entire code (to be issued on the many-core) to the paralleliser such that in an environment of such abundant parallelism it can detect and exploit all opportunities (even where this involves executing final administrative tasks say at the beginning of an application where possible alongside initialisation serial tasks). While this work will be introducing one such autoparallelising approach to meet this need, a further

motivation to consider this approach viable more broadly is that the software infrastructure for alternative investigative projects is already well established.

  - Given the view put forward in Chapter 1, that many-core-like architectures would fit best into clusters as node processors. It is apparent that code development and execution at higher levels than these many-core nodes should continue to be carried out using existing system appropriate parallelisation models and infrastructure, including manual parallelisation of codes on a macro level. Such an approach will be highly advantageous in achieving performance, industry adoption, and code ports of legacy applications.
  - Similarly, the adopted interconnect will need to equate to very high bandwidth memory access and off-chip IO, and will ideally further enable easy multi-chip integration. While bulk transfer to and from a system is an efficient approach it requires an application be segmented into units for which the complete data set can be passed to the processor. This has been seen to work provided the blocks are large enough to be efficient. The alternative is to share a cache level with a host CPU which has its own advantages but in the instances reviewed these were not worth the performance or scaling cost incurred.

- Given the current pursuit of and the high value placed on energy efficient computation, and this work's focus on general purpose HPC specifically, it appears unlikely that a reconfigurable design (with the energy and programming or compiler complexity costs incurred by such) would be advantageous in this particular case. This is other than as an accelerator where the performance gains of custom hardware may outweigh the energy costs of reconfigurability. A further obvious requirement for an HPC architecture is floating point support.

- An aspect that does not appear to have been explored in the literature is that of extensively enabling different PEs differently. This work will explore this briefly for practical resource limitation reasons. In a similar vein fine-grained power control, of the nature needed in a many-core, was not specifically addressed in any of the architectures reviewed.

- An optional point of concern lies in if a many-core architecture is implemented using NTV technology, the two concepts being a very appropriate match for one another. Many-core's focus being on performance via parallelism rather than on fast deep pipelines. If NTV operation is intended, however, an increase in faulty transistor operation needs to be anticipated with redundancy and recovery structures included in the designed.

While there is no guarantee that taking into account all of the above points will result in a better HPC computing node, there will be sufficient novelty in the result that it appears worth exploring.

Chapter 3

APPRASE Pipeline

3.1 Introduction

Based on Chapters 1 and 2, the hypothesis proposed is to explore one avenue amongst many possibilities in the largely unexplored domain of finding a means to significantly greater computing efficiency. Chapter 2 particularly raises far more questions around both hardware and software than this work is going to answer. New solutions in both are needed and while this work's focus is on the hardware the one cannot be developed or used in isolation from the other. As a result the investigation carried out is into the joint APPRASE-Fynbos system. It is in response to the discussed complexities and requirements of programming a many-core architecture that an autoparallelising solution, and particularly one that has capacity to address legacy codes, is seen to be highly advantageous and therefore worth this investigation.

While ideally an architecture and compiler will be developed alongside one another, APPRASE has a history dating much further back than Fynbos to the 1960's. The one advantage of this, however, is that its techniques have already been shown to be effective. This chapter therefore serves to firstly provide context by describing the origins of the APPRASE pipeline so as to justify its use, and secondly to detail the pipeline as it now exists adapted to Fynbos and Fynbos to it. While the fundamental principles of the compiler stack are historical, as is inevitable it has evolved and been further developed during the hardware architectural design and implementation phase of Fynbos. Even as it stands now further work is already required and more would be needed if a second improved version of Fynbos were to be created, as the conclusions will detail. To be clear the work described here is not the authors but detailed only for the reasons given above.

3.2 Compiler History

APPRASE has its origins in an era prior to the easy scaling of Moore's law. The algorithmic basis for it weres developed at Applied Dynamics International (ADI)[1] (a hardware-in-the-loop simulation systems company) in 1975 for a machine called the AD-10. Use cases for the AD-10 included simulation of jet and helicopter flights, missile guidance systems, space vehicles behaviour, and a nuclear power station control system [Peter, 1983].

The AD-10's software stack is significant now as it made use of a programming model similar to the more recently proposed implicitly parallel programming models. In one of the few papers now still available discussing the AD-10, the following quoted statement bears evidence to this while listing the AD-10's advantages:

> "...Fourth, the sequentially coordinated operation of the different processors minimizes programming difficulties inherent in parallel processing. This sequential orientation is consistent with usual programming concepts and avoids the very difficult task assignment and data communication problems which characterize other parallel architectures."[Gilbert and Howe, 1978]

Due to the relevance of this stack, the AD-10's architecture and stated purpose are also worth review as these will inform later discussions and analyses of results and design. The following details on the AD-10's architecture and functioning are taken primarily from the above mentioned [Gilbert and Howe, 1978], along with [Howe, 2005], [Peterson, 1984] (reporting a users experiences in using the machine), and finally from crucial conversations with one of the compiler writers for the machine and co-supervisor in this work, J. Collins.

At a high level, the AD-10 was a 16-bit word, fixed-point multiprocessor consisting of five different base functional processor modules on a shared bus. Each processor used an independent program counter and program memory, but all ran on the same 10MHz clock and execution of a simulation was carried out with all processors operating simultaneously.

The implementation of the system in five different functional processors served two purposes. Firstly it allowed for each to be designed solely for its function, greatly reducing complexity and thereby gaining operating speed. Secondly, as the separate

[1] www.adi.com

modules are run simultaneously a level of parallelism is provided along with the ability to pipeline activities to further improve on performance.

Programming the AD-10 involved making calls to a library of macro-files containing optimised assembly routines which made use of the pipelining and parallelism inherent in the AD-10. A language, MPS10, was developed to abstract the programming environment away from assembly language.

In MPS10 the programmer was required to divide the application into operational categories, a set of which constituted an applications program. Amongst others these included summation, numerical integration, function generation, coordinate transformation, vector cross product, addition, and multiplication. The program would be stored in the AD-10's data memory module, but the instructions within a particular operational category would be stored in the program memory of the functional processor that was to execute them.

In a manner similar to streaming processors during execution one set of operations was called to run at a time but its operations were applied to as many variables as possible so as to amortise the execution overheads of initiation. Dealing with a stream of values each functional processor was internally pipelined. Depending on the category, multiple functional processors might be further pipelined together. To provide a means of optimally finding the largest group of independent similar operations to run together a utility named DAREA was developed.

3.2.0.1 DAREA

DAREA parsed the MPS10 source code building a DAG in the form of a table of all variables referenced to the equation (if any) which produced it. With the dependencies represented as such, DAREA uses an adaptation of a scheduling approach proposed in [Kuck et al., 1981] to group operations for the optimal execution sequence and data area assignments.

The scheduling algorithm used was described by then ADI employed J. Collins in a paper [Collins, 1986] submitted for publication to P.A.R.L.E 1986. It was rejected, however, on the basis that it would not work. It did, however, and was used to pack the AD-10's data memory module far more efficiently than a human as evidenced in [Peterson, 1984]. In this case study of a jet engine simulation by General Electric, DAREA achieved 2.4X improvement in processing frame rate (performance improvement), and a saving of 30% on the use of data areas.

As discussed parallelising code requires a number of stages, here the scheduling algorithm specifically is summarised in Figure 3.1, and the following explanation.

To begin a topological sort of the DAG is done assigning every statement to an execution level as follows. Parsing the table all constants and state variables (results of integration) are marked and assigned to level 0. There after an iterative process is carried out of parsing the table for all statements with already marked input variables and assigning such to the next level.

The above sorting schedules statements for execution as soon as their inputs exist and results in the fewest number of execution levels possible, in this case five versus the original sequential ten. But this assumes no limits on the number of parallel execution units which is unrealistic. Figure 3.1 assumes a 2-way parallel machine meaning levels 0 and 2 must lose an instruction to one of the later levels, conveniently there are two remaining open slots meaning the program length remains the same. In the same way whether a wider or narrower parallel machine is better or worse (taking into account both execution time and logic and power costs) becomes a point of discussion in Chapter 5 where the joint system is analysed. Often allowing one level to be significantly wider allows many more to be moderately wider and the program length to be shortened, but this comes at the cost of many empty slots and therefore an apparently inefficient execution. This is a problem that has troubled all wide parallel processing machines, proposals for how APPRASE-Fynbos could address it efficiently are made later but not implemented.

While this example is relatively trivial the domain of scheduling research offers many different approaches that might be taken in determining how to fit the sort to the 2-way hardware. DAREA uses a method that begins at the highest or last execution level, selecting what operations will be carried out in the final level first, and continuing to schedule each level in reverse order. While doing so it is also continuously looking through the lower (earlier) levels for instructions which may be promoted to a higher (later) level. This is possible as while the topological sort will place a variable (statement) that is only dependant on variables in the level immediately prior (n-1), it may only be needed at level n+5. To keep track of such ranges after every promotion DAREA makes use of a heuristic termed the *float* of a variable which gives the range of levels to which each could be assigned. In cases where there are a number of variables that could be promoted DAREA will select the one with the most parents (inputs) so as to free up as much space as possible in the earlier levels.

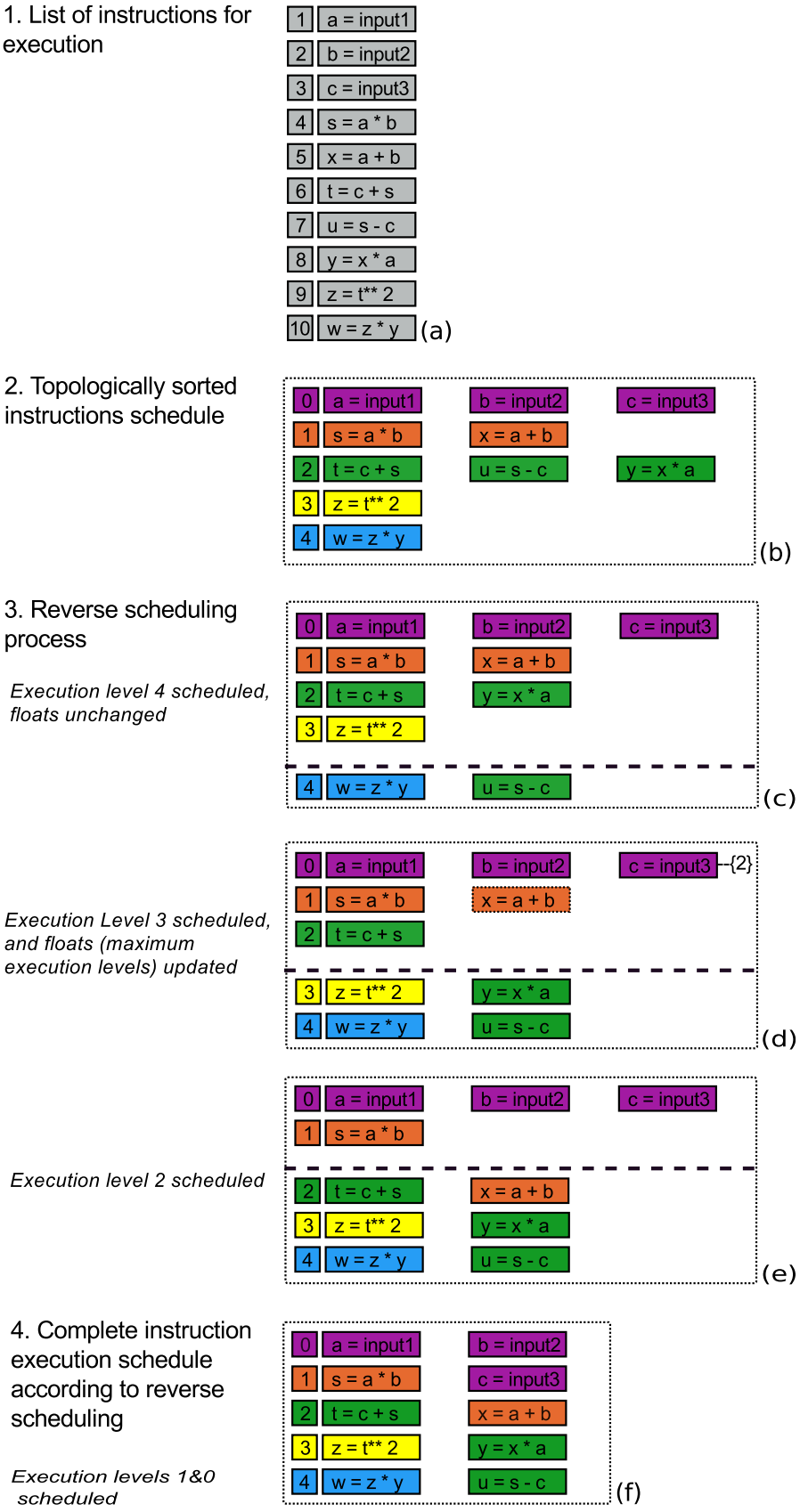


Figure 3.1 – Reverse scheduling algorithm demonstration given a 2-way execution pipeline. Values below the dashed dividing line have been scheduled.

The above methodology is based on the consideration that often programs begin with many inputs, and funnel down to less outputs. If an instruction can be promoted (it's execution delayed), then its dependencies can be delayed, creating more space and flexibility in earlier execution levels, and arriving at a more efficient (fewer more densely packed) execution levels. As an example, in Figure 3.1 (c), w must be scheduled in level 4, and based on their float ranges, u should be scheduled alongside it. Because the variables u depends on (s & c) are also already needed earlier none of the remaining float ranges change in this step. In the following iteration (Figure 3.1 (d)), however, with level 3 scheduled, due to the promotion of y the float range of x is updated such that it can now be scheduled in level 2 (shown in Figure 3.1 (e)). The process continues until level 0 is scheduled. By scheduling instructions in this reverse order the search space for the next instruction to be scheduled is greatly narrowed (there is no backtracking) making it easier and computationally less expensive to apply other constrains such as operator latency (used in Fynbos), memory latency, or register requirements.

3.3 APPRASE

While the subject of code optimisation and autoparallelisation has been long investigated currently there is more focus on C and other more modern languages. Section 2.3 detailed the relevant works on this subject showing that there appears to be no other similar published work on trying to autoparallelise Fortran at a many-core granularity. Further while not exhaustive of the tool suites and approaches most relevant and therefore reviewed none appeared to make use of a similar scheduling algorithm.

The above background explains where APPRASE comes from and the type of machine and applications its original forms were used on. It was first published as APPRASE (Automated Pipeline for the Parallelisation of RAndom Scalar Expressions) in brief in [Collins John, 2008]. Adapting the pipeline for current sequential Fortran and Fynbos has involved the use of a commercial Fortran analysis tool developed by J. Collins, WinFPT.[1] Algorithm 1, gives a shortened version of the stages this tool takes the code through when targeting Fynbos, with greater details to be given later in Algorithm 2.

[1] http://www.simconglobal.com/winfpt.html

Algorithm 1: The primary software stages of the APPRASE pipeline

1 Optimisations.
2 Separation of *lives*.
3 Decomposition into *triplets*.
4 Conversion to Fynbos notation.
5 Scheduling by DAREA.
6 Execution in a simulator or packing for execution on the Fynbos hardware.

Beginning with stage one optimisations, in general these are common tactics used in many other tool chains such as loop unrolling, code in lining, and function checking in preparation for mapping to the hardware. In stage two the code is converted into a form which removes many dependencies (analogous to the commonly used SSA form discussed earlier in Section 2.3). The transformation models a variable's period of use as a *life*. For every life of each variable identified the variable is renamed with a unique identifier making a new variable with a single life only. The rules for life identification are [Farrimond and Collins, 2007];

1. *Every variable life starts with an initial state or with an assignment, and ends with a new assignment or the end of a scope (its use).*

2. *Every right-hand-side use is checked. Two assignments or the initial state belong to the same life if they may both reach the same right-hand-side use.*

Both SSA form and life analysis make use of control flow graphs and remove all WAW and WAR dependencies enabling greater parallelism. SSA form differs subtly, however, in that every time a variable is assigned it is always given a new name, while life analysis examines a variable's future to determine which assignments lead to the same right hand side use point. In SSA form therefore, for every use of an original variable on the right hand side, a decision function (phi) must be inserted to decide which of the previously assigned values will be used. With the use of lives, two assignments may constitute the same life if both can reach the same right-hand-side use. As the same life is under consideration, no decision function is needed. This is perhaps most clearly shown with the example in code Listing 3.1.

Continuing at stage 3 of Algorithm 1, with loops unrolled, functions in-lined, and dependencies optimised, the code is decomposed into the target granularity of instructions to be parallelised. Termed *triplets*, the code is represented in expressions predominantly of the form: $a = b < operator > c$ (see Algorithm 2 for more cases).

```
IF (Q)
        X = 5     becomes X1 = 5                        (1)
ELSE
        X = 4     becomes X2 = 4                        (2)
ENDIF
Z = X             becomes X3 = PHI(X1,X2)  (3)
                                                  Z3 = X3
```

Listing 3.1 – SSA form vs Life analysis example. SSA form will adopt all the shown transformations but in this code segment X will only have one life and therefore one name as both assignments reach the same right hand side use.

Up until this point the code remains syntactically correct Fortran and may therefore be re-run for the purposes of verifying functionality post-transformation. Stage 4 involves converting the code into an IR that DAREA (used in stage 5) will be able to interpret. In stage 5, DAREA schedules the code according to the reverse algorithm described earlier and produces an xml representation which may be passed to both, a representative Fynbos simulator for debugging and testing purposes (created by a third collaborator, B. Farrimond), and to a compiler script responsible for generating a Fynbos load stream.

3.3.1 Matching APPRASE to Fynbos

Fynbos was designed to match the pre-existing APPRASE, and given the above pipeline various features were dictated to the hardware. Most of these will be detailed in Chapter 4's architecture description, but some reverse influences of the hardware on software were also made as trade-offs and mitigations.

In part due to the limitations of APPRASE and in part a decision in pursuit of simple hardware, but one of the biggest impact design trade-off decisions made was to use lock-step operation to synchronise all PEs. On an operational and performance level lock-step simplifies the hardware and software control required but comes at a performance cost. In theory significant performance compensating measures are possible and can be added in software but limited development time on the developer's part has meant they have not yet been implemented. Therefore, while Chapter 4 will discuss the positive and negative implications of lock-step on the hardware and

Algorithm 2: Stages of processing taken in autoparallelising Fortran with APPRASE.

- A list of the transformations made by WinFPT, these are enacted strictly in the following order:
  i. All intrinsic functions and subroutines are checked against a list of available Fynbos hardware supported operators. Those supported in Fynbos (for instances ADD, MUL, ABS,...) are left in place. Any operator not supported is declared EXTERNAL thereby turning them into function and subroutine calls for which a code routine needs to be supplied.
  ii. All sub-programs are expanded in-line. The process begins with the most deeply nested but runs to completion making the order irrelevant. As a side effect the process moves values into COMMON blocks on the way, but this ultimately unimportant as in the end there is only one remaining scope as once complete all variables are local to the main program.
  iii. All DO loops are unrolled. In the case of nested loops, unrolling begins with the outer loop in order to catch any inner loop bounds that may depend on the value of an outer loop index.
  iv. All other control constructs are replaced by constructs of the form: IF <logical expression> GOTO label.
  v. Scalarisation: All array elements are replaced by scalar variables.
  vi. Separation of lives: All variable lives are identified and each separate life of each variable is renamed to create a unique variable. The rules for life identification are again:
    (a) Every variable life begins with an initial state or with an assignment, and ends with a new assignment or the end of a scope (its use).
    (b) Every right-hand-side use is checked. Two assignments or the initial state belong to the same life if they may both reach the same right-hand-side use.
  vii. All expressions are decomposed into multiple statements termed triplets and in the forms of;
    (a) $a = b < operator > c$, or $a =< operator > b$
    (b) In the case of a control operation this is extended to the formats: $IF(a)\ GOTO\ label$, or $IF(.NOT.a)\ GOTO\ label$
  viii. At this point, WinFPT regenerates the codes as modified Fortran which may be run to confirm that the behaviour is unchanged.
  ix. The f2hydra plugin to WinFPT then converts the code to Fynbos notation for submission to drxh (DAREA).
- Some order dependencies of the transforms:
  1. Classification of intrinsics must take place before in-line expansion because the sub-programs for the intrinsics are themselves expanded in-line.
  2. Scalarisation must take place after loop unrolling so that the array indices have literal values.
  3. Identification of lives must take place after in-line expansion, loop unrolling and scalarisation because of the variable name transformations this causes.
  4. Decomposition into triples is easier after scalarisation as it can then operate on the final target objects.

the potential improvements that are possible with software changes, the negative impacts are discussed here.

With all PEs operating according to the same schedule and clock any operation that takes significantly longer than the average will keep all other PEs waiting. In Fynbos and due to the nature of the operation division is the operator with the longest latency (~9 cycles versus an average of 3 in Fynbos) with anywhere between 9 and 60 cycles [Kwon and Draper, 2009] for the same reported in other processors. A measure of mitigation already enacted by APPRASE is to simply promote the scheduling of multiple division operations to a common level. Once one division operation is scheduled into a level DAREA values scheduling more division operations in the same level higher than alternative operations. Across a wide execution width architecture, however, there will inevitably still be some efficiency loss but the results presented in Chapter 5 will show this feature makes a significant difference in performance metrics.

Another factor of consideration in examining operators like division, is that of hardware complexity, the longer latency being indicative of more complex processing, more registers, and longer logic chains. Division and its compatriot square root have traditionally been sidelined as infrequent enough to not warrant significant research, and tests show this to be on average true in the majority of SPEC benchmarks for instance, the exception being spice2g6 (a circuit simulation code) [Kwon and Draper, 2009]. They are, however, more prevalent in CAD packages and 3D graphics rendering, two applications which are increasingly becoming users of HPC hardware. Both operations are costly in die area requiring varying collections of multiplication, addition, shifting and other such units. In line with the move to simple hardware and this partial assumption of infrequent use, not all PEs in Fynbos therefore support division, while the equally resource intensive square root is implemented instead as a software routine of Heron's method. Both adaptations require software handling and serve primarily as test case prototypes for future work.

Not offering division capabilities in every PE in Fynbos adds to APPRASE's task of now not only scheduling as many divisions in parallel as possible but of also ensuring the operations are scheduled to a capable PE and that the operands reach that PE timeously. While a needed means of conserving hardware given the available platform constrictions (detailed later in Chapter 4), this also serves as a test platform for software handling heterogeneity in the PEs.

Handling of the square root subroutine is covered by Algorithm 2 step (i), the first step carried out in the APPRASE pipeline which involves checking for operations not supported in hardware and marking them as EXTERNAL (requiring a subroutine implementation). Given the same pursuit of reduced hardware complexity, it would be advantageous if complex operators could themselves be parallelised exploiting the abundance of parallelism a many core offers. While Heron's method is fundamentally serial (see Appendix A.1.3.1 for a literature review showing this to be the case for division and square root), other complex operators such as transcendentals are not. As a software test case therefore, Heron's method involves a series of dependant division operations. Therefore even while sequential by breaking an operation down into substeps it becomes possible to schedule these sub-steps in parallel with other unrelated operations.

Traditionally transcendental functions are also handled by specialised and expensive hardware modules due to their performance advantage over software alternatives. GPGPUs are one example of such where their special function units are more sparsely represented than the general purpose ALUs. The many-core architecture, however, may perhaps be able to switch the balance to be in favour of an appropriately parallel software implementations and APPRASE's capability to do such has been prototyped first using square root as a demonstrator. Again software developer time limits mean parallel software Taylor series approximations for instance have unfortunately not been developed and tested.

As a comparison to these techniques, confronting both the hardware cost and latency problems division and square root bring and as a representative VLIW architecture sharing the parallel trait with many-cores, Itanium adopted a tangentially similar two part approach. In hardware a reciprocal estimator is used which leaves the software to refine such to sufficient accuracy[Greer et al., 2002]. In the context of a VLIW, and further one containing two fully pipelined FMA modules, this approach provided three distinct advantages. Firstly, with latency directly linked to software controlled precision, there exists flexibility to lower latencies where less than standard precision values are sufficient, this theoretically could also be added to APPRASE. Secondly, the multi-operation iterative nature of such refinement means compilation opportunities existed for scheduling other independent instructions alongside those involved in carrying out a software implemented operation. While the deep pipelines in a VLIW open up greater scope for such than in Fynbos this is what using Heron's method on

Fynbos has attempted to accomplish even while the complete function is fundamentally serial. Finally, with the deep pipeline facilitating high throughput for consecutive operations, the software refinement and simpler hardware at reduced performance option (relative to a full hardware expensive but high performance operator) is more appealing as it frees up die area for more beneficial work. Such deep pipelines are not possible in Fynbos and the current design fails to advantage or make possible feeding results directly back into the ALU. Again hardware proposals are made later that would change the evaluation of this conclusion. Therefore while very appropriate for the deeply pipelined VLIW, only a portion of similar techniques are appropriate for the lock-step and much wider many-core being proposed by this work.

As indicated all further adjustments to APPRASE specific to matching it to Fynbos will be detailed in Chapter 4 where the hardware component description will motivate for these changes.

3.4 Host software

At the conclusion of the APPRASE pipeline an xml file is produced representing the scheduled application as targeted at a specific configuration of Fynbos. To complete the software process and execute an application in the hardware there remains the relatively trivial steps of compiling and loading the program. The low level details of these steps would require a full description of the hardware and while non-trivial sections they are neither innovative nor significant contributors and so are simply covered briefly here.

3.4.1 Fynbos Compiler (xml converter)

As this work is based entirely on static scheduling the xml file created includes directions of the exact placement (by PE and memory location) of each instruction and data word. Included in this is also an ordering of instructions and a record of the configuration of the Fynbos that was targeted during scheduling. To execute the program on Fynbos, a load file is created using the xml file as an input. This load file is the binary executable equivalent for Fynbos but given Fynbos's location as a co-processor and the interface involved (10Gbe) it is really a list of UDP packets ready for transmission by a further script. These packets contain the commands Fynbos needs to load a program, execute it, and finally return the results to a host if required. Stand alone

```
<row id="16">
. . .
    <tile id=" 6">
        . . .
        <strip id=" 2">
            <operator id="MUL_R4" output="R2_2_7" input_0="RTEMP_171" input_1
                ="S2_3_LF11" input_strip_0="2" input_strip_1="4" value
                =0.1496000000000000 0E+12"/>
            <export export_symbol_0="RTEMP_307" export_symbol_1="S2_3_LF19"
                export_row_0="14" export_row_1="11"/>
            <import import_symbol="R2_4_6" source_tile=" 1" destination_row
                ="44"/>
        </strip>
        . . .
    <tile>
. . .
<row>
```

Listing 3.2 – Small sample section from a xml file output of APPRASE pipeline.

commands may be sent to Fynbos outside of this file, it simply makes execution easier and automates the generation of such commands.

As an example Listing 3.2 shows a multiply operation to be located in Tile6, Strip2, instruction memory addresses 16 as it would be represented in the xml output of APPRASE. The basic functional operation of the contents of this listing are summarised as:

$R2\_2\_7 = RTEMP\_171 * S2\_3LF11$

Based on Listing 3.2, operand $RTEMP\_171$ is to be sourced from the local data memory, address 14 ($input\_strip\_0 = "2", export\_row\_0 = "14"$). While the second will be fetched from within Strip4 within the same tile ($input\_strip\_1 = "4"$), and imported to this operation. The operation's result will be stored locally in data memory address 16 by default, while the result of whatever operation takes place in Strip1 will also be imported and stored in this Strip2's local data memory at address 44 ($import\_symbol = "R2\_4\_6", source\_tile = "1" destination\_row = "44"$). The listing further indicates that on initialisation, Tile6, Strip2, data memory address 16, will be loaded with value 149.6e12 ($value = 0.1496000000000000E + 12"$).

Creating a load file from the above requires extracting the above information, forming Fynbos instructions from the information therein, converting the decimal data values given into IEEE754 FP representation, and formatting both into a hex repre-

sentation that is suitable for transmission over a 10Gbe UDP interface and into a 64b wide FIFO on a remotely hosted FPGA board. Having generated such a second script orders and packs the instruction and data words into the single load file, inserting the required host commands as appropriate (LOAD_DATA, LOAD_INSTRUCTIONS, RUN, OFF_LOAD explained in the following section).[1]

3.4.2 Communication

As indicated the primary means of communication with Fynbos is via a 10Gbe connection between a host server and a FPGA board. To create the interface within the FPGA one of the 16 hard-block GTP Gb transceivers is overlaid with a pay-for Xilinx supplied link-layer. A UDP transport layer created on top of this in conjunction with a software stack running on a neighbouring PowerPC core hosted on the same board were both supplied with the board. While an unreliable protocol such as UDP is not appropriate for Fynbos functionality (a consequence of available resources rather than choice), given the physical length, direct nature, and operating environment of the connection there have been no problems even with no additional packet validation hardware. Given the trade-offs involved this is considered acceptable for a prototype but in any real world service the particular FPGA board, remote co-processor model, UDP, and even 10Gbe interface would not be selected for use.

The structure of the load file is a consequence of this interface, the first word to be transmitted is a command to Fynbos that it should prepare to receive and load data or instructions. Whether data or instructions are loaded first is inconsequential but the commands are distinct. To avoid overflowing the receive buffer within the FPGA, the host will always wait for a request for more before transmitting a packet, but it does not expect any acknowledgement. Once loaded with a program's data and instructions a command to execute is delivered.

On completion notification is sent to the host which may or may not then need to request results be off loaded. To conserve on-board memory resources and improve operating efficiency data results may be transmitted back to the host during execution. The host must therefore be listening for such.[2] Alternatively a further command can initiate a process of reading out the contents of data memories for transmission

[1] The Python scripts and functions for carrying out these processes can be found in the attached software as indexed in Appendix B.

[2] In order to avoid receive buffer overflows on the host server's part, the default buffer size (in Linux: /proc/sys/net/core/rmem_default) may need to be increased. (in Linux set to /proc/sys/net/-core/rmem_max)

back to the host once execution is complete. In summary a load file initiates the following sequence and also contains the associated data and instructions in correct order:[1]

i. Load data

ii. Load instructions

iii. Execute program

iv. Off load data from specified memories (optional)

Depending on the program and Fynbos configuration targeted, significant proportions of the instruction packets transmitted could be filled with NOPs. This is an inefficiency which in a reconfigurable environment might be counteracted. The bitstream might be compiled to pre-fill the BRAMS with true NOP instructions. But this becomes impractical once programs are of such a size as to require multiple load-execute cycles (inevitable), as intermediate results would then have to be off-loaded and re-loaded in order to allow a re-flashing of the FPGA, or it may simply prove that an ASIC is more appropriate in any case.

3.5 Summary and Conclusions

To conclude therefore the AD-10 made use of the simplifications possible in using distributed computing to ensure scalability. To overcome the overhead of configuring the distributed system streaming and pipelined operation were used. These required dependence analysis and operation isolation similar to that done in parallelising an application. The programming model used, however, of calling functional modules sequentially, retains the ease of sequential programming and leaves the dependence analysis and data optimisations to a scheduler DAREA.

This approach is analogous to modern implicitly parallel programming models where the base sequential programming model is retained as far as possible. In the case of APPRASE, however, parallelisation is solely dependant on the compiler leaving only appropriately parallelisable algorithm selection to the programmer.

In addition to traditional optimisations and a SA form-like IR DAREA makes use of a reverse scheduling algorithm. This schedules the final instructions first and iteratively seeks to schedule instructions as late as possible, thus gaining as long a period

[1] The scripts required to transmit and receive packets are indexed in Appendix B.

as possible to schedule the instructions that generate the values they are dependant on. To the best of our knowledge this particular approach is novel. However, as was acknowledged earlier the algorithm is not the primary contribution of this work and so a fully extensive literature review of this aspect was not carried out.

Comparing APPRASE on a high-level to the other parallelising tools. While the APPRASE pipeline also contains steps that are source-to-source transformations a much lower level of parallelism than those reviewed targeted is the final output. For instance comparing APPRASE to Polaris/Cetus briefly. All three use array privatisation Tu and Padua [1994] where distinct instances of array elements may be allocated to a thread's private storage so as to prevent memory related dependence. But where Cetus and Polaris have done so by breaking the arrays into sections, APPRASE completely scalarises arrays into distinct elements. This has the implication that APPRASE tracks dependencies on a per element basis and in consequence a finer-granularity of parallelism is possible, but APPRASE will also eventually be overwhelmed by the size of some arrays. Or while Cetus does do sub-program dependency analysis it analyses them in isolation. In contrast APPRASE in-lines sub-programs such that their dependencies and operations will be included in the main programs dependency analysis.

Examining the compilers for some of the architectures reviewed, where others looked for parallelism within sub blocks (hyperblocks) of code APPRASE searches across the entire code (as do binary writers). It is able to do so as it can assume the complete program and data set is available to it. While this is another significant advantage of not employing a cache, once programs too large to fit within Fynbos are targeted sub-block partitioning of the code will also be needed. In this case, however, it is not needed to guarantee independence or order of flow (as the above examples do). As such it will be possible to break an application up into such sub blocks after parallelisation thereby minimising the lost opportunities.

To gain further performance the AD10 included application specific variable representation and compensating arithmetic to increase clock speeds. The APPRASE-Fynbos system currently offers only FP numerical representations, although theoretically a similar optimisation could be added if Fynbos remains reconfigurable.

Adapting APPRASE to Fynbos and vice versa, and considering the idealised goals for a many-core programming model set out earlier.

- How a many-core programming model interacts with memory depends on the hardware model adopted In this case memory is predominantly controlled through

complete foreknowledge of all data movement, APPRASE exploits the adopted distributed memory system in ways that are only possible given the scale and granularity involved in a single-chip many-core. The practicalities of such will be explained alongside the supporting hardware later but communication is overlaid to execute primarily in parallel with operations. No cache and foreknowledge of all latencies is exploited for performance and parallelism, scaling and capacity to feed data to so many ALUs is catered for, exploits such as duplication of values to increase performance are used wherever advantageous. As with any static scheduling approach dynamic events are a potential source of poor performance. Chapter 4 will examine other hardware mitigations and branching capacity but APPRASE is significantly aided, relative to the Itanium compiler, by a lack of cache and therefore complete foreknowledge of the latency of every operation and step.

- Based on lower frequency of use complex operations are catered for in Fynbos via heterogeneity in sparse hardware modules or overloading of function calls with parallel software implementations. As with VLIW architectures the goal is to parallelise sub-steps in a complex operation, at least with other independent operations if not within the operator itself. Contrasting to all other compilers reviewed APPRASE is targeting an architecture with a very reduced set of intrinsics, meaning far more operators need to be implemented in software. Given the goal of PE abundance this is potentially advantageous as it affords the opportunity to parallelise sub-steps of such software operators with other operations. The compiler capacity for such has been developed and tested with an overloading of square root but no further operators have been created.

- A lock-step control model is used to keep the hardware simple but this means APPRASE must take steps to ensure equally long latency operations are scheduled along side one another as far as possible. Given the logic constraints of the hardware development environment, division capacity is also constrained for which APPRASE must further compensate.

- APPRASE assumes higher level infrastructure management exists governing where its operation fits within a multi-level concurrency hierarchy. It has been proposed that APPRASE be integrated into existing models that require manual parallelisation on the higher cluster and node levels. Nevertheless clearly this is yet

to be done meaning no conclusions, other than that it is an attractive concept, can be drawn.

- APPRASE's sequential programming model for the programmer is clearly one of its greatest attractions. It meets both the requirement for programmability and productive verification techniques simply because sequential coding and debugging is already a well understood and supported domain. The multi-stage code transformation that produces functional Fortran at intermediated stages (baring the final few) for easy testing further enables easy debugging.

- While APPRASE is being proposed to be only as a component in a hypothesised larger infrastructure, as a convenient result of its heritage APPRASE clearly has the capacity to deal with a significant proportion of legacy HPC code given the supported language is Fortran. While integrating APPRASE into such a larger structure still needs to be done, APPRASE is providing one instance of the final crucial step in supporting legacy Fortran code porting to many-core. Clearly support for C and other languages, however, is still needed.

- While APPRASE is based on fine-grained control of application execution, as noted it fails to address the need for fine-grained power management control. This is more the result of a lack of development time and a function of the development hardware environment, however, than any fundamental flaw in the approach.

- While APPRASE cannot address architecture independence (its techniques are for the most part too tied in to the Fynbos architecture), DAREA's scheduling process could be adapted to alternative architectures. It is, however, difficult to imagine what relevant trade-off there would be to make it worth while having too different architectures with the same parallelising technique. As a model therefore the approach presented is architecture independent in that there is no reason the DAREA scheduling algorithm could not be applied to other system. But in the context of many-core the need to find all possible efficiency gaining opportunities (in applications and hardware targets) and the desirability of abstracting the parallelism away from the programmer, it is argued that it is a warranted trade-off to require greater compiler development work and recompilation of codes on a per target basis. As was concluded in Chapter 2 it is more likely that a combination of automation and manual intervention will be used

in matching compilers and code and applications to architectures in any future heterogeneous system, than that autoparalleisation would manage the whole hierarchy.

Chapter 4

Fynbos Architecture

4.1 Introduction

The following chapter describes the Fynbos architecture in detail. In this new era of microprocessor design in addition to performance, energy efficiency is now a second primary driving design goal. This is one which might be achieved through a number of tactics. In many-cores and Fynbos efficiency is bought with area in the form of as many parallel ALUs as possible. This and the era imposes a number of design principles that differ from convention, the following is a short list of such as proposed collectively in the recent literature.

1. *Pipelining is not as useful as it was in the past*: In the past pipelining bought higher clock speeds at the cost of additional registers and control logic. While area is cheaper than ever, in a context that is already artificially dropping clocking speeds to fit within the TDP pipelining might well gain nothing while still costing power. Further even when a maximum clock speed can be tolerated, it will likely only exacerbate the already existing speed gap between memories and the ALUs.

2. *Logic multiplexing is no longer beneficial*: Where dark silicon is an imminent surety, computational resource sharing generally only adds control complexity and lengthens routing lines.

3. *Memory bandwidth is now more important than latency*: Again with dropping clock frequencies it becomes easier to keep the processing logic fed with data from the slower memories. With growth in number of independent processing threads, however, memory bandwidth requirements grow significantly.

4. *Keep growing on-chip memories:* While lower latencies is no longer the motivation, large RAMs remain beneficial. Relative to other logic, on-chip caches with a far lower switching rate are a form of dim silicon and are therefore a power efficient use of transistors, provided the application domain can make use of such.

5. *Scalability is a requirement*: If parallelism is buying performance for less power than frequency, designs must scale to as wide as possible. Data paths, memory structures and hierarchies, and ALU capabilities should all take this into account. A sub category to this is the need for scalable data sharing tactics which go beyond the efficiencies possible with traditional cache coherence protocols. Depending on the application, memory multiplexing, multi-level novel interconnects, multi-tiered caches, PE hierarchical groupings, and even software hardware management are amongst the proposals for such.

6. *It is no longer worth conserving transistors*: If a use for transistors can be found that increases performance in an energy efficient way, implement it. The monetary and area price for such will no longer out weigh the value gained.

7. *Data sharing costs are no longer insignificant*: With multiple orders of magnitude increase in computational width, the power cost of sharing data between so many ALUs is no longer insignificant. With the on-chip interconnect now consuming a notable portion of the TDP its optimisation should be given greater consideration than in the past.

The version of Fynbos described here attempts, but certainly does not completely succeed, to take into account all of the above. While not succeeding in all, it does serve to examine some of the trade-offs involved, and highlight options for further development. Credit must be given to Nallatech[1] designer Richard Chamerblain, who prior to the authors involvement in the project, proposed the high-level Tile and Strip structure used in the following to match APPRASE. The practical implementation and detailed design of (instruction set, data and control paths, interfaces, control infrastructure, I/O and similar), however remain the authors.

[1] www.nallatech.com

4.1.1 Development Environment Hardware

The following section outlines the infrastructure environment Fynbos has been prototyped in. By definition a prototype is both; faulty in design, and a compromised representation of even that design. Prototyping hardware on an FPGA requires a range of compromises due to the substrate's limitations, these need to be recognised. In some cases, such as logic element consumption, they may be quantified and taken into account within a reasonable measure. In others, such as the I/O design implemented here, they force a less than ideal design. In either case, the process of practical implementation beyond simulation, has served to provide insights of practicality and a place from which a hypothetical ASIC implementation might be inferred, and will contribute to the wider industry discussions on the use of reconfigurable fabrics in HPC and many-cores.

During design, Fynbos was successively targeted at three different FPGAs, each as they became available for use. Obviously, for the purposes of examining a many-core architecture, the larger the FPGA the better. While this has required redesigns, the result is a architecture that is fully parametrised in terms of memory sizes, instruction width and structure, array shape and size, and homogeneity. Chapter 5 uses this to extrapolate what parameters a version on the latest Virtex 7 FPGAs and an ASIC might have.

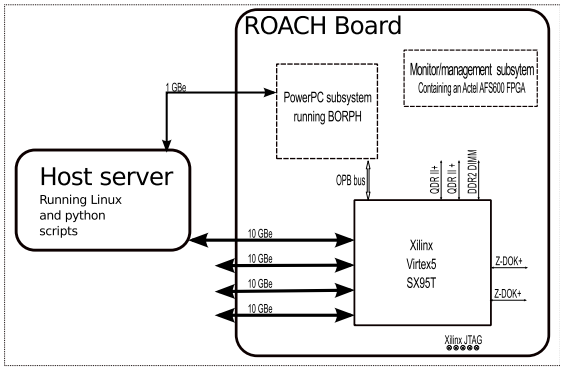


Figure 4.1 – Hardware development environment: A Linux server hosts a ROACH board[CASPER, 2009] supporting a Xilinx 5VSX95T FPGA.

Initial work targeted a Nallatech H101-PCIXE board[Nallatech, 2007] supporting a Virtex 4 chip and notably providing a PCI-X interface. This was followed by a Xilinx development kit supporting the much larger DSP focused SX50T Virtex 5, but with no direct off-board access to the FPGA. Finally, the Reconfigurable Open Architecture

Computing Hardware (ROACH)[CASPER, 2009] board, shown in Figure 4.1, supporting the even larger SX95T, and direct 10Gbe access, is what the following is targeted at. Designed for real-time processing as the back-end to radio telescopes in the Square Kilometre Array (SKA) project [SKA, 2013], the ROACH contains four 10Gbe ports connect directly to the FPGA, while a single 1Gbe connection serves to communicate with a PowerPC subsystem. [1]

Running on the PowerPC is the BORPH operating system[So and Brodersen, 2006]. This is an extended Linux kernel which handles the Ethernet stacks, and most significantly is capable of dynamically reading and writing registers on the FPGA. This subsystem is incidental in the operation of Fynbos but advantageous in this environment as it both handles the Ethernet stacks saving valuable FPGA resources, and provides an additional debugging avenue. To be clear, there was never any intention to use it as host in a host-coprocessor type configuration as seen in many similar works. With no means of host DMA, and all I/O carried out via one of the 10Gbe connections, the system is unrepresentative of how a many-core would likely be hosted, but is sufficient for prototyping purposes.

Examining the SX95T, the chip comes from Xilinx's high performance signal processing range, supporting a large proportion of high speed I/O and DSP slices. While the floating point units (FPUs) in Fynbos can be shaped to make use of the DSPs, the architecture does not yet make use of the generous I/O provisions. Regarding memory, in an effort to release as much configurable logic as possible for ALU and control use, all memories are implemented in the BRAM blocks, setting the total on-chip memory size to just over 1MB. Due to the fixed nature of the BRAMs, however, it is seldom possible to use each BRAM in its entirety as memories must be multiples of nine bits wide. Finally, other than the obvious limit in configurable logic, the on-chip communication infrastructure will affect system scaling. While not a factor on the SX95T, in later design extrapolations on larger chips the number of clock regions and supporting logic for crossing such become limiting factors in scaling up the array size.

4.1.2 Architecture Overview

Following other architectures, in order to be scalable Fynbos is constructed in a two-tiered array as shown in Figure 4.2. Basic PEs termed *Strips*, each contain a local data memory, local instruction memory, decoder, and ALU. Grouping Strips into *Tiles* and

[1] This work has no intentional connection to the SKA project, but only uses their hardware due to its availability to the author.

stacking Tiles forms a row and column structured array. As the system is parametrised, the option of matching an optimal array configuration to an application exists. Later discussion will suggest, however, that the costs of such reconfiguration is too high relative to the performance possible on an ASIC. In such a case, the optimal compromise configuration would need to be derived considering the impact of such on a range of HPC applications, this is beyond the scope of this work.

Strips are very simple by design, with no pipelining, branch prediction, or advanced instruction issue, in line with the many-core hypothesis. Saving logic and power in order to instantiate many instances of such, the compiler is instead responsible for the bulk of work regarding instruction issue and data movement. Differing with other many-cores, however, Fynbos has no cache or shared memory and has no form of message passing. In common with some other many-cores, communication is via direct memory to memory, or memory to ALU data paths. Consequently, with no cache and a simple data path, all possible latencies in the system are known enabling optimal static instruction scheduling.

With HPC applications in view, Fynbos must avoid the constraints experienced in using GPGPUs and other SIMD-like architectures, and instead operate in a MIMD fashion. To enable such while maintaining synchronisation, the array operates in lockstep sharing a common instruction address that acts as a program counter (PC). As each Strip is sourcing an instruction from a private local memory, despite sharing a common instruction address all Strips execute a unique instruction, all of which are executed in parallel and synchronisation with each other. This execution of the same instruction address in parallel is referred to in the following as executing a *row*. The reference being the common instruction memory location, or row in each memory module, or the view that the array, while hierarchical, is effectively a long vector of PEs as far as execution control is concerned.

The above use of lockstep operation ensures simplicity of control, but comes at a performance cost in two forms primarily. Firstly, all Strips are required to wait for which ever Strip is performing the longest operation, potentially leaving a majority of Strips idly waiting while a few or even one complete computations. As noted in Chapter 3, division is the most significant example of this with a much longer latency than other operations and as also noted as a means of mitigation APPRASE aims to schedule operations of common latency for concurrent execution. Secondly, lockstep requires the use of universal branching. That is, no form of independent threads exists, rather either the entire array must branch to a new instruction address or not.

This is a consequence of the compiler's heritage in large simulation applications that held little need for such. In a densely branched code, however, a lack of fine-grained dynamic control will be the largest cost of static scheduling. While the majority of loops will have been unrolled this universal branching is the only form of dynamic control currently supported in Fynbos and will limit the degree of parallelism achievable. Considering the architectures reviewed, however, two approaches are suggested to remedy such.

- Static compilation always comes at the cost of lost dynamic opportunities. In VLIW architectures, trace scheduling attempts to remedy the situation by using branch prediction to fill up empty slots and gaining performance when predictions were correct. The branch prediction algorithms are used to determine what sequence of instructions are scheduled for the entire program trace. In the case of incorrect predictions, rather than hardware roll back and clean up additional instructions for such are available in the schedule. For branches that are difficult to predict predication is instead used, where both branches are scheduled to execute with the results of the incorrect one being disabled on evaluation of the condition. Obviously if the branch is very imbalanced or if both sides are really long then it is probably more worth while performing a real branch.

  While compiler support for similar branch prediction and scheduling is complex, an easier approach available to many-cores in general, may be to exploit their characteristic abundance of PEs, and use a form of predication. If both branches are scheduled, regardless of balance or length, greater parallelism with the following code may be released improving performance. Once a branch's result is known specially enabled Strips capable of predicative copy operations (comparison dependant copies) can move the correct branch's calculated values to pre-known locations, ready for continued code execution. While this approach incurs increased code size costs, so does trace scheduling and in this case there is no need for any roll back support. Given fine-grained power control of Strips, however, experimentation would be needed to determine the value of such in a performance-power trade-off.

Returning to Fynbos operation controlling the lockstep execution (and also shown in Figure 4.2) is a central system controller which serves to coordinate three stages of operation: LOAD (instructions and data), RUN (execute application), and OFF_LOAD

(empty data memories). All instruction and data values are pre-placed by the compiler into specific memory locations during LOAD. The RUN stage therefore only requires the controller to broadcast issue to all strips an instruction address (PC) and synchronising control signals. Barring a branch operation imposing a new PC value, execution advances sequentially on receipt of done signals from each Strip. Branching is accomplished by carrying out the same branch operation in two specially enabled Strips, guaranteed to be mutually exclusive in supplying a new PC value to the controller.

On reaching the end of a program or the instruction memory results may be offloaded if required and a new set of instructions (and possibly data) loaded in. A further specially enabled COPY_OUT Strip, also shown in Figure 4.2, provides a further optional means of moving results off-chip during execution, saving on OFF_LOAD time later, and on-chip memory during execution.

In summary therefore, Fynbos is a MIMD array architecture which relies on a fully known data dependency graph and execution latency path in order to perform optimal parallel static scheduling of all operations. The following sections examine the finer details of trade-offs made in the design of the ALU and strip functionality, on-chip interconnect, memory hierarchy, and system operation.

4.2 Control and Data Movement

Given the scalability and minimised control infrastructure design objectives, alternative control and data sharing mechanisms are needed for a many-core.

Scalability requires short paths, through few levels of logic, with minimal fan out, in a replicable form. Minimised control logic, requires that some mechanism other than complex hardware carry out the required work. These factors impacts most strongly on the control model adopted and the design of data sharing infrastructure.

As seen in other architectures reviewed, control may be carried out by software (either as dynamic run-time control or as static compilation and scheduling) when coupled with a simplified control model. Such models include dataflow, streaming, vector, or in Fynbos' case, lockstep operation.

Addressing the need for data sharing other architectures have created various NOCs, shared memories, and buses. Considering the target of HPC a means of all-to-all communication is essential. Point-to-point interconnects that connect a PE to each of its

neighbours offer infinite scalability at the cost of down to single cycle latencies per intermediate PE. All-to-all communication infrastructures and buses offer far lower latencies for communications between far apart PEs, but severely limit scalability and are subject to potential congestion. Application domain specific many-core like architectures are able to adopt middle ground design that take advantage of aspects of their application domain. For instance streaming processors and their exploitation of the streaming nature of their problem sets, or dataflow architectures that exploit the same application characteristic to avoid the need for all-to-all like communication. In the case of Fynbos a more general purpose design is clearly needed. Taking the lead from other architectures a PE and memory hierarchy is adopted that enables cheap fast communication within subgroups of PEs and memories (intra-Tiles), and a higher cost (only if a copy operation is used) for communication with PEs that are more remote (inter-Tile). Fynbos differs from these, however, in the use of simple direct ALU-to-memory, memory-to-ALU connections, along with memory-to-memory communication, without any form of shared memory units or hierarchies. Fynbos also makes use of a form of programmable interconnect which has a precedence, but is able to incorporate and overlay such directly into ALU instructions and does not require dedicated instruction memory or decoding for such.

To elaborate on the purposes of this design approach a Fortran code parsed to run on Fynbos will have been written as a sequential program assuming a shared memory structure. Doing so avoids the programming hazards of a distributed memory structure, but on Fynbos such a code will run on hardware in which no such shared memory exists. Rather, a shared memory is in effect emulated, or the distributed memory is hidden from the programmer, in a manner that also avoids the usual problems of parallel execution with a shared memory structure (race conditions or deadlocks for instance).

4.2.1 Memory Infrastructure

Traditionally the most simple approach to data sharing in a parallel machine is to make use of shared access memories, with either hardware or software control ensuring coherency. While new shared memory models are being developed, such as PGAS discussed earlier, shared memory systems do not scale well as an increase in node numbers causes an exponential increasing in interconnect traffic.

As seen earlier some architectures have attempted to overcome these limitations using multiple small clusters of PEs around small shared memories with even deeper hierarchical memory chains back to a larger shared cache. This comes with an infrastructure and software burden that limits performance and scalability in manners unsuitable to the generic many-core model.

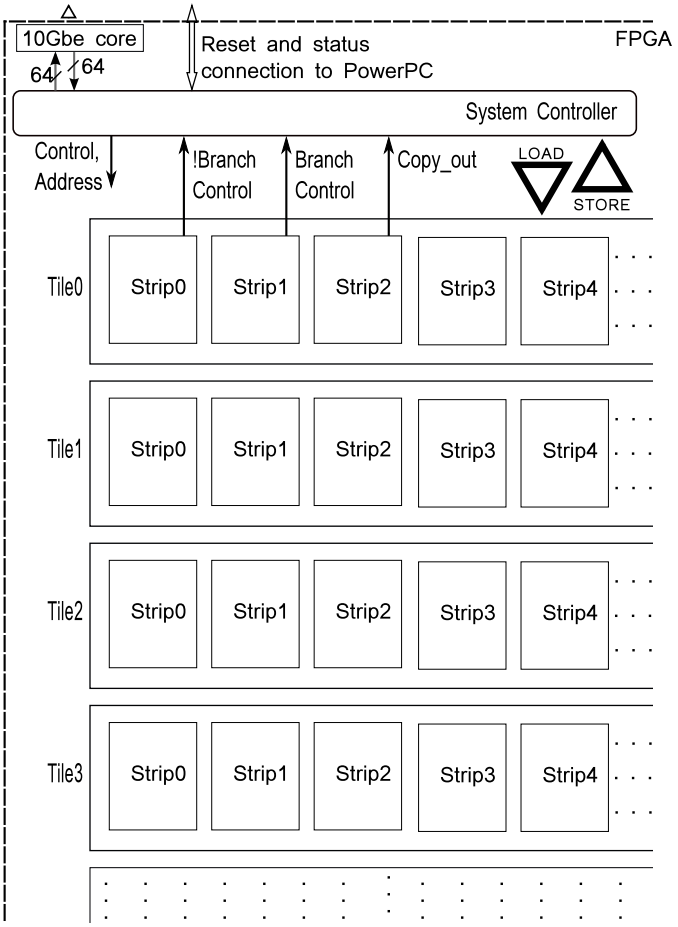


Figure 4.2 – A simple overview representation of Fynbos showing an arbitrarily sized array and symbolic control and I/O infrastructure. The array is depicted so as to show that the number of strips per tile and the total number of tiles is configurable. I/O between the host and array is clearly via a system controller intermediate and limited to control, load, and store lines. Concerning debugging, only the system controller is directly accessible to the PowerPC.

Yet the many-core model shares the challenge traditional vector and streaming processors have, of keeping so many ALUs supplied with data and instructions. Traditionally vector register files, data stream transfers, and a subdivided hierarchical memory structure have supplied the required bandwidth. These techniques make use of the predictable and foreknown memory access patterns associated with the highly data parallel application domains their architectures best serve. Based on such fetch and store operations may be aggregated and pipelined minimising their latency cost. While Fynbos also makes use of data movement predictability it does so in a manner that avoids the high cost of moving smaller quantities of data around that these incur. As vector register files and data streams are fixed in size and shape, they lack the flexibility sometimes required for inter-PE communication. If only one element in a register file or data stream is required, it would be wasteful to reload the

whole unit, generally these architectures avoid such situations through appropriate coding, or reform the units when necessary to contain only required data.

Relative to both of these, the fully distributed memory system and address space of local instruction and data memories in each Strip in Fynbos provides a very flat memory hierarchy. Considering Fynbos' static scheduling control system, however, any cache with non-deterministic access times is both a hindrance and unnecessary, while this flat distributed approach potentially offers greater scalability and fine-grained communication. For Fynbos the limit on scalability will instead either be due to the application's inherent limits (in which case Fynbos is non-causative) or the data and instruction paths from the host's main memory to each Strip's memories. Figure 4.3 shows the instruction and data memories along with the data lines used to access each.

The protocols for accessing such are detailed more fully in a later section but in brief. Control, address, and a data word's width of data lines run from the system controller to every Strip. While of a limited number these lines are clearly the longest in the system and therefore presented as the most likely scalability-limiting factor (although as Chapter 5 will find FPGA resource limits are reached first in the prototype).

When loading the memories a Strip and memory register is addressed using the address lines, triggering the relevant Strip to write the data on the 32 bit wide LOAD lines to the appropriate location. In the case of instructions, as indicated the width is dependant on the array size and memory depths. Therefore provision is made for up to 64 bit wide instructions, which are loaded in two cycles via the same 32 bit data lines.

Given this work's attempt to position one or more Fynbos arrays as a replacement to cluster nodes, however, the following is proposed. That either multiple maximum size Fynbos arrays are created within a chip, or that a larger Fynbos array be divided into multiple sections (perhaps quadrants) of which each is separately loadable but which execute in synchronisation. In the case of the first each array would most likely be loaded with a separate OpenMP like thread via its own I/O interface and system controller. In the case of the second the goal would purely be to divide up the memory loading infrastructure thereby improving scalability and reducing application setup time as multiple load interfaces might carry out the task in parallel. Implementing such a system, however, is beyond the scope of this work and detailed as a future work recommendation.

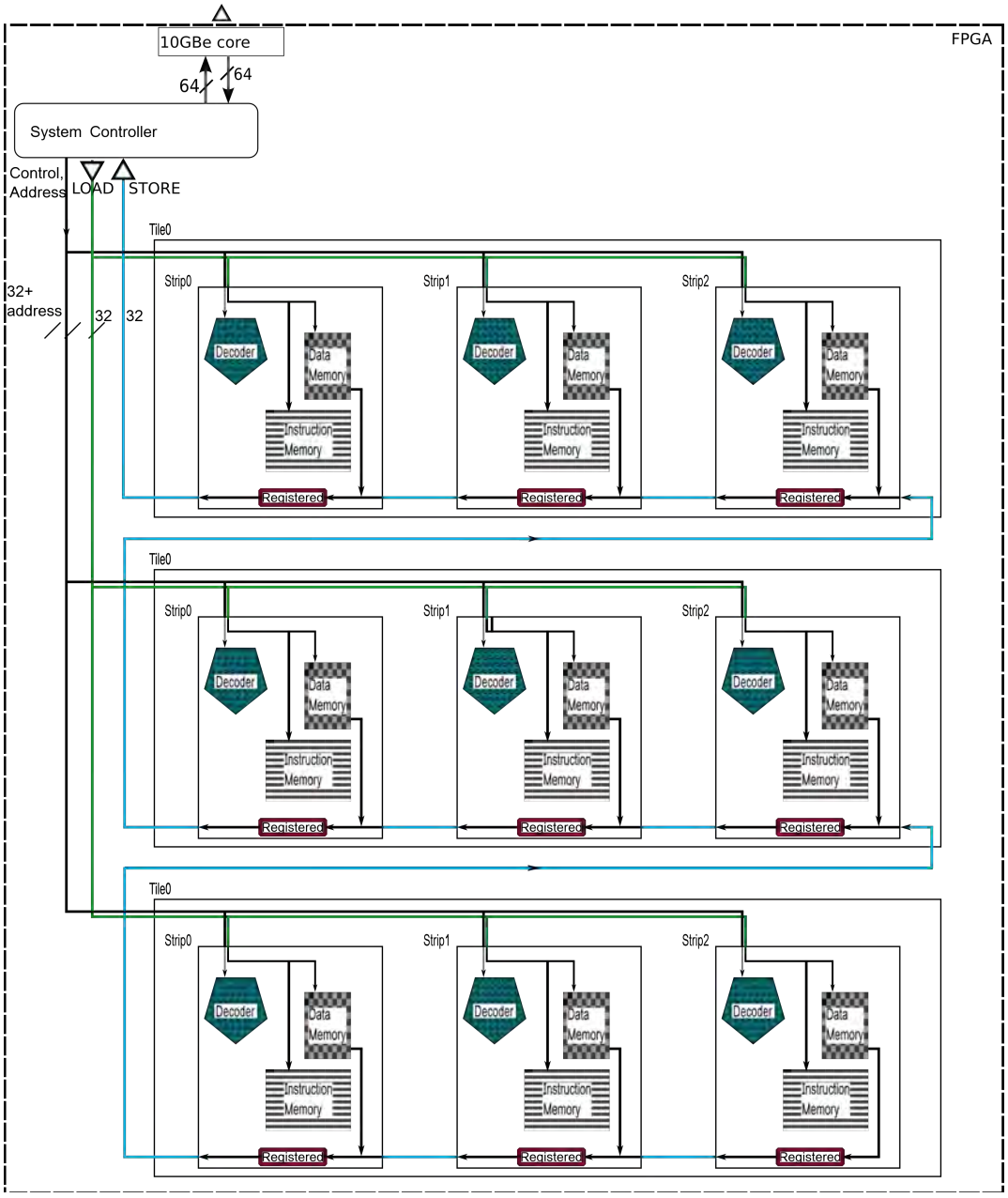


Figure 4.3 – A 3x3 Fynbos array depicting the separate instruction and data memory modules in each strip, along with the infrastructure required for the host to access them in the LOAD and OFF_LOAD operating stages.

The STORE line also shown in Figure 4.3 alternatively does not present the same challenge to scalability. If off-loading of the data memories is required the STORE lines represent a push through pipeline whereby memory addresses are triggered in reverse order from the last Strip and address in the array, and values are passed sequentially through each Strip eventually exiting into the system controller and 10Gbe core via Tile0 Strip0. Given that results may be returned to the host dynamically during execution a more rapid off-loading infrastructure is unnecessary and this slower but more scalable and less resource intensive approach is sufficient.

Practically on the FPGA, BRAMS add further memory related limits to scalability due to their fixed nature which define the size, shape and quantity of memories. Chapter 5 will later show this, rather than the LOAD paths, will be the limit encountered in this work as far as memory is concerned. Nevertheless clearly on larger FPGAs or an ASIC the above limits would eventually be reached without the further many or divided Fynbos approaches proposed.

FPGA BRAMs exist as 36x1024 bit arrays which may be subdivided or joined in width (by factors of nine at the cost of complexity or more simply into 18 bit wide units), and depth (by factors of 1024 or less depending on port usage) to form the memory shape required. This is inherently inefficient where ECC memory is not implemented or non-standard bit widths are used. While the data formats used in Fynbos are either 32 bit or 64 bit (ECC is not implemented), the instruction width is dependant on the array size and memory depths, which both affect the number of addressing bits required. For instance, every factor of two increase in memory depth adds one bit to the address lines running throughout the array, and three bits to the instruction width, as the number of bits required to address data memory is increased by one and the instruction contains three data addresses.

Initial designs for the instruction memory on the Virtex 4, where memory resources were severely limited, made use of all 36 bits available by intertwining instructions across multiple memory addresses, at the cost of encoding and decoding on read and write operations. Later instances did away with this however, as the larger FPGAs became available and the need to be able to fully parametrise the design became more apparent. As such the current implementation sees the tools infer a joining of the minimum number of BRAMs as required to fit one instruction per memory address, and use a 36 bit wide BRAM for the data.

4.2.2 Data Sharing, Array Interconnectivity

Given the above memory structure. As suggested some of the architectures reviewed earlier also approached the data sharing problem without shared memories, determining rather to have a maximum path length of the width of one PE (achieving infinite scalability within resource limits), or by using a form of on-chip network (limited by addressing, a different form of resource limit). In the first case the problem of path length is completely removed but PEs are limited to exchanging data directly with their nearest neighbours only. Communication across greater distances requires multiple hops through the intervening PEs, at a latency cost which may or may not be pipelined away depending on the scheduler or compiler's abilities. Similarly the use of an on-chip network also achieves scalability and a much higher (all-to-all in some cases) degree of interconnectivity, also at the cost of latency, and in some cases the potential for congestion. If programmable, a compiler may be capable of preemptively sending data and thereby masking latencies. Architectures that use networks successfully generally also have single or groups of PEs capable of independent execution, and often use the network as a means of synchronisation greatly enhancing their ability to handle dynamic code. In the case of Fynbos the lockstep operation while a result of other push factors, also ensures synchronisation.

Considering that the majority of applications will not be able to scale infinitely Fynbos does not adopt the minimalistic nearest-neighbour approach, and in seeking simplicity of logic also avoids the NOC approach despite its advantages. Instead, as outlined, the Tiles serve as subgroups of PEs within which all-to-all inter-Strip communication is possible, and between which an all-to-all inter-Tile communication is possible. Figure 4.4 demonstrates a subset of these connections. This middle ground approach also plays a role in keeping the number of global signals to a minimum, as discussed regarding loading the memories. By keeping such to a minimum routing and resource conservation are assisted enabling greater scalability.

Examining Tile2 in Figure 4.4, OutputA_s and OutputB_s from each Strip represent copies of the operands fetched from local memory in that Strip. These are made available to all other Strips within Tile2 as InputA_sx and InputB_sx shown. Internally a given Strip selects operands to operate on from those loaded from its own memory and those made available to it from the other Strip memories.

Examining Tile2, Strip0, OutputC_t is loaded with a copy of the result from Strip0's current ALU operation. This value is made available to every other Strip0 (column

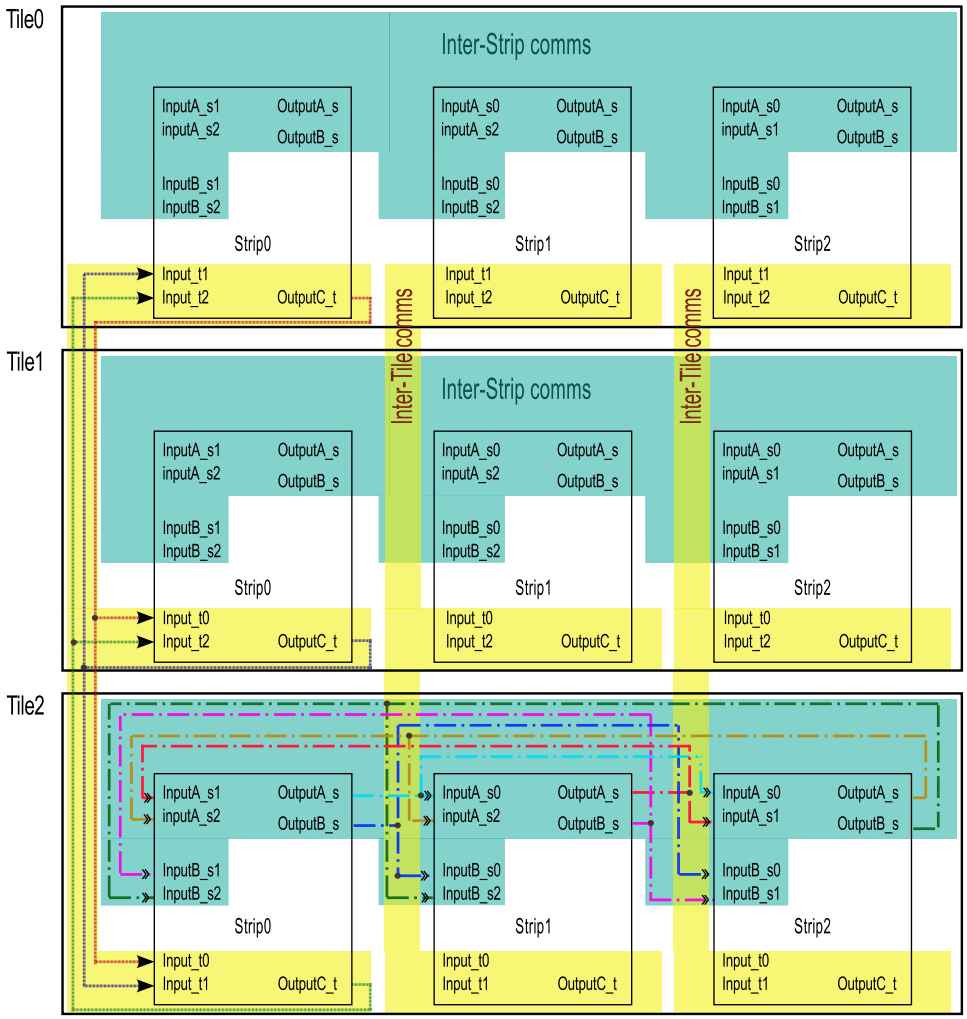


Figure 4.4 – A 3x3 Fynbos array providing a representation of the data sharing interconnect only. Inter-Tile communication lines appear for the first column of strips only, similarly the inter-strip communication lines are shown in Tile2 only. In both cases for clarity the remaining interconnect lines are indicated as shading otherwise. In summary, within a Tile two data words are made available from each Strip to every other Strip within the same Tile, between Tiles one data word is made available to every other Tile's column index corresponding Strip.

Table 4.1 – Table of the bit arrangement of a Fynbos instruction word. All fields except for opcode are subject to change in the case of a different Fynbos configuration. This particular configuration consists of eight Tiles each containing eight Strips and as a result all data word selection fields are three bits wide. Similarly the data address fields are 11 bits wide due to the data memory units being 2048 words deep.

Bits	47-37 (11)	36-34 (3)	33-28 (6)	27-25 (3)	24-22 (3)	21-11 (11)	10-0 (11)
Name	addrq1/addr2	Selq	Opcode	Sel1	Sel0	addr1	add0
Definition	Inter-Tile result storage address	Inter-Tile result MUX selector	Opcode	Operand1 MUX selector	Operand0 MUX selector	Local memory address1	Local memory address0

index0 in the array) in the remaining Tiles, as Strips are saving the result of an operation. Internally a Strip is therefore able to optionally select to store any result from any other Strip of the same column index in any other Tile, as well as optionally additionally selecting to store its own result. In order to save on instruction width, rather than including a further memory address in the instruction the storage location of a Strip's own result is predefined to be the same as the current instruction memory row but in the data memory module (termed the implied address). If, however, a different location is required, a Strip can refrain from storing another Tile's Strip's result and store its own at a local memory location included in the instruction (termed the specified address).

During execution all data movement (all of the above selection and location options) are predetermined by the compiler, and the schedule encoded in the loaded instructions. Execution of an instruction offers three opportunities all of which may occur simultaneously. Data may be passed between strips within a tile, while operand values (not necessarily the same as those being passed) are also registered into ALUs for processing, and finally the result may be non-exclusively passed to another tiles and optionally saved into a local memory.

Table 4.1 shows the instruction encoding for an 8x8 array with 2048 deep data memories. Included are, an opcode, addresses for the two values accessed either as local or intra-Tile operands, selector bit fields for the local operands as well as the address specified result, and finally the address for such (the inter-Tile transferred or local result).

Using Figure 4.5 which provides a more detailed reference for what the internals of a Strip in an 8x8 array might look like to elaborate. On instruction decode, two local data memory addresses fetch two values first into two multiplexers (MUX1 and

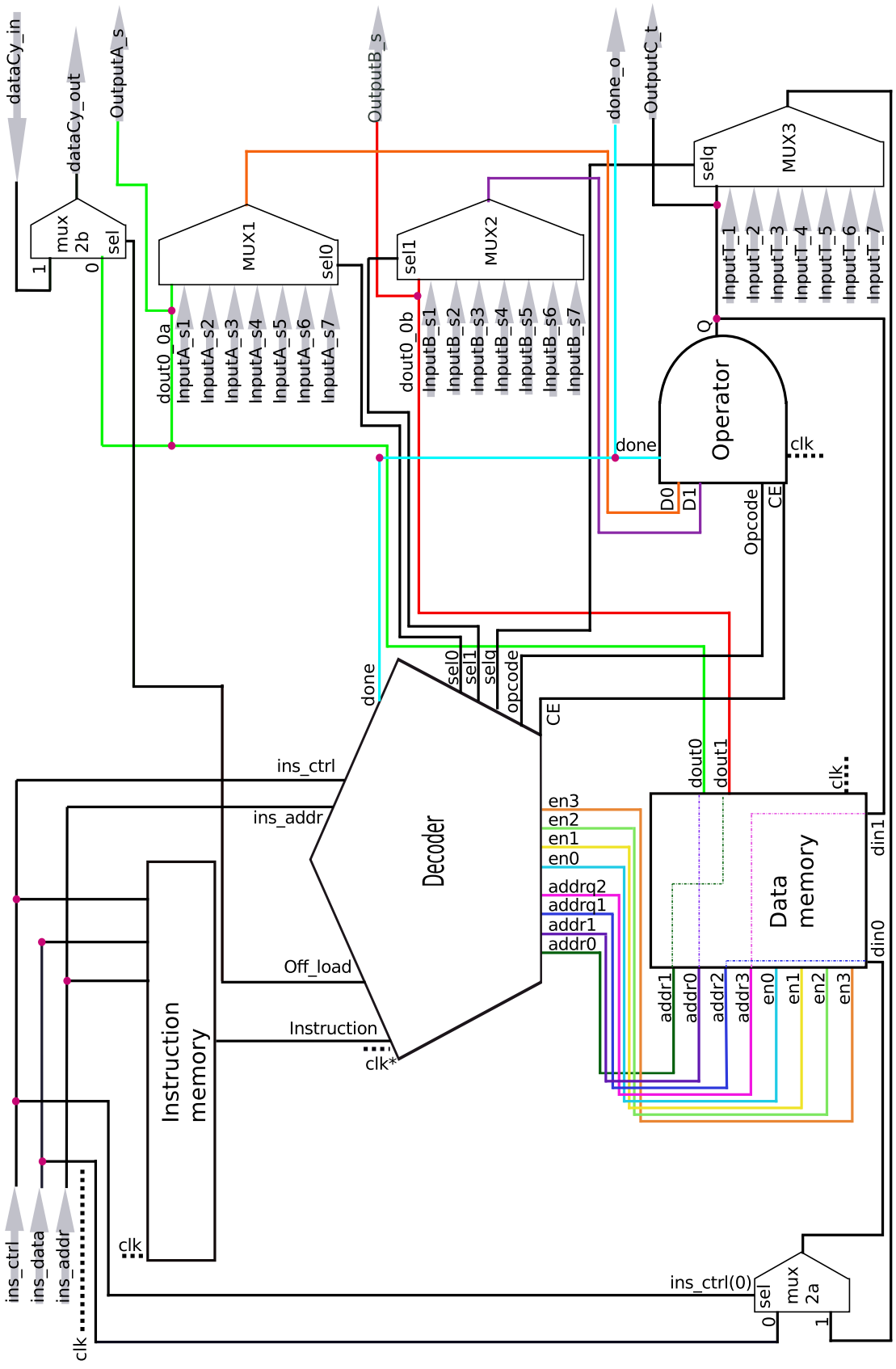


Figure 4.5 – Schematic representation of the contents of Tile0, Strip0 in a 8x8 Fynbos array.

[*] Only the instruction is registered all other inputs, decode logic, and outputs are combinatorial.

MUX2) that also source the corresponding values fetched from each data memory in all other Strips in the local Tile. These multiplexers feed operands into the ALU according to selector signals (Sel0 and Sel1) in the current instruction. As such data is shared directly between PEs without an explicit copy operation being required or passage through memory. With this functionality the scheduler is free to either double purpose a Strip, having it operate on data at the same times as it is fetching data for another Strip, or to use a Strip that would otherwise have been idle to fetch and move data.

In an analogous manner, a Strip's local result (Q) is made available to its local memory, and to multiplexers (MUX3) in all column index corresponding Strips in the other Tiles. During the STORE stage of execution in each Strip two values may be stored. The first is selected from the inter-Tile multiplexers (MUX3) additionally sourcing the results generated in other Tiles and ultimately entering the local data memory via din0. The second is always the local result which is optionally written to the data memory via din1 (under control of the combinatorial decoder). The scheduler may therefore again optionally double purpose a Strip to both import another data value or operation result for future use, while still storing a locally generated result.

Both interconnects therefore perform an all-to-all broadcast within the same row or column respectively. Over the course of an operation up to two values may be imported, and three exported. A value may therefore move from corner to corner of an array in the space of two operations. Again this system is an attempt to find a balance between scalability and freedom of data sharing. The use of a distributed memory address space, and lack of an all-to-all interconnect, ensures greater scalability. To match this the interconnect described above provides greater than nearest-neighbour only freedom (the infinitely scalable alternative), and logically resource cheap sharing. This is important as the more readily available the operands are the greater freedom the scheduler has for finding parallelism.

While direct ALU-ALU data movement is not possible, such as some of the nearest neighbour architectures are able to offer, only one pass through a memory unit is required when moving data between PEs, either as a fetch from memory or as a store of a result. This is in contrast to more traditional designs that will see a word stored in local memory, fetched, passed to, and written to another PEs memory, before being fetched again from such for use. Fynbos could support a direct ALU-ALU mode, both between Strips and internal to a Strip. Given the lockstep operation, however, the

compiler would need to be adapted to take advantage and cognisance of such, parallelising ALU-ALU operations as and advantage gained through such would be lost if any one Strip is not executing in the same manner. In effect the register pipeline between Strips would also be reduced by such most likely additionally forcing a drop in clock speed.

4.3 A Many-core PE

Part of the purpose of this work is to examine the architecture of a single many-core PE. Fynbos Strips are notable primarily for their simplicity, three control bits, and shared address lines direct the memories and ALU through all states. As already seen ultimately the control model selected determines much of the contents of a PE, as it in many ways determines the possible data path, control path, interconnect, and memory infrastructure designs. The only remaining significant aspect for consideration is that of the practical numerics.

4.3.1 ALU Design

Previous work reviewed showed architectures that adopted an even finer granularity of operation than Fynbos, where by ALUs are constructed from or selected out of optional bit width ALUs as the application needs. Considering the target domain in this work is general HPC, however, floating point (FP) support is a mandatory requirement. Further double precision is needed in the vast majority of applications.

While fixed point or other numerical representations simply lack the range required by HPC, double precision operation can be implemented directly in hardware (as traditional CPU cores do), or by joining single precision hardware units (as GPGPUs do), or finally in software using numerical scaling and error correction[Baboulin et al., 2009; Dekker, 1971; Linnainmaa, 1981]. This final option is far more appropriate to the many-core hypothesis. Single precision hardware in any case often operates twice as fast as its double precision equivalent, and reducing hardware complexity via software sophistication follows directly in the many-core principle. Further such an approach will advantageously exploit the available parallelism of a many-core. While the APPRASE compiler used here does not yet include the sophistication to perform such operations, it is for these reasons that the Fynbos hardware contains only single precision floating point support.

Similarly, due to the lock step operation and a lack of compiler capability to take advantage of such, integer operations are done in floating point instead of increasing logic costs by adding a dedicated integer pipeline. In this case software checking is needed to ensure such a conversion will still result in the same functionality. On a substrate with greater resources, however, this may not be the correct decision as an integer pipeline serves to provide specific functionality and, given the compiler functionality, could be exploited to valuable performance gains.

Table 4.2 lists the operations supported in Fynbos, along with notes on their use locations and conditions. In general the standard operators (+, -, *, and logical comparators) are implemented using Xilinx's FP core v5.0, configured for single precision, and available in all Strips. These cores are IEEE 754 [IEEE, 1985] compliant with two caveats, they do not support denormals, and only implement round-to-nearest rounding.

Alternative floating point cores are available, such as FloPoCo [de Dinechin and Pasca, 2011], or the range available at opencores.org. Unlike all of these, however, the Xilinx cores are free, complete in the operator support, well integrated into the tool suite, and fully customisable in terms of both latency-logic trade offs and the mantissa and exponent widths. For the purposes of a prototype they were therefore optimal, further work could be done evaluating and comparing the other FP cores available.

As noted in the description of APPRASE (Chapter 3), on average division and square root (SQRT) are used far less frequently than the other standard operators [Kwon and Draper, 2009]. Considering this lower frequency the silicon resources these require (each more than doubles the logic resource consumption of a ALU), and their disparately long latencies, division (DIV) is only optionally enabled in some Strips and square root (SQRT) is implemented instead in software. Fundamentally sequential and dependant on division, Heron's method or any other SQRT algorithm, cannot be extensively parallelised but also as was discussed it serves as a test case for the proposed future work of similarly overloading transcendental and other complex functions with parallel software routines.

In the case of division,[1] APPRASE is made aware of which Strips are division capable Strips (DCSs), and as indicated earlier, will take steps to optimise performance by limiting the number of rows containing division to as few as possible.

[1] see Appendix A.1.3.2 for further details on Xilinx's implementation of such.

Table 4.2 – Table listing the operation available in Strips and their limitations or implementations.

operator	Opcode	Notes on use case and availability
+	000000	Universal
-	000001	Universal
<	001100	Universal
==	010100	Universal
<=	011100	Universal
>	100100	Universal
!=	101100	Universal
>=	110100	Universal
Copy	000101	Universal
Copy_out	001001	Only available in Tile0 Strip2
Maximum	000110	Universal
Minimum	000111	Universal
No Operation	000010	No operation is performed in the ALU but data will be fetched for the benefit of other local Strips. The implied address will not be written to. The specified address may be written to (unless prevented by it being the same as the implied address in which case nothing will be written, making it a true no operation)
*	000011	Universal
/	000100	Only available in specifically enabled Strips (configurable)
Branch	001000	Only available in Tile0 Strip0 (branch on false) and Strip1 (branch on true). The implied address is not written to. The specified address may be written to (unless prevented by it being the same as the implied address in which case nothing will be written)
Absolute	001010	Implemented as copy sign
True No operation	- - - - - -	This does not have an opcode but rather is a special mode engaged when a NOP is instructed and the specified and implied result addresses are the same. In any operation this condition signals that no value should be imported and written to the specified address. In the case when a NOP is being executed this means nothing is operated on and nothing is written to local memory. This mode is necessary for Strips taking no part in a row execution at all, to prevent unwanted values being written to random memory locations.

While Table 4.2 briefly discusses the special case operators (to be elaborated on further in the next section), a notable lack in Fynbos involves NOP support. One of the proposed tenets for many-cores is fine-grained power control. While Fynbos ALUs are clock gated, beyond this there is no mechanism to allow for complete Strips or even Tiles to be powered off when unneeded. There are a number of ways in which such could be implemented, from compiler created instructions initiating a sleep mode state through to the system controller powering down specific Strips. The first offers simplicity of hardware but will mean an incomplete power down as Strips watch for a signal to reawaken. The second suggests the need for further undesirable global signals, but perhaps might be implemented alongside the earlier proposed array partitioning where reducing the scope of global LOAD lines is already being sought. Such would instead see large swaths of a system being powered off and on at the behest of a local controller itself directed to do so by the compiler.

4.4 Operation

Figures 4.2 and 4.3 earlier showed a system controller module interfacing between the 10Gbe core and the array. Along with administrating the loading and offloading of memories, this module controls the 10Gbe core and orchestrates program execution. Part of performing the later includes managing branches, raised exception flags, and program termination, as well as directing results off loaded during execution to the 10Gbe transmit buffers.

The basic flow of operation was detailed in Chapter 3's description of the load file, (load the memories, execute, and return results if necessary). To manage both the array and the 10Gbe core through these stages, the controller module jointly operates two FSMs. The following outlines these FSMs and the additional special case handling hardware included in the array.

4.4.1 Execution Flow

Figure 4.6 shows the FSM responsible for operating the array. State (CTRL) transitions are dependant on the valid state of data arriving through the 10Gbe core, exception flags from the array, and internal values. Figure 4.7 shows the second FSM which manages the TX port of the 10GBe core. State (TXRX) transitions are on CTRL, exception flags from the array, and internal values. These transition dependencies are a

consequence of the need to use minimal states, arising from the goal of minimal logic use.

As the operating flow has already been described, the figures are predominantly self explanatory along with the following expository notes regarding the first (Figure 4.6).

- The command to be decoded, as referenced in CTRL=CMD, refers to the host commands that determines the mode Fynbos enters. Full details of each command can be found in Appendix A A.1.4, in brief they can be summarised as containing the following[1].
  - Load Data: A starting Tile-Strip-Register address, and the number of words to be loaded. There is no need to address each strip with a separate command (although this is possible), as the controller is aware of the depth of each memory and therefore able to move on to the next strip once a prior one is full. If necessary, however, each memory register can be addressed directly. Such access comes at the minimal administrative cost of command fetching and decoding (~10ms).
  - Load Instructions: The same as Load Data but for the instruction memories which may be of a different depth to the data memories.
  - Run: An instruction register address at which to begin execution, and an instruction register address at which to halt.
  - Off Load Results: A starting Tile-Strip-Register address from which off loading will count down to zero, and the register address within each strip to begin off loading from. With no tri-state buffers internally on the FPGA, the load data path cannot be reused for off loading. As detailed earlier, a pipeline infrastructure through each strip is used. From a given starting address each register in each strip, counting down to 0-0-0, is instructed to read out its data. These data words are then pushed from one strip to the next eventually arriving at Tile0 Strip0 from where it is pushed into the TX buffer. The initial latency incurred by this has a maximum cycle count equal to the total number of Tiles plus the number of Strips.[2] Once full,

[1] The design description is parametrised such that the exact length of these commands can change as needs depending on the shape and configuration of the Fynbos array targeted.

[2] A side effect of this is an equivalent number to the latency, of zero values, will also be transmitted back to the host before the pipeline is full. It was concluded that it was cheaper to remove such in software on the host than in logic in Fynbos.

however the pipeline streams one data word per cycle into the TX buffer. The second register address given in the command enables more rapid off loading of only a lower portion of each data memory if desirable.

- While instructions are retrieved from the 10Gbe RX buffer in 64b wide words, the data path used to load both data and instruction words into all strips, is only 32b wide (the size of data words). As alluded to elsewhere, the width of instructions is dependant on the shape and configuration of a given Fynbos array. As described, in order to handle instruction words wider than 32b (generally the case), instructions are loaded in two stages (states LI1 and LI2), as an upper and lower word.

- The details of raising an exception, branching, copying results out of the array while execution is taking place, and program termination, which are indicated here, are given in the section following.

4.4.2 Exceptions and Events

Due to the simplicity of each strip, the system controller becomes responsible for handling all exceptions to the normal lockstep statically scheduled events. Within the limited scope of Fynbos such events include: branching, program termination, the occurrence of a floating point overflow or underflow, and while not strictly within the vein, transmission of results to the host during execution.

4.4.2.1 Moving Results Off-chip

Beginning with the last of these. On its current platform Fynbos is a co-processor that is required to transfer data and results to and from a host. It therefore supports a mechanism for returning results to the host for storage during program execution. This both parallelises the return of results with their generation, and conserves on-chip data memory. Due to the serial 10Gbe output port, only one Strip (Tile0 Strip0) is currently capable of copying data out of the array, but on a different platform there is no reason more exit interfaces could not be added with more strips so enabled.

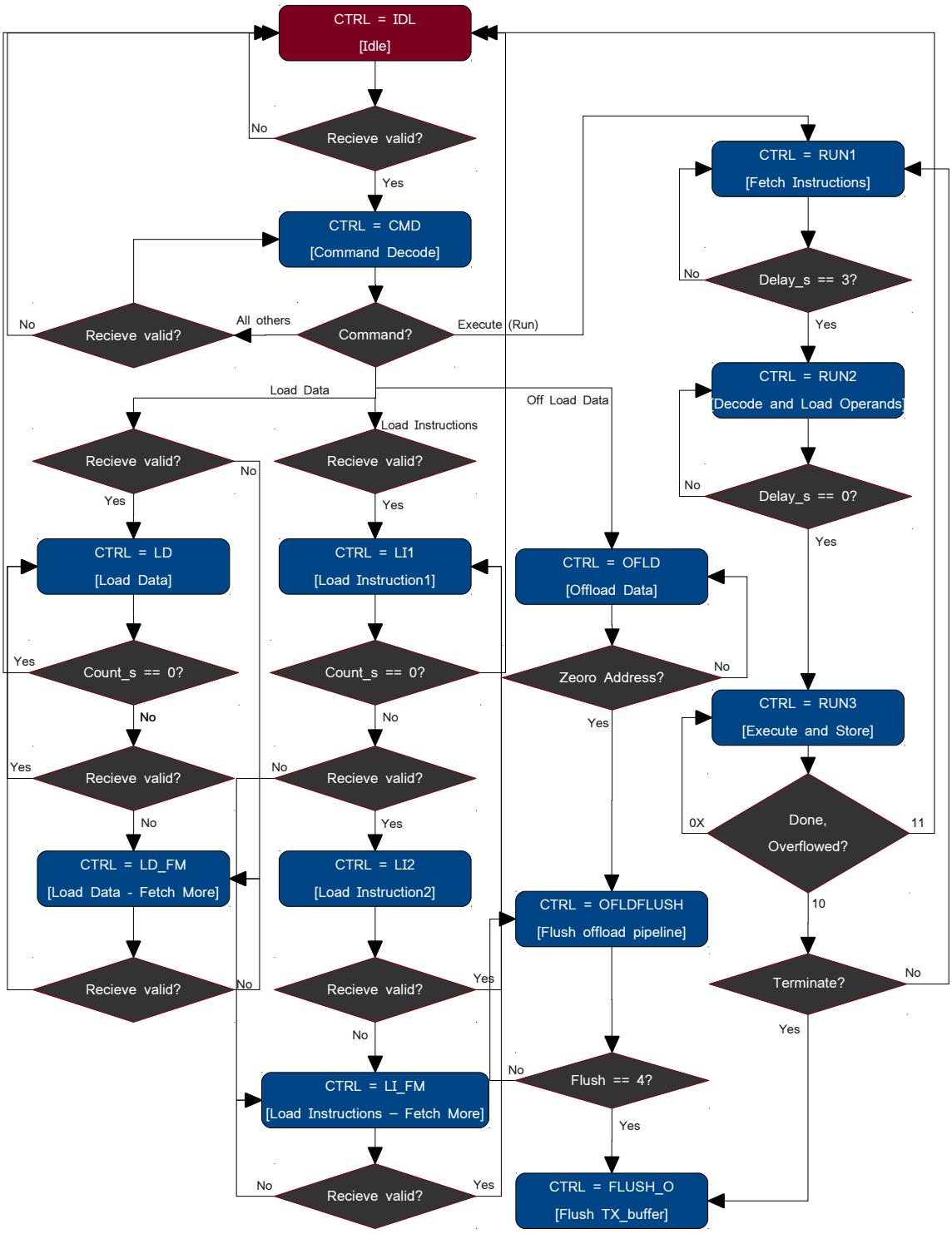


Figure 4.6 – State diagram of the array control FSM in the Fynbos system controller. State=CTRL. On reset CTRL=IDL. Remaining states include; CMD (command decode), LD (Load Data), LD_FM (fetch more data to load), LI1 (Load upper word of an instruction), LI2 (load lower word of an instruction), LI_FM (fetch more instructions to load from the host), RUN1 (Issue an instruction address to the array, RUN2 (Decode fetched instructions and fetch operands addressed), RUN3 (Execute the operation on fetched operands and store results), OFLD (Initiate an off-load process retrieving data from array data memories for return to the host), OFLDFLUSH (Flush the off-load pipeline), FLUSH_O (Flush the 10Gbe transmit buffer).

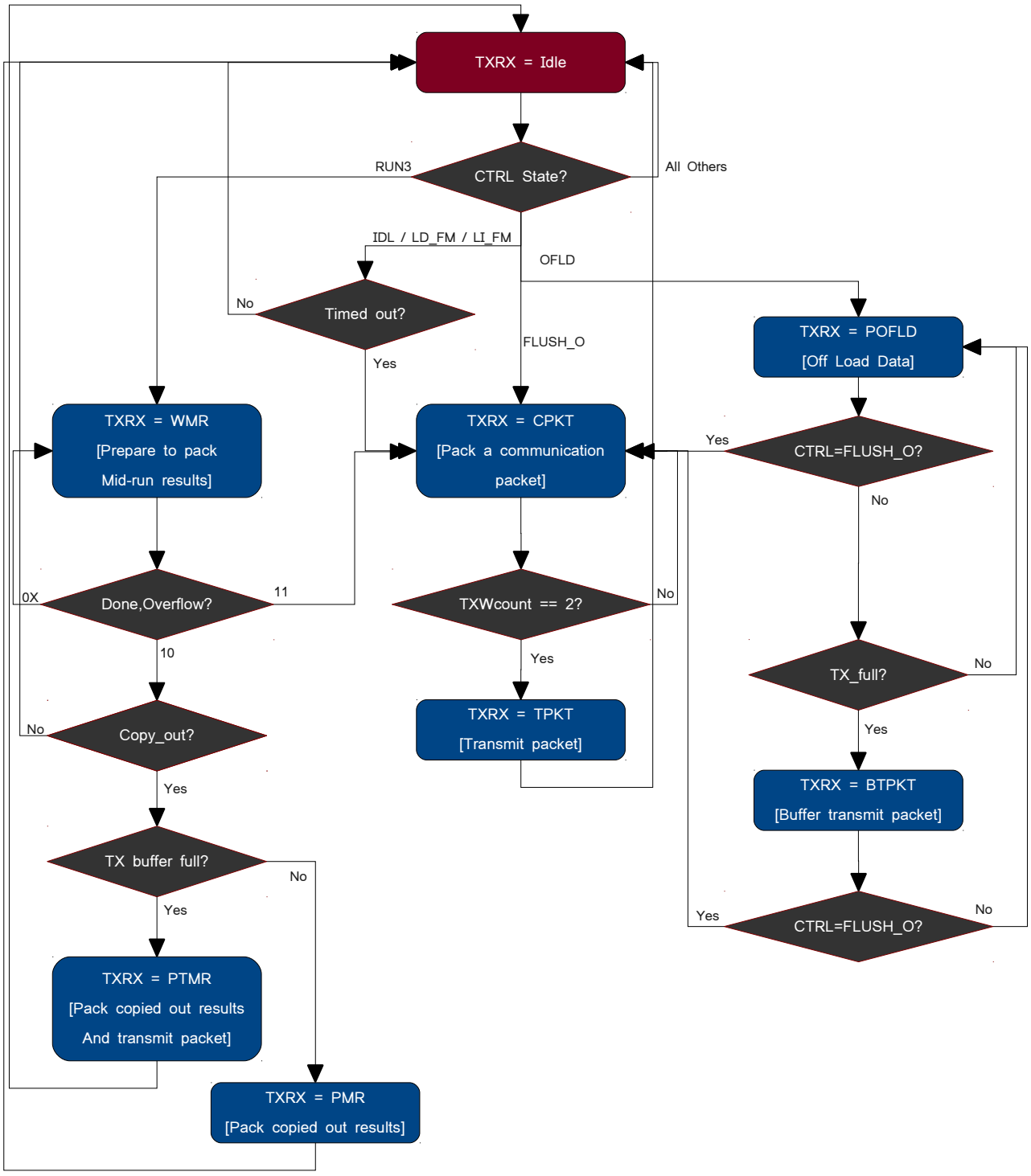


Figure 4.7 – State diagram of the 10Gbe control FSM in the Fynbos system controller. State=TXRX. On reset TXRX=Idle. Remaining states include: CPKT (Pack communication packets for transmission to the host. Communications include: requesting more data or instructions, raising an exception, delivering results created during execution, delivering off-load data, or indicating a flush of the off-load pipeline), TPKT (Transmit contents of TX buffer), WMR (Sit in a state of readiness to receive results copied out of the array during program execution), PMR (Push result into TX buffer), PTMR (Transmit a full packet of copied out results), POFLD (Push off-load data from the array into the TX buffer), BTPKT (Transmit TX buffer full of off-load data).

4.4.2.2 Floating Point Exceptions

The IEEE 754 floating-point standard includes five exceptions, of these the Xilinx floating point cores support signalling of four: invalid operation, division by zero, overflow, and underflow. Inexact rounding signalling is not supported in hardware although notification of such is given when converting input data to hex FP values during load file creation. For simplicity the core implementations used in Fynbos include only over- and under- flow signalling, although the others could be easily included.

For now signalling involves raising a bit all the way in the system controller. Collecting a signal bit from every Strip requires long signal lines that could become the critical path (the logical path that sets the minimum possible clock period) with greater scaling. A slower but more appropriate mechanism in such a future case could involve writing to designate memory registers for later explicit checking. As Fynbos stands, however, if an exception is signalled, the controller immediately aborts execution and transmits a communication packet indicating such.

4.4.2.3 Branching

In line with the static schedule and lockstep operation, a branch in Fynbos involves branching the whole array. To cause such a branch condition is first evaluated and the result passed on to both of two branch enabled Strips (Tile0 Strips0 and 1). In the following operation both will evaluate the branch result by carrying out a compare operation. With one looking for a false result and the other a true, one of them in mutual exclusivity will submit a new PC value (instruction address) to the system controller.

Fynbos uses non-relative branching as it removes the need for the required additional mathematical hardware, and because there is no need for it. While PC-relative addressing is more common the reasons for such (nearby addressing means less than a whole address's worth of bits are needed, and a program can be placed anywhere in memory) are irrelevant in the context of Fynbos and static scheduling.

4.4.2.4 Program termination

A program has reached its end when the PC points to the instruction address supplied as the terminate address in the RUN instruction, usually arrived at via a branch. There is no graceful means of interrupting a program while running, other than to reset the FPGA which looses all data.

4.5 Verification

Without industry standard verification software full code and state coverage verification was not possible. As an alternative Xilinx's Isim HDL simulator and custom in-hardware test routines were instead used to verify the functionality and hardware implementation. Isim enabled modular functionality test during development, and ultimately served in debugging the complete system in conjunction with python scripts that generated appropriate test bench input stimuli. In-hardware tests included:

- Generated in hardware test;
  - Correct functioning of all operators
  - Correct behaviour of all inter-Strip connections
  - Correct exception raising and handling
  - Correct and complete off loading of results using both mechanisms
  - Correct branching
  - Correct memory operation of all addresses
- Generated 10Gbe test:
  - Appropriate Fynbos recovery from lost or incomplete UDP packets
  - Appropriate host responses to received requests and data
  - Correct transmission and capture of results

4.5.1 Design discards

It is perhaps worth noting here some of the design decisions not described above due to their targets being ultimately discarded. But for reference Table 4.3 contains some of the more significant ones.

4.6 Summary and Conclusions

Considering this work's hypothesis the following initial conclusions and design summary are notable.

1. The Fynbos Strips are representative of an appropriate HPC many-core PE given;

Table 4.3 – Table cataloguing a selection of the design avenues pursued but later discarded for various reasons.

Target	Goal or purpose	Reason for discard
Inter-weaved instruction memory	Given that the ECC bits are available in BRAM by default it would have been most efficient to pack the data and instructions into the BRAMS in a manner that used these. This was possible through an encoding in the load command that meant an instruction was split over multiple BRAM rows according to a set pattern relating to its time of arrival.	It became clear that the length of an instruction would need to be parameterisable which in turn meant the encoding would also need to be parameterised which was not possible
Alternative interfaces	As discussed over the course of development this work targeted multiple FPGAs each within its own development environment. Apart from moving the target this meant first implementing a system controller capable of interfacing with the Nallatech GUI environment blocks, and second investigating DMA transfers through RAM modules on a dev-kit. Before creating the 10Gbe based controller used here.	Change in the target FPGA.
Single FSM interface	The first system controller and host interface to use the 10GBe core was created prior to the addition of mid-execution off-load of results. When this functionality was added to APPRASE the single FSM became to clumsy to manage both the 10GBe core and the Fynbos array. In response the dual FMS version described above was implemented.	Changes to APPRASE
Equal memory depths	Originally the APPRASE design anticipated that the instruction and data memories would be the same depth and exploited this fact in how it stored data. Later when APPRASE was given greater freedom and was not confined to this approach the memory interface and decoder had to be redesigned and parameterised.	Changes to APPRASE.
Static Array	Given that the first FPGA targeted was very small it was clear at the time that a very limited number of Strips would fit within it. The interconnect, instruction memory, and decoder were created assuming a static largest array size. The interfaces and functionality of each were therefore created to react appropriately to default zeros for cases when a smaller array than the maximum was needed. When larger FPGAs became available all of these interfaces required redesign for full parameterisation and the functionality of such.	Change in the target FPGA.
Fixed precisions ALUs	The APPRASE developer also has a fixed-point scaling algorithm that allows fixed-point arithmetic to be used as an alternative to floating point. In brief the approach works by creating a scaling factor on all inputs and tracking the operations carried out on each variable through a sophisticated bookkeeping scheme. With this approach in mind a fully parameterised fixed-point ALU was developed by hand.	It became apparent that the fixed-point scaling was not going to be incorporated into APPRASE in the immediate future.
Optimal tool flow settings	Prior to Vivado being released the best configuration of ISE tool suit flags was determined. This configuration set produced the shortest critical path for the maximum size array. In addition to the synthesis, map, place and route tool settings, the optimal latency, DSP use configuration, and rate settings had also been catalogued as regarded the FPU modules.	Discarded when Vivado was released and the potential to synthesis for a V7 made available.

(a) The target application domain of HPC and its breadth in application characteristics, supports the need for IEEE754 standard floating point infrastructure.

(b) The many-core ethos of reducing hardware complexity, and the hardware and performance trade-offs involved support the intended use of single precision hardware and with software emulated double precision.

(c) The lower frequency of division and SQRT operations, the hardware costs of such and other complex operators such as transcendentals, the abundance of parallelism native to a many-core, and APPRASE capacity to optimise for such, support the limitations on availability of division, software implementation of SQRT, and proposed parallel software implementations of other complex operators.

(d) The use of distributed memories, and decentralised control (enabled by static scheduling end the tiled architecture), which support the many-core scalability requirements and push to move complexity out of hardware and into software.

(e) The Strip offers a simple arithmetic pipeline capable of independent instruction execution and memory access.

2. Fynbos Strips are deficient as many-core PEs for HPC in the following areas.

(a) Strips lack a integer pipelines, due primarily to the limited resources available on the FPGA and need for APPRASE optimisation for such.

(b) The hardware and software control required for fine-grained power gating on individual Strips is not present in the current design.

3. The Fynbos array is sufficiently and appropriately scalable as a prototype many-core architecture given;

(a) The middle ground approach adopted on the inter-connect provides a logic efficient means of effective all-to-all communication while enabling the compiler with a means of finding and exploiting the most efficient communication schedule circumventing the traditional problems of congestion and extensive costly connection lines. It further offers, in conjunction with the scheduler, a means of masking all data movement latencies behind useful operations through the multi-purpose capabilities of a Strip during instruction execution.

(b) The use of distributed memory. While in this prototype the maximum memory capacities should register concern (18MB in total in BRAM), a larger FPGA or ASIC would not be subject to such harsh constraints.

4. The interconnect on Fynbos is capable of supporting the communication needs of a non-streaming[1] HPC application given and provided;

   (a) The simple and direct ALU-to-memory, memory-to-ALU and memory-to-memory communication options.

   (b) Point 3(a) above.

   (c) And provided additional I/O interfaces are made available in a real world implementation.

5. The Fynbos system architecture is deficient in the following areas and should be remedied as described.

   (a) For greater scaling the proposed sub-divisioning of a Fynbos array or clustering of multiple maximum size arrays, is needed to shorten and minimise the number of global signals used and improve I/O rates with multiple I/O and control points. APPRASE would also obviously have to be adapted to such.

   (b) As a tenet of improving architecture efficiencies, in addition to individual Strip power gating, frequency scaling on an entire array is needed in the context of a multi-Fynbos or sub-divided Fynbos environment. This is not realised here due to the limitations of the available hardware environment, and while it could have been designed for given the conditions it was considered beyond the scope of a prototype.

   (c) While Fynbos exploits static scheduling to gain simplistic hardware, static scheduling's inherent poor support for dynamic code is a non-trivial problem in targeting HPC. To truly meet the needs of HPC a means of achieving performance in the face of dynamic code is needed. For such in Fynbos or any many-core, a form of predication is proposed that exploits the abundance of parallel PEs available.

[1] The Fynbos architecture described would not suit a streaming application, but aside from GPGPUs neither would most HPC targeted architectures as it is not a common application framework in HPC at this level of application parallelism.

6. While sharing aspects of other architectures, as far as can be determined Fynbos represents a novel architectural contribution to the domain of many-core architecture research given;

   (a) While sub-groups of PEs is a known technique, the interconnect between and within Tiles in Fynbos is unique.

   (b) While distributed memories is becoming a recognised approach to many-core architectures, the combination and implementation of such with static scheduling is unique.

   (c) While heterogeneous systems exist, the use of such simply to save resource costs of division based on its frequency has been proposed elsewhere it has not been utilised in a many-core architecture.

   (d) While the use of software implemented operators is not in itself unique, the use of such in a many-core is.

   (e) While programmable interconnects exist that are capable of overlaying data movement over computation, Fynbos manages to do so while incorporating the instructions for such into the same control path as the computation.

Chapter 5

APPRASE-Fynbos Evaluation

5.1 Introduction

Chapter 3 reviewed how the APPRASE compiler pipeline approaches the needs of a many-core programming model as well as HPC applications and Chapter 4 detailed how Fynbos attempted to practically implement the ideals of the many-core architectural model. It remains, however, to determine the practical result of joining the two, and to verify any earlier suppositions with numerical evidence. To do so the following chapter examines firstly how well Fynbos scales towards the hypothesised thousands of PEs on an FPGA or hypothetical ASIC substrate, and secondly how corresponding compute performance and energy efficiency values compare to a sequential x86 based architecture.

5.2 Fynbos Hardware Evaluation

Quantifying Fynbos' hardware scalability means examining its scaling limits and the reasons behind such, but also its scaling in logic and power resource use. Each of these components can further be examined on both a FPGA substrate and a hypothetical equivalent ASIC implementation. Both platforms will contribute independent conclusions, in addition to contributing to a third discussion around whether Fynbos and many-core chips should be created as reconfigurable or static platforms. In both realisable hardware scalability and efficiency are explored considering, the logic use, critical path delays, and power consumption of Fynbos in varying configurations.

Fynbos was designed to be reconfigurable in terms of how, the array hierarchy is structured, memory was allocated, and the availability of division. All of these vari-

ables will affect the metrics under consideration in manners that should scale according to a trend and thereby allow for defensible predictions to be made regarding configurations not realisable on the currently available platforms.

To allow this the following first compares implementations of Fynbos on the ROACH XC5VSX95T Virtex5 (tested in hardware) and a much larger FPGA the XC7V2000T Virtex7 (tested in software only). From here on these are referred to as the V5 and V7 respectively. From these implementations hypothetical ASIC parameter estimates can also be derived using the scaling values reported in [Kuon and Rose, 2007]. Before doing any of the above, however, the methodologies employed in tool chain configuration and ASIC scaling are presented.

5.2.1 Tool Chain Configuration and ASIC Scaling

Comparing an RTL design on different substrates involves many complicating tool chain, hardware, and implementation parameters. The following attempts to summarise the difference involved and the actions taken to mitigate, compensate, or account for such.

5.2.1.1 FPGAs

Comparing a RTL design on two different FPGAs is complicated by the differences inherent to each model. As part of distinctly different generations, the V5 and V7 differ in more ways than raw logic cell count (see table 5.1 for specification differences). Further in generating a loadable FPGA binary there are an enormous number of possible parameter setting combinations in the generating tool suites. From RTL code compiler pragmas and coding style that adapt a RTL description to a particular target FPGA, through to the array of tool options given at each software stage of synthesis, mapping, placement, and routing. Finally the reconfigurable design of Fynbos obviously adds further to this range. The following therefore lists the measures taken in an attempt to standardise the comparisons.

- Beginning with the tool chains, Xilinx recently completely re-engineered their software tool chain, beginning a process of deprecating the older ISE suite, and replacing it with Vivado. Unfortunately, the earlier ISE based tools do not support Virtex 7 devices and Vivado equally does not support Virtex 5 devices. As a result, all V5 values reported are generated in ISE 13.4 and all V7 results were created using Vivado 2013.2.

Table 5.1 – Specification differences between the V5[Xilinx, 2011c] and V7[Xilinx, 2012]. Both use 6-input LUTs and CLBs containing two slices although the contents of such slices differ as shown.

Specification	XC5VSX95T (V5)	XC7V2000T (V7)
Fabrication process feature size	65nm	28nm
Logic cells	94,208	1,954,560
Slices	14,720	305,400
Flip-Flops per slice / Total on chip	4 / 58,880	8 / 2,443,200
LUTs per slice / Total on chip	4 / 58,880	4 / 1,221,600
BRAMs (36Kb) / Total on chip KB	244 / 1,098	1,292 / 5,814
DSP48E slices	640	2,160
Area	35 x 35 mm	45 x 45 mm
Speed Grade Used*	-1	-2L

[*] The speed grade of the target FPGA will affect maximum critical path delay and therefore maximum operating frequency. Given that the design has been tailored to fit within the ROACH V5 (-1 speed grade device), and that many timing relevant architectural changes would be made in targeting a different device, there is little value in drawing conclusions on maximum possible frequency of Fynbos on the V7. For the purposes of simply pushing the design as is, as far as possible, the V7 designs are therefore compiled targeting a -2L speed-grade (maximum performance and support for operating at a lower voltage for lower power, although the latter feature is unused here).

– Apart from differences in underlying compilation algorithms this discrepancy in tool version introduces difficulties in specifying the many compile stage parameters that influence fit and performance as some settings differ or are completely absent from one or other tool. To standardise as far as possible it was found that rather than attempting to approximate near equivalent settings it was best to generally use the tool defaults. Table 5.2 records the few instances where this is not done and where a setting was only available in one tool chain, the default for such was also chosen.

– A further consequence of this tool divergence appears in the use of IP cores for the floating-point arithmetic. It was found that common parameters could be used in both, but the cores are generated with different Xilinx Floating-point IP versions, 5.0 and 7.0 respectively. Table 5.3 gives the common parameters selected and the reasons for deviating from the produc-

	ISE 13.4 V5	Vivado 2013.2 V7
Synthesis	Optimisation effort (-opt_level): high <normal>	The alternatives to default makes use of fewer timing and RTL optimisations indicating the default includes a degree of optimisation which is not the case in ISE hence the use of "high" there. (synth_design -directive default)
Mapping	Global Optimisation (map -global_opt): speed <off>	Mapping is not an explicit step in Vivado, but this physical resource cognisant optimisation offers some similar functions. The alternatives to "default" optimise for area and logic instead suggesting the default optimises for speed hence its use in ISE. (opt_design -directive default)
Placement and Routing	Extra Effort (par -xe -power): normal <none> Power reduction: on <off>	The alternatives to default are single aspect goals or reduced optimisation effort. (place_design, & route_design, -directive default) Optimising is non-default for both ISE and Vivado primarily due to long run-times but was added for both sets of compiles. (power_opt_design)

Table 5.2 – Table indicating the few divergences made from default settings in the compilations made for comparisons between the V5 and V7. Values shown under ISE 13.4 V5 indicate the change made, < > contains the default and ( ) indicate the Tcl command of the option. Values listed under Vivado 2013.2 V7 give the default used here showing justification for the deviation from default in the ISE runs.

tion values (also shown) in these comparative compilations.

- As discussed the ROACH board prescribed the I/O interfaces available to Fynbos. The interface modules used in Fynbos when targeting ROACH do not map to the V7 architecture but are too entwined with the design to be easily removed.[1] Therefore, the interfaces are absent from all V7 compiles and the quantities assigned to the interfaces are subtracted from the V5 values when comparing the two. Again, given a real world use case where hardware was optimally selected, it is unlikely that these particular interfaces would be used, a hard-block PCIe interface, or multiple ports and a DRAM controller would be more appropriate and as such including the current I/O in the evaluation here is considered valueless.

- Given the range of parameters to be altered and scope of configurations available for these scaling comparisons, generally increments of multiples of two

[1]The 10GBe core is included using a separate ngc file that also includes the clocking hardware all of which was generated with ISE System Generator 11.4 as later versions no longer the OPB bus used on ROACH.

Table 5.3 – Table showing the core latency and DSP use in both the V5 and V7 comparative compiles. Bracketed values indicate the value used in the ROACH production versions.

	Latency (cycles)	**DSP48E consumption**
Floating point Add_Subtract	2 (3*)	2, Full usage
Floating point Multiplier	2	2, Full usage (3, Max usage**)
Floating Point Comparator	0	0, No usage
Floating point Division	9***	0, No usage
Floating Point to Fixed Point converter	0	0, No usage
ISE 13.4 core version	Floating_point 5.0	
Vivado 2013.2 core version	Floating_point 7.0	

[*] Having a higher latency conserved register resources which are at a premium on the V5. For the comparative runs where registers were not a constraint on the V7, it was more appropriate to use 2 cycles as representative of what would be done given the scope to choose an FPGA.
[**] DSP use was limited in the comparative runs in an attempt to achieve greatest array size scaling before hitting a DSP resource limit, but in the production compiles using more DSP slices is primarily a space saving mechanism releasing other resources that are at a higher premium.
[***] For the production compiles 9 cycles is an optimal logic and latency trade-off on the V5 chip. This is not necessarily the case on the V7 but the same latency settings are retained in the comparative runs for consistency and lack of a justification for a different trade-off.

were used simply because the binary nature of logic means most resource use changes occur on binary borders. For instance memory address length increases by one bit for a doubling in depth and BRAM use, or changing the array size means changing the number of selector bits needed for each multiplexer which in turn changes the quantity of interconnect logic needed and the width of instructions. Where changes made do not match binary borders there will be an inherent degree of inefficiency, again such implementations would not be used in a production design but are merely used for demonstration purposes here.

- Changes in hard-block consumption are linear and can be calculated prior to any attempt at synthesis. Table 5.3 lists the number of DSP48E slices per floating point core for the comparative compiles as four. Given the size of data and instruction words BRAM resources are similarly distributed. Table 5.4 elaborates on the limits both types of hard-block impose on scaling in the V5 and V7 separately.

  Where DSP48E use trade-offs release resources for more strips, BRAM use (lack-

ing any notion of caching) has little effect on hardware performance and is rather a base constraint on application size. Due to addressing, however, memory size does affect decode logic and recursively instruction width. To confine the number of parameters involved and evaluate realistic circumstances therefore, two memory depths of 1024 and 2048 entries each, are chosen for evaluation. Instruction and data memories are also matched in a given configuration. This fixes Strip local data memory capacities at 4KB (32b*1024) and 8KB (32b*2048) each respectively. Instruction memory calculations are more complex with the already mentioned scaling of instruction width with array and data memory depth scaling. The matter is further complicated by the fixed form of BRAM blocks which can be divided into units that are multiples of nine bits wide. As such where the instruction width is not exactly divisible by nine there is a measure of inefficiency with implemented memory going unused. AppendixA table A.2 details the calculations of such showing examples of 0-22% of implemented instruction memory going unused (the loss is not a function of scaling). This is not to say that an ASIC implementation would not also incur similar inefficiencies for similar reasons, but greater flexibility would enable better efficiencies.

Table 5.4 – Table showing the maximum number of Strips possible in a Fynbos array according to the hard-block limits of the V5 and V7. These values ignore any other resource, timing, or power limits that may or may not become relevant. The values shown in brackets give the total number of a resource available on the given chip.

		Required per strip	V5 (strips)	V7 (strips)
BRAM 36KB blocks*	**1024 entries deep**	2.5	97 (244)	516 (1292)
	2048 entries deep	5	n/a**	258 (1292)
DSP48E		4	160 (640)	540 (2160)

[*] Units of 0.5 represent 18Kb blocks.

[**] Comparative compilations for the 2048 deep memories were considered unnecessary with resource limits on the V5 too tight for real scaling and the V7 results showing the effect of such sufficiently.

Unrelated to standardising, of final note is that Fynbos configuration labels in all comparisons will use the form: <Tiles>x<Strips>x<DCSs –optional> (multiplied out total number of Strips –optional). Eg: 4x8x12 (32) represents a Fynbos array containing 4 Tiles, each containing 8 Strips, for a total array size of 32 Strips, 12 of which are DCSs. Due to the time consuming nature of such of generating compile data, not all configurations can be generated.

5.2.1.2 ASICs

As many parameters need to be accounted for in the above inferring what characteristics an ASIC implementation of a FPGA design would have is all the more fraught with complicating factors. [Kuon and Rose, 2007] is the most recent published attempt to quantify the area, speed, and energy gap between FPGAs and ASICs. Their approach improves on past attempts by comparing area, speed, and power consumption of 23 complete application circuits (as opposed to gates per LUT), and by quantifying the impact of FPGA hard-blocks.

Due to the multitude of parameters involved comparing an ASIC and FPGA is extremely difficult. The manner of comparison therefore depends heavily on the purpose. For instance, using an FPGA and ASIC pair of comparable process technologies and supply voltages, and ignoring any monetary cost factor, consider the aspect of energy consumption. Using the appropriate tool chains it is possible to determine the energy cost of a specific RTL design, such as Fynbos, on both. However, this fails to take into account the inherent gap in clock speeds between the two platforms and the resulting performance cost difference. It may be possible to remove such a performance gap, however, by instead using platform appropriate RTL designs, but at a cost of logic which changes the energy consumption comparison. Alternatively, the energy consumption of a specific functionality might be compared on the two platforms using the performance optimal platform appropriate RTL design and maximising clock speeds on each, and performances may or may not match. In this latter scenario logic costs, energy consumption costs, and the resulting performance, can all be taken into account. However, such a comparison is only valid for the single case study application implemented. In the case of [Kuon and Rose, 2007], therefore, the former approach of comparing near identical RTL designs is taken making a performance comparison near but not quite meaningless, offering at best an approximate order of magnitude.

Adding to the complications of using their results is the fact that, as was time appropriate for the paper, they compare designs on CMOS 90nm FPGAs to designs on CMOS 90nm standard cell ASICs. As was already listed the V5 and V7 are 65nm and 28nm devices respectively, while a contemporary Xeon or GPGPU might be fabricated using a 22nm. While the earlier paper [Zuchowski et al., 2002], suggested that the gap between FPGA and ASIC remained near constant from 0.25um through 90nm processes, it cannot simply be assumed that such will be the case for more recent pro-

cesses. Two further qualification made by [Kuon and Rose, 2007] include that while IO potentially has a considerable impact on analysis they excluded it for tractability (this suits the work here as IO is also ignored), and that the gap between standard cell ASIC designs and full custom implementations is in itself large,[1] which would make the gap to FPGAs even larger. However, as it is less common to use such they do not use is as the comparator, meaning their estimates are simply even more conservative.

Given that in Fynbos's design the hard blocks are used extensively it is also notable that [Kuon and Rose, 2007] found such hard-blocks directly affect the power and area gap between FPGA and ASIC, but had less of an impact on critical path delay. None of which takes into account the impact of reduced flexibility (because that would be is application specific). Finally, other factors that would affect the comparison but which cannot be easily quantified or for which data is unavailable include whether a multi-threshold voltage process is used or not in either, the nominal supply voltage, and fabrication monetary costs.

Taking all of the above into account, [Kuon and Rose, 2007]'s ratios for power and logic can only be considered at most as approximate guides. But every compromise made and assumption taken is justifiable and conservative meaning this guide is not unrealistic and therefore the values generated using it will be a conservative order of magnitude estimate.

[Kuon and Rose, 2007] offers different scaling factors under different conditions, based on the above the factors selected for use in characterising a hypothetical Fynbos ASIC here are summarised as follows.

- Power:
  - This is the most difficult metric to estimate given that it is affected by so many factors including even the specific foundry.
  - Accurate test benches were not available and so constant toggle rates across the board were used. In support of this, where test benches did exist the results showed the constant toggle rate approximations to be unbiased and sufficiently accurate giving confidence to such.
  - The FPGA generated power estimates are for the complete chip including sections not used, while an ASIC by definition only includes used logic.

[1] Standard cell designs can be 3-8 times slower, use 3-10 times more power, and are around 14.5 times less dense, than full custom designs.

  - Whether ASIC designs should be examined considering worst or typical operating conditions is purpose specific. Static power will differ wildly with such although this is less of a concern for dynamic power as temperature variation has significantly less impact on such. For this and other reasons detailed in the paper, a comparative ratio was only derived for dynamic and not static power.
  - Considering these points, Fynbos' proportional use of hard-blocks, and Fynbos' proportional chip resource use, a best approximate scaling factor of 1/10X FPGA dynamic power is adopted for the hypothetical Fynbos ASIC.

- Area:

  - FPGA area consumption in a design being higher than an ASIC is fundamentally due to all the additional logic needed to support reconfigurability. As such the use of hard-blocks significantly reduces the gap.
  - Given a larger design the tools will try harder to fit it in and meet timing than they would otherwise.
  - Considering that Fynbos makes considerable use of hard-blocks, an average factor of 1/18X FPGA area is adopted for the hypothetical Fynbos ASIC.

- Delay:

  - In the study hard-blocks always saved space, but only ever marginally increased speed, more often slowing a design down. This is due to either a poor fit of the algorithm to the blocks or, where the gain is marginal, it is consumed in extra logic required to reach the hard-block.
  - FPGA designs with significant hard-block use on the highest speed grade devices were found to be on average 3.1X slower than ASICs, but considering ASICs are usually designed for worst case, the lowest speed grade FPGAs offer a more honest comparator resulting in a larger gap.
  - Considering again Fynbos's extensive use of hard-blocks, and the slower (-1) speed grade of the V5 in use, an average factor of 4X FPGA clock speed is adopted for the hypothetical Fynbos ASIC.

- Performance:

– Again due to all the customisation opportunities and scope for more optimal redesign when targeting an ASIC versus an FPGA, any attempt to draw raw performance comparisons has even less certainty than the metrics derived using the above factors. Nevertheless a comparison will be made offering at the very least an order of magnitude estimate. At worst this will only be an underestimate of what a hypothetical Fynbos ASIC might achieve.

5.2.2 Hardware Comparisons

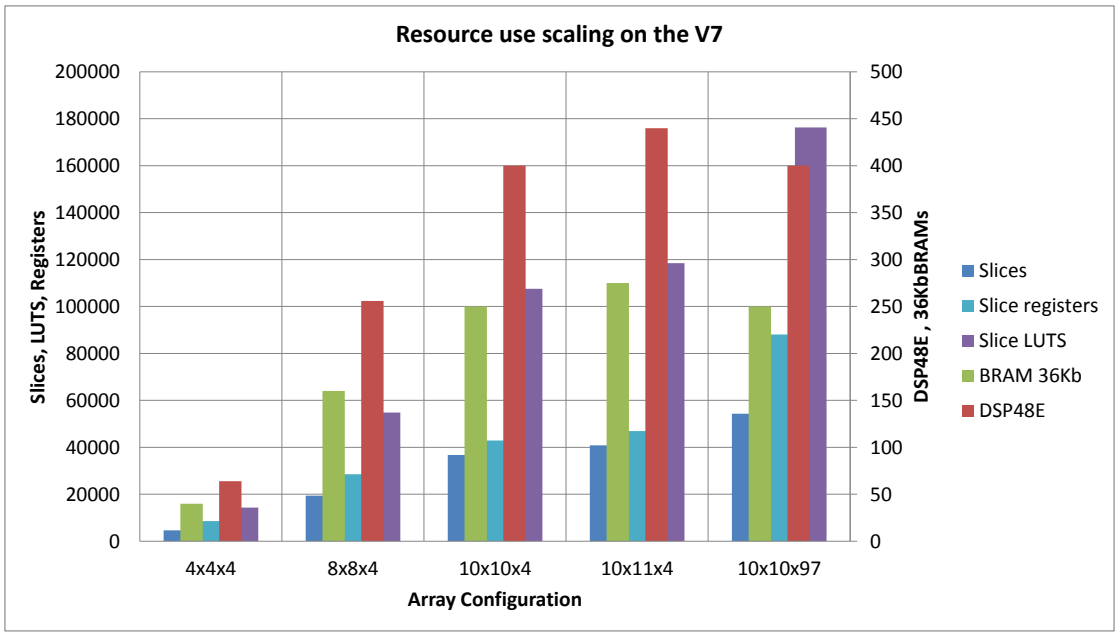
The following section examines Fynbos' hardware performance as an HPC many-core architecture. To begin with scalability is examined considering logic, power, and critical path delays, after which the memory hierarchy, data and control paths, and ALU design are evaluated with the same.

5.2.2.1 System Scalability

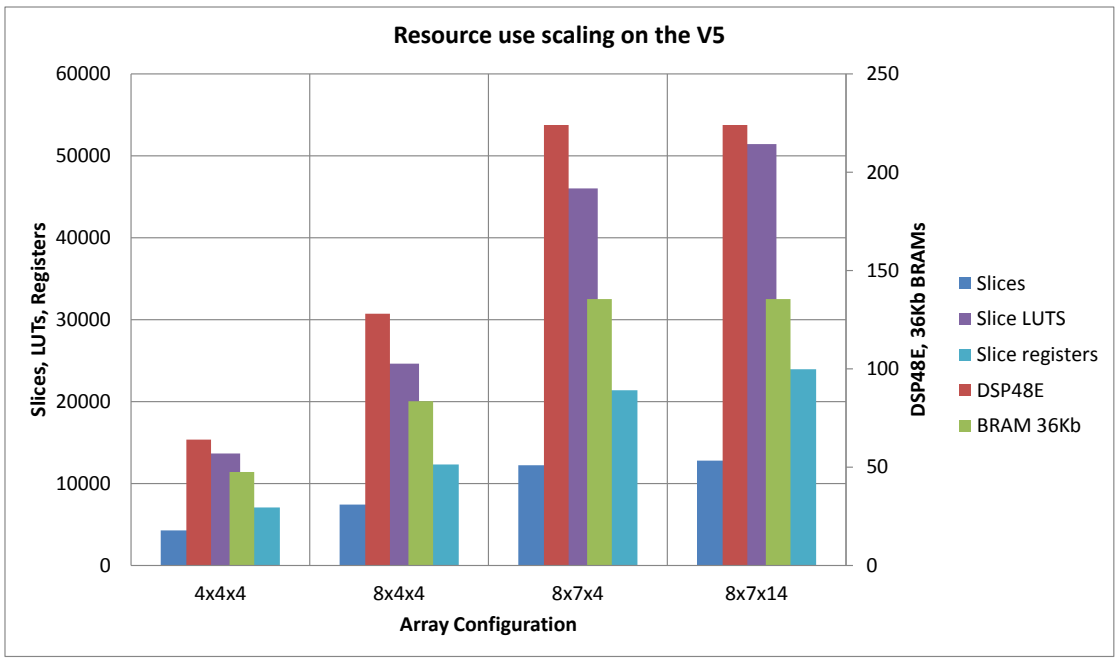
Looking at maximums figure 5.1 shows the resource usage on both the V7 and V5 in varying configurations that support 1024 deep memories and constant DCS capacities (aside from the last configurations). In both graphs DSP and BRAM scaling is exactly predictable, and the LUT and register values show unsurprising increases with array size. The 8x7 and 10x11 arrays represent the largest arrays that meet timing on each target respectively. Comparing the last configurations with their preceding maximums, division is shown to consume greater quantities of LUTS than other resources.

On each target the limits on scaling are due to different causes. Figure 5.2 shows the proportional (percentage of the total available resource) consumption of each resource in these maximum size arrays. In the case of the V5, slice consumption is above 80% in implementing the 8x7 designs, while the same design on the V7 uses only 5% of available slices. In real terms the same design on the V5 is also implemented with less resources but this is primarily a product of the tools working harder given the greater constraints.

In the case of scaling on the V7 the limit is clearly therefore not logic resources but rather timing. If timing is ignored the array size could continue scaling until 512 Strips



(a)



(b)

Figure 5.1 – Bar graphs showing resource use of scaling Fynbos configuration sizes on the V7 (a) and V5 (b). In both all but the last configurations contain four DCSs for equivalence. The largest array to fit and meet timing on the V5, the 8x7x4, is close to half the size of the same on the V7 (56 Strips vs 110). The last data points instead shows an alternative maximum; the largest array supporting maximum DCS capacities

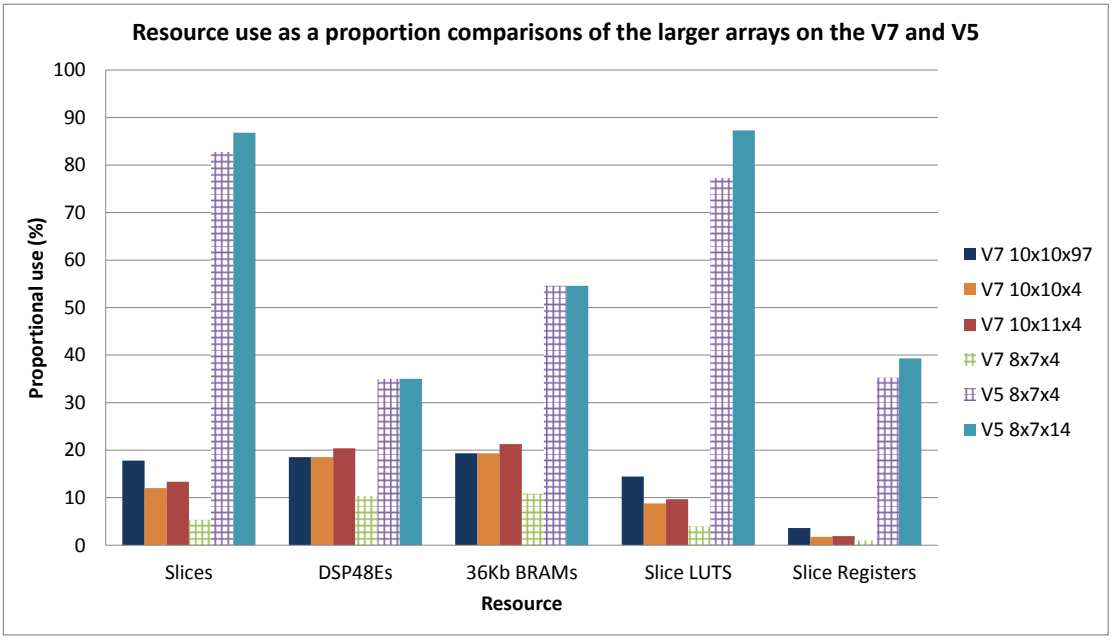


Figure 5.2 – Bar graph comparing the proportional resource use of the maximum scaling Fynbos configurations. The values for an 8x7x4 configuration are shown on the V7 and V5 in hatched colours for comparison. Clearly resources are not a limiting factor on the V7, while on the V5 LUTS in particular are a limit.

before any hard resource limits were hit.[1] The limiting factor rather is critical path delays as the same 100MHz clock is matched in all V7 designs.[2] In this case the speed limit encountered is specific to the Virtex7 family, as the Vivado tools report running out of super long lines (SLLs), a resource specific to the Virtex7 devices which facilitates interconnections between the new super dense logic regions via a passive interposer layer. With placement directions and greater configuration testing a slightly larger array is most likely possible, a 128 Strip 4 DCS array achieved 95MHz.

While device specific the limits' cause is indicative of substantial or excessive quantities of interconnections in the Fynbos architecture. Fynbos is indeed a densely interconnected design but this was a known design compromise between scalability and data communication. These results show that given this compromise and a 100MHz

[1] While such an array cannot be placed it is synthesisable, and based on trend differences between synthesis and post placement utilisation results (a gap of 7 and 5% for LUTs and Registers respectively), using the same RTL and tool settings, a 512 Strip Fynbos array will hit DSP and BRAM limits long before the logic resources available are exhausted on the V7.

[2] As with comparisons to an ASIC, targeting so different an FPGA means a range of fundamental design changes could be made that would affect maximum speed making a speed comparison uninformative

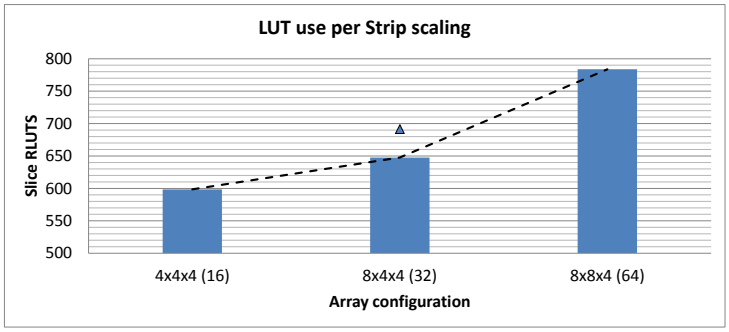
target the scalability limit is around 100 PEs on the V7.

In the case of the V5 the scaling limits are instead clearly linked to resource availability as the more dense a design the more difficult to correctly route. Timing analyses indicated the limiting critical paths were most often tool and IP settings depending, either within the DSP units used for the ALUs, or in the data path directly to or from an ALU and its local data memory. Initially this suggests that a simple lack of pipelining within the ALUs is at fault. However, adding further registers to these pipelines fails to further increase speeds, which is in line with Kuon and Rose [2007]'s assessment that in a net analysis hard-blocks generally fail to add speed to designs. That is when 80% full, the routing required to reach DSP blocks or additional registers inserted logically before them is simply too lengthy due to the congestion for any net gains. Clearly implementing the ALUs using slices instead of the DSP blocks is also not an option given the current 80% consumption.

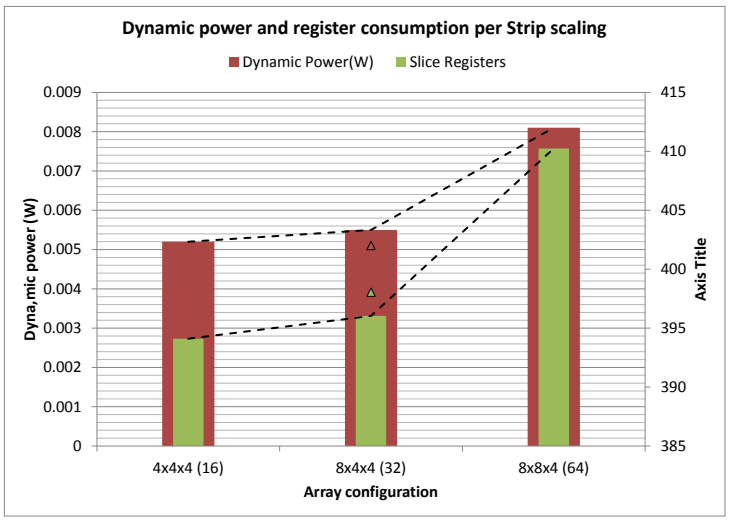
5.2.2.2 Configuration Variants

As already discussed Fynbos includes a number of parametrised reconfigurable characteristics, these are used in the following to compare the effect of various configurations on scalability, resource use, and power consumption.

Interconnect and PE Hierarchy are anticipated to affect efficiency of resource use as Strip distribution in Tiles will impact the interconnect implementation. It also clearly has the potential to affect application performance as it concerns inter-Strip communications but this factor is left for later discussion. Examining the hardware only, table 5.5 shows the 4x8x4 configuration to be moderately more resource intensive than its reverse 8x4x4 (7% more slices consumed). This is also as expected considering increasing the number of Strips per Tile will increase the quantity of intra-Strip connections (two additional data word paths per additional Strip). This is relative to increasing the number of inter-Tile connects which involves only lengthening existing data paths. Interestingly the more resource intensive Strip dominant configuration uses marginally less dynamic power than its counterpart which could be due to the long paths required in the latter. Given the lack of certainty in the power estimates and that the difference is 0.1% of the total, however, further testing with better tools would be required to confirm any such hypothesis.



(a)



(b)

Figure 5.3 – Graphs showing the per-Strip consumption of LUTs (graph a), registers and dynamic power (graph b), in doubling array size configurations on the V7. Values plotted are averages across all general purpose Strips in each array so as to avoid representing any amortising of administrative logic or representing resource re-use. In each, values for the reverse configurations, the 4x8x4 (32) array, are overlaid as single data points on the 8x4x4 bars showing the trend would differ if this configuration was used instead. Most notably the difference is sufficiently large that it would visibly equalise the LUT gradient across all three array sizes, but the knee would remain visible in the power and register series.

To examine the efficiency of the interconnect in more detail, Figure 5.3 plots the changes in per-strip resource (LUT, registers and dynamic power) use as array size increases. To observe a trend a constant rate of increase in array size is needed. With the available data this means only the 16, 32 and 64 Strip configurations are useful, given the expected difference between the 8x4 and 4x8 arrays, the second is overlaid on the first as a single data point. To ensure changes observed are not a result of further amortising the logic costs of the clocking, system control, division, and special case Strips, the values used are the result of averaging what was consumed only by all the general purpose non-DCS Strips in each array. The averaging is needed as resource sharing means in reporting resource use for a given Strip the tools may count the same resource in multiple Strips or vice versa.

Table 5.5 – Table comparing the resource consumption of a pair of Fynbos configurations so as to compare the effect of implementing many Tiles with few Strips with the reverse. Values are as generated for the V7 target.

Array	Slices	Slice LUTS	Slice Registers	Dynamic Power (W)
4x8x4	9082	28198	15017	0.279
8x4x4	8477	26577	14952	0.282

Despite the values plotted being on a per Strip basis, an increase in consumption is still evident in all three. The increases are due to the increasing degree of required interconnectivity. As a result of the compromise between connectivity and scalability this is expected and evidence of why the architecture will have a scaling limit. The greater rate of increase between the 32 and 64 Strip arrays indicates the limit will be reached asymptotically. Employing higher Tile counts with fewer Strips each may extend the range slightly, but only if an application's communication needs will allow it.

Memory size affects address lines and instruction width, potentially also impacting resource consumption. To evaluate the effects of increasing memory sizes Table 5.6 gives the ratios of resource consumption in a Fynbos configuration using 2048 addresses deep data and instruction memories and

Table 5.6 – Table showing the cost of larger memories through ratios of the resource consumption in arrays supporting 2048 and 1024 deep data and instruction memories. The resulting ratios are all near one but with no meaningful visible trend. Consequently it can be concluded that memory size, relative to the rest of the architecture, has no significant impact on resource consumption.

Array	Slices	Slice LUTS	Slice Registers
4x4x4	1.059	1.007	1.010
8x4x4	1.156	1.009	1.011
8x8x4	1.058	1.011	1.014

the same size arrays with 1024 addresses deep memories. Clearly DSP use will not change and BRAM use will double so neither is reported. The comparison of Slices, LUTS, and registers, however, show that the impact of larger memories - slightly wider address lanes and instructions - on the overall logic costs of the array are inconsequential. Given the other scaling limits already encountered it cannot be shown where increasing numbers of address lines in the global communication infrastructure would ultimately become a limiting factor. In conclusion it is therefore unlikely that program size will be a limiting factor provided the chip space can be made for larger memories.

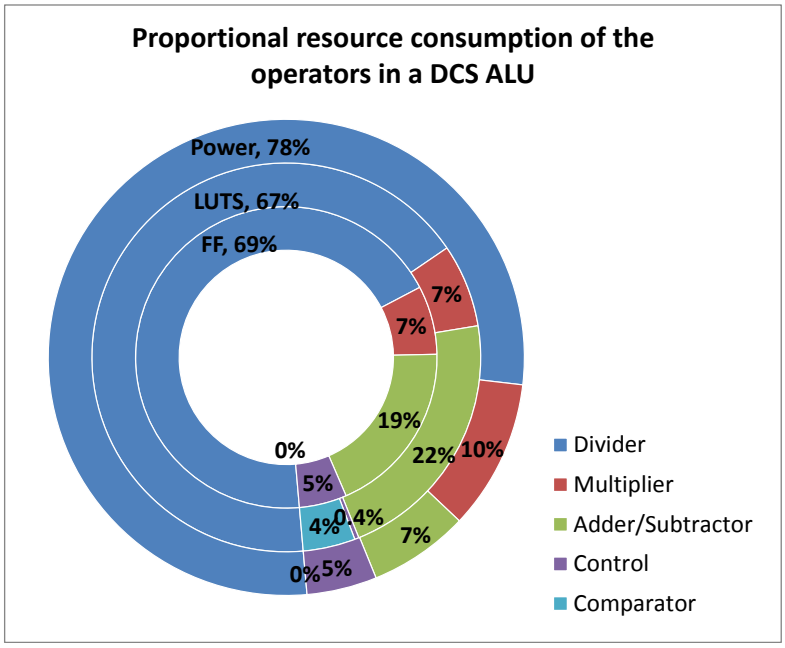


Figure 5.4 – Chart representing the proportional resource cost of each ALU operating unit. Power, LUTS, and registers are shown from the outer most to the inner most ring respectively. Special functions such as copy or branch are not included. To account for the resource sharing the source values used here are averages of the cost of all of each unit across the relevant ALUs in a 10x10x97 configuration targeting the V7 FPGA.

ALU design has been a topic of much consideration in the design process. Figure 5.5 will show that ALUs consumed the major share of resources but Figure 5.4 first compares the resource cost of the different operands. The singularly large cost of division is most obviously apparent, representing 78% of the dynamic power cost of this ALU. Multiplication appears to use less logic while consuming the second highest quantity of power because the use of DSP48E hardware is not represented. Consisting solely of combinatorial logic the comparator unit uses less power than is within the precision recoded by the Xilinx tools.

5.2.3 Power Distribution

Considering a system breakdown of how resources are consumed, Figure 5.5 shows the logic and power consumption of the major Fynbos sub-systems collated together. Examining the outer power ring, ALUs consume 46% of power relative to the combined 26% (1+25) of decoders and system controller costs. These values are in reference to a 10x4x4 Fynbos configuration with 1024 deep memories, considering that division consumes 78% of a division capable ALU's total power (see earlier Figure 5.4), this comparison is moderate relative to what would appear in a configuration containing greater division capacity.

The global control path costs are not included in this due to how the tools report values and the code hierarchy which mean the interconnect and control paths cannot be separated. However, while together these consume a further 15%, using the LUT and FF rings it can be seen that this is predominantly a result of register use, which considering the design means this is predominantly a function of the interconnect rather than the global control path. Based on such the ALU power consumption, in line with the goals of a many-core, is the dominant consumer.

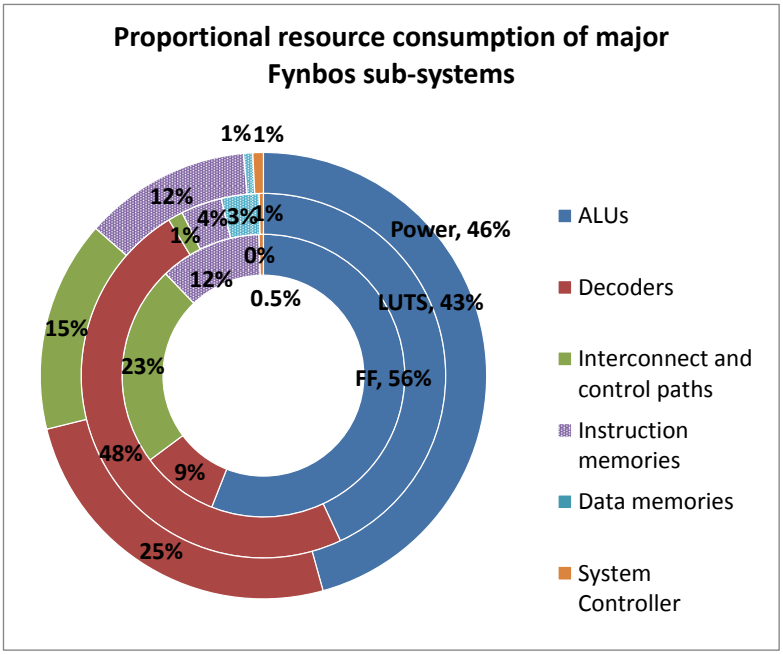


Figure 5.5 – Chart showing the proportional resource consumption of each major Fynbos component. Power, LUTs, and register use is shown in each consecutive ring from the outer most to the inner. The values used are taken from a 10x10x4 configuration with 1024 deep memories targeting the V7 FPGA. The significantly dominant proportion of resources use dedicated to the ALUs is evident, in line with the goal of many-core architectures for minimal control and maximal ALU logic.

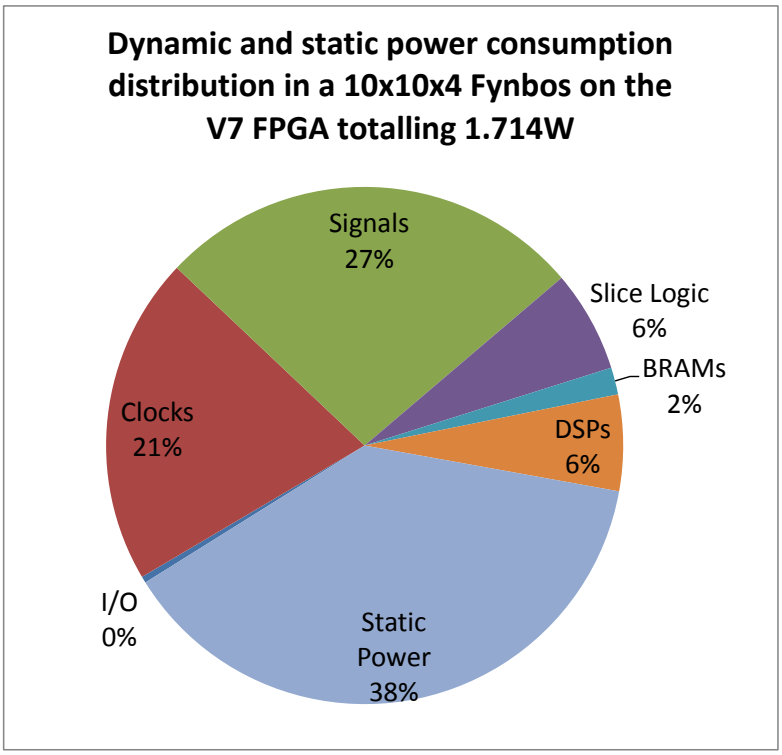


Figure 5.6 – Chart showing the proportional distribution of power when a 10x10x4 1024 deep memory Fynbos configuration is implemented on the V7 FPGA. The value shown for I/O is entirely unrepresentative due to no realistic I/O being included in the V7 targeted design for reasons discussed earlier.

In logic consumption the decoders specifically consume 48% to the ALU's 43% but this ratio is significantly skewed by the previously discussed use of hard-block DSPs by the ALUs. In register use the ALUs are the largest consumer by a significant margin, using 56% of all registers in the design.

The memory systems occupy the final 13% with instruction memories consuming 12% and the data memories only 1%. The inner LUT and FF rings again show the reason for this. The instruction memories larger use proportion is primarily due to register use. This is in turn due to the need to store each word as it arrives so as to join the upper and lower portions of each instruction, further LUT logic is additionally needed to control such.

In Figure 5.5 the systems represented consume a total of 1.057W in dynamic power. Showing an alternative FPGA fabric specific division and thereby including static power, Figure 5.6 presents a break down when the 38% static power is included bringing the total to 1.714W. The diminutively small value assigned to I/O here is clearly unrepresentative of a real world implementation and a result of no I/O in the design. In the V5 implementations that included the 10Gbe core, however, the 10Gbe core alone consumes 1.852W, more than doubling the total consumption of a system. While implemented on a different process and already identified as inappropriate for this context, this does reveal I/O as potentially considerable cost. In addition to selecting a lower

power and higher bandwidth form of I/O, any real world implementation would further need to match a sufficiently large array to the I/O costs to ensure net value on a system.

5.3 APPRASE-Fynbos Software Evaluation

In examining compute performance and energy efficiency, the industry standard is comparative execution of standard domain appropriate benchmark suites. Of the architectures reviewed, many of the designs included published HPC relevant and alternative benchmarks (for instance RAW was evaluated with SPEC and STREAM [Taylor et al., 2004], and between them the dataflow architectures have been evaluated on a mixture of SPEC, SPLASH2 and Mediabench[Lee et al., 1997]). For HPC, depending on the exact application domain different benchmarks are more useful than others, but for general purpose ranking the top500 uses LINPACK.

Based on such, LINPACK and SPEC (as the most commonly run suite on the architectures reviewed), would be the most appropriate means of performance analysis for Fynbos but such an analysis is not practical for a number of reasons. Firstly there are language constraints, while Fortran versions of LINPACK are available the same is not true for large portions of SPEC. Secondly, as was noted earlier, significant direct comparisons would not be possible as between the reviewed architectures a diverse and incomplete set of the suite applications is covered. Thirdly, the possibility of running either in hardware is muted by the largest implementable size of Fynbos, requiring the so far only proposed multi-load operation capability in the software stack.

Lacking the above the BLAS routines on which LINPACK relies would also serve as an appropriate alternative considering that they are available in Fortran, well known as performance indicators, and can be given any scaling size input. However due to the beta nature of APPRASE, and dependence on collaborators for such, this secondary ideal is also not possible.

Lacking these ideals the alternative ultimately used in the following is a theoretical performance quantifier, and the one real world representative application to have been passed through APPRASE. For the theoretical quantifier therefore, a theoretical representation of the BLAS level 1 operation SAXPY (Single-Precision A multiplied by X Plus Y) is used. Of the BLAS routines this is a common standard for quantifying an architecture's theoretical maximum performance, for which I/O timing costs are not included. For the real-world case study, as a proof of concept exercise an n-body

simulation of the solar system's planetary bodies is used, scaling up to seven bodies. N-body applications are one of the original dwarves of computing as identified by [Asanovic et al., 2006]. Within this classification the domain is considered compute (rather than memory) bound, which is appropriate for Fynbos.

The Fortran code for the solar system is simply sequential, requires SQRT and division, makes function calls, requires data sharing, and is easily scaled. This makes it a good case study testing many of the functionalities required. As a n-body simulation the system falls into one of the many HPC application domains. As a solar system simulation the number of bodies is small relative to other applications of the approach. For comparison DLpoly is a molecular dynamics simulation package which will typically be used to simulate multiple thousands of molecules, or GADGET is a cosmology simulation package which may simulate millions of bodies. However, the smaller nature fits more easily into the available Fynbos instances, and will give a representative demonstration of the communication and processing costs and performance of APPRASE-Fynbos in its current form. These results will be indicative of how a larger body count application on a larger instance of Fynbos would preform (considering the communication infrastructure will be equivalent and the processing capacity simply scaled with the application's requirements). What will not be demonstrated, however, is how a multi-node instance would preform, as discussed later. This is a significant further evolution of the system which has not yet been investigated and which is beyond the scope of this thesis.

Using a standard Fortran compiler, gfortran in this case, the same code may also be run on an x86 architecture. Given the argument previously made for many-core processors to replace cluster core nodes within clusters rather than clusters, and the secondary focus on autoparallelisation, executing an explicitly sequential section of code on a sequential x86 hardware thread is a more appropriate comparator than a parallelised version of the code.

In combination these two performance quantifiers, SAXPY and the solar system simulation, enable a low level analysis of the Fynbos-APPRASE system which while less useful in comparing such to other work, does serve to provide insights in considering future versions of this and other future many-core designs in general.

The following section is therefore divided into three discussions. In the first the theoretical quantifier SAXPY is used to discuss theoretical peak efficiency and performance of Fynbos. In the second, the case-study solar system is used to examine the

APPRASE-Fynbos system's ability to find and exploit parallelism in an application. Finally the third reports the real world performance of the solar system application on APPRASE-Fynbos and an x86 Xeon.

5.3.1 Theoretical Maximums of a Scaling Fynbos

Vivado unfortunately does not support timing simulations of VHDL designs (it only supports switching activity interchange format (SAIF) file generation of the I/O ports and not the internal wire declarations)Simulation [2013] which means timing simulation power analysis is not possible. The power estimate values used in the following were therefore instead generated using Vivado's power analyser assuming constant toggle rates and static probabilities for all the nets in the design and are therefore considered less accurate. No quantifier of this error exists, however, as such would be designed specific. Considering that Kuon and Rose [2007] did find the values to be insignificantly different for their test cases. However, the following assumes the values are correct to within sufficient limits for conclusions to be drawn. All other environmental variables and software options used in generating the results reported here are also the same as discussed earlier. Similarly all inferred ASIC values given are calculated using the same ratios motivated for earlier. For the purposes of the following discussion, FLOPS is used as a performance metric, and FLOPS/W as a measure of efficiency.

SAXPY performs the vector operation: $y = \alpha x + y$. Due to its frequency of use, where a product of two numbers is added to a third, many general purpose processors have dedicated hardware capable of performing this in a single step. The operation, referred to as a MAC (multiply-accumulate), may be performed with one or two rounding stages (in the case of the former it is then referred to as a fused MAC or FMA). In either case the joining provides a performance advantage by combining two steps into one resulting in the execution of fewer pipeline stages. In Fynbos, however, with the simplified pipeline and use of triplet parallelism a similar 3-input operation is not possible. Consequently two separate operations are required on Fynbos which comes with a cost, although Figure 5.7 shows the difference to be minimal relative to the absolute performance. Also shown is the predictable scaling increase in performance with an increase in array size, achieving just under 1GFLOPS in a 128 Strip array implemented on the V7 FPGA.

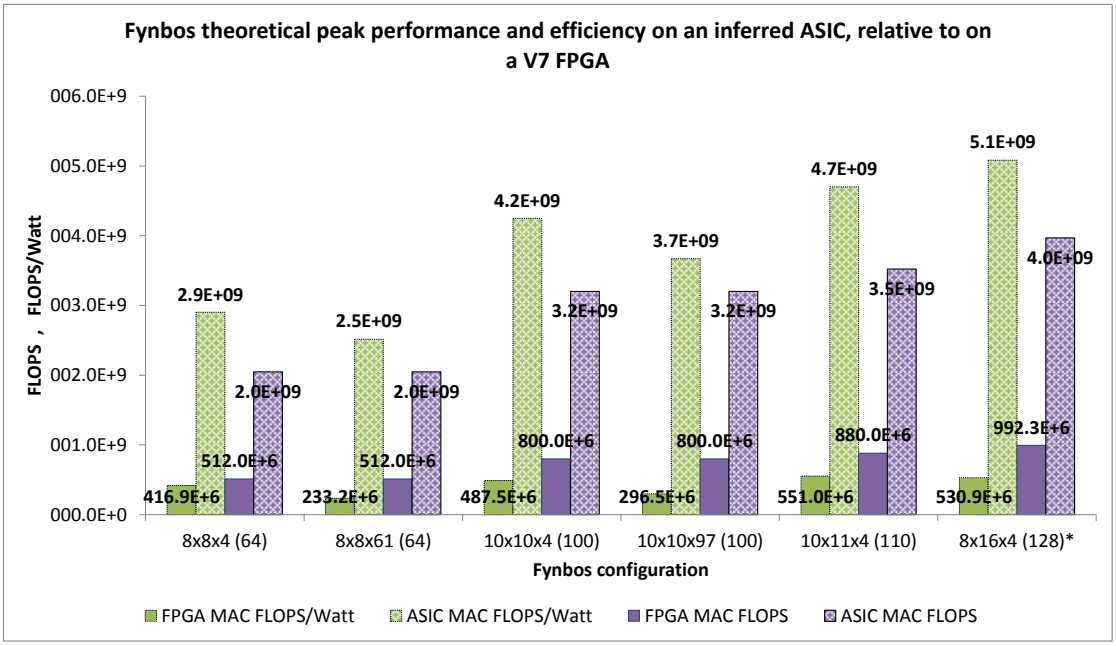


Figure 5.8 – The values plotted here are of theoretical peak FLOPS and FLOPS/Watt values for a range of Fynbos configurations. Plotted in solid colour are values achieved on the V7 (and one instance of the V5) FPGA. Performance is calculated assuming SAXPY as a benchmark where the add and multiply operations cost 12 and 13 cycles respectively for the complete fetch-decode-execute-store sequence. The efficiency calculations takes into account total dynamic and static power consumption. In dotted series, the equivalent theoretical peaks for the hypothesised ASICs are plotted using 4X increase in clock speed and 10X decrease in dynamic power consumption. The V5 and V7 based comparative 8x7x4 values show the efficiency cost of including I/O and comparative 10x10x4 and 10x10x97 configurations show the efficiency cost of greater division capacity. Given the benchmark no performance cost is incurred for such.

[*] Values calculated using a 96.9MHz clock rather than 100MHz

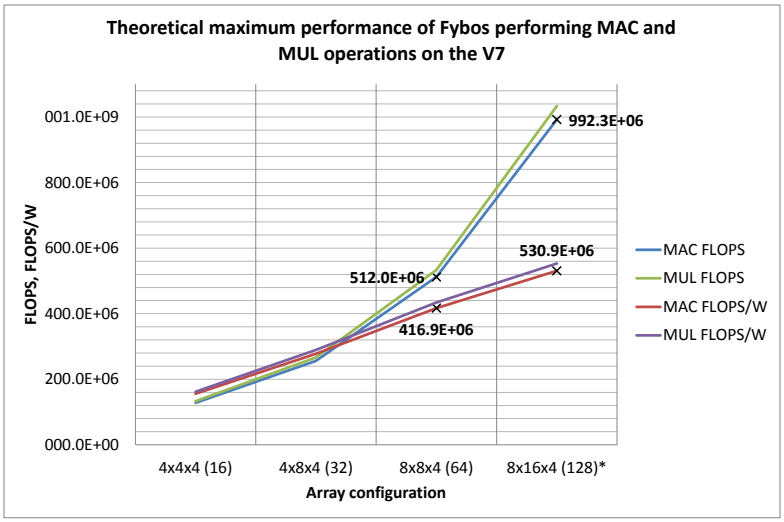


Figure 5.7 – Plot showing the minimal gap in performance, relative to peak, when comparing MAC and MUL as benchmarks. The Fynbos arrays on the x-axis double in size resulting in the y-axis showing a predictable exponential increase in theoretical peak performance. The efficiency graph alternatively follows a linearly increasing trend reflecting the increasing costs as

Figure 5.8 compares the peak performance and efficiencies possible on a range of array configurations on the V7, with an inferred hypothetical ASIC equivalent that takes into account no design changes or cus-

tomisations to the new fabric and assumes a similar 28nm process. Generating the ASIC values used the four fold increase in clock speed and ten fold drop in power consumption. While the largest array to meet 100MHz clock speed timing on the V7 is the 110 Strip array, a 128Strip array is also reported and its values calculated using the tools provided maximum clock speed of 96.9MHz. Relative to the 110Strip, this slightly larger and slightly slower array achieves higher performance and efficiency metrics on the ASIC but drops in efficiency on the FPGA (530 vs 551MFLOPS/W).

Comparing the highest efficiency rated FPGA design with the highest efficiency rated inferred ASIC therefore (the 10x11x4 on the V7 with the 8x16x4 on the ASIC), the ASIC achieves a half order of magnitude gain on performance (880MFLOPS vs 4.0GFLOPS) with a full order gain in efficiency (551MFLOPS vs 5.1GFLOPS/W). If instead identical arrays are compared, the expected 10x and 4x gains in efficiency and theoretical peak performance are also visible.

The remaining configurations in Figure 5.8 are given to demonstrate the efficiency cost or overhead of carrying greater division capacity. Comparing the 8x8xY arrays, a 15X increase in DCSs results in a 44% drop in efficiency on the FPGA or a 14% drop on the ASICs. Comparing the 10x10xY configurations, a 24X increase in DCSs results in a 39% drop in efficiency on the FPGA or a 12% drop on the ASICs. Given the benchmark being used, peak performance in the same is unaffected.

Figure 5.9 plots the inferred ASIC values derived from four Fynbos configurations fitted to the V7. These four arrays are sequenced to double in size producing a trend in performance and efficiency. Taking into account the further possible area reduction gains of conversion to ASIC, Figure 5.9 uses this trend to project plausible values for Fynbos arrays that utilise the additionally available logic to implement more Strips. The scaling is done including the control logic but this will only result in more conservative values. Continuing the doubling of array size the final projected point plotted scales the array size 16 times, short of the 18 fold ratio available. The final 4096

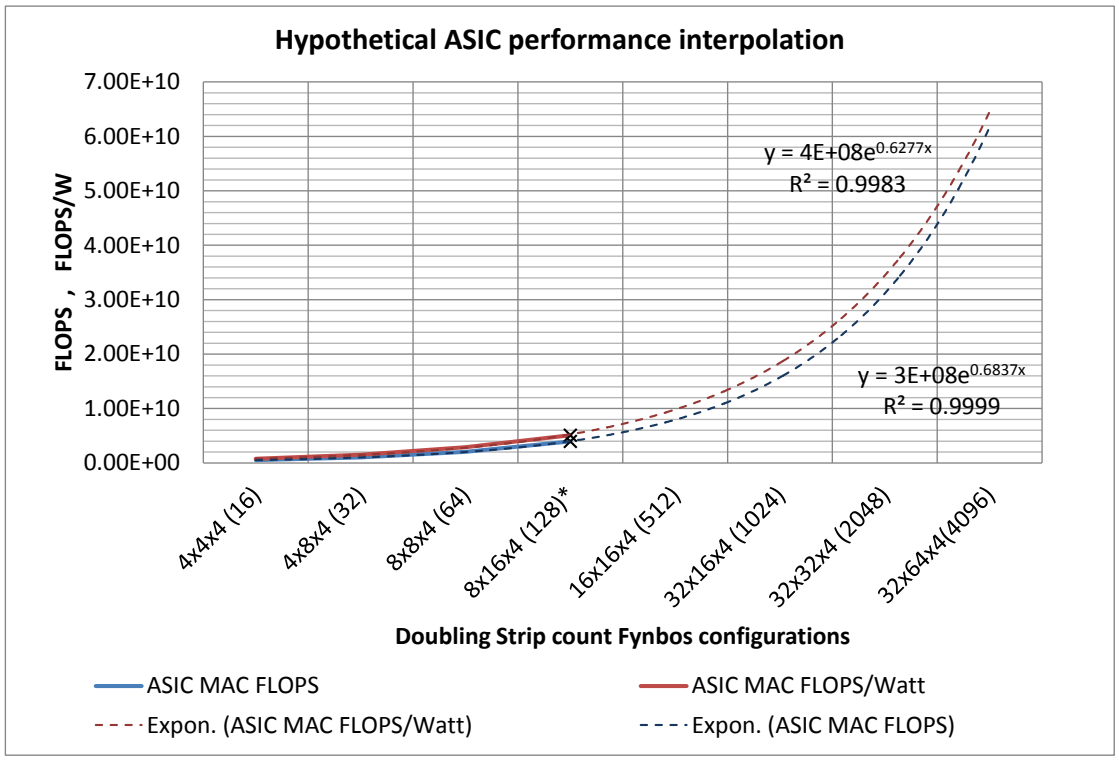


Figure 5.9 – Projections of Fynbos performance and efficiency on an 28nm ASIC using the exponential trend in four Fynbos configurations realisable on the V7. Solid lines indicate real data that has been scaled according to the ratios set out. The dashed lines extend these series according to the exponential function they indicate plotting arrays up to 16 times the size of the 8x16x4 array.

[*]Values calculated using a 4*96.9MHz clock rather than 4*100MHz

Strip array plotted therefore hypothetically achieves 60.5GFLOPS at 62GFLOPS/W on a 28nm ASIC with a TDP of less than 4W conservatively.

In an encouraging result the efficiency achieved here is greater than any of the other architectures reviewed, but Table 5.7 shows that Fynbos' raw theoretical peak performance is absurdly inadequate even in this largest hypothetical 4096 Strip ASIC array. Customising the design for an ASIC platform and implementing a version of the proposed multi-Fynbos version to shorten the maximum longest path would likely enable a higher clock speed. Yet even a 1GHz clock would only produce 327.7GFLOPS on the 4096 array for an unknown power cost. This remains far off from the likes of the K40 or XeonPhi's TFLOP range and while it would bring Fynbos to within range of a conventional Xeon's performance, with an unknown efficiency the result is valueless. This is, however, an expected outcome given the submissions of the many-core

concept.

Table 5.7 – Table listing approximate performance and efficiency values of a number of the architectures reviewed showing where Fynbos sits amongst such. These comparisons are difficult to make as most architectures fail to state what calculation is being used to represent peak performance, and what the conditions of operation are for the stated power measurement. The FLOPS/W values listed are a simple division of the first two values although the power values used are simply TDP rather than exact calulation costs. Given the data available and these caveats, however, the above are listed so as to position the architectures at least within an order of magnitude, but should be considered as such and not considered official representations.

[*] Values based on projection plotted in Figure 5.9

Architecture	GFLOPS	TDP(W)	GFLOPS/W	Clock frequency
Intel E5-2692	371	115	3.2	2.2GHz
AMD Opteron 6274	282	115	2.5	2.2GHz
IBM PowerPC A2 (BlueGene/Q)	205	55	3.7	1.6GHz
SPARC64 VIIIfx	128	58	2.2	2.0GHz
RAW	6	18	0.3	425MHz
Rigel	2400	92	26	1.2GHz
Kalray	230	5	46	400MHz
Epiphany	100	2	50	800MHz
P2012	76	4	19500MHz	
XeonPhi	2416	300	8.05	1.2GHz
Tesla K40	4290	235	18.3	745MHz
V7 implemented Fynbos 10x11x4 (110)	0.88	1.59	0.55	100MHz
28nm ASIC* Fynbos 32x16x4 (4096)	60.5	0.9	62	387.6MHz

A low clock speed is fundamental to a many-core architecture. A slow clock keeps power consumption within a realistic TDP range enabling a greater proportion of, or even all, transistors available to be turned on and in use, and means a NTV is a possibility. While theoretical peak performance is low application performance and FLOPS/W values should be high. In the case of Fynbos, unfortunately this is not entirely true due to the lack of even simple pipelining to reduce operation latencies. Pipelining of the Fynbos equivalent of fetch-execute-store, was discarded as not possible in Chapter 4 due to lack of software support for such. While the compiler deficiency might have been compensated for entirely in hardware, it would have gone against the minimalist design goals. However, in order to compete this is clearly needed.

Kalray is a near equivalent architecture that achieves 230GFLOPS, at 46 GFLOPS/W, operating at 400MHz and also implemented in 28nm. Kalray is all the more significant as a comparator given it supports POSIX, OpenMP and C/C++, is aiming to reach scaling of up to 1024 PEs, and supports a range of fully integrated standard I/O

protocols. As such it offers stiff competition. The comparison is not entirely fair, however, as Kalray's PEs are really pipelines in VLIW cores, only one of which is a FPU, and code does need to be rewritten with their dataflow model in mind to achieve optimal performance. Only a direct application performance comparison would show one or other to be ahead, but simply by virtue of being available for purchase Kalray is certainly ahead of Fynbos in many respects.

Given that every latency is foreknown and that only one branching mechanism exists with minimal additional hardware and further software changes the following is proposed to allow Fynbos a measure of simple pipelining and execution stage overlapping. Adding a specialised local instruction memory to the system controller containing maximum row latencies for each row address would enable this unit to issue new instruction addresses and control logic as soon as the longest latency in the previous row had passed, rather than waiting all the way until done signals were received. This would see an overlapping of fetch-instruction and fetch-data operations but only require minimal further registers on some of the multiplexer controls. In the case of branches, the same special instructions might prevent the pipeline from filling for the necessary number of instructions following a branch until it has been fully resolved. The additional software required would therefore simply be to represent these row maximum latencies in a specialised system controller instruction and perhaps optimise the instructions scheduled alongside a branch (it is already optimising for common latency instruction in a row). With such a system in place results would be generated every 3 cycles (depending on the operation) rather than every 11 operations as is now the case.

5.3.2 APPRASE-Fynbos Parallelisation Efficiency

The above section gives the boundaries of what is possible in hardware. Using these limits and the 7-body solar system program, this section examines the joint APPRASE-Fynbos system so as to comment on its potential to find sufficient parallelism within an application.

Amongst the factors that the APPRASE algorithm is able to take into account are flexibility in the configuration of Strips and Tiles, the net array size, and the quantity of DCSs available. Changing the ratio of Tiles to Strips influences how much power and possibly time is spent on data movement. Changing the array size affects the maximum degree of parallelism possible and thereby instruction promotion, data movement, and ultimately program length. In the face of the lock-step control system and

long latency of division, changing the number of DCSs changes how efficiently division operations can be done in parallel with each other rather than with other shorter operations. Exploiting and circumventing these factors involves APPRASE's instruction promotion algorithm taking into account the latency costs of division, and utilising the copy (move) operations to achieve greater parallelism.

To examine the effects of each of these performance metrics of the the largest program available are generated for varying Fynbos configurations. Given that the largest program available is only a seven bodies system, it is difficult to see obvious trends in the following data or confirm deductions. As previously suggested the analysis should ideally be performed using a wide range of applications with different native communication requirements, but as has already been indicated this is not a possibility for now and instead only reasoned hypotheses can be made.

Restricting the scope Fynbos test configurations are limited to those that might plausibly be implemented in the V7, and using parameters corresponding to such rather than the hypothetical ASIC described earlier. Further, due to an error in implementation[1], the compiler version used here is limited in flexibility to reliably configuring only whole Tiles worth of Strips as DCSs as opposed to portions of Tiles. Unfortunately this makes it impossible to hold the number of DCSs constant while changing the number of Strips per Tile. Any comparisons of varying numbers of Strips per Tile would therefore be inconclusive as the effect of more or less DCSs could not be ruled out, this aspect of APPRASE is therefore not explored here.

Given these constraints the Fynbos configurations used in the following are as follows. Due to the compiler restriction in order to have a constant rate of increase in DCS capacity, Strip size is fixed for all configurations with array size grown solely by increasing the Tile count. The Strip size is fixed at four simply because it is the smallest value practically realisable in Fynbos, and a smaller increment size is desirable for gaining the finest granularity of data points possible. The configurations therefore include every array size from eight to 104 Strips in increments of four Strips (2 to 26 Tiles in increments of 1). Within each array size, an instance of each multiple of four (forced to match the number of Strips per Tile) DCSs possible is further included. For example given the array size of 20 Strips, configurations of 5 Tiles and 4 Strips are created with 4, 8, 12 and 16 DCSs each (notated as earlier as 5x4x4, 5x4x8, 5x4x12...). Clearly

[1]The problem is fixable, however, time constraints on the compiler writer have meant it was not possible for this analysis and must be left for future work

the larger arrays will include more configurations as greater DCS capacities are possible, this has a visible effect in some of the following overview graphs where metrics appear to be cyclic with a growing period as the larger array size sets include more data points. It should be noted that because of this and because not all data points can be visibly labelled, despite the data being discrete, continuous plots are used so as to more clearly visually portray the trends and demarcate the periods discussed.

Given the above to begin with the following ideals can be anticipated as a comparison.

- An increase in array size should result in a decrease in execution time as a result of a shortening in program row count (length) through an increase in execution parallel width. A limit on the performance gains should exist either where the inherent parallelism available within the application is encountered or where the additional communication costs (copy operations) required to operate at greater levels of parallelism out weighs the benefits.

- An increase in DCS capacity should result in a decrease in execution time as a result of fewer rows containing division. That is as more rows execute more division operations in parallel the number of rows that cost the longer latencies of division should decrease. As with the array size a limit will exist in the inherent division parallelism available in the application, and while less of a possibility the potential for communication costs to overwhelm parallel division gains too also exists in theory.

- A point of optimal efficiency, for a given application, should exist when optimising for power and for execution time, the two points may or may not be common.

Figure 5.10 gives a high level view of the effects of both DCS quantity and array size on, parallelisation achieved and resulting execution times (for a 100 year simulation using a time step of 0.25years). For ease of reading, specific details are only examined in the caption but two higher level points of significance follow. Firstly, the steep rate of decrease in execution time in the smaller arrays, slowing quickly to near inconsequential further performance gains (<1ms for corresponding DCS capacities). Secondly, the expected lower frequency periodic fluctuations in the series which are a visualisation of the step wise repetitive listing of DCS capacities, a stepped graph of array size is shown indicating the step widths. This shows at a high level how, within

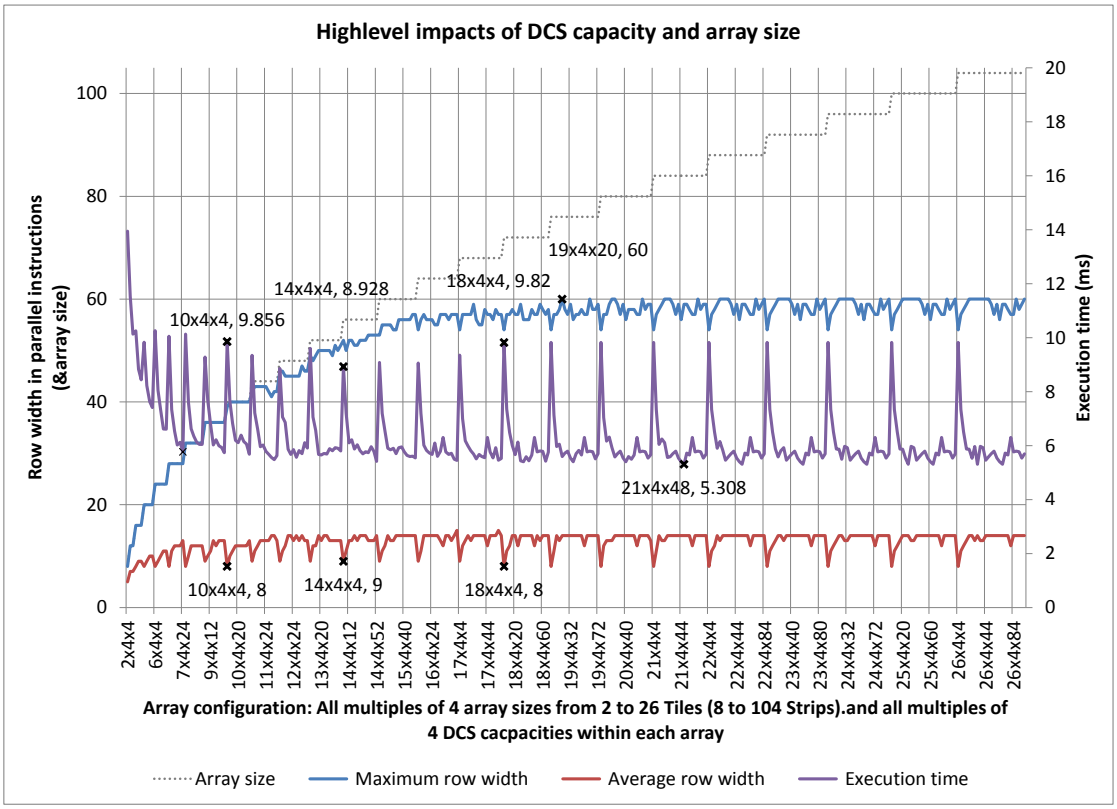


Figure 5.10 – Graph summary of the seven body solar system program compiled for each Fynbos configuration possible given a strip size of four. An obvious cap on the inherent parallelism is visible in the maximum row widths and average row widths series. In the case of this application, the maximum is 60 operations for at least one row in the program and is first reached in 19x4x20. Such maximum row widths must be infrequent, however, considering the much lower average row width series values. While the first instance of the lowest execution time (5.3ms) occurs in the 21x4x48, arrays much smaller and with far fewer DCSs achieve execution times within <1ms of this. While there are discernible dips in execution times of the 4 DCS capacity configurations (see two dips between the 10x4x4 and 18x4x4 before it plateaus), this graph is too high level to show if the same pattern is present for other DCS capacities.

a given array size, DCS capacity influences performance in an unexpected manner where greater DCS capacity can cause a drop in performance. Similarly, performance for the *x4x4 configurations visibly plateau on a local high rather than an asymptotic low, which is equally unexpected and perhaps occurring in other DCS capacities too.

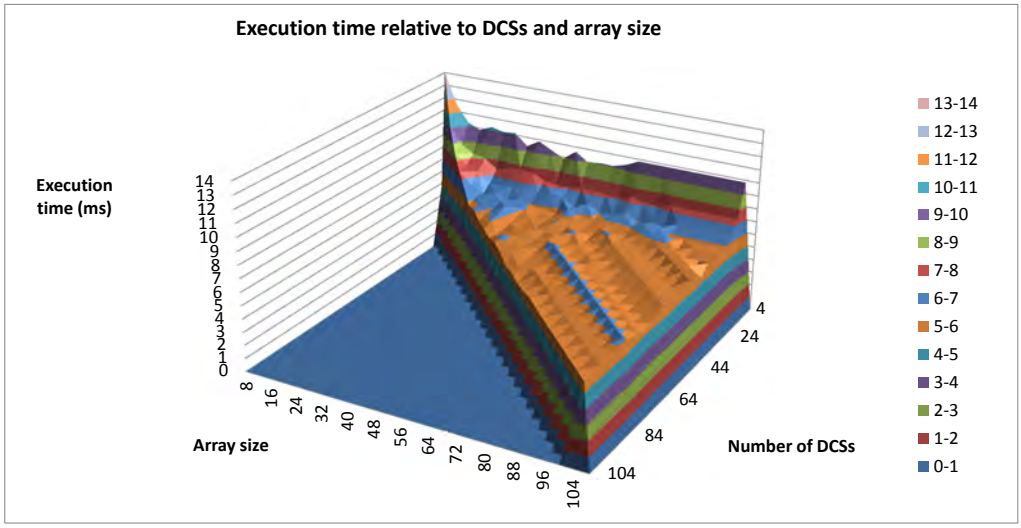
Plotting execution times for all configurations generated, Figure 5.11a again shows the initial steep improvement in performance with an increase in resources. The rough plateau it flattens out into illuminates a composite relationship between array size and DCS capacities and its effect on execution time. Figures 5.11b and 5.11c plot the same data but each removes one axis for clarity.

The regular ridge lines in the plateau show a linear response to specific ratios of array size to DCS capacity but this is not exhibited across the full range of configurations. Using Graph 5.11b, a notable knee around 16 DCSs is visible, showing that there is little more to be gained in using any more than 16 DCSs (for this application).

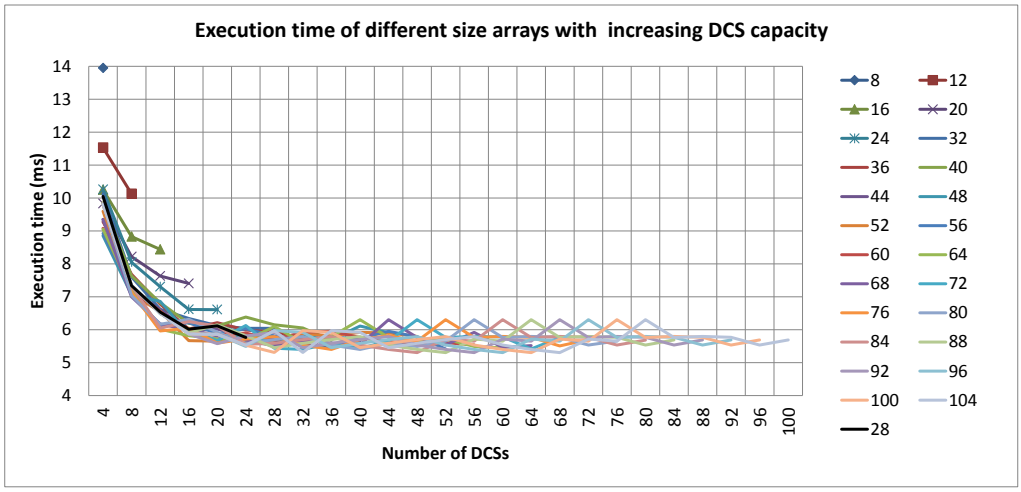
Graph 5.11c alternatively shows the plateauing in performance as different array sizes are reached for different lower DCS capacity arrays. The 16 DCS knee is instead visible here as a vertical gap of separation between those configurations with less than 16 DCSs and the rest which merge into much closer proximity floating between 5 and 6ms execution times. The knee in 5.11c less obviously appears to instead highlights 28 Strip arrays the hinge point in array size making a significant impact on execution times. The 28 Strip array size series is highlighted in black for further comparison in 5.11b too.

Both 5.11b and 5.11c show the regular ridge lines as shifting peaks occurring as once off highs in each configuration. Why performance should drop in larger or higher DCS capacity arrays is surprising and undesirable behaviour, and a matter for further investigation. It appears that in some cases, array size and DCS capacity dependant, APPRASE makes an error in judgement when parallelising the code. Whether this error relates to poor instruction promotion, excessive communication, or a lack of DCS capacity is so far indiscernible, the latter clearly plays a role considering the change in behaviour in the higher DCS capacity configurations.

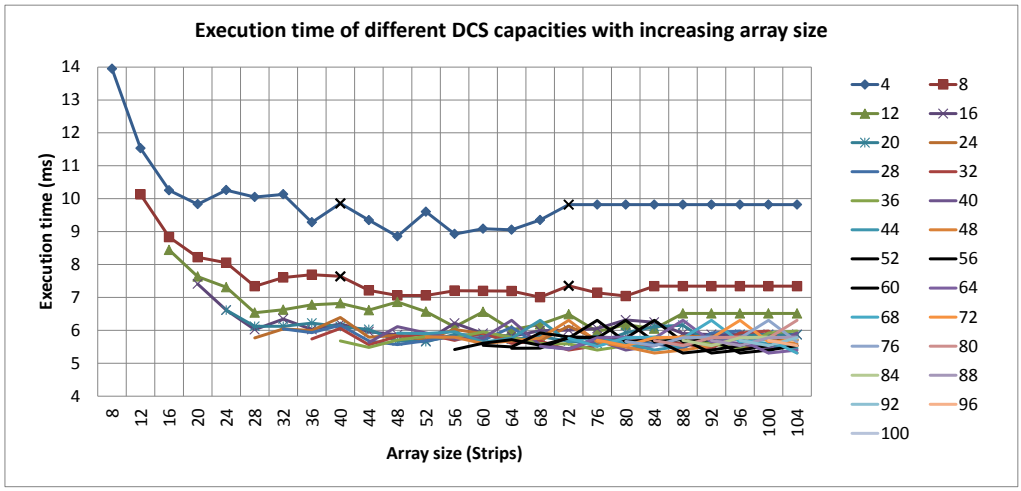
Admittedly the time differences under consideration here are less than a millisecond, but given that the program is only simulating 100 years and seven bodies and executes in between 5 and 14ms (between 7-20% of the total), it may be indicative of something that would manifest as a much more significant problem given a larger program and longer run time. Given only this application set, it is impossible to say



(a)



(b)



(c)

Figure 5.11 – Representations of the execution times for the seven body program compilations on uniformly incrementing Fynbos configurations. Graph (a) plots array size and DCS capacity on the two horizontal axis with execution time in the vertical. Graphs (b) and (c) contain the same data with one or the other horizontal axis removed.

whether these trends are evidence of an error in the APPRASE algorithm or if further analysis with larger problem sets would show them to simply be inconsequential.

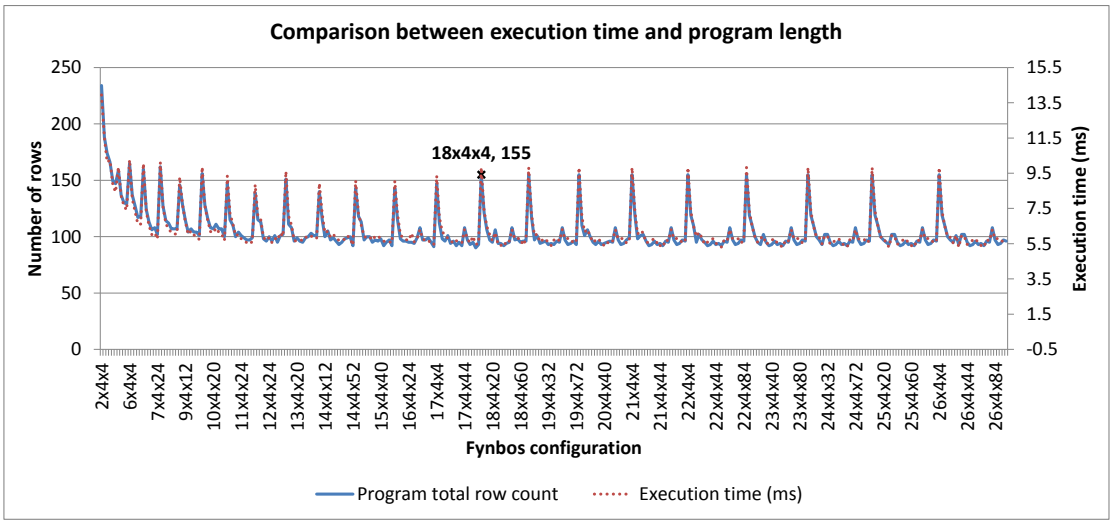
Assuming the later is not the case, with performance so dependant on exact array configuration Fynbos could only ever be used as a reconfigurable architecture. If each application required an exactly tuned array the question of pushing the design down into an ASIC with the performance gains of such as outlined earlier is quickly muted.

In considering the causes for these unexpected trends in performance the first two anticipated outcomes serve as a starting point that sees program length, communication trends, division, and parallel efficiency examined in the following. Execution time will increase if either the program length (number of rows) increases, or if the cumulative latency of a fixed number of rows is increased due to other factors, or a combination of both.

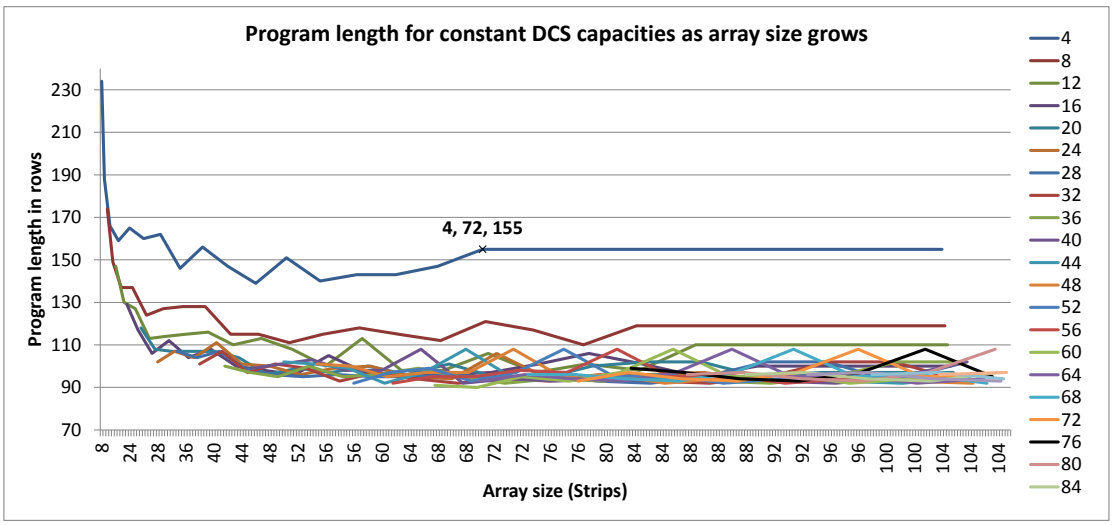
5.3.2.1 Program Length

Examining the first of these, program length, a larger array should enable shorter wider programs for a decrease in execution time, yet Figure 5.12a plots program length against the array configurations and the previously noted unexpected behaviours are again visible. Increased program row length, however, is not necessarily a conclusive indicator of decreased performance. Yet overlaying the execution time series on the same graph, also shows that two metrics track each other very closely. This suggests that the advantage to be gained through efficient division parallelisation is not generally appreciable relative to the total execution time, or that such advantage gains are accompanied by similar program length reduction, or that there is a mutual combination of both. The last of these being most likely considering the series are not identical in form only very similar.

Figure 5.12b shows how array size affects program length for each DCS capacity. The graph is too dense to see easily but if broken down there are three ranges of DCS capacities apparent 4:36, 40:80, 84:100 (see Figure A.2 for full plots of each). The first range sees program lengths plateau at local highs, the marked point in Figure 5.12a shows the 4 DCS series' plateau starting point. The second covers the shifting peaks also visible in the dense graph, these shift regularly with each increment of DCS capacity, the peak occurring at the next array size up (for instance at 10x4x64, 11x4x68,...). Finally the third sees a relatively near flat progression where neither DCS nor array size changes make any appreciable difference in program length. The same



(a)



(b)

Figure 5.12 – Graphs plotting the number of rows in the schedule produced by APPRASE for the seven body system on varying Fynbos configurations. In graph (a) these values are plotted for all configurations with the corresponding execution times overlaid on the same axis. Graph (b) plots the same program lengths in row numbers for constant DCS capacities as the array size grows.

ranges are visible in the equivalent execution time graphs, with slight differences in presentation.

5.3.2.2 Communication

The factors impacting program length are the same as those affecting parallel efficiency, copy operations, common division parallelisation, and parallelisation of other instructions. Beginning with the copy operations, while the number of copies fluctuates relative to array size and DCS capacity there is no direct correlation between these values and increased program length suggesting that communication costs are at least not directly responsible for the performance drops either. With no clear pattern it is difficult to succinctly show the lack of correlation. Figure 5.13 shows a surface plot of the number of copy operations used in the program for a each Fynbos configuration, comparison of this with the earlier given execution time surface plot (Figure 5.11a) which closely tracks program length shows they are not directly linked.

Independently removing each axis from the surface graph shows no obvious pattern in APPRASE's use of copy operations relative to array size and DCS capacity, and no similar pattern ranges as were found in program lengths. For further verification that copy operations do not affect performance as far as increasing program length in row count, Figure A.3 contains three profile plots of the program execution row widths. The first plots only copy operation widths, the second all operations, and finally one of all operations with copy operations subtracted. The program lengths for the various configurations are the same with and without the copy operations showing APPRASE is instead able to hide all communication in parallel width.

Figure 5.14 more clearly indicates that DCS capacity has a greater influence on copy operation use than array size. More significantly, however, it also reveals a key step at 10x4x4 from which point on all configurations, regardless of array size or DCS capacity, have at least one row in which 21 copy operations are performed in parallel. In conclusion therefore copy operations contribute to parallelism not program length impacting performance indirectly. There use will therefore impact apparent efficiency due to the increased parallelism. Given that there is no link between copy operations and program length, they also have no connection to the unexplained fluctuations, or the pattern ranges seen in program length. Finally, perhaps due to the limited size of application available, there is no clear evidence of APPRASE using copy operations to exploit increased array width only of it exploiting increased DCS capacity.

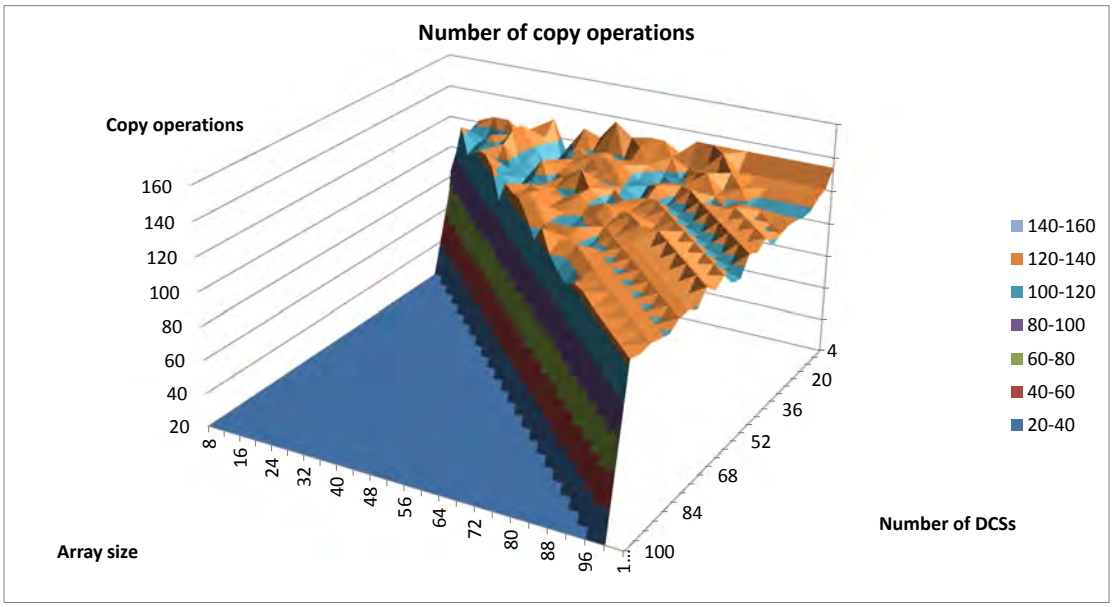


Figure 5.13 – Surface graph of the number of copy operations required to execute the seven body system on varying Fynbos configurations.

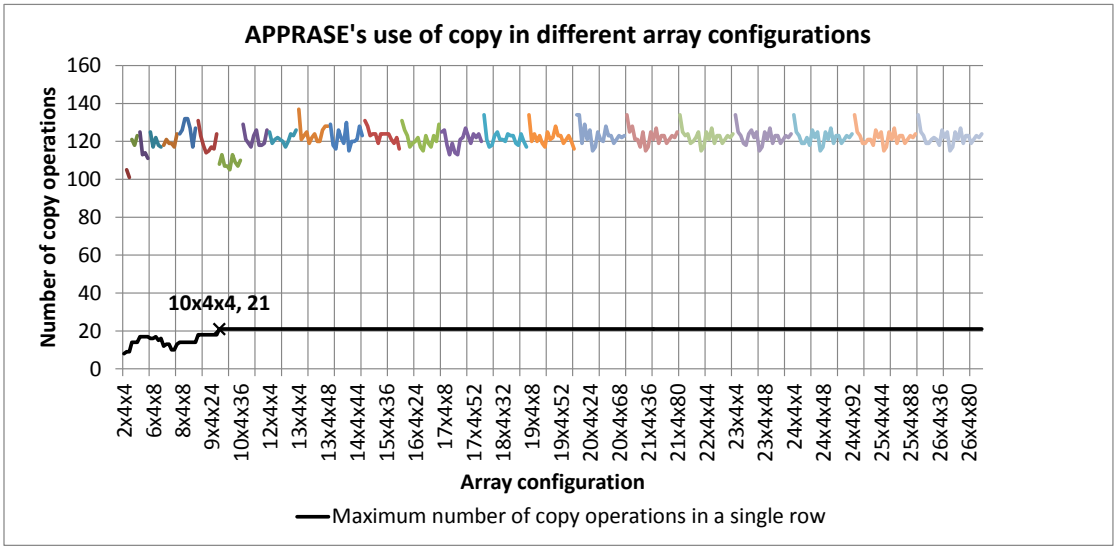


Figure 5.14 – Graph showing the number of copy operations required in various configurations of Fynbos for the seven body application. The coloured series each represent a constant array size for changing DCS capacity, and portray the total number of copy operations used in executing the seven body on each configuration. The black series instead plots the maximum number of copy operations used in a single row for each configuration. The first shows how DCS capacity rather than array size primarily influences the number of copy operations used. The step function at 10x4x4 in the second, however, shows a threshold in array size is reached from which point on to some degree a cap on the exploitable parallelism is reached. This is purely indicative and not conclusive as further discussions show.

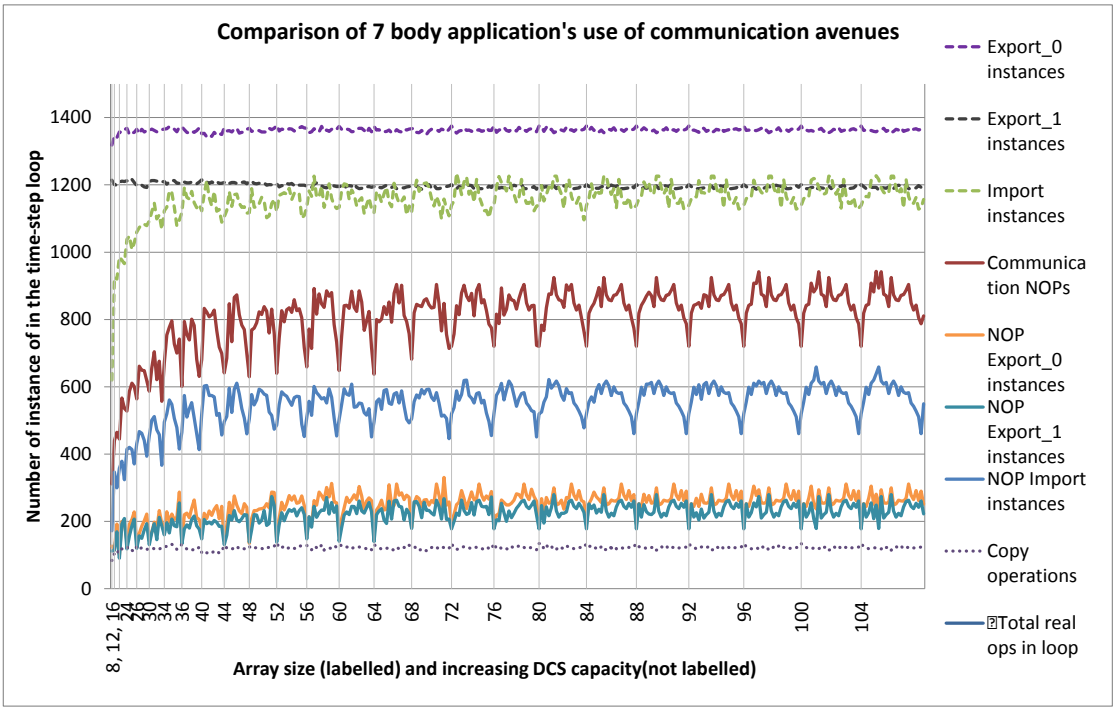


Figure 5.15 – A comparative plot of the different forms of communication used by APPRASE when scheduling the seven body solar system application. Values for all array configurations are plotted but only the smallest DCS capacity configuration for each array size is labelled on the x-axis, as with previous graphs DCS capacity for a given array size starts at four and increments in units of four. The Export_0 and Export_1 series refer to a Strip inter-Strip operand communications within a tile, while import refers to inter-Tile communications.

While APPRASE is able to hide copy communication operation in parallel width, more significantly the larger proportion of communication is carried out using the ability to move data while an operation is taking place. Due to how the schedule xml file is structured it is not reasonably possible to determine the number of dual purposing operations where data is fetched in one Strip for another Strip while the first Strip sources its operands elsewhere. However, the number of instance in which an instruction only causes inter and intra Tile communication take place with a NOP being executed by the sourcing Strip, is shown in Figure 5.15. The number of communication only NOP operations fluctuates with array size in a similarly unexplainable manner to other metrics, more detailed analysis of the influence of DCS capacity and array size shows a similar lack of pattern and at a high level a levelling-off of averages once once array sizes reach a certain magnitude.

The series labelled with the term export refer to a Strip fetching data from its lo-

cal memory. In the series labelled “Export_0 Instances” and “Export_1 Instances” (in dashed lines) represent all instances of data being fetched from a local memory and therefore are not representative of communication exclusively. In the case of “Import Instance” (also a dashed line), however, all instances represent a transfer of data between Tiles. For the seven body solar system application inter-Tile communication clearly dominates over intra-Tile communication but this is likely influenced by the compiler fault enforced limitation to increasing total Tile count rather than Strip counts per Tile. For the same reason intra-Tile communication between Strips shows as a near constant and influenced only by DCS capacity and not array size.

To eliminate false representations of intra-Tile communication within Tiles, those instructions containing NOP operations but which also fetch values from their memories are examined as they clearly must be doing so only so as to communicate a value to another Strip. Examining these series (in solid lines) reveals a number of behaviours.

- Communication NOP instructions are far more prevalent than explicit copy operations (dotted series), showing APPRASE is capable of not only hiding copy communication in parallel but also of utilising the dual functionality capabilities of Fynbos instructions.
- Inter-Tile imports remain more common than intra-Tile exports.
- Comparing the number of imports in communication only NOP instructions with the instances of imports in all non-NOP instructions (“Import Instances”), on average nearly twice (1156 vs 549) as many mathematical operation instructions as NOP only importing instructions, are performing both a mathematical operations and an import operations. APPRASE is clearly capable of utilising the dual operate and communicate functionality of Strips.

5.3.2.3 Division Capacity and Parallelism

If copy operations do not influence program length but do appear to be influenced by DCS capacity, Figure 5.16 shows there is a relationship between program length (or execution time) and parallelisation of division. For the range 8:44 of array sizes, APPRASE manages to gain performance as expected, an increase in DCS capacity results in a decrease in the number of rows performing division and thereby a reduction in net execution time. The array size ranges beyond this point achieve the same in the

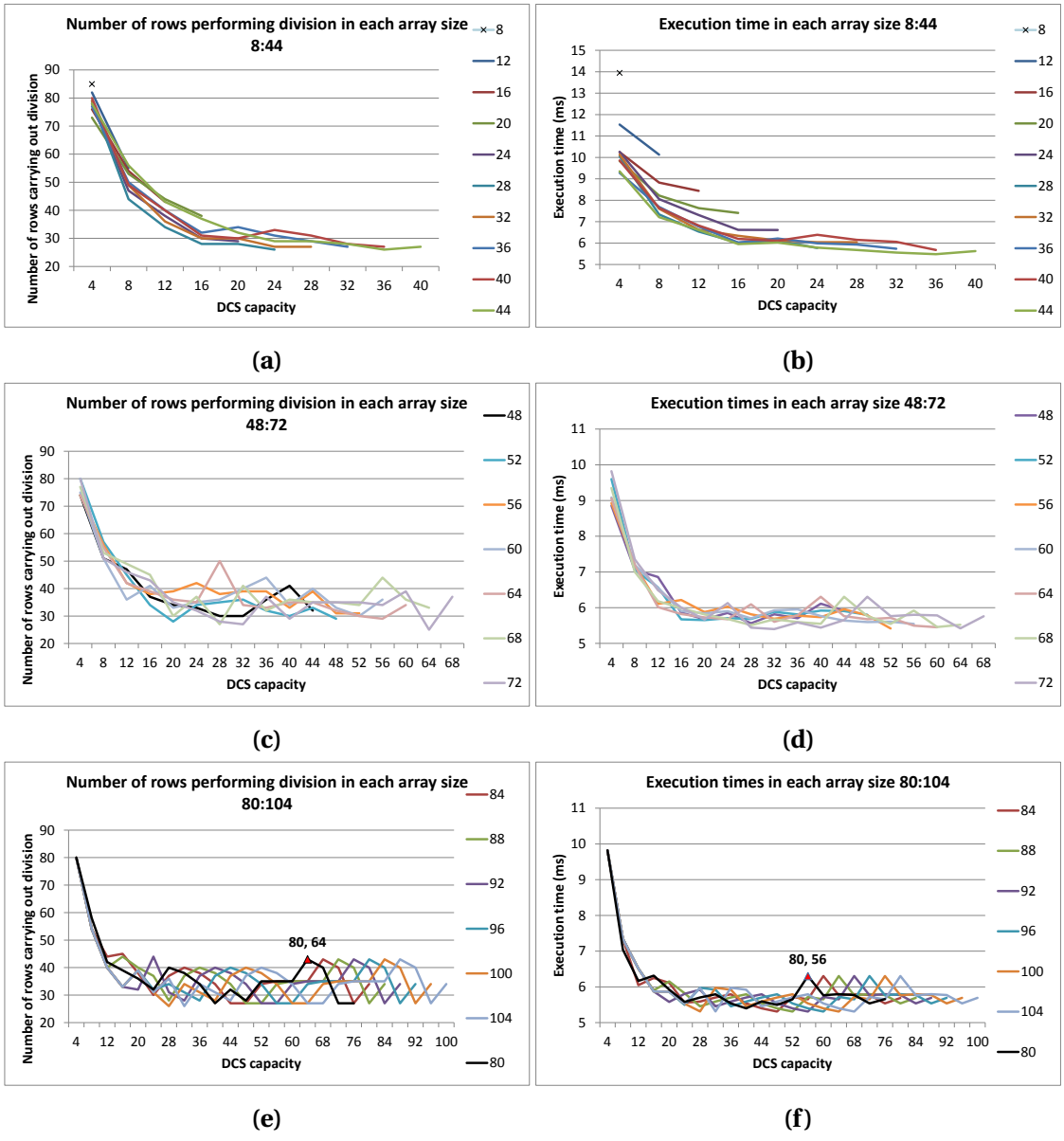


Figure 5.16 – These six graphs offer a comparative view of the number of rows performing division in various configurations (a,c,e), relative to the execution times for the same configurations (b,d,f). Series are presented for each array size relative to DCS capacity.

lower DCS capacities. However, beyond the 16 DCS knee the unexpected behaviour (of increased DCS capacities causing spikes in execution time or division row counts) returns. In the case of these the two metrics fail to correlate. For instance in Figures 5.16e and 5.16f, where parallel division is worst in the 80 Strip array (43 rows versus 27 at its best) is at 64 DCSs, yet execution time for the same size array spikes at 56 DCSs (6.3 vs a best 5.4 ms).

In conclusion therefore while DCS capacity obviously influences division parallelism (and up until configurations containing around 44 DCSs the number of rows containing division operations decreases relatively steadily) in larger arrays containing more DCSs the effect appears more random and is currently not understood. While this additional parallelism in the smaller arrays cannot be tied directly to any reduction in program length in rows, it is clearly linked to the reduction in execution time at least in the lower array sizes.

5.3.2.4 Power Efficiency

Having shown that increased DCS capacity and array size, with the accompanying increase in division parallelisation, improves the execution times it remains to include the equally important component of power consumption. Lacking accurate power data for executing the solar system on Fynbos efficiency in resource use is instead adopted as a substitute.

Figure 5.17 plots an efficiency measure *ALU packing efficiency*, achieved by APPRASE when parallelising the seven body solar system against the range of Fynbos configurations. ALU packing efficiency represents the ratio of *Total instruction*[1] *: Total instruction opportunities*, and is representative of how well APPRASE and an application have used the resources available.

[1]Packing efficiencies are calculated using the xml schedules produced by DAREA. The values reported here are conservative representing the efficiencies of the program main loop (a time step iteration) and excluding all NOP operations that exist purely to afford opportunities for Strip export or import of data. Only the main loop operations are included as the few additional statement exist purely to indicate to the LOAD file constructor where various initial condition data values should be loaded but play no part in execution. The NOP operations are excluded as operations performing Fynbos specific work rather than work specific to the application. As a point of reference the packing efficiency for the 7 body application running on the 7x4x16 configuration has a packing efficiency of 46%, if the NOP operations that serve only to enable data movement and loading are included this rises to 68%. On average inclusion of the communication NOPs raises efficiency by 12% with the gap decreasing with array size.

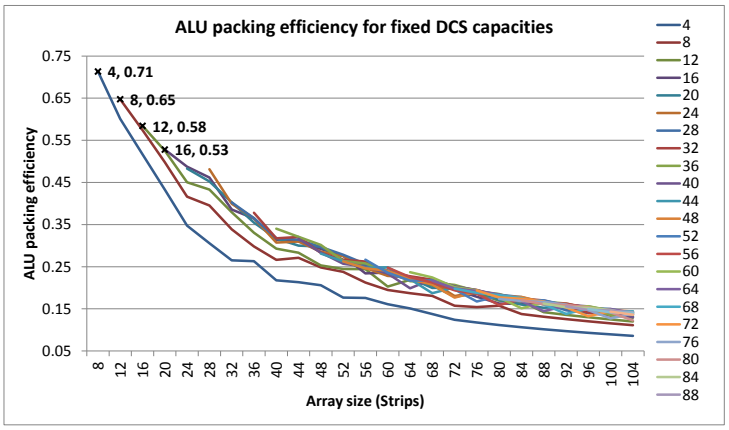


Figure 5.17 – ALU packing efficiency for fixed DCS capacities across a range of array sizes. ALU packing efficiency is calculated as $ProgramInstructions/\left[RowsInProgram * No.Strips * No.Tiles\right]$ The data point labels indicate in order; DCS capacity (series), and ALU packing efficiency(value).

This is most useful in comparing how different applications make use of the same resources, but Figure 5.17 uses it to reiterate once more the mutual dependence DCS capacity and array size have on one another in achieving performance and efficiency. The knee at 16 DCSs is again visible, this time as the merging of the series containing more than 16 DCSs, but more obvious is the rapid drop in efficiency with increasing array size. These values, however, need to be considered in conjunction with execution times. Unfortunately, to achieve the shortest execution times significant array size is needed but only very temporarily leading to what is measured here as inefficient resource use. As would be the case in hardware where dynamic and particularly static power losses increase on larger chips, a larger array, if not fully utilised throughout the application execution, carries a great deal of resources that are only used briefly for a very small portion of the work.

To consider both metrics together, Figure 5.18 plots ALU packing efficiency and execution time together for a range of Fynbos configurations around the apparent optimum for the seven body system (20-44 Strip arrays). As the horizontal represents increasing DCS capacity, based on the above it is expected that the execution time series (dotted lines) will merge beyond the 16 DCSs knee point, which they do around 6ms. Alternatively the efficiency series (solid lines), as expected, exhibit dropping efficiencies for larger arrays. Reading both the 28 Strip array appears to be a reasonable compromise for this application specifically at 16 DCS achieving 46% packing efficiency and 6.02ms execution time.

It has already been noted and is clearly visible in the data already examined that the seven body solar system does not stretch the resource limits of the available Fynbos configurations. But to show how APPRASE is capable of effectively using resources

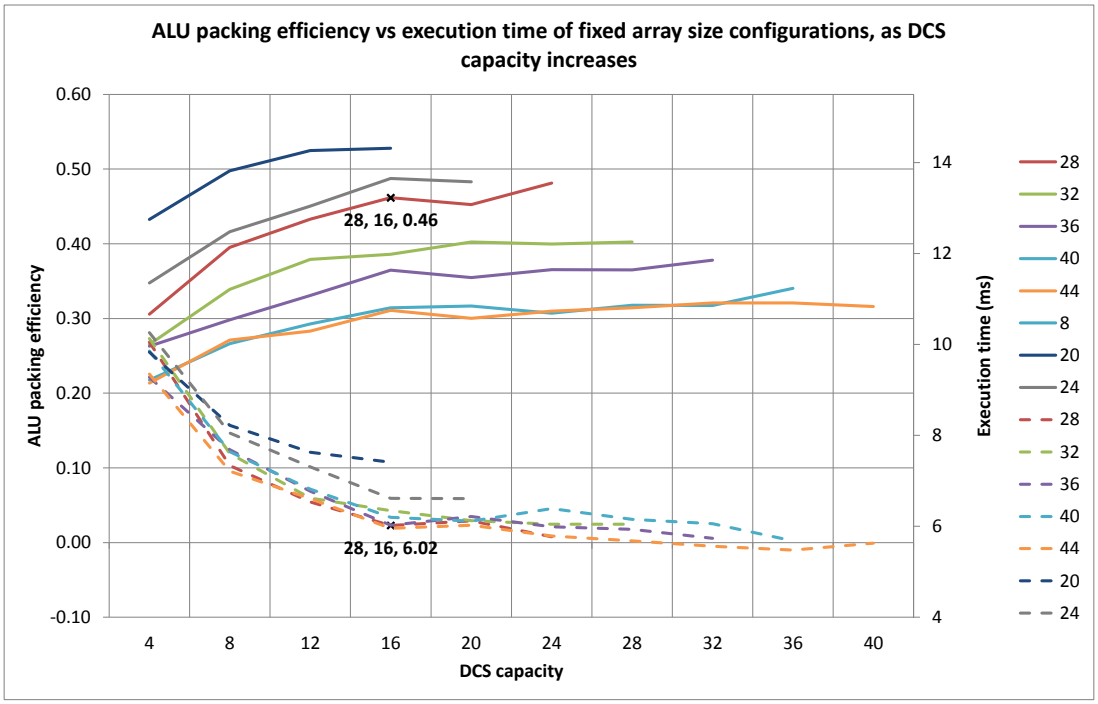


Figure 5.18 – In solid lines ALU packing efficiency for a limited range of array sizes is shown. On the second vertical axis and in dashed lines the corresponding execution times for the test case seven body system are shown. The horizontal axis represents the progressive increase in DCS capacity. Labels in the graph indicate in order, array size in Strips (series), DCS capacity (category), axis appropriate value

as a task is scaled, the following Figure 5.19 compares the ALU packing efficiency (solid lines) and execution times (dotted lines) of incrementing numbers of solar bodies simulated on various Fynbos configurations. As is to be expected execution time increases with problem size, but satisfyingly so does efficiency in a matching manner. The effect of array size shows the same efficiency drop with array size, but the 28 Strip 16DCS array of Figure 5.18, stands out as both offering the lowest execution times for all problem sizes and interestingly the smoothest scaling progression. The sharp transitions in the alternative series (all of which only support 4 DCSs) is perhaps a useful indicator of less than optimal configuration for an application where APPRASE is required to make borderline decisions.

5.3.3 APPRASE-Fynbos Real-world Performance

Finally while industry standard benchmark comparisons cannot be run the following compares the power and execution times of the solar system application on Fynbos to

Figure 5.19 – In solid lines ALU packing efficiency for a range of array configurations is shown as the application problem size is increased. On the second vertical axis and in dashed lines the corresponding execution time series are shown.

a standard server system representative of what a HPC node might consist of. The Fortran run on both systems is identical except that on the server native SQRT rather than Heron's method is called, and the output write statements are commented out. In the case of the latter, while the write statements cause Fynbos to copy its results back to the host this is done in parallel adding nothing to its execution time. Including the final time delay of sending the final packet and writing from the server's receive buffer to memory is difficult to measure and is unlikely to require cost time than if the server side code was required to write its results back to main memory on every iteration. To ensure sufficient run times for reliable power consumptions results simulations were set to run for 100 000 years using a four hour time step.

The Xeon is clearly considerably faster, between 100 and 4X times faster, depending on the number of bodies considered and if the FPGA or hypothetical Fynbos ASIC is the comparator. However, given the differences in clock speed (3.6GHz vs 400 and 100MHz), and the technology base (32nm vs 65nm), it is notable that the Xeon only achieves a 4.4X speed up relative to the hypothetical 65nm ASIC Fynbos and with only 7 bodies in the system. Given the qualifications around how the hypothetical ASIC

Table 5.8 – Execution time in minutes of the solar system application as its problem size is scaled. System specifications for the server system are as follows; 32nm 4 CPU (two hardware threads each) 3.6GHz Xeon X5687, 64GB DDR3, Ubuntu 12.04.3 LTS, gfortran 4.6.3 compiler optimisation level O2

Number of solar bodies	X5687 Xeon (32nm, 3.6GHz)	7x4x16 Fynbos (65nm FPGA, 100MHz)	7x4x16 Fynbos (Hypothetical 65nm ASIC, 400MHz)	Slow down relative to the Fynbos ASIC
1	0.2	18.0	4.5	22.05
2	0.2	19.7	4.9	24.50
3	0.5	23.3	5.8	11.60
4	1.0	27.4	6.8	6.80
5	1.6	34.6	8.6	5.40
6	2.3	44.9	11.2	4.90
7	3.1	55.0	13.7	4.40

clock speed was derived this result could be either an over or under estimate.

While execution times lengthen in all cases as the problem size is increased, the difference between performances on the different systems narrows showing Fynbos-APPRASE's ability to take advantage of the increasing available parallelism in the applications. Admittedly, the Xeon version is entirely sequential and is compiled with minimal optimisations (using the gfortran -funroll-loops option worsened run times by ~3X). But this is an appropriate comparison when positioning Fynbos where it would be used under similar circumstance, a node within a cluster running sequential code, traditional parallelism having already been applied at a higher level so far as node divisions are concerned. The only further step that might be carried out is threading the application into between 2 and 16 threads, which is still shy of the 28 Strips Fynbos is utilising.

Examining the power costs involved Dell's OpenManage Server Administrator tool which monitors CPU power consumption registered 140W when idle with only a single user logged in, and 189W during execution of all of the application sizes. Subtracting the two the Xeon consumes 49W when executing any one of the solar system application (1-7 bodies). This is in stark contrast to the <4W conservatively of Fynbos on the V5 (including I/O) to do the same.

Therefore comparing the 7 body application on the Xeon and the V5 FPGA (see Table 5.8); a 18X (55.0/3.1) slowdown achieves a 12X (49/3.9) decrease in power costs. More impressively, comparing the Xeon with the hypothetical 65nm FPGA; a 4X (13.7/3.1) slow down achieves a 176 (49/(3.9/14)) decrease in power costs.

5.4 Summary and Conclusions

Considering the research goals originally set out in Chapter 1, the following summarises what has been shown in this chapter regarding the hardware components, software components, and joint APPRASE-Fynbos system.

5.4.1 Hardware

- The limits on Fynbos' scaling on the V5 (maximum 8x7x4) were due primarily to a lack of logic resources leading to congestion and a maximum clock speed of 100MHz. On the V7 (maximum 10x11x4) while not limited by logic the number of fast interconnects between regions meant higher speeds or larger arrays were not possible. In both cases the limit was reached due to limits in the substrates that may have been removed given an ASIC, but both also hint at the fundamental design limit anticipated by the design compromises made.
- Within the scaling limits encountered division capacity was not found to be a primary scalability limiting factor (consider that both a 100 STrip array with 97 DCSs and a 110 Strip array with 4 DCSs met timing on the V7) and only enacted a falling 12% drop in efficiency for a 24 fold increase in DCS capacity (in the ASIC implementation of 10x10x4 vs 10x10x97).
- In the case of the interconnect exponentially scaling costs can be tempered by employing higher Tile counts with fewer Strips but obviously only to the degree an application and required performance will allow for.
- Aside from the costs in BRAM it was shown that increasing memory capacity has a near inconsequential impact on resource consumption. In an ASIC context it could therefore be scaled to as large as is worth while for a given application and considering the trade-off will be for further Strips.
- Fynbos achieves the many-core goal of spending the largest proportion of resources on ALUs with 46% of power consumption going to the ALUs and only

26% to control. Considering the division module in a division capable ALU consumes 78% of that ALU's power this ratio is conservative and would favour the ALUs even more in an array supporting division in more than 10% of its ALUs. In resource use this comparison is more complicated on FPGAs but ALUs consume 43% and 56% of LUTS and registers while 48% and 9% of same are spent on control.

- Where the power costs of data movement are forecast to be a point of considerable concern in designing exascale systems, Fynbos' interconnect spends only 15% of its small power expenditure on the interconnect and control paths.

5.4.2 Software

- Again considering the goal of scalability and this time including efficiency, the largest array justifiably scaled and implemented on a hypothetical 28nm ASIC, is a 387MHz 4096 Fynbos providing 60.5GFLOPS at 62GFLOPS/W (calculated with the given assumptions and constraints these values should be considered as only approximate and within the correct order of magnitude). While this efficiency is higher than any other architecture reviewed (the next closest is Epiphany at 50GFLOPS/W), its raw performance is absurdly low. This indicates that with the current design, in order to compete multiple Fynbos cores would have to be joined in one of the earlier proposed manners, or a significantly higher clock speed would be needed dropping its efficiency.

- The limiting factor on Fynbos' raw performance is the latency of each operation. While pipelining as a technique was discarded in Chapter 4 due to lack of compiler support, this is clearly needed if Fynbos were to ever be competitive. While the compiler deficiency might have been compensated for entirely in hardware, it would have gone against the minimalist design goals.

- While peak theoretical performance is not the only relevant measure, baring a means of comparative benchmarking, the commercial architecture Kalray, on specifications alone appears as if it would offer greater performance than Fynbos for similar power efficiency and has the added advantage of already being available for purchase. Its FPU and ALU pipelines, and support of C/C++, POSIX, and OpenMP, while requiring code modification to match their model, make it a recognisable competitor for the HPC domain. As a commercial product lacking

an application benchmarking opportunity or low-level details of the architecture a conclusive comparison cannot be drawn.

- Considering the expected software outcomes:
  - An increase in array size did result in a decrease in execution time through a combination of shortening in program row count and an increase in execution parallel width.
  - Parallel width was most influenced by DCS capacity rather than array size, up until a maximum for a given application was reached. Beyond this point parallel width, program row length and the resulting execution times tended to plateau.
  - The increase in parallel width of divisions resulted in a decrease in execution times.
  - Within the effective plateauing, however, an unaccounted for behaviour exhibits in which APPRASE produced longer execution time schedules for some particular more highly resourced arrays. In execution time, program length, and average row widths, these spikes are regular but have no obvious cause. The configuration in which they occur are innocuous other than to be a regular distance apart. Neither the quantity of copy operations or number of rows containing division shows any direct correlations to such. While of little consequence in the given application, with no resolution on the cause further testing is warranted to determine such and if larger applications will see similar behaviour amplified until it is of concern.
  - Communication is not a limiting factor on performance as APPRASE is able to effectively hide it in parallel width, avoiding any lengthening in the programs row counts.
  - A point of optimal efficiency and performance does exist for the given application, although which exact Fynbos configuration is best will depend on the power budget and maximum acceptable execution time as the two factors do not converge on a common optimal configuration.
  - Execution times for the same application appear to fluctuate in an unexpected manner with unexplained peaks for more and less resourced arrays. While apparently having a small effect in this case study, they are plausibly indicative of a problem in the APPRASE algorithm that would cause larger

differences in a larger or longer application. If this is the case arrays would either need to be finely tuned to an application (muting the question of an ASIC implementation) or the cause resolved and removed.

  - Program lengths track execution time fluctuations nearly identically meaning the fluctuations are at least linked to program length and how APPRASE parallelises code and not a function solely of division parallelisation.
  - There are different ranges of array configuration (in array size and DCS capacity) within which these unexplained spikes in execution time occur with a regularity. Examining the number of divisions per row three ranges of shared behaviour are visible (array size ranges:0:44, 48-72, 80-104 Strips). Examining execution time three ranges of shared behaviour are also visible (DCS capacity ranges:4-36, 40-80, 84-100 DCSs). However, the peaks in execution time or dips in division row width, do not correlate with each other when comparing the same array configurations

- Concerning communication:

  - Copy operations have no obvious link to the fluctuations in execution time and have no direct impact on performance as far as making the program longer in row count is concerned.
  - DCS capacity influences copy operation use more than array size, with low DCS capacity causing a spike in copy presumably to enable operand to reach them, once a DCS threshold is reached, however, the quantity of copy operations shows no explicable trend.
  - There is no clear evidence of APPRASE using copy operations to exploit increased array width.
  - Copy operation use does not play a major role in influencing program length and thereby does not directly at least influence execution time. This means APPRASE appears able to hide communication costs, which presumably indirectly facilitates performance gains through greater parallelism although this is not directly visible as a correlation.
  - Copy operations do contribute significantly to parallelism (showing APPRASE is effective at hiding the operations in parallelism) and so will be a secondary force increasing efficiency.

  - Communication of data via the provisions for such in instructions is utilised in ten times as many instance than the copy operations showing APPRASE is further capable of utilising the dual operate and communicate functionality of Strips.
  - The quantity of inter-Tile communication revealed is clearly influenced by the constraint on Strip counts within Strips suggesting the number of copy operations may also have been artificially lowered.
  - Communication only NOPs, on average, added a further 60% more instructions to a program's contents but are also hidden in parallelism.
  - A step in the plot of maximum number of copy operations in a single row at 10x4x4 (Figure 5.14), does not appear to directly influence or cause any of the changes in behaviour discussed.

- Concerning division capacity:

  - For the seven body solar system application DCS capacity has a significant and predictable influence on execution time that is greater than the impact of array size. It reduces execution time particularly by parallelising more division operations into fewer rows as opposed to only by finding greater parallelism.

- Concerning efficiency:

  - In the absence of accurate power values the ALU packing efficiency metric provides an analogous metric. As was found with execution times, array size and DCS capacity affect consumption. Smaller arrays achieve higher efficiencies than larger but also result in longer execution times, both metrics need to be read together.
  - The optimal configuration for the solar system appears to be 7x4x16, for 1 to 7 bodies. For larger body counts presumably there will be a further threshold beyond which greater array and DCS capacity will be beneficial.
  - APPRASE is capable of using resources efficiently as problem size scales. It will be most effective if given a deduced optimal configuration. Sporadic behaviour with an increasing problem size may be indicative of less than optimum Fynbos configurations, the specifications of which will inevitably change as certain thresholds are reached with the problem size.

  - Neither DCS or array size have an obviously greater impact on efficiency or performance.

- Comparing the seven body application on the Xeon and the V5 FPGA a 18X (55.0/3.1) slowdown achieves a 12X (49/3.9) decrease in power costs. More impressively, comparing the Xeon with the hypothetical 65nm FPGA a 4X (13.7/3.1) slow down achieves a 176 (49/(3.9/14)) decrease in power costs.

- Without an appropriate comparator it is not possible to conclude on whether or not APPRASE is finding all parallelism available and if it is producing the best (performance and energy efficiency wise) schedule. The above does, however, show it to be capable of concealing overt communication in parallel with functional operations (the number of data transfers carried out within instructions is not available data), be capable of exploiting greater parallelisim and achieving greater efficiencies as a problem size grows, and be capable of exploiting additional width and DCS capacity in conjunction to gain a reduction in execution time through optimal operator parallelism in addition to parallelism in general.

- Finding the optimal configuration for a given application, with the current version of APPRASE, will involve a degree of testing and will mean selecting an acceptable efficiency and execution time range.

- Finally in the solar system case study, comparison of the X and Y co-ordinate results generated on different show a difference. When running the Fortran program on an x86 based CPU, and the APPRASE schedule of the same fortran program on Fynbos or its simulator, in single precision, the results begin to diverge as the solar simulation is run for lengthening time periods. While it is known that single precision is often insufficient for HPC applications, it needs to be established whether this divergence is due to APPRASE or not. Comparing the results generated on the FPGA with those of the Fynbos simulator shows no difference at all in results comparing double or single precision versions), indicating that the underlying FPGA FPU hardware is not the cause despite its incomplete implementation of the IEEE754 FP standard. Comparing results generated on a FPGA implemented Fynbos configuration supporting double precision FPUs, with a version of the application using REAL*8 variables running on an x86 system, also shows no difference. Comparing results generated on a FPGA implemented Fynbos configuration supporting only single precision FPUs, with

a version of the application using REAL*4 variables running on an x86 system, however, reveals the growing divergence noted.[1] As an anticipated problem, for which software double precision routines that exploit many-cores abundance of PEs has already been proposed, and brevity this problem is not examined in more detail here. It must be noted, however, as a point of concern and further work recommended, theoretically proving that the schedule produced by APPRASE is not any more of an influencing factor in this error beyond the consequences of reordering that result from floating point not guaranteeing associativity.

[1] See Appendix A.3 Graph A.1 and the spreadsheets attached digitally as indexed in AppendixB

Chapter 6

Conclusions

This work began by presenting the theoretical benefits of the many-core model. This model was proposed by others in the face of the ILP, memory, power and utilisation walls being encountered by industry, and the resulting clear need for a new approach to computational processing. For any new approach to counter these barriers it will need to offer significant energy efficiency gains alongside a practical avenue to further performance. Fabrication process improvements, application parallelisation, and many-core architectures can theoretically achieve the required hardware efficiency requirements of this by counteracting a drop in clock speed with extensive fine-grained parallel processing in the form of logically simple PEs. Switching to any new model will entail encountering the problem of legacy code, amongst many others, given the nature of the many-core model the problem of software migration and programmability is further exacerbated by the real difficulties involved in finding, coding, and scheduling the fine degree of parallelism required.

Having been provided the APPRASE pipeline, a compiler possibly capable of not just finding the degree of parallelism many-core requires and of using static scheduling to replace hardware control, but also of operating on specifically legacy serial Fortran codes, an attempt to apply this model to HPC was selected as an avenue of investigation. HPC lends itself to being a many-core target, this is partly due to its users familiarity with parallelism, but more significantly because of the possibility of side stepping Amdahl's law with Gustafson's law as a consequence of the nature of the applications.

Fynbos as the result of this investigation, is an attempt at creating both an architecture representative of the hypothesised many-core model, and also one capable of meeting the needs of the HPC domain. Targeting HPC means both FP support

and MIMD operation are necessities which add complications and restrictions to the hardware design. Having considered the implications of both the design goals of many-core and HPC, Fynbos is adapted to be scalable but not infinitely so, and is instead targeted at the position and role of a HPC cluster node core.

In designing Fynbos to match APPRASE (rather than creating both a compiler and architecture simultaneously) compromises were inevitable and have lead to recommendations for changes in both. The following first summarises these recommendations again, following which conclusions regarding Fynbos, APPRASE, and how each serves the many-core hypothesis are given separately. Finally the conclusions that have been drawn throughout the document are drawn together so as to answer the research questions originally made in Chapter 1, and based on this draw final conclusions regarding the given hypothesis.

6.1 Further Work Required

The following section is divided into recommendations pertaining solely to hardware, solely software, and finally those that are only relevant if effected in both.

1. Changes and additions required to the hardware only of Fynbos.
   (a) Implementing hardware support for fine-grained power control of PEs.
2. Changes and additions required only for the software stack:
   (a) Software support for utilising excess parallel hardware to circumvent hardware defects.
   (b) The unexplained behaviour exhibited in the performance of the solar body application on some Fynbos configurations remains unexplained. Further testing and tuning are needed to determine why, in specific instances, greater resource capacity in both array size and DCSs, leads to a worsening in performance of the same application, relative to similarly but differently resourced arrays. All tests thus far indicate there is no single source cause but that array size, array configuration, and DCS capacity are all influencing factors in how APPRASE responds. Further it has been established that the sporadic increases in execution times are not solely due to an increase in program row length, or poor parallel scheduling of division, or otherwise

similar latency operations but a combination of both. Due to testing limitations it has not been possible to determine if this behaviour is simply a function of a small application in large Fynbos configurations or if it is indicative of an underlying problem.

(c) Fix the problem restricting flexibility in number of Strips per Tile.

(d) In order to target a cluster of Fynbos nodes as proposed, the software infrastructure to use such needs to be developed. Such code would be responsible for splitting codes into units for issue to different Fynbos array nodes based on standard compiler pragmas, and correctly distributing the data accordingly. In addition, this would most likely also entail targeting and selecting between a range of architectures in addition to Fynbos arrays.

(e) Along with the above point, if any significantly large codes were to be run on Fynbos, in which multiple memory load stages were required (for instruction or data memories or both), the suggested investigation into and implementation of a re-load protocol is needed. Such an investigation may also require consideration of not just quantity but also speed of reloading with memory bound applications in mind.

(f) As stated in Chapter 1, regardless of APPRASE's capabilities parallelism in an algorithm will ultimately be limited by the underlying algorithm. New algorithms taking into consideration both the high and low levels of parallel execution are needed.

(g) In line with the many-core target of simple PEs, in order to provide sufficient precision for HPC applications, software implemented double precision (and other application specific precisions) was proposed as the appropriate means considering also the abundance of PEs. Implementing this is both something that has already been done by others and beyond the scope of this work but is nevertheless still noted as required future work.

(h) A simple further optimisation possible would be to include compile time checking for expressions for which the results are already known such as $a*0$.

3. Further work required involving both the hardware and software stage in conjunction:

(a) Implementing hardware and software support for multi-Fynbos array designs with multiple I/O ports and control units.

(b) Implementing hardware and software for frequency scaling on Fynbos arrays in the hypothesised multi-Fynbos system.

(c) Implementing hardware and software support for a form of predication such as the one described that exploits the abundance of parallel PEs available.

(d) Implementing hardware and software support for pipelining and overlapping of execution cycles.

6.2 Fynbos and Many-core Architecture Conclusions

Conclusions drawn in the following regarding a hypothetical porting of Fynbos to an ASIC, and how such a chip would preform, should be read within the context of the assumption made (see Section 5.2.1.2). These assumptions while justified and reasoned do mean a measure of error is expected in the exact values derived. However, in all cases the assumptions taken were all cautious and conservative, and the conclusions drawn are believed to be appropriate.

Matching the hardware requirements set out in Chapter 1 regarding the many-core model:

1. The Fynbos Strip is representative of an appropriate HPC many-core PE in simplicity and capabilities lacking only further functionality not possible in the given environment and situation.

2. The clock speed on the FPGA is lower than in similar designs but if ported to an ASIC simple scaling means the clock speed would be on par with these other designs which are also implemented as ASICs. The low clock speed of 400MHz for such a design is in line with the many-core target.

3. The design, while not infinitely scalable, should theoretically scale to 4096 PEs on a ASIC meeting the target of thousands of PEs. This is considered more appropriate than an infinitely scalable design as the data sharing capabilities of the Fynbos design are essential if the degree of fine-grained parallelism required to achieve performance is to be found in applications, and it is already known that applications can be distributed with a degree of module independence and

there is no reason to forgo exploiting such higher levels of parallelism. This scalability is largely a result of static scheduling enabling the use of a distributed memory system (rather than a shared cache) without incurring the hardware or communication costs of managing it.

4. As best as can be determined Fynbos is a novel contribution particularly to the many-core architecture design domain. Fynbos' combination use of static scheduling with its unique interconnect (a compromise in long lines and interconnectivity, managed by static scheduling and programmability) and use of distributed memory, achieve significant scalability without sacrificing the data sharing capacity necessary if the level of fine-grained parallelism required in applications is to be exploitable. Further novel contributions include the use of software implemented operators in a many-core to achieve simplified hardware and find greater application parallelism, and the incorporation of overlaid communication into the same control and data pipeline as the arithmetic pipeline.

5. The interconnect does not exploit any domain specific communication characteristic. While certain applications will perform better on it than others, in a reconfigurable environment the Tile-Strip configuration will provide flexibility in tuning the array to an application's communication pattern. Unfortunately given the circumstances described this aspect could not be as fully explored as desired in Chapter 1.

6. This investigation has only added weight to the conclusion that a reconfigurable target is inappropriate for a many-core HPC architecture. This conclusions is draw based on a number of factors; the design limits encountered on the FPGAs targeted, the efficiency and area gains possible with an ASIC, and the determination that the interconnectivity rather than logic costs of implementing extensive division support was the scalability limiting factors in Fynbos. In the case of the last of these the reasoning is as follows. Without configurability DCS capacity will need to be statically enabled in all or most PEs, but in conjunction with fine-grained power control this division logic could be used as dim silicon. This would therefore be following the advisory to exploit available transistors for efficiency and performance gains where ever possible given their abundance. The net result being that if division logic in each PE (and other similar operators within reason) is not going to restrain scalability there is no further reason to

retain reconfigurability. A final factor in this equation that cannot be evaluated here, however, is the cost of increased static power from a larger chip.

7. The similar advisory to spend unused transistors on memory as a form of dim silicon, can also be applied to Fynbos as the costs of larger memories are inconsequential.

8. Fynbos achieves the many-core goal of spending the largest proportion of resources on ALUs with the conservative ratio showing 46% of power consumption going to the ALUs and only 26% to control.

9. Fynbos succeeds at keeping the interconnect power costs low (15% of total consumption).

10. While only just over 100 PEs were achievable on the available FPGAs, area, power, and speed scaling parameters predict that up to 4096 Strips could be implemented on an ASIC meeting the original goal of "hundreds to thousands of PEs".

11. Fynbos is likely to run into the same problems as Imagine [Ahn et al., 2004] did in that when an application is memory rather than compute bound its performance will drop when the problem set fails to fit on-chip. Multi-Fynbos systems may be able to avoid this but in such cases a streaming or memory appropriate architecture is likely to simply be more appropriate.

12. Comparing Fynbos to the only other architecture to rely so extensively on the compiler and find use in HPC, where VLIW architectures have traditionally struggled to fill the width of the processors, Fynbos is much less restricted in what can be parallelised as all PEs can support the same range of operations and even when not the case the width of Fynbos is so much greater than a VLIW (lacking the restriction to fitting everything into a single instruction) that the slight limitation on DCS capacity for instance is less of a hindrance. Further, without a cache Fynbos is able to predict the latencies of everything thereby finding additional exploits and avoiding any complexities of stalling or flushing data paths. Where Itanium has support for branch prediction and predication, however, Fynbos is still lacking such.

6.3 APPRASE and Programming for Many-core Conclusions

Matching the requirements regarding many-core programming set out in Chapter 1 and elaborated on more fully in Chapter 2:

1. **Software Design**

   (a) There is a clear line of influence on APPRASE from the original AD-10 architecture and the requirements of that system and its applications. However, given the results available, there is no obviously negative impact from such, only constraints and avenues taken that may or may not be worse than another. The largest impacting hereditary factor is most likely the use of static scheduling and the performance hurdles dynamic code can encounter with such.

   The use of static scheduling and its exploitation of detailed low-level knowledge of the target architecture (Fynbos in this case) represents a design decision (albeit an imposed one). This decision enables Fynbos' unique programmable communication infrastructure, which tests have shown is used effectively by APPRASE to overlay communication and operations in parallel with one another. Further, static scheduling enables APPRASE to ensure optimal temporal and spacial locality of data, allowing for the distributed memory system needed for scalability to be used.

   The disadvantages of static scheduling, however, have not been compensated for in this work, and without implementing the revisions proposed above it is difficult to judge whether they are a removable barrier or not. It can be argued, however, that static scheduling in combination with the correct dynamic code mitigating tactics is highly advantageous to a many-core based design. This is based on; the results presented in the literature reviewed which argued that static scheduling affords many optimisations appropriate to a many-core that are otherwise unavailable, what static scheduling has enabled in this work, and the complexities of many-core programming explored in this work.

   (b) In various ways APPRASE meets or would easily fit into a model created to meet six out of the eight ideals laid out for a many-core programming model in Chapter 2. It is difficult to separate how each of these is handled by static scheduling and the use of autoparallelisation, each being a

result of the combination created in APPRASE. But broadly, the complexities of many-core memory management, multi-level PE hierarchy, and heterogeneity (of PEs within a many-core but not of different processor architectures within a larger system) are handled by the scheduling process. And ease of programmability, support of heritage codes, and avoidance of the complexities of extreme parallelism debugging, are each side-stepped by enabling the traditional sequential programming model to be used unguided as regards parallelisation, and through the support of the Fortran language.

The two ideals not met are enabling fine-grained control and portability. As discussed in the original statement, the first of these is tied to the division of labour between software and hardware and given the system design created, hardware control not easily possible in the prototyping environment used was necessary, and has therefore been identified as recommended further work for both domains. In the case of the second, APPRASE is clearly not language portable in its current form, whether the same techniques applied in the APPRASE pipeline could be applied to other languages requires further study to determine. Regarding functional portability, and heterogeneity of multiple different architectures in the same system. With such low level knowledge being key to performance the characteristic of architecture independence is difficult to add. APPRASE's flexibility and ability to adapt to varying configurations of Fynbos does indicate that it could be made to accommodate a limited range of hypothetical many-core architectures, but it is unlikely that it would be able to optimise the schedules sufficiently for such without extensive reconstruction work providing it with the low-level factors in each to be accounted, such as how important parallelising division operations with each other is in Fynbos. This target of multi-architecture support is a higher level systems software problem and not within this work's domain of targeting programming a node in a larger cluster. Instead the work reviewed on DSLs [Chafi et al., 2011] and a compiler capable of selecting the optimal hardware target is noted as a promising approach into which APPRASE and similar systems might be fitted. Doing so, however, would loose the benefits of Fortran support. Finally performance portability, which is intrinsically linked to this matter

of functional portability, is most likely with an automated process if functional portability can be achieved, but given that finding performance is now a matter of finding ever very low level optimisation available, it is unlikely.

(c) It is note-worthy that in comparison with other parallelising compilers, the lack of a cache specifically opens up many opportunities for performance gaining exploits and optimisations as there is no possibilities of stalls.

(d) The potential of automatically parallelising a language so dominant to the HPC community as Fortran was a primary motivation for this work. This has now been shown to be possible, that extracting parallelism from APPRASE requires no guidance from the user on best parallel opportunities, and the code therefore requires no editing other than to ensure appropriate I/O and operator calls.

2. **Software Analysis**

(a) The theoretical peak performance of Fynbos is far below its competitors due to its long operation latency rather than clock speed. Pipelining of such is necessary if it is to be competitive, this can be implemented with minimal additional hardware and further software control.

(b) While Fynbos is more efficient than any of these competitors, relative to the stated required exascale pico Joules per operation, the predicted ASIC efficiencies for even Fynbos remain an order of magnitude out at nano Joules per FLOP.

(c) APPRASE is capable of using additional DCS capacity and array size for increasing parallel gains and thereby reducing application execution time, this is also true when the application problem set is increased in size.

(d) APPRASE is capable of hiding communication work behind arithmetic operations, using this means of data movement ten times more often than the more explicit copy operations.

(e) APPRASE uses the more obvious form of data movement communication, copy operations, predominantly to reach division capable Strips rather than to exploit greater array size. These operations are always hidden in parallel width and do not increase the latency of an application but rather reduce

it by improving the use of available DCS capacity and parallelising greater numbers of division operations with each other.

(f) APPRASE is able to optimise an application schedule by scheduling as many division operations in parallel as possible. Part of enabling an optimal version of this is having an array size sufficiently large to enable a handful of very wide execution rows that occur perhaps even only once in a complete application.

(g) APPRASE is effective at transforming sequential Fortran code into parallel code for execution on the Fynbos architecture. The sequential seven body solar system application executes on the hypothetical Fynbos ASIC in 12 times the time but using 149x less power by executing with an average parallel row width of 12.[1]

6.4 Hypothesis Conclusions

In light of the above therefore, the following conclusions to the research questions are therefore the hypothesis (re-stated here) are drawn:
That a many-core architecture designed to match the legacy APPRASE Fortran autoparallelising pipeline, can achieve improved compute efficiency over a traditional superscalar sequential core, while still providing performance when executing HPC codes, making the APPRASE-Fynbos system approach a viable option in tackling the walls currently facing the computing industry.

Based on the literature reviewed, including the papers first proposing the many-core model and other architectures with similar characteristics, Chapter 1 described a range of characteristics expected and desirable in a many-core. It has been shown how Fynbos has targeted these characteristics as design goals throughout the development process, and succeeded in matching the majority of them. Where characteristics have not been met the reasons for such have been explained (predominantly limitations of the development environment and APPRASE). Further fully MIMD operation is possible as specified, as are the other numeric and generality of functionally requirements of the HPC domain. The largest discrepancy between the many-core model proposed in the literature and what has been described here is the lack of inclusion of a high-speed traditional core within the sea of simple PEs. Instead Fynbos

[1] On the 7x4x16

has presented not as a sea of PEs to be run with a traditional sequential core but as a co- processor, which could nevertheless still be implemented on the same chip as a traditional core. If such were to be done, however, based on the manner of operations APPRASE-Fynbos takes and literature reviewed, it appears inadvisable to incorporate a shared memory between the traditional CPU and Fynbos other than in a manner that ensures atomic access by both.

Considering the challenges of programming a many-core architecture in a HPC environment, this work was limited to the problems of programming the many-core only, proposing only that traditional cluster management and application distribution tools be retained to continue exploiting higher levels of parallelism and facilitating integration of new architectures. At the level addressed therefore, is has been shown that APPRASE is effective at transforming sequential Fortran code (highly prevalent in HPC legacy codes which will need porting to any new architecture) into parallel code for execution on the Fynbos architecture. Given APPRASE's development stage this is, however, not true for the complete scope of the Fortran language but a practical subset. APPRASE is further effective at hiding data movement (inter-PE communication) in both parallel width (not increasing application latency) and in arithmetic instruction execution using Fynbos' unique dual functionality instructions that enable both communication and arithmetic operations to be encoded in a single instruction, control path and data path. This resulted in a high efficiency and low power cost of communication, a point of particular concern in the case of a many-core architecture in which computation is spread out.

Provided further application testing and algorithm development finds the parallelism required, many-core architectures are clearly an important step in the correct direction with the hypothetical ASIC Fynbos achieving greater efficiencies than any of the other architectures reviewed. But while Fynbos, Epiphany, Kalray, and P2012 all achieve ratings in the tens of GFLOPS/W, this translates into nano Joules per FLOP, and is therefore still an order of magnitude out from the exascale required pico Joules per operation. Further, even when scaled to operate on an ASIC Fynbos, while efficient, is far from producing the compute performance of even current systems. Given all the further alterations proposed for Fynbos and APPRASE, and the existing gap in performance between Fynbos and the likes of Epiphany, Kalray and P2012, it is implausible that the Fynbos design would ultimately significantly out perform any of these in efficiency or theoretical peak performance. Given Fynbos's architecture and control model and the domain specific design of these competitors, however, it is

plausible that Fynbos may be able to outperform these in practical HPC application execution. Due to testing limiting circumstances beyond our control this could not be thoroughly examined or established leaving only design review and discussion as a basis for evaluations.

The many-core hypothesis, however, has a scaling back in processing hardware complexity as a fundamental principle. Determining exactly how far to scale and where the line between too simple and simple enough for optimal efficiency is, is a matter of design deduction but also experimentation. In this work it appears that this line was overshot as Fynbos is too simple to achieve the necessary performance. Measured changes have therefore been proposed, however, which will increase hardware complexity slightly but will also add significantly to performance predominantly through further partnering software exploits. While, having overstepped the mark, this work has shown that pushing the burden of control to a compiler, in conjunction with a many-core architecture, can increase efficiency, and be done in a fashion that could realistically see adoption but the HPC industry.

In carrying out this work the following novel contributions are made. A many-core processor for general purpose HPC has been developed and its use performance metrics analysed for work as an HPC cluster node. The use of a Fortran autoparallelising compiler, which exploits static scheduling, to port sequential Fortran to a many-core architecture for the HPC domain is unique, and has demonstrated the potential advantages of such an approach considering the programmability and architecture challenges facing industry.

Finally, therefore, Fynbos can be considered a prototype processor that meets the requirements for many-core classification as understood in this work. Designed to match the legacy APPRASE Fortran autoparallelising pipeline, it has significantly improved compute efficiency over traditional superscalar sequential cores, but together APPRASE-Fynbos as a system is not yet developed enough to deliver the compute performance necessary for HPC codes. Also, even this improved efficiency is found to still be below the requirements for exascale computing.

References

Adapteva. Epiphany Architecture Reference (G3), 2012. URL http://www.adapteva.com/support/docs/e3-reference-manual/. 59

Adapteva. Epiphany SDK Reference, 2013. URL http://www.adapteva.com/support/docs/esdk-manual/. 59

JH Ahn, W.J. Dally, and B. Khailany. Evaluating the imagine stream architecture. ... *Computer Architecture* ..., pages 14–25, 2004. doi: 10.1109/ISCA.2004.1310760. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1310760http://dl.acm.org/citation.cfm?id=1006734. 36, 42, 200

Massimo Aldinucci, Marco Torquati. FastFlow: programming muli-core. URL http://sourceforge.net/projects/mc-fastflow/. 69

Farhana Aleen and Nathan Clark. Commutativity analysis for software parallelization. *ACM SIGPLAN Notices*, 44(3):241, February 2009. ISSN 03621340. doi: 10.1145/1508284.1508273. URL http://dl.acm.org/citation.cfm?id=1508284.1508273. 71, 84

A Amaricai and O Boncalo. Implementation of very high radix division in FPGAs. *Electronics Letters*, 48(18):1107–1109, 2012. ISSN 0013-5194. doi: 10.1049/el.2012.0721. 223

AMD. AMD PowerTune Technology, 2011. URL http://www.amd.com/us/Documents/PowerTune_whitepaper_WEB.pdf. 53

AMD. AMD Grapics Cores Next Architecture, 2012. URL http://www.amd.com/us/Documents/GCN_Architecture_whitepaper.pdf. 52

AMD. AMD FirePro S10000, 2013. URL http://www.amd.com/us/Documents/FirePro_S10000_Data_Sheet.pdf. 49

AMD. What is Heterogeneous System Architecture (HSA)?, n.d. URL http://developer.amd.com/resources/heterogeneous-computing/what-is-heterogeneous-system-architecture-hsa/. 50

Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. doi: 10.1145/1465482.1465560. URL http://doi.acm.org/10.1145/1465482.1465560. 3

Sebastian Anthony. Intel unveils 72-core x86 Knights Landing CPU for exascale supercomputing, 2013. URL http://www.extremetech.com/extreme/171678-intel-unveils-72-core-x86-knights-landing-cpu-for-exascale-supercomputing. 50

Krste Asanovic, Ras Bodik, Bryan Catanzaro, Joseph Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, Lester Plishker, John Shalf, Samuel Williams, and Katherine A Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html. 3, 5, 164

Marc Baboulin, Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Julie Langou, Julien Langou, Piotr Luszczek, and Stanimire Tomov. Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications*, 180(12):2526–2533, December 2009. ISSN 00104655. doi: 10.1016/j.cpc.2008.11.005. URL http://dx.doi.org/10.1016/j.cpc.2008.11.005. 130

L Benini, E Flamand, D Fuin, and D Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 983–987, 2012. doi: 10.1109/DATE.2012.6176639. 58

S Benker, S Pllana, J Larsson, P Tsigas, A Richards, R Namyst, B Bachmayer, C Kessler, D Moloney, and Sanders. The PEPPHER Approach to Programmability and Performance Portability for Heterogeneous many-core Architectures. In *Parallel Computing*, 2011. 68, 75

Geoffrey Blake, Ronald Dreslinski, and Trevor Mudge. A survey of multicore processors. *IEEE Signal Processing Magazine*, 26(6):26–37, November 2009. ISSN 1053-5888. doi: 10.1109/MSP.2009.934110. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5230801. 34

W Blume, R Doallo, R Eigenmann, J Grout, J Hoeflinger, and T Lawrence. Parallel programming with Polaris. *Computer*, 29(12):78–82, December 1996. ISSN 0018-9162. doi: 10.1109/2.546612. 82

Mark Bohr. A 30 Year Retrospective on Dennard's MOSFET Scaling Paper. *IEEE Solid-State Circuits Newsletter*, 12(1):11–13, 2007. ISSN 1098-4232. doi: 10.1109/N-SSC.2007.4785534. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4785534. 1, 11, 12

Uday Bondhugula, Albert Hartono, J Ramanujam, and P Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375595. URL http://doi.acm.org/10.1145/1375581.1375595. 82

REFERENCES

Shekhar Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, 1999. ISSN 02721732. doi: 10.1109/40.782564. URL http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=782564. 1

Shekhar Borkar. Thousand core chips. In *Proceedings of the 44th annual conference on Design automation - DAC '07*, page 746, New York, USA, June 2007. ACM Press. ISBN 9781595936271. doi: 10.1145/1278480.1278667. URL http://dl.acm.org/citation.cfm?id=1278480.1278667. 22

Shekhar Borkar. The Exascale challenge. In *Proceedings of 2010 International Symposium on VLSI Design, Automation and Test*, pages 2–3. IEEE, April 2010. ISBN 978-1-4244-5269-9. doi: 10.1109/VDAT.2010.5496640. URL http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=5496640. 11

Shekhar Borkar and Andrew A Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May 2011. ISSN 0001-0782. doi: 10.1145/1941487.1941507. URL http://doi.acm.org/10.1145/1941487.1941507. 11

Michael Wolfe ,Brent Leback, Douglas Miles. Tesla vs. Xeon Phi vs. Radeon A Compiler Writer's Perspective. *PGI Insider*, September 2013. URL http://www.pgroup.com/lit/articles/insider/v5n2a1.htm. 78

Romain Brillu, Sébastien Pillement, Aymen Abdellah, Fabrice Lemonnier, and Philippe Millet. FlexTiles: A Globally Homogeneous but Locally Heterogeneous Manycore Architecture. In *Proceedings of the 6th Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '14, pages 3:1—-3:8, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2471-7. doi: 10.1145/2555486.2555489. URL http://doi.acm.org/10.1145/2555486.2555489. 58

Mihai Budiu, Girish Venkataramani, Tiberiu Chelcea, and Seth Copen Goldstein. Spatial computation. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XI, pages 14–26, New York, NY, USA, 2004. ACM. ISBN 1-58113-804-0. doi: 10.1145/1024393.1024396. URL http://doi.acm.org/10.1145/1024393.1024396. 24, 27

CAPSL. Open64, 2014. URL http://www.open64.net/. 85

CASPER. CASPER: Collaboration for Astronomy Signal Processing and Electronics Research, 2009. URL https://casper.berkeley.edu/wiki/ROACH_Architecture. 115, 116

B Catanzaro, A Fox, K Keutzer, D Patterson, Bor-Yiing Su, M Snir, K Olukotun, P Hanrahan, and H Chafi. Ubiquitous Parallel Computing from Berkeley, Illinois, and Stanford. *Micro, IEEE*, 30(2):41–55, 2010. ISSN 0272-1732. doi: 10.1109/MM.2010.42. 11, 22

Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming - PPoPP '11*, page 35, New York, New York, USA, February 2011. ACM Press. ISBN 9781450301190. doi: 10.1145/1941553.1941561. URL http://dl.acm.org/citation.cfm?id=1941553.1941561. 68, 80, 202

B L Chamberlain, D Callahan, and H P Zima. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007. ISSN 1094-3420. doi: 10.1177/1094342007078442. URL http://dx.doi.org/10.1177/1094342007078442. 74

Leland Chang and Wilfried Haensch. Near-threshold operation for power-efficient computing?: it depends... In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1159–1163, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1199-1. doi: 10.1145/2228360.2228573. URL http://doi.acm.org/10.1145/2228360.2228573. 61

Yonggang Che, Chuanfu Xu, and Zhenghua Wang. Analyzing the Efficiency and Bottleneck of Scientific Programs on Imagine Stream Processor by Simulation. *2008 IEEE International Symposium on Parallel and Distributed Processing with Applications*, pages 89–98, December 2008. doi: 10.1109/ISPA.2008.17. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4725139. 41

Kiyoung Choi. Coarse-Grained Reconfigurable Array: Architecture and Application Mapping. *IPSJ Transactions on System LSI Design Methodology*, 4:31–46, 2011. 34, 56

Clearspeed. CLEARSPEED WHITEPAPER: CSX PROCESSOR ARCHITECTURE. Technical report, ClearSpeed, 2007. URL https://www.cct.lsu.edu/~scheinin/Parallel/ClearSpeed_Architecture_Whitepaper_Feb07v2. 54

John Collins. An algorithm for instruction scheduling for parallel computation. 1986. 95

Sharma Ashutosh Collins John, Farrimond Brian. APPRASE: Automatic Parallelisation of FORTRAN to Run on an FPGA. In *Summer Computer Simulation Conference*, Edinburgh, Scotland, 2008. 98

Convey. Convey computer, 2013. URL www.conveycomputer.com. 8, 57

CriticalBlue. Criticalblue Accelerating Embedded Software, n.d. URL http://www.criticalblue.com/. 67

R Cytron, J Ferrante, B K Rosen, M N Wegman, and F K Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 25–35, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2. doi: 10.1145/75277.75280. URL http://doi.acm.org/10.1145/75277.75280. 84

William J. Dally, Timothy J. Knight, Ujval J. Kapasi, Francois Labonte, Abhishek Das, Patrick Hanrahan, Jung-Ho Ahn, Jayanth Gummaraju, Mattan Erez, Nuwan Jayasena, and Ian Buck. Merrimac. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing - SC '03*, page 35, New York, New York, USA, November 2003. ACM Press. ISBN 1581136951. doi: 10.1145/1048935.1050187. URL http://dl.acm.org/citation.cfm?id=1048935.1050187. 36, 43

Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. Cetus: A Source-to-Source Compiler Infrastructure for Multicores. *Computer*, 42(12):36–42, December 2009. ISSN 0018-9162. doi: 10.1109/MC.2009.385. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5353460. 82

F de Dinechin and B Pasca. Designing Custom Arithmetic Data Paths with FloPoCo. *Design Test of Computers, IEEE*, 28(4):18–27, July 2011. ISSN 0740-7475. doi: 10.1109/MDT.2011.44. 131

École Polytechnique Fédérale de Lausanne. Scala, 2014. URL www.scala-lang.orghttp://www.scala-lang.org/. 80

T. J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, June 1971. ISSN 0029-599X. doi: 10.1007/BF01397083. URL http://www.springerlink.com/content/h918421wt4480530/. 130

R.H. Dennard, F.H. Gaensslen, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, October 1974. ISSN 0018-9200. doi: 10.1109/JSSC.1974.1050511. URL http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1050511. 2

Javier Diaz, Camelia Munoz-Caro, and Alfonso Nino. A Survey of Parallel Programming Models and Tools in the Multi and Many-core Era. *IEEE Transactions on Parallel and Distributed Systems*, 2012. ISSN 1045-9219. doi: 10.1109/TPDS.2011.308. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6122018&contentType=Early+Access+Articles&sortType=asc_p_Sequence&filter=AND(p_IS_Number:4359390)&rowsPerPage=50. 71, 72, 74

R G Dreslinski, D Fick, B Giridhar, Gyouho Kim, Sangwon Seo, M Fojtik, S Satpathy, Yoonmyung Lee, Daeyeon Kim, N Liu, M Wieckowski, G Chen, D Sylvester, D Blaauw, and T Mudge. Centip3De: A 64-Core, 3D Stacked Near-Threshold System. *Micro, IEEE*, 33(2):8–16, 2013. ISSN 0272-1732. doi: 10.1109/MM.2013.4. 62

Jing Du, Xuejun Yang, Guibin Wang, and Fujiang Ao. Scientific Computing Applications on the Imagine Stream Processor. *Advances in Computer Systems Architecture*, 4186(Lecture Notes in Computer Science):38–51, 2006. URL http://dx.doi.org/10.1007/11859802_5. 41

Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceeding of the 38th annual international symposium on Computer architecture - ISCA '11*, volume 39, page 365, New York, New York, USA, June 2011. ACM Press. ISBN 9781450304726. doi: 10.1145/2000064.2000108. URL http://dl.acm.org/citation.cfm?id=2000064.2000108. 5

Christian Fabre, Iuliana Bacivarov, Eth Zürich, Ananda Basu, Martino Ruggiero, David Atienza, and Éric Flamand. PRO3D: programming for future 3D manycore architectures. In *Proceedings of the 2012 Interconnection Network Architecture: On-Chip, Multi-Chip Workshop*, INA-OCMC '12, pages 47–50, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1010-9. doi: 10.1145/2107763.2107776. URL http://doi.acm.org/10.1145/2107763.2107776. 62

Brian Farrimond and John Collins. Dimensional Inference Using Symbol Lives. In *International Conference on Software Engineering Theory and Practice*, number X, Orlando, Florida, USA, 2007. 99

GASPI. Global Address SPace Programming Interface GASPI, 2014. URL http://www.gaspi.de/en. 74

Geoff Gastor. Project Denver-based SoC due in 2015, 2013. URL http://techreport.com/news/24530/. 50

GCC Wiki. Automatic parallelization in Graphite, 2009. URL http://gcc.gnu.org/wiki/Graphite/Parallelization. 86

Mark Gebhart, Bertrand A Maher, Katherine E Coons, Jeff Diamond, Paul Gratz, Mario Marino, Nitya Ranganathan, Behnam Robatmili, Aaron Smith, James Burrill, Stephen W Keckler, Doug Burger, and Kathryn S McKinley. An evaluation of the TRIPS computer system. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 1–12, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-406-5. doi: 10.1145/1508244.1508246. URL http://doi.acm.org/10.1145/1508244.1508246. 24

E. O Gilbert and R. M Howe. Design considerations in a multiprocessor computer for continuous system simulation. In *National Computer Conference*, pages 385–394, Anaheim, 1978. Afips Press. 94

Robert E Goldschmidt. *Applications of division by convergence*. Msc, Massachusetts Institute of Technology, 1964. URL http://hdl.handle.net/1721.1/11113. 223

N Goulding-Hotta, J Sampson, Qiaoshi Zheng, V Bhatt, J Auricchio, S Swanson, and M B Taylor. GreenDroid: An architecture for the Dark Silicon Age. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pages 100–105, 2012. doi: 10.1109/ASPDAC.2012.6164926. 7

Green500. Ranking the World's Most Energy Efficient Supercomputers., 2013. URL www.green500.org. 8

Bruce Greer, John Harrison, Greg Henry, Wei Li, and Peter Tang. Scientific computing on the Itanium processor. *Sci. Program.*, 10(4):329–337, December 2002. ISSN 1058-9244. URL http://dl.acm.org/citation.cfm?id=1240058.1240063. 103

John L Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988. ISSN 0001-0782. doi: 10.1145/42411.42415. URL http://doi.acm.org/10.1145/42411.42415. 10

Linley Gwennap. ADAPTEVA : MORE FLOPS , LESS WATTS. *Mircroprocessor Report*, (June), June 2011. URL http://www.linleygroup.com/. 59

T R Halfhill. Ambric's new parallel processor. *Microprocessor Report*, pages 1–8, October 2006. URL www.MPRonline.com. 33

Tom Halfhill. Elixent Improves D-Fabrix. *Mircroprocessor Report*, 2005. URL http://linleygroup.com/newsletters/newsletter_detail.php?num=4352. 34

REFERENCES

S Hanson, B Zhai, K Bernstein, D Blaauw, A Bryant, L Chang, K K Das, W Haensch, E J Nowak, and D M Sylvester. Ultralow-voltage, minimum-energy CMOS. *IBM Journal of Research and Development*, 50(4.5):469–490, 2006. ISSN 0018-8646. doi: 10.1147/rd.504.0469. 61

Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Toward Dark Silicon in Servers. *IEEE Micro*, 31(4):6–15, July 2011. ISSN 0272-1732. doi: 10.1109/MM.2011.77. URL http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=5967003. 5

Tim Harris and Satnam Singh. Feedback Directed Implicit Parallelism. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 251–264, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-815-2. doi: 10.1145/1291151.1291192. URL http://doi.acm.org/10.1145/1291151.1291192. 79

L Hochstein, J Carver, F Shull, S Asgari, V Basili, J K Hollingsworth, and M V Zelkowitz. Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, page 35, November 2005. doi: 10.1109/SC.2005.53. 67

Houman Homayoun, Vasileios Kontorinis, Amirali Shayan, Ta-Wei Lin, and Dean M. Tullsen. Dynamically heterogeneous cores through 3D resource pooling. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12. IEEE, February 2012. ISBN 978-1-4673-0826-7. doi: 10.1109/HPCA.2012.6169037. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6169037&contentType=Conference+Publications&sortType=desc_p_Publication_Year&searchField=Search_All&queryText=Multicore+CPUs:+Processor+Proliferation. 58

R M Howe. Analog computers in academia and industry: a history of analog computing at the University of Michigan and the founding of Applied Dynamics International. *Control Systems, IEEE*, 25(3):37–43, June 2005. ISSN 1066-033X. doi: 10.1109/MCS.2005.1432597. 94

B Hutchings, B Nelson, S West, and R Curtis. Comparing fine-grained performance on the Ambric MPPA against an FPGA. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 174–179, 2009. doi: 10.1109/FPL.2009.5272505. 34

Wen-mei Hwu, Shane Ryoo, Sain-Zee Ueng, John H Kelm, Isaac Gelado, Sam S Stone, Robert E Kidd, Sara S Baghsorkhi, Aqeel A Mahesri, Stephanie C Tsao, Nacho Navarro, Steve S Lumetta, Matthew I Frank, and Sanjay J Patel. Implicitly parallel programming models for thousand-core microprocessors. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 754–759, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-627-1. doi: 10.1145/1278480.1278669. URL http://doi.acm.org/10.1145/1278480.1278669. 14, 79

IBM. Cell Broadband Engine, 2012. URL https://www-01.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine. 36

IBM. X10: Performance and Productivity at Scale, 2013. URL http://x10-lang.org/. 69, 74

IEEE. IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Std 754-1985*, page 0_1, 1985. doi: 10.1109/IEEESTD.1985.82928. 131

Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Number April. Intel, 2012a. 222

Intel. *Intel Itanium Processor 9500 Series Reference Manual*. Number July. Intel Corporation, 2012b. 38

Intel. Automatic Parallelization with Intel Compilers. Technical report, Intel, 2012c. URL http://software.intel.com/en-us/articles/automatic-parallelization-with-intel-compilers. 86

Intel. Intel Fortran Compiler XE 13.1 User and Reference Guide, 2013a. URL http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/fortran-mac/. 76

Intel. Intel Xeon Phi Coprocessor datasheet, 2013b. URL http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-phi-coprocessor-datasheet.pdf. 8, 49, 53

Intel. Intel Xeon Phi Coprocessor datasheet, 2013c. URL http://www.intel.com/content/dam/www/public/us/en/documents/performance-briefs/xeon-phi-product-family-performance-brief.pdfhttp://download-software.intel.com/sites/default/files/managed/b5/83/intel-xeon-phi-systemssoftwaredevelopersguide.pdfhttp://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-phi-coprocessor-datasheet.pdf. 53

Intel. Intel Cilk Plus, 2013d. URL https://www.cilkplus.org/. 69

Intel. Intel Parallel Studio XE Suites, 2013e. URL software.intel.com/en-us/intel-parallel-studio-xe. 67

Intel. Intel Threading Building Blocks, 2013f. URL https://www.cilkplus.org/. 69

Intel. Intel Array Building Blocks, 2014. URL http://software.intel.com/en-us/articles/intel-array-building-blocks. 69

Nikolas Ioannou and Marcelo Cintra. Complementing user-level coarse-grain parallelism with implicit speculative parallelism. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-44 '11*, page 284, New York, New York, USA, December 2011. ACM Press. ISBN 9781450310536. doi: 10.1145/2155620.2155654. URL http://dl.acm.org/citation.cfm?id=2155620.2155654. 79

ISSCC. ISSCC 2013 TECHNOLOGY TRENDS, 2013. 4

S Jain, S Khare, S Yada, V Ambili, P Salihundam, S Ramani, S Muthukumar, M Srinivasan, A Kumar, S K Gb, R Ramanarayanan, V Erraguntla, J Howard, S Vangal, S Dighe, G Ruhl, P Aseron, H Wilson, N Borkar, V De, and S Borkar. A 280mV-to-1.2V wide-operating-range IA-32 processor in 32nm CMOS. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pages 66–68, 2012. doi: 10.1109/ISSCC.2012.6176932. 61

C.-P. Jeannerod, H Knochel, C Monat, G Revy, and G Villard. A New Binary Floating-Point Division Algorithm and Its Software Implementation on the ST231 Processor. In *Computer Arithmetic, 2009. ARITH 2009. 19th IEEE Symposium on*, pages 95–103, June 2009. doi: 10.1109/ARITH.2009.19. 224

D R Johnson, M R Johnson, J H Kelm, W Tuohy, Steven S Lumetta, and S J Patel. Rigel: A 1,024-Core Single-Chip Accelerator Architecture. *Micro, IEEE*, 31(4): 30–41, 2011. ISSN 0272-1732. doi: 10.1109/MM.2011.40. 44

Kalray. Kalray, 2013. URL http://www.kalray.eu/. 33

Yi Kang, Wei Huang, Seung-Moon Yoo, D Keen, Zhenzhou Ge, V Lam, P Pattnaik, and J Torrellas. FlexRAM: Toward an advanced Intelligent Memory system. In *International Conference on Computer Design (ICCD), 1999*, pages 192–201, 1999. doi: 10.1109/ICCD.1999.808425. 63

Yi Kang, Wei Huang, Seung-Moon Yoo, D Keen, Zhenzhou Ge, V Lam, P Pattnaik, and J Torrellas. FlexRAM: Toward an advanced Intelligent Memory system. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pages 5–14, 2012. doi: 10.1109/ICCD.2012.6378608. 62, 63

U.J. Kapasi, W.J. Dally, S. Rixner, J.D. Owens, and B. Khailany. The Imagine Stream Processor. *Proceedings. IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 282–288, 2002. doi: 10.1109/ICCD.2002.1106783. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1106783. 41

UJ Kapasi, Scott Rixner, WJ Dally, and B Khailany. Programmable stream processors. *Computer*, 2003. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1220582. 41

Himanshu Kaul, Mark Anders, Steven Hsu, Amit Agarwal, Ram Krishnamurthy, and Shekhar Borkar. Near-threshold voltage (NTV) design: opportunities and challenges. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1153–1158, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1199-1. doi: 10.1145/2228360.2228572. URL http://doi.acm.org/10.1145/2228360.2228572. 5, 61

John H Kelm, Daniel R Johnson, Matthew R Johnson, Neal C Crago, William Tuohy, Aqeel Mahesri, Steven S Lumetta, Matthew I Frank, and Sanjay J Patel. Rigel: an architecture and scalable programming interface for a 1000-core accelerator. *SIGARCH Comput. Archit. News*, 37(3):140–151, June 2009. ISSN 0163-5964. doi: 10.1145/1555815.1555774. URL http://doi.acm.org/10.1145/1555815.1555774. 44

Ken Kennedy, Charles Koelbel, and Hans Zima. The rise and fall of high performance Fortran. *Commun. ACM*, 54(11):74–82, November 2011. ISSN 0001-0782. doi: 10.1145/2018396.2018415. URL http://doi.acm.org/10.1145/2018396.2018415. 69

KHRONOS Group. KHRONOS group - OpenCL, 2013. URL http://www.khronos.org/opencl/. 53, 69

Mario Kicherer, Fabian Nowak, Rainer Buchty, and Wolfgang Karl. Seamlessly portable applications. *ACM Transactions on Architecture and Code Optimization*, 8(4):1–20, January 2012. ISSN 15443566. doi: 10.1145/2086696.2086721. URL http://dl.acm.org/citation.cfm?id=2086696.2086721. 75

Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. SD3: A Scalable Approach to Dynamic Data-Dependence Profiling. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 535–546, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4299-7. doi: 10.1109/MICRO.2010.49. URL http://dx.doi.org/10.1109/MICRO.2010.49. 71

Aparna Kotha, Kapil Anand, Matthew Smithson, Greeshma Yellareddy, and Rajeev Barua. Automatic Parallelization in a Binary Rewriter. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 547–557, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4299-7. doi: 10.1109/MICRO.2010.27. URL http://dx.doi.org/10.1109/MICRO.2010.27. 86

E Krimer, R Pawlowski, M Erez, and P Chiang. Synctium: a Near-Threshold Stream Processor for Energy-Constrained Parallel Applications. *Computer Architecture Letters*, 9(1):21–24, 2010. ISSN 1556-6056. doi: 10.1109/L-CA.2010.5. 61

D J Kuck, R H Kuhn, D A Padua, B Leasure, and M Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '81, pages 207–218, New York, NY, USA, 1981. ACM. ISBN 0-89791-029-X. doi: 10.1145/567532.567555. URL http://doi.acm.org/10.1145/567532.567555. 95

Ian Kuon and Jonathan Rose. Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, February 2007. ISSN 0278-0070. doi: 10.1109/TCAD.2006.884574. URL http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4068926. 146, 151, 152, 157, 165

Taek-Jun Kwon and Jeffrey Draper. Floating-point division and square root using a Taylor-series expansion algorithm. *Microelectronics Journal*, 40(11): 1601–1605, 2009. ISSN 0026-2692. doi: 10.1016/j.mejo.2009.03.004. 102, 131, 222

Marco Lanuzza, Stefania Perri, and Pasquale Corsonello. MORA: a new coarse-grain reconfigurable array for high throughput multimedia processing. In *Proceedings of the 7th international conference on Embedded computer systems: architectures, modeling, and simulation*, SAMOS'07, pages 159–168, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-73622-0, 978-3-540-73622-6. URL http://dl.acm.org/citation.cfm?id=1776200.1776224. 33

C Lattner and V Adve. LLVM: a compilation framework for lifelong program analysis transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86, March 2004. doi: 10.1109/CGO.2004.1281665. 85

Chris Lattner. The LLVM Compiler Infrastructure, 2013. URL llvm.org. 76

Chunho Lee, Miodrag Potkonjak, and William H Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 30, pages 330–335, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-7977-8. URL http://dl.acm.org/citation.cfm?id=266800.266832. 29, 163

Junghee Lee, Hyung Gyu Lee, Soonhoi Ha, Jongman Kim, and Chrysostomos Nicopoulos. A programmable processing array architecture supporting dynamic task scheduling and module-level prefetching. In *Proceedings of the 9th conference on Computing Frontiers*, CF '12, pages 153–162, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1215-8. doi: 10.1145/2212908.2212931. URL http://doi.acm.org/10.1145/2212908.2212931. 31

REFERENCES

Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *ACM SIGARCH Computer Architecture News*, 38(3):451–451–460–460, June 2010. ISSN 0163-5964. doi: 10.1145/1816038.1816021. URL http://portal.acm.org/citation.cfm?id=1816038.1816021. 9

Seppo Linnainmaa. Software for Doubled-Precision Floating-Point Computations. *ACM Transactions on Mathematical Software*, 7(3):272–283, September 1981. ISSN 00983500. doi: 10.1145/355958.355960. URL http://dl.acm.org/citation.cfm?id=355958.355960. 130

LLNL. Rose compiler infrastructure, 2014. URL rosecompiler.org. 82

Eugene Loh. The Ideal HPC Programming Language. *ACMQueue*, 8(6):30:30—-30:38, 2010. ISSN 1542-7730. doi: 10.1145/1810226.1820518. URL http://doi.acm.org/10.1145/1810226.1820518. 10

D B Loveman. High performance Fortran. *Parallel Distributed Technology: Systems Applications, IEEE*, 1(1):25–42, 1993. ISSN 1063-6552. doi: 10.1109/88.219857. 69

Scott A Mahlke, David C Lin, William Y Chen, Richard E Hank, and Roger A Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th annual international symposium on Microarchitecture*, MICRO 25, pages 45–54, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press. ISBN 0-8186-3175-9. URL http://dl.acm.org/citation.cfm?id=144953.144998. 25

Peter Markstein. Software division and square root using Goldschmidt's algorithms. *Proceedings of the 6th Conference on Real Numbers ...*, pages 146–157, 2004. URL http://www.m1c4a1.wz.cz/docs/goldschmidt.pdf. 224

A Marowka. Back to Thin-Core Massively Parallel Processors. *Computer*, 44(12):49–54, 2011. ISSN 0018-9162. doi: 10.1109/MC.2011.133. 11, 22

Micron Technology. Micron Technology, Inc., 2011. URL http://www.micron.com/products/hybrid-memory-cube. 63

Microsoft. C++ AMP (C++ Accelerated Massive Parallelism), 2013. URL http://msdn.microsoft.com/en-us/library/hh265137.aspx. 76

Mahim Mishra, Timothy J Callahan, Tiberiu Chelcea, Girish Venkataramani, Seth C Goldstein, and Mihai Budiu. Tartan: evaluating spatial computation for whole program execution. *SIGOPS Oper. Syst. Rev.*, 40(5):163–174, October 2006. ISSN 0163-5980. doi: 10.1145/1168917.1168878. URL http://doi.acm.org/10.1145/1168917.1168878. 24, 27

A Miyoshi, T Kitayama, and H Tokuda. Implementation and evaluation of real-time Java threads. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 166–175, 1997. doi: 10.1109/REAL.1997.641279. 69

G Moore. Cramming more components onto integrated circuits. *Electronics*, 38(38), April 1965. 2

Ananya Muddukrishna, Artur Podobas, Mats Brorsson, and Vladimir Vlassov. Task scheduling on manycore processors with home caches. In *Proceedings of the 18th international conference on Parallel processing workshops*, Euro-Par'12, pages 357–367, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-36948-3. doi: 10.1007/978-3-642-36949-0_39. URL http://dx.doi.org/10.1007/978-3-642-36949-0_39. 48, 71

Dheya Mustafa, Aurangzeb Aurangzeb, and Rudolf Eigenmann. Performance analysis and tuning of automatically parallelized OpenMP applications. In *Proceedings of the 7th international conference on OpenMP in the Petascale era*, IWOMP'11, pages 151–164, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-21486-8. URL http://dl.acm.org/citation.cfm?id=2023025.2023041. 71

Nallatech. H101-PCIX Reference Guide, 2007. 115

NVIDIA. Cuda c programming guide, 2013a. URL http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. 69, 223

NVIDIA. Kepler GK110, 2013b. URL https://www.google.co.za/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&ved=0CCsQFjAA&url=http%3A%2F%2Fwww.nvidia.com%2Fcontent%2FPDF%2Fkepler%2FNVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf&ei=q_-qUqyrFMS5OQW7t4GADw&usg=AFQjCNHa4sr8_tBwNKzKvzo2u89WwLrrsQ&sig2=m8CSnc_hKTorTxnc-GUZTg&bvm=bv.57967247,d.d2k. 50

NVIDIA. Tesla K40 active accelerator, 2013c. URL https://www.google.co.za/url?sa=t&rct=j&q=&esrc=s&source=web&cd=4&cad=rja&ved=0CEAQFjAD&url=http://www.nvidia.com/content/PDF/kepler/Tesla-K40-PCIe-Passive-Board-Spec-BD-06902-001_v05.pdf&ei=CbuqUoWlFKeY1AWKtIHYBw&usg=AFQjCNFcrB_ZSGVsqGTI_C8uEo4pon150A&sig2=Lg-FaFw4tV1dsQHqFYIlHg. 49, 53

Stuart F Oberman and Michael J Flynn. An Analysis Of Division Algorithms And Implementations. *IEEE Transactions on Computers*, 46:833–854, 1995. 223

K Olukotun and L Hammond. The future of microprocessors. *Queue*, pages 67–77, 2005. doi: 10.1145/1941487. URL http://dl.acm.org/citation.cfm?id=1095418. 1

OpenACC. OpenACC Directives for Accelerators, 2013. URL http://www.openacc-standard.org/. 76

D Perlmutter. Sustainability in silicon and systems development. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pages 31–35, 2012. doi: 10.1109/ISSCC.2012.6176867. 63

Wright Peter. Digital computers in time-critical simulation. *Data Processing*, 25(9):30–33, 1983. 94

K Peterson. The use of DAREA, the Data Area Optimizer, on a Jet Engine Simulation. Lynn, MA, June 1984. General Electric Company, ADIUS. 94, 95

PGI. PGI Compiler User's Guide, 2013. URL https://www.pgroup.com/doc/pgiug.pdf. 86

José-Alejandro Piñeiro, Javier Díaz Bruguera, J.-A. Pineiro, and Javier Díaz Bruguera. High-Speed Double-Precision Computation of Reciprocal, Division, Square Root and Inverse Square Root. *IEEE Trans. Comput.*, 51(12):1377–1388, December 2002. ISSN 0018-9340. doi: 10.1109/TC.2002.1146704. URL http://dx.doi.org/10.1109/TC.2002.1146704. 223

Nathaniel Pinckney, Korey Sewell, Ronald G Dreslinski, David Fick, Trevor Mudge, Dennis Sylvester, and David Blaauw. Assessing the performance limits of parallelized near-threshold computing. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1147–1152, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1199-1. doi: 10.1145/2228360.2228571. URL http://doi.acm.org/10.1145/2228360.2228571. 3

Fred J. Pollack. New microarchitecture challenges in the coming generations of CMOS process technologies (keynote address)(abstract only), November 1999. URL http://dl.acm.org/citation.cfm?id=320080.320082. 9

Prakash Prabhu, Soumyadeep Ghosh, Yun Zhang, Nick P. Johnson, and David I. August. Commutative set: a language extension for implicit parallel programming. *ACM SIGPLAN Notices*, 46(6):1–1–11–11, June 2011. ISSN 0362-1340. doi: 10.1145/1993316.1993500. URL http://dl.acm.org/citation.cfm?id=1993316.1993500. 10, 11, 15, 71

Benoit Pradelle, Alain Ketterlin, and Philippe Clauss. Polyhedral parallelization of binary code. *ACM Trans. Archit. Code Optim.*, 8(4):39:1—-39:21, January 2012. ISSN 1544-3566. doi: 10.1145/2086696.2086718. URL http://doi.acm.org/10.1145/2086696.2086718. 86

Md rezaur rahman. Intel Xeon Phi Coprocessor Vector Microarchitecture. Technical report, Intel, 2013. URL http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-vector-microarchitecture. 53

Richard M Russell. The CRAY-1 Computer System. *Commun. ACM*, 21(1):63–72, January 1978. ISSN 0001-0782. doi: 10.1145/359327.359336. URL http://doi.acm.org/10.1145/359327.359336. 37

Mohamed M Saad, Mohamed Mohamedin, and Binoy Ravindran. HydraVM: extracting parallelism from legacy sequential code using STM. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, HotPar'12, page 8, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2342788.2342796. 84

Karthikeyan Sankaralingam, Ramadass Nagarajan, Robert McDonald, Rajagopalan Desikan, Saurabh Drolia, M.S. Govindan, Paul Gratz, Divya Gulati, Heather Hanson, Changkyu Kim, Haiming Liu, Nitya Ranganathan, Simha Sethumadhavan, Sadia Sharif, Premkishore Shivakumar, Stephen Keckler, and Doug Burger. Distributed Microarchitectural Protocols in the TRIPS Prototype Processor. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 480–491. IEEE, December 2006. ISBN 0-7695-2732-9. doi: 10.1109/MICRO.2006.19. URL http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4041870. 24

Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky, and Olivier Tardieu. The Asynchronous Partitioned Global Address Space Model. Technical report, Toronto, Canada, June 2010. URL http://www.cs.rochester.edu/u/cding/amp/papers/full/TheAsynchronousPartitionedGlobalAddressSpaceModel.pdf. 74

Frank Schimbac. Intel Concurrent Collections for C++ 0.9 for Windows and Linux, 2013. URL http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc. 69

Seaforth. Seaforth, 2013. URL http://www.intellasys.net/index.php?option=com_content&task=view&id=60&Itemid=75. 33

Mingoo Seok, S Hanson, Yu-Shiang Lin, Zhiyoong Foo, Daeyeon Kim, Yoonmyung Lee, Nurrachman Liu, D Sylvester, and D Blaauw. The Phoenix Processor: A 30pW platform for sensor applications. In *VLSI Circuits, 2008 IEEE Symposium on*, pages 188–189, June 2008. doi: 10.1109/VLSIC.2008.4586001. 61

John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. *Computing for Computational Science*, pages 1–25, 2011. URL http://link.springer.com/chapter/10.1007/978-3-642-19328-6_1. 6

Gaurav Sharma and Jos Martin. MATLAB: a language for parallel computing. *Int. J. Parallel Program.*, 37(1):3–36, February 2009. ISSN 0885-7458. doi: 10.1007/s10766-008-0082-5. URL http://dx.doi.org/10.1007/s10766-008-0082-5. 69

Aviral Shrivastava, Jared Pager, Reiley Jeyapaul, Mahdi Hamzeh, and Sarma Vrudhula. Enabling Multithreading on CGRAs. In *2011 International Conference on Parallel Processing*, pages 255–264. IEEE, September 2011. ISBN 978-1-4577-1336-1. doi: 10.1109/ICPP.2011.77. URL http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6047194. 57

Logic Simulation. Vivado Design Suite, 2013. URL http://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_2/ug900-vivado-logic-simulation.pdf. 165

SKA. Square Kilometre Array Africa, 2013. URL www.ska.ac.za. 116

Aaron Smith, Jon Gibson, Bertrand Maher, Nick Nethercote, Bill Yoder, Doug Burger, Kathryn S McKinle, and Jim Burrill. Compiling for EDGE Architectures. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 185–195, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2499-0. doi: 10.1109/CGO.2006.10. URL http://dx.doi.org/10.1109/CGO.2006.10. 24

K C Smith, A Wang, and L C Fujino. Through the Looking Glass: Trend Tracking for ISSCC 2012. *Solid-State Circuits Magazine, IEEE*, 4(1):4–20, March 2012. ISSN 1943-0582. doi: 10.1109/MSSC.2011.2177577. 4

Hayden So and Robert Brodersen. Improving Usability of FPGA-Based Reconfigurable Computers Through Operating System Support. In *2006 International Conference on Field Programmable Logic and Applications*, pages 1–6. IEEE, 2006. ISBN 1-4244-0312-X. doi: 10.1109/FPL.2006.311236. URL http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4100998. 116

Stanford University. Pervasive Parallelism Laboratory Stanford Engineering, 2013. URL http://ppl.stanford.edu/main/. 80

REFERENCES

Stutter and Herb. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), 2005. 1

Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. WaveScalar. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 291–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2043-X. URL http://dl.acm.org/citation.cfm?id=956417.956546. 24, 25

Steven Swanson, Andrew Schwerin, Martha Mercaldi, Andrew Petersen, Andrew Putnam, Ken Michelson, Mark Oskin, and Susan J Eggers. The WaveScalar architecture. *ACM Trans. Comput. Syst.*, 25(2):4:1—-4:54, May 2007. ISSN 0734-2071. doi: 10.1145/1233307.1233308. URL http://doi.acm.org/10.1145/1233307.1233308. 24, 25

M B Taylor, J Kim, J Miller, D Wentzlaff, F Ghodrat, B Greenwald, H Hoffman, P Johnson, Jae-Wook Lee, W Lee, A Ma, A Saraf, M Seneski, N Shnidman, V Strumpen, M Frank, S Amarasinghe, and A Agarwal. The Raw microprocessor: a computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, March 2002. ISSN 0272-1732. doi: 10.1109/MM.2002.997877. URL http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=997877. 46

Michael B Taylor. Is dark silicon useful?: harnessing the four horsemen of the coming dark silicon apocalypse. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1131–1136, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1199-1. doi: 10.1145/2228360.2228567. URL http://doi.acm.org/10.1145/2228360.2228567. 5, 8, 57

Michael B Taylor. A L ANDSCAPE OF THE NEW D ARK SILICON DESIGN REGIME. *Micro, IEEE*, 33(5):8–19, 2013. doi: 10.1109/MM.2013.90. 5

Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 2—-, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2143-6. URL http://dl.acm.org/citation.cfm?id=998680.1006733. 47, 163

The Portland Group. *CUDA Fortran Programming Guide and Reference*. Portland Group, 2013. 76

TILEPro. TILEPro64 Processor Product Brief, 2011. 48

Tilera. TILE-Gx Instruction Set Architecture, 2013. URL http://www.tilera.com/products/processors/TILE-Gx_Family. 39, 46

Philip Top and Maya Gokhale. Application Experiments: MPPA and FPGA. In *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, pages 291–294. IEEE, 2009. ISBN 978-0-7695-3716-0. doi: 10.1109/FCCM.2009.37. URL http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=5290896. 33, 34

Top500. Top500 November 2013 list, 2013. URL http://www.top500.org/lists/2013/11/. 6, 7, 8

Peng Tu and David Padua. Automatic array privatization. In Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 500–521. Springer Berlin Heidelberg, 1994. ISBN 978-3-540-57659-4. doi: 10.1007/3-540-57659-2_29. URL http://dx.doi.org/10.1007/3-540-57659-2_29. 108

UPC. Unified Parallel C, 2005. URL http://upc.gwu.edu/. 69, 74

Teruo Utsumi, M Ikeda, and Moriyuki Takamura. Architecture of the VPP500 parallel supercomputer. *Supercomputing'94., ...*, pages 478–487, 1994. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=344311. 37

S R Vangal, J Howard, G Ruhl, S Dighe, H Wilson, J Tschanz, D Finan, A Singh, T Jacob, S Jain, V Erraguntla, C Roberts, Y Hoskote, N Borkar, and S Borkar. An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *Solid-State Circuits, IEEE Journal of*, 43(1):29–41, 2008. ISSN 0018-9200. doi: 10.1109/JSSC.2007.910957. 63

Vitalii Vanovschi. Parallel Python. 2014. URL http://www.parallelpython.com/. 69

Vector Fabrics. Vector Fabrics, 2013. URL http://www.vectorfabrics.com/. 67

Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: reducing the energy of mature computations. *SIGARCH Comput. Archit. News*, 38(1):205–218, March 2010. ISSN 0163-5964. doi: 10.1145/1735970.1736044. URL http://doi.acm.org/10.1145/1735970.1736044. 5, 57

L Verdoscia, M Danelutto, and R Esposito. CODACS prototype: CHIARA language and its compiler. In *Distributed Computing Systems Workshops, 2004. Proceedings. 24th International Conference on*, pages 864–870, 2004. doi: 10.1109/ICDCSW.2004.1284134. 28

Lorenzo Verdoscia and Roberto Vaccaro. D3AS project: a different approach to the manycore challenges. In *Proceedings of the 9th conference on Computing Frontiers*, CF '12, pages 261–264, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1215-8. doi: 10.1145/2212908.2212948. URL http://doi.acm.org/10.1145/2212908.2212948. 22, 28

M Villalon, B Amini, S Creusillet, R Even, O Keryell, S Goubier, J Guelton, F Mcmahon, G Pasquier, and Péan. Par4All: From Convex Array Regions to Heterogeneous Computing. In *IMPACT 2012 : Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012*, Paris, 2012. URL http://hal.archives-ouvertes.fr/hal-00744733/. 82

Richard Vuduc, Aparna Chandramowlishwaran, Jee Choi, Murat Guney, and Aashay Shringarpure. On the limits of GPU acceleration. In *Proceedings of the 2Nd USENIC Conference on Hot Topics in Parallelism*, page 13. USENIX Association, June 2010. URL http://portal.acm.org/citation.cfm?id=1863086.1863099. 9

Alan Wallcraft. Co-Array Fortran, 2005. URL http://www.co-array.org/. 69, 74

Tadashi Watanabe. Architecture and performance of {NEC} supercomputer {SX} system. *Parallel Computing*, 5:247–255, 1987. ISSN 0167-8191. doi: http://dx.doi.org/10.1016/0167-8191(87)90021-4. URL http://www.sciencedirect.com/science/article/pii/0167819187900214. 37

Robert P Wilson, Robert S French, Christopher S Wilson, Saman P Amarasinghe, Jennifer M Anderson, Steve W K Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W Hall, Monica S Lam, and John L Hennessy. SUIF: an infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Not.*, 29 (12):31–37, December 1994. ISSN 0362-1340. doi: 10.1145/193209.193217. URL http://doi.acm.org/10.1145/193209.193217. 85

Michael Wolfe. The PGI Accelerator Compilers with OpenACC. *PGI Insider*, March 2012. URL http://www.pgroup.com/lit/articles/insider/v4n1a1.htm. 76

Xilinx. LogiCORE IP Divider Generator v3.0. 2011a. 224

Xilinx. LogiCORE IP Floating-Point. 2011b. 224

Xilinx. Virtex-series FPGAs, 2011c. URL www.xilinx.com. 147

Xilinx. 7 Series FPGAs Overview Summary of 7 Series FPGA Features Notes : Artix-7 FPGA Feature Summary, 2012. URL http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf. 147

Xuejun Yang, Xiaobo Yan, Zuocheng Xing, Yu Deng, Jiang Jiang, and Ying Zhang. A 64-bit stream processor architecture for scientific applications. *ACM SIGARCH Computer Architecture News*, 35(2):210, June 2007. ISSN 01635964. doi: 10.1145/1273440.1250689. URL http://portal.acm.org/citation.cfm?doid=1273440.1250689. 36

Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A High-Performance Java Dialect. In *In ACM*, pages 10–11, 1998. 69, 74

Tjalling J Ypma. Historical development of the Newton-Raphson method. *SIAM Rev.*, 37(4):531–551, December 1995. ISSN 0036-1445. doi: 10.1137/1037125. URL http://dx.doi.org/10.1137/1037125. 223

Ying Zhang, Xuejun Yang, Guibin Wang, Ian Rogers, Gen Li, Yu Deng, and Xiaobo Yan. Scientific Computing Applications on a Stream Processor. *ISPASS 2008 - IEEE International Symposium on Performance Analysis of Systems and software*, pages 105–114, April 2008. doi: 10.1109/ISPASS.2008.4510743. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4510743. 36, 41

Hongtao Zhong, Kevin Fan, Scott Mahlke, and Michael Schlansker. A Distributed Control Path Architecture for VLIW Processors. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 197–206, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2429-X. doi: 10.1109/PACT.2005.5. URL http://dx.doi.org/10.1109/PACT.2005.5. 39

P S Zuchowski, C B Reynolds, R J Grupp, S G Davis, B Cremen, and B Troxel. A hybrid ASIC and FPGA architecture. In *Computer Aided Design, 2002. ICCAD 2002. IEEE/ACM International Conference on*, pages 187–194, November 2002. doi: 10.1109/ICCAD.2002.1167533. 151

Appendices

Appendix A

Additional information

A.1 Additional Explanatory Notes

A.1.1 GPGPU Programming Model and Hardware Mapping

As independent operations, the threads of a kernel may be distributed across multiple VPUs. Keeping a kernel within one SMX with a shared L1 cache, however, is advantageous, both in terms of exploiting locality for memory read and writes, and for data sharing between threads. A thread block is therefore defined as that in which all threads execute the same kernel concurrently on the same SMX. Threads within a thread block may cooperate with each other via the shared memory and synchronisation barriers.

The size of thread blocks is dependant foremost on the resource limits of a single SMX. Given the resources, multiple thread blocks may be executed concurrently on the same SMX up to a limit dependant on; kernel memory and register requirements, and the maximum number of threads per block and blocks per SMX, set by the control hardware capacity of an architecture.

Once a kernel's problem set reaches beyond the limits of a single SMX, however, a kernel must be spread across multiple SMXs in multiple thread blocks creating what CUDA defines as a grid. Grids are , arrays of one or more thread blocks executing the same kernel by reading and writing to the L2 cache, and global kernel-wide synchronisation between dependant kernel calls. The execution of a grid of thread blocks may entail multiple accesses to the host memory, making its size limit, within reason, the problem set size.

Tying this to execution on the hardware, is the SIMT concept described in Subsection 2.2.3.5. Threads of the same kernel are issued for execution in warps, batches of

32 threads. Within a SMX, four warps, or 128 threads, or eight VPU instructions, can be issued per cycle. The issuing of the warps take advantage of the fact that the threads within a thread block must be independent and warps can therefore be executed in any order. If the next eight VPU instructions to be issued within a thread block are waiting on memory, an alternative set or thread in the conventional meaning, of warp instructions may be issued so as to hide the memory latency, with execution of another set of operations. It is therefore, these warps of threads that are interweaved making the SIMT form of operation.

Summarising the connections between processing, programming abstraction, and memory scope again therefore. Thread instructions are executed on a GPU core or DP or SFU. Each thread has a thread and block ID, a program counter, registers, and per-thread private memory in L1 cache. Threads are grouped into thread blocks in which all threads operate within the same SMX and share data in a per-block shared allocation of the L1 cache. Multiple blocks, executing the same kernel, operate in a grid across multiple SMXs sharing data in the L2 cache via kernel wide synchronisation barriers. Finally, multiple grids may operate in an independent and overlapping manner within a single GPU provided the resources for such exist.

Table A.1 – A comparison of the hardware and programming models used in NVIDIA and AMD GPGPUs. Where alternative titles are give they refer to; first the NVIDIA terminology, second the OpenCL equivalent, and third a descriptive name given and used in this document. The vertical progression is representative of the associated hierarchy, while the horizontal matches the various models, or levels of abstraction, involved, to one another

Machine Object	Program Abstraction	Processing Hardware	Memory Hardware
PTX Intructions: Assembler level SIMD instructions executed by CUDA threads across the lanes of a GPU core.	***CUDA Thread/ Work item:*** A single lane of execution in a GPU core, executing an instance of a kernel on a single vector element index. i.e a vertical slice of a Warp/ Wavefront. A register mask may control the writing of results generated by a thread/item. This will have its own ID	***Thread processor/ Processing Element/ GPU Core:*** The hardware ALU executing instructions on single elements in the SIMD Lane, a datapath and register file. Executes the PTX instructions in CUDA threads.	***SIMD Lane Registers:*** Dedicated to a single thread processor (SIMD lane), registers are per-allocated to a set of threads in different warps to serves as a multi-threaded register file.
Warp/ Wavefront: A grouping of CUDA threads that execute the same instructions concurently. Individual threads may diverege on branch conditions but at the cost of efficiency as other theads are disabled until the diverged thread re-merges or completes.	***Thread Block/ Work Group:*** Depending on hardware capacity and desired data sharing, threads executing the same kernel are grouped into an array for execution on a single SMX/CU. Threads witjhin a block/group are able to communicate via shared memory. This will have its own ID.	***Streaming Multiprocessors (SMXs)/ Compute Units (CUs)/ PE:*** A multithreaded SIMD processor. Clusters of thread processors form 16-wide SIMD VPU like units, which are collated to form the multiple hardware threads in a SMX.	***Shared/ Local memory:*** Fast local memory private to a SMX. Used to share data within a thread block at points of synchronization. Allocated to a thread block and deallocated when thread block's threads complete.
			Local/ Private memory: A portion of the L1 cache not shared but private to a thread and serving as a register overflow.
	Grid/ Index Range: One or more thread blocks executing the same kernel sequentially, or concurrently, or a mixture of both.	***Scheduling:*** The warp and grid scheduling in both architectures is very different. Sufice it to say dedicated hardware modules exist for the purpose at both an SMX and global level.	***L2 cache:*** Cache available to all SMXs, and used to in data sharing between thread blocks.
			Global/ GPU memory: DRAM cache available to all SMXs. Accessible to all threads in any thread bock in any grid and available to the host to read and write.

A.1.2 BRAM Analysis Table

Table A.2 – Cost of instruction BRAM form limits. As is to be expected, in certain boarder cases where the instruction width is exactly divisible by nine, there is no waste.
*Can be subdivided to a degree into 18b wide 0.5 units

Depth	1024				2048			
Array (TilesxStrips)	16 (4x4)	64 (8x8)	256 (16x16)	512 (32*16)	16 (4x4)	64 (8x8)	256 (16x16)	512 (32*16)
Instruction Width	42	45	48	49	45	48	51	52
Total instruction memory per strip in KB (number of 36kb modules*)	5.25 (1.5)	5.6 (1.5)	6 (1.5)	6.125 (1.5)	11.25 (2.5)	12.0 (3)	12.75 (3)	13.0 (3)
Total instruction memory implemented in array in KB	108	432	1728	3456	180	864	3456	6912
Total implemented instruction memory unused due to form limits in KB	24 (22.2%)	72 (16.6%)	192 (11.1%)	320 (9.2%)	0	96 (11.1%)	192 (5.5%)	256 (3.7%)

A.1.3 Division Hardware

A.1.3.1 Serial Nature of Division

In terms of the SQRT and DIV algorithms used in practise, most are optimised for hardware implementation due to the obvious speed advantage of such. As they share stages of computation, SQRT is generally implemented as a variant of whatever DIV algorithm is chosen, potentially even reusing sections of the DIV module hardware. Only the DIV algorithms are therefore discussed in detail here.

some of their conclusions around algorithm dominance are slightly dated and a range of subtle improvements have subsequently been proposed but the core algorithms used in new processors have not changed significantly. [Kwon and Draper, 2009] reports Intel's core i7 to be using a Radix-16 SRT algorithm, AMD's Phenom II and IBM's Power6; Goldschmidt's, and [Intel, 2012a] shows examples of how their SSE instructions use reciprocal estimates and few iterations of Newton-Raphson for division and square root. Xilinx documentation unfortunately fails to detail which algorithm their floating point division core uses. Nvidia Kepler GPUs use reciprocal division and square root in a manner that they note is non-standard relative to the

IEEE 745 floating point standard [NVIDIA, 2013a].[Oberman and Flynn, 1995] uses four categories to group the common algorithms used:

The first, digit recurrence routines, most commonly SRT (named for Sweeney, Robertson and Tocher who independently published it), use shift and subtraction operations to resolve 1 bit per iteration. As such they converge lineally on the answer, but occupy the smallest die area of the four and have the added advantage that a remainder is available on completion. Each iteration involves four stages looping n times for an n digit word.

The second, functional iteration algorithms alternatively includes Newton-Raphson [Ypma, 1995], and Goldschmidt [Goldschmidt, 1964]. These require more die area but converge on the quotient quadratically. The larger die area is occupied by multiplication units, as the primary operation, and a means of forming an initial estimation value. Such estimations are generally made with a look-up-table (LUT) or partial array. Within the classification, Newton-Raphson is self correcting but requires two dependant multiplications, while Goldschmidt requires error correction, but uses 2 independent multiplications which can be executed in parallel.

The third, very high radix division, is a grouping of methods that are able to retire around 10 quotient bits per iteration. Such comes at the cost of generally more multipliers, meaning the approaches lies in a middle ground concerning die area usage and latency. Due to its natural use of FMAs and BRAMs, the approach has been shown to be a good approach for FPGAs[Amaricai and Boncalo, 2012].

Finally, [Oberman and Flynn, 1995] specifies a last grouping as variable latency algorithms, which consists of methods of reducing the latency when possible. That is, these methods shorten the latency of an operation when the particular operands allow for such thereby reducing a programs overall latency. These methods include; result and reciprocal caching for reuse, quotient bit speculation and self-timing to minimise the critical path.

The functional iteration and high radiix algorithms all require an initial approximation to be made, either of the reciprocal or another related metric. The most simple source of such is a look up table (LUT) or a partial product array whose sizes are directly proportional to the precision of the estimate. Or an alternative approaches, not occupying memory area but logic, includes using a taylor series approximation [Piñeiro et al., 2002].

[Oberman and Flynn, 1995] draws the conclusion that while more area can reduce latency, predominantly when used to create larger LUTs, the latency of an architec-

ture's multipliers eclipses most other effects and is the dominant factor in determining latency as the algorithms are otherwise generally within range of each other.

Looking alternatively, for software approaches to division, where the motivation for such lies in a partial or complete lack of floating point division hardware, the desire to dynamically change precision requirements or possibly the opportunity to schedule other operation in parallel. In both cases the algorithms used appear to only be variants and optimisations of the methods already discussed. For embedded systems supporting only integer arithmetic for instance, [Jeannerod et al., 2009] contributed towards a floating point C library for integer processors. Alternatively [Markstein, 2004] offers a software version of Goldschmidt's algorithm that achieves the same performance as a hardware implementation provided the hardware used supports FMAs.

None of these approaches are parallelisable beyond a degree of pipelining or two concurrent multiplications occupying an inner iteration loop's contents leaving nothing to be exploited by a many-core architecture. Implementing Newton-Raphson or Goldschmidt's methods in software, would incur an additional cycle latency of around 30-40 cycles depending on the exact approach taken, initial estimates, and the availability of an FMA operation or not. The significant increase is predominantly due to the fetch and store stages included in every Fynbos fetch-execute-store cycle and the fact that the process is serial.

Like Itanium, therefore, the APPRASE-Fynbos system has adopted a middle ground solution. Assuming that both are infrequent we have implemented a division module but only in a limited number of PEs, and chosen a software solution for square root.

A.1.3.2 Xilinx Division Hardware

The datasheet for Xilinx's Floating-Point IP core [Xilinx, 2011b]does not describe what algorithm exactly they have selected used but the equivalent integer divider core has a detailed explanation of the high radix with prescaling algorithm it uses[Xilinx, 2011a]. We assume the floating point core uses the same for mantissas division. The description given for the integer divider therefore, indicates that operands are prescaled, pushing the divisor closer to 1. Thereafter, the operands are normalised, before both are multiplied by an estimate reciprocal driving the divisor towards 1 and the numerator towards the quotient and the cycle repeated. The iterations themselves are done in using a carry-save multiplication and subtraction making use of the XtremeDSP slices. Regardless of the fundamental algorithm, [Xilinx, 2011b] is clear about the core being fully pipelined, however this is not of any use in the context of Fynbos. One of

the drawbacks of using Xilinx IP cores is DSPs may or may not be shared, depending on synthesis settings and how code is written.

A.1.4 Host commands

Table A.3 – Table listing the components of the host issued commands directing Fynbos into the four modes of operation required to execute a program. The exact bit range for each component are not included as they depend on the shape and configuration of Fynbos. All commands are standardised in length for a given configuration, however, with Run and Off load including a 'don't care' section each to make up for being shorter than Load data and Load instructions.

Command\Component	**Defined Command Sections**		
Load Data	Number of words to be loaded	Start address (Register address:Strip address:Tile address)	010 (command code)
Load Instructions	Number of words to be loaded	Start address (Register address:Strip address:Tile address)	001 (command code)
Run	Termination row number (register address)	Start row number (register address)	011 (command code)
Off Load Data	Depth of data memories in this Fynbos configuration	Start address (Register address:Strip address:Tile address)	100 (command code)

A.2 Literature review comparison tables

Table A.4 – Reference comparison table for architectures reviewed. Memory categorisations are as follows; Distributed-shared: One memory address space accessible by all, Distributed-shared-coherent: Cache is spread out but allows for multiple copies and higher more conglomerate levels to be coherent. Also one address space, Distributed: each cache and memory address space is independent as far as processing goes. The might be accessible to each other depending on interconnect, SRF-unshared: Streaming file loaded from an off chip memory accessible by the host but not cache coherent

Architecture[1]	Memory Model	L2 Cache per Chip	L1 Data Memory on Chip (per PE)	L1 Instruction Memory on chip (per PE)	INT[2] / SFP/ DFP	Languages	Threading[3]
TRIPS	Distributed	1MB	64 (2) KB	128 (4) KB	Y/Y/N	C, Fortran	Up to 4
WaveCache[4]	Distributed	4MB	128 (1) KB	64 (0.5) KB	Y/Y/N	C	Up to 64
Tartan[5]	Distributed	256KB	8K	8K	Y/N/N	C	1
RAW	Distributed-shared	2MB	512 (32) KB	1.5MB (32cpu+64router) KB	Y/Y/N	C, Java, StreamIt	Up to 16
Tile-GX8072	Distributed-shared-coherent	18MB	2304 (32) KB	2304 (32) KB	Y/Y[6]/Y[6]	C, C++	Y
Rigel	Distributed-shared-coherent GAS	4MB	D&I joined: 8MB (8 KB)	D&I joined: 8MB (8 KB)	Y/Y/N	Rigel Task Model API	Y
Kalray	PGAS	2MB				C, C++, OpenMP, POSIX	256
Epiphany	Distributed-Shared	2MB	NA	NA	Y/Y/N	C, C++, OpenCL	N
FlexRAM	N/A	NA	NA	128 (2) KB (equivalent)	Y/N/N	Assembly	N/A

[1] Some of the architectures here have more levels than implied in terms of functional unit hierarchy.
[2] In some cases INT is fixed-point
[3] A confusing metric because in some cases this means pthreads and in others just independent streams of instructions
[4] "cluster" = "processor"
[5] CPU1
[6] Limited

P2012	PGAS	1MB	1x 256KB shared with all PEs (32Bank)	16KB	Y/Y/N	OpenCL, OpenMP, NPM	Y
Merrimac	GAS	512KB	128 (8 per cluster, + 6KB per LRF) KB				
Imagine[1]	SRF-unshared	128KB SRF[2]	9.7KB (LRF) + 8 (scratchpad) (2.2) KB	144KB	Y/Y/N	StreamC	N

[1] Evaluating the Imagine stream architecture
[2] 16KB per cluster of SRF, 256*32 scratchpad per cluster, LRF per ALU

Table A.5 – Reference comparison table for architectures reviewed

Architecture	Implemented	ALU per PE[1]	Elements per level of hierarchy	PE per chip	Frequency	Base architecture
TRIPS	Fabricated (2006)	2 (32 bit)[2]	1:16:2	32	366MHz	Dataflow (EDGE)
WaveCache	Simulation (2007)	1 (32 bit)[2]	1:2:4:4:4	128	1GHz	Dataflow (WaveScalar)
Tartan	Simulation (2006)	16 (128 bit)[2]	1:16:16:16	800 (pages)	No clk	Dataflow(Spatial Computing)
RAW	Fabricated (2002)	2 (32 bit)[2]	1:16	16	425MHz	MIPS
Tile-GX8072	Fabricated (2007)	3 (64 bit)[3]	1:72	72	1.2GHz	VLIW
Rigel	Simulation (2009)	2 (32 bits)[3]	1:8:128:1024	1024	1.2GHz	RISC
Kalray	Fabricated (2012)	4 (32 bit)[3]	1:16:16	256	400MHz	VLIW
Epiphany	Fabricated (2011)	2 (32 bit)[3]	1:64	64	800MHz	RISC
FlexRAM	Simulation (1999)	1 (32 bit)[2]	1:64	64	400MHz	RISC
P2012	Fabrication (2012)	3 (32 bit)[3]	(16+1):4	64+4	500MHz	RISC
Merrimac	Simulated (2003)	4 (64 bit)	1:4:16:16:32:32	16+1	500MHz	Streaming (SIMD)
Imagine	Fabricated (2002)	6 (32-bit)[3]	1:6:8	8 (clusters)	200MHz	Streaming (SIMD VLIW)

[1]ALU meaning any arithmetic unit - floating point or integer and ignoring shared units
[2]Single issue
[3]Multiple issue

Table A.6 – Reference comparison table for architectures reviewed

Architecture	Power per chip	FLOPS per chip	Multi-chip integrable[1]	ASIC transistors	Fabrication	Processor size	Total chip area
TRIPS	30W		N	170 Million	130nm	92mm^2	330mm^2
WaveCache	N/A		N	N/A	90nm	199mm^2	247mm^2
Tartan	N/A		N	N/A	90nm	400mm^2	
RAW	18.2W	6G	Y	122 Million	180nm	256mm^2	331mm^2
Tile-GX8072			N		40nm		2025mm^2
Rigel	92W	2.4T	N	N/A	45nm	207mm^2	320mm^2
Kalray	5W	230G	Y		28nm		
Epiphany	2W	100G	Y		28nm		225mm^2
FlexRAM	36W		Y		180nm	96mm^2 (PEs only)	505mm^2
P2012	4W	76G	N		28nm	15.2mm^2	26mm^2
Merrimac	31W	128G(DP)	Y		90nm		100mm^2
Imagine	6.8W	7.6G	Y		180nm		260mm^2

[1] Easily or inherently without a message passing protocol but rather direct connection - simply an upscaling array size

A.3 Additional graphs

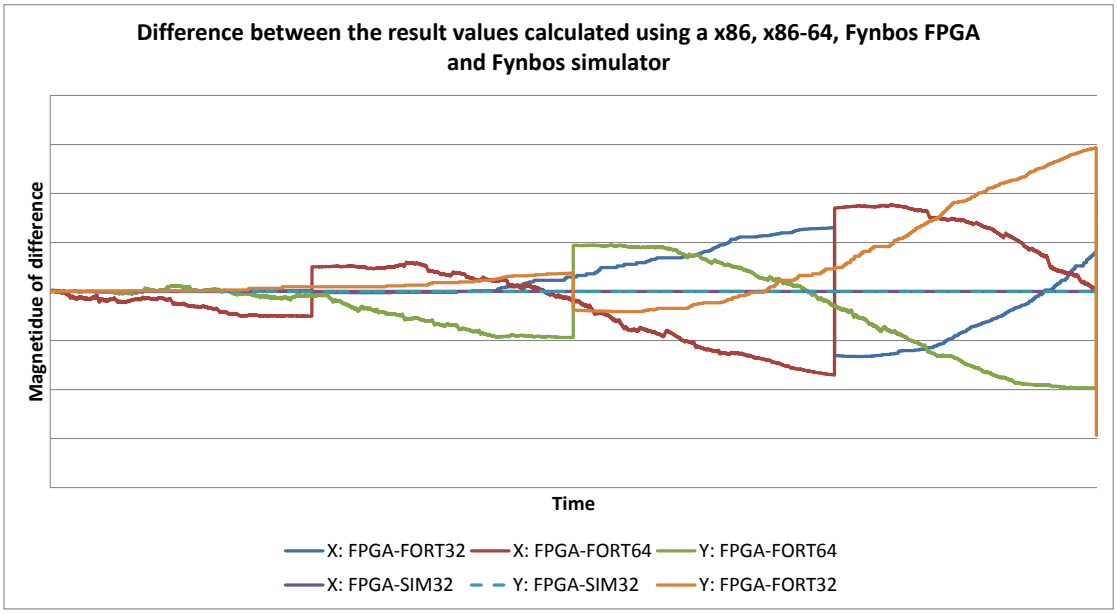
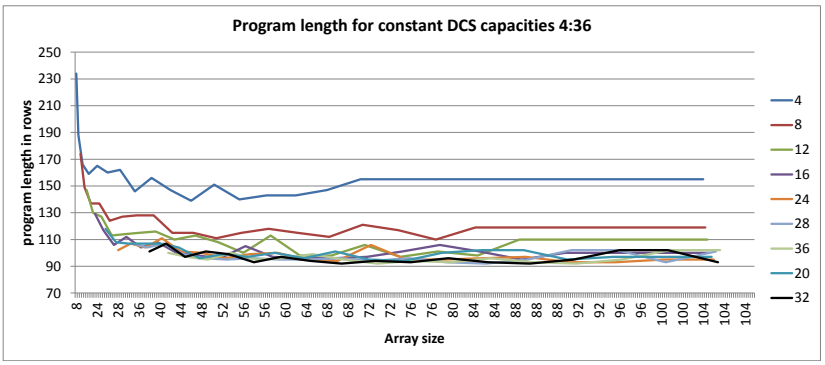
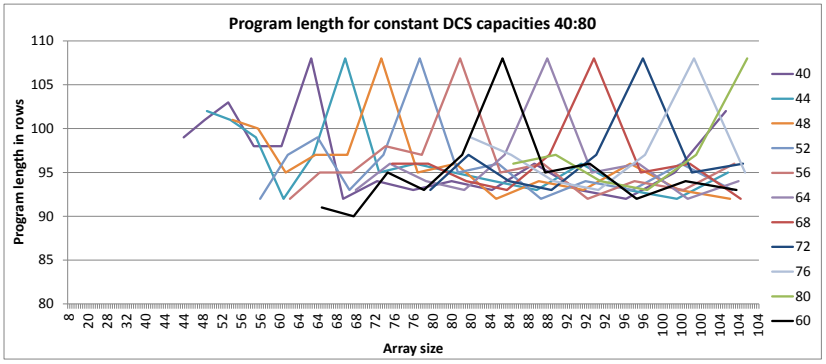


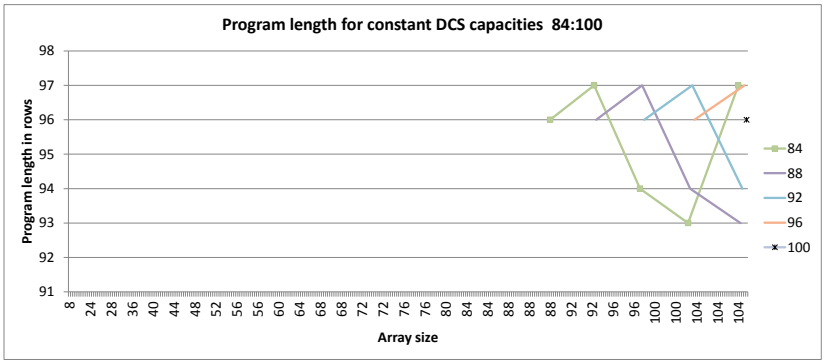
Figure A.1 – Comparing the X and Y co-ordinate values of the single body solar system calculated using different systems, the values plotted are all absolute differences. Both X and Y co-ordinate result differences are shown. In each instance the FPGA values used were generated using a Fynbos configuration supporting sinlge precision FPUs only. The series labeled FPGA-SIM compares the FPGA values with those generated in a single precision version of the simulator. Registering zero difference indicates the FPGA hardware is functioning correctly. The series labled FPGA-FORT32 compares FPGA values with those generated on an x86 system running Fortran code using REAL*4 type variables. The series labled FPGA-FORT64 compares FPGA values with those generated on an x86 system running Fortran code using REAL*8 type variables. It is clear the absolute error (difference) grows with the number of iterations executed (time).



(a)

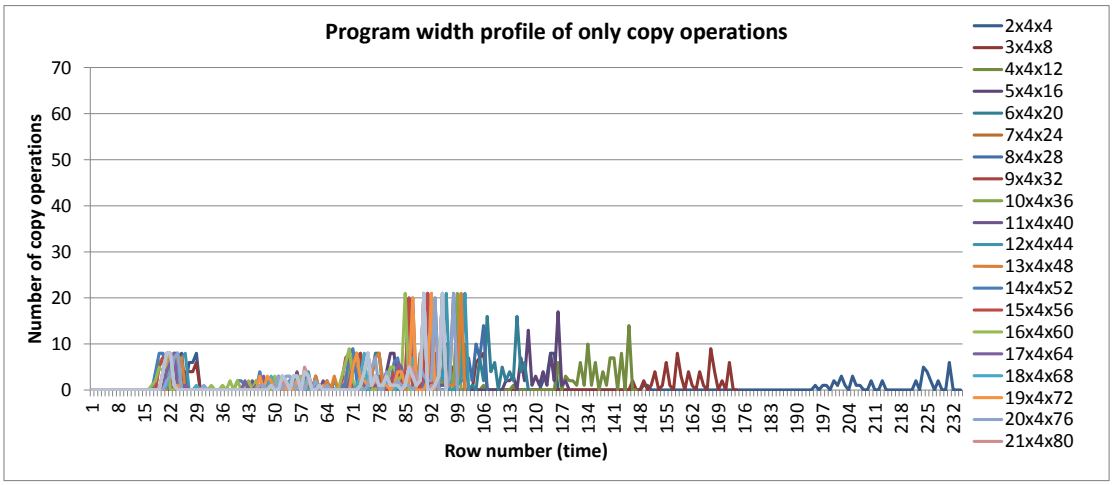


(b)

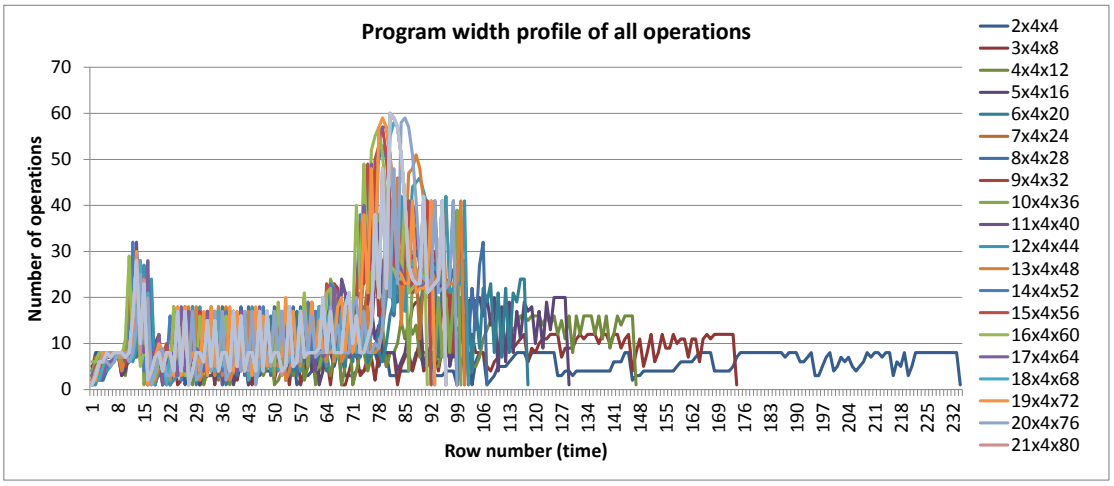


(c)

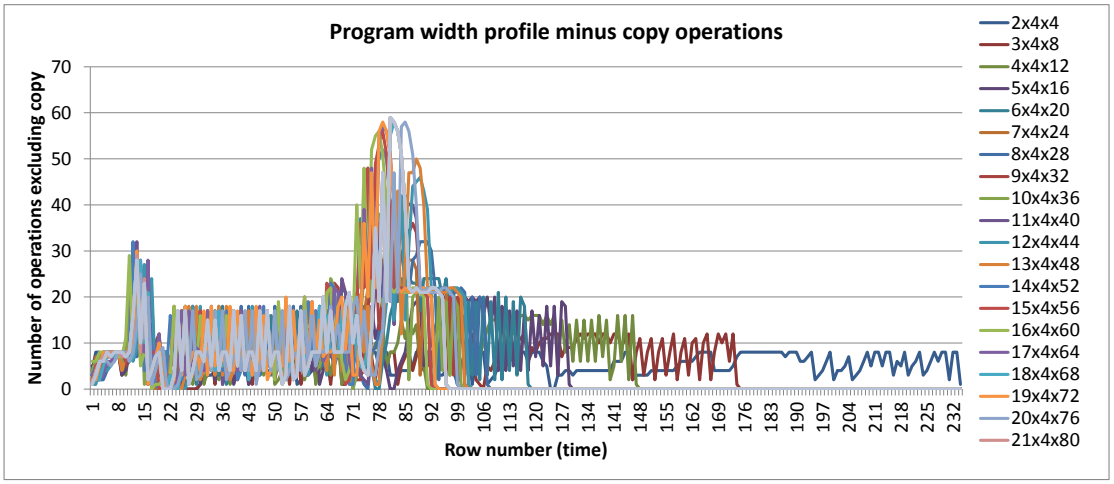
Figure A.2 – Graphs of three ranges array size showing different program length behaviour for constant DCS capacities. Note the same y-axis range is not used in each graph, graph (b) and (c) each have consequitively smaller ranges so as to show the behaviour best in each but fluctuations are significantly smaller in these graphs despite visual appearance.



(a)



(b)



(c)

Figure A.3 – The above three graphs each plot the width of execution rows for the seven body system compiled for a range of Fynbos configurations. In (a) only copy operations are included in the count. In (b) all operations are included. Finally (c) contains (a) subtracted from (b) plotting the complete program profile minus all copy operations. Clearly program length is unaffected by copy operations so far as they might cost additional rows.

Appendix B

Index to Digital Attachment

The following is an index of the significant files supplied with this work in digital form. Not every file is indexed here but the README.txt file found in each directory includes greater information about its contents and their purposes. All paths are given with reference to the top level directory "Appendix-Digital-Attachment/".

All code included here was created by this thesis's author, the codes pertaining to APPRASE up until xml generation are the property of J. Collins and SimCon Lts and not supplied although example outputs of the process are.

1 ./Fortran_Transformation/ Contains documents on the details on the transformation processes and example outputs of the transformation processing carried out on the solar body test cases including intermediatory files.

./Doc/ Details on the transformation process.

./sqrt_heron/ Source for square root Heron's method implementation.

./ss_series/ Source and intermediatory files for each of the body count test cases.

2 ./Fynbos/ HDL code for Fynbos and python scripts for running the system:

./Communication/ Python scripts for converting FPGA bin file into a ROACH loadable bof file, loading the bof file and communicating with ROACH to execute a program.

./HDL/ Fynbos implemented as VHDL code

./IO_Generation/ Python scripts for converting *.xml to Fynbos load file.

./Post_Processing/ Python scripts for converting the Fynbos returned data to decimal and corectly ordered format.

./Test_Generators/ Python scripts for generating Fynbos hardware system tests

3 Hardware_Tests HDL test projects and scripts for scraping and collating data from the Xilinx report files.

./1024/ Comparative compiles of Fynbos array on the V7 using 1024 deep instruction and data memories.

./2048/ Comparative compiles of Fynbos array on the V7 using 2048 deep instruction and data memories.

./ISE/ Comparative compiles of Fynbos array on the V5 using 1024 deep instruction and data memories.

4 Software_Tests Software tests, data scraping scripts and resulting spread sheets

./FPGA_32b_compared_to_everything_else.ods :Spreadsheet containing comparative X and Y co-ordinate results from the solar system application (single body) generated using Fynbos (single and double precision), the Fynbos simulator (single and double precision), and x86 and x86-64 based architectures.

./Increasing_Bodies/ Analysis of APPRASE's schedules when the number of solar bodies is increased.

./Seven_Bodies_Scaling_Fynbos/ Analysis of APPRASE's schedules when the Fynbos configuration is changed.

./Xeon_Comparison/ Code, binaries, and hardware system specifications of the solar applications run on a Xeon platform.:q