# Accelerated Adjoint Algorithmic Differentiation with Applications in Finance

**Jarred de Beer**

A dissertation submitted to the Faculty of Commerce, University of Cape Town, in partial fulfilment of the requirements for the degree of Master of Philosophy.

University of Cape Town

March 7, 2017

*MPhil in Mathematical Finance,*
*University of Cape Town.*

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

# Declaration

I declare that this dissertation is my own, unaided work. It is being submitted for the Degree of Master of Philosophy in the University of the Cape Town. It has not been submitted before for any degree or examination in any other University.

Signed by candidate

March 7, 2017

# Abstract

Adjoint Differentiation's (AD) ability to calculate Greeks efficiently and to machine precision while scaling in constant time to the number of input variables is attractive for calibration and hedging where frequent calculations are required. Algorithmic adjoint differentiation tools automatically generates derivative code and provide interesting challenges in both Computer Science and Mathematics. In this dissertation we focus on a manual implementation with particular emphasis on parallel processing using Graphics Processing Units (GPUs) to accelerate run times.

Adjoint differentiation is applied to a *Call on Max* rainbow option with 3 underlying assets in a Monte Carlo environment. Assets are driven by the Heston stochastic volatility model and implemented using the Milstein discretisation scheme with truncation. The price is calculated along with Deltas and Vegas for each asset, at a total of 6 sensitivities.

The application achieves favourable levels of parallelism on all three dimensions implemented by the GPU: Instruction Level Parallelism (ILP), Thread level parallelism (TLP), and Single Instruction Multiple Data (SIMD). We estimate the forward pass of the Milstein discretisation contains an ILP of 3.57 which is between the average range of 2-4. Monte Carlo simulations are *embarrassingly parallel* and are capable of achieving a high level of concurrency. However, in this context a single kernel running at low occupancy can perform better with a combination of Shared memory, vectorized data structures and a high register count per thread.

Run time on the Intel Xeon CPU with 501 760 paths and 360 time steps takes 48.801 seconds. The GT950 Maxwell GPU completed in 0.115 seconds, achieving an 422× speedup and a throughput of 13 million paths per second. The K40 is capable of achieving better performance.

# Acknowledgements

With thanks to my supervisors Associate Professor Peter Ouwehand and Associate Professor Michelle Kuttel. GPU computing has been a fascinating journey, and without your help would not have been possible. Further, thank you to the MPhil team for the immensely satisfying body of work and guidance presented during the year. Also, I wish to thank my parents for their support over the years.

# Contents


**1. Introduction** . . . . . 1

**2. Rainbow Options and Adjoint Differentiation with examples** . . . . . 4
- 2.1 Rainbow Options . . . . . 4
- 2.2 The Heston Stochastic Volatility Model . . . . . 5
- 2.3 Adjoint Differentiation . . . . . 7
- 2.4 Example Rainbow option with pathwise Adjoint Differentiation for GBM and Heston models . . . . . 8

**3. Multithreading** . . . . . 17
- 3.1 Graphics Processing Units . . . . . 18
- 3.2 Compute Unified Device Architecture . . . . . 21
- 3.3 Measuring scalability with Strong and Weak scaling . . . . . 23

**4. Implementation** . . . . . 25
- 4.1 Development Methodology . . . . . 26
- 4.2 Testing Methodology . . . . . 27
- 4.3 The Application Design . . . . . 28

**5. Results** . . . . . 30
- 5.0.1 Discussion . . . . . 34

**6. Conclusion** . . . . . 36

**Bibliography** . . . . . 37

**A. Milstein Discretisation** . . . . . 40
- A.1 Milstein Discretisation of the Heston Model . . . . . 42

# List of Figures

2.1 Progression of constant volatility and stochastic volatility processes . 9
(a) Constant volatility process . . . . . . . . . . . . . . . . . . . . . 9
(b) Stochastic volatility process . . . . . . . . . . . . . . . . . . . . . 9
2.2 Accumulation of adjoint results between time steps . . . . . . . . . . 10
(a) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
(b) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.3 Heston Asset and volatility curves for three correlated assets . . . . . 13
(a) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
(b) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

3.1 Code snippet using OpenMP for a Monte Carlo simulation . . . . . . 17
3.2 Streaming multiprocessor layout for Fermi architecture . . . . . . . . 19

5.1 Optimisation speedups on the GT950 (a) and K40 (b) graphics cards. 31
(a) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
(b) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
5.2 Execution dependency stalls visualised from NVIDIA's visual profiler 32

# List of Tables


2.1 Asset parameters . . . . . 9
2.2 Forward and adjoint elemental operations for GBM. . . . . 11
2.3 *Call on max* rainbow option using Geometric Brownian Motion . . . . 12
2.4 *Call on max* rainbow option using the Heston model . . . . . 14
2.5 Vanilla Call results for each asset using the Heston Model . . . . . 15
2.6 Elemental operations for Heston using Milstein discretisation . . . . 16

3.1 Generations of CUDA compute architectures . . . . . 21

4.1 CPU Hardware . . . . . 28
4.2 GPU Hardware . . . . . 28

5.1 *Call on max* run time in seconds . . . . . 30

# Chapter 1

# Introduction

Graphics Processing Units (GPUs) were initially designed to meet the demands of real-time graphics rendering. Thousands of simple arithmetic calculations need to be performed with millisecond turnaround times. The GPU achieves this with parallelism, centered on a multicore architecture to process similar instructions concurrently, much like a vector machine. GPUs were initially closed to fixed function units, but as photorealistic effects developed so did the need for more flexibility. Some of this flexibility came in the form of small programs to calculate different effects, called programmable shaders. These marked the first evolution of General Purpose Graphics Processing (GPGPU) as users began to leverage them for scientific and engineering tasks. In 2007, NVIDIA released the first truly general purpose language for GPU programming called Compute Unified Device Architecture (CUDA).

Financial applications began to leverage GPUs in the derivatives market. In 2006, Hanweck Associates developed a real-time volatility engine which evaluated all U.S listed equity options in under a second (Kirk (2007)). In 2007 Giles and Xiaoke (2008) created a LIBOR Model with a portfolio of swaptions which achieved a 149× speedup calculating 80 Deltas for 80 initial forward rates. Without Delta calculations they achieved a 400× speedup. In the same year Giles and Glasserman (2005) applied Adjoint Differentiation (AD) as a technique to calculate sensitivities for a LIBOR market model. The technique had been known to engineering for some time and is an efficient approach at calculating the gradient of a mathematical function. The authors were subsequently awarded Risk Quant of the year in 2007 (The Risk Awards 2007 (2007)).

Around 2008 GPUs began appearing in some of the world's top 500 supercomputers. Satoshi Matsuoka commented on the first adoption: "In testing our key applications, the Tesla GPUs delivered speed-ups that we had never seen before." (Humber (2008)). By 2015 one fifth of the supercomputers on top500.org (2016) contained GPUs (Brown (2015)) which contributed one third of the combined com-

puting power. They are also the most energy efficient. Bloomberg (2009) claim to have saved 10× the energy consumption using 48 servers equipped with GPUs by not having to scale up to a 1000 CPU cluster.

Recently, Gremse *et al.* (2016) used GPUs and AD techniques to price a basket call option of 10 FX rates with a 10 factor local volatility model. They used an algorithmic differentiation library called *dco* to assist in calculating sensitivities for 438 input parameters. The run time of their simulation reduced from 2 hours to 522ms using a single GPU with 66% of that time spent generating the *dco* data structure. Bernemann *et al.* (2011) measured the throughput of their Heston Model which managed 21 000 present value calculations on the CPU which scaled by 70× to 1.5 million on the GPU.

In this dissertation we calculate the price of a *Call on Max* Rainbow option with 3 underlying assets. In addition we calculate its Deltas and Vegas, for a total of 6 sensitivities. We do this with Monte Carlo simulation using the pathwise method and adjoint differentiation. Assets are simulated using the Heston Model with stochastic volatility. We use GPUs to analyze the acceleration of the entire process. Adjoint differentiation is implemented manually to avoid the overhead from third party libraries. This approach allows for a flexible implementation that scales to any simulation size or time step specified by the user. It also avoids the overhead from third party libraries. However, the solution is not generalizable.

Our implementation uses a single call to the GPU. This is different to most other listed approaches, such as Gremse *et al.* (2016), but it provided better results for our purpose. NVIDIA provides a library for generating random numbers on the GPU and its documentation advises pre-generating random numbers with a separate GPU call (Nvidia (2010)). We instead generate random numbers during calculations using the Philox random number generator, and again find this suites our purpose. Our implementation is largely inspired by the works of Volkov (2010) and Volkov and Demmel (2008) who draw on knowledge from vector machines to obtain speedups on the GPU with relatively unconventional techniques.

We test performance on two GPU architectures provided by NVIDIA: the GT950 from the Maxwell generation of GPUs intended for fast graphics; and the K40 from the Kepler generation built for professional computing. We find the Maxwell device provides competitive performance and is less than one tenth the price of the K40.

Chapter 2 introduces Rainbow Options, the Heston Model and Adjoint Differentiation. We present two examples for applying adjoint differentiation in a pathwise scenario. First we calculate a rainbow option using Geometric Brownian motion (GBM) with constant volatility, and then extend the concept to stochastic

volatility with the Heston Model. Chapter 3 discusses the concept of multithreading with models for the CPU that are also adopted by the GPU. Two measurements for scalability of an application are also discussed. Chapter 4 explains the development cycle, testing and implementation strategy. Chapter 5 contains the results from CPU and GPU performance experienced at various stages during development, with a discussion. We conclude with Chapter 6.

## Chapter 2

# Rainbow Options and Adjoint Differentiation with examples

## 2.1 Rainbow Options

An option is a financial instrument that gives the holder the right, but not the obligation to buy or sell an underlying asset at a predetermined time (the expiry time) at a predetermined price (the strike price). Call options give the right to buy while put options confer the right to sell. Rainbow Options are options that involve more than one underlying asset and the name associates each of these assets with a colour of the rainbow. For example, a rainbow call on the max with strike $K$ and expiry $T$ gives the holder the right to pay $K$ for the asset with the highest value.

The theory underlying rainbow options builds on the work of Black & Scholes. Margrabe (1978) evaluated the exchange of one risky asset for another at expiry and Stulz (1982) built on this to develop the two asset rainbow option, while Johnson (1987) generalized the concepts to multiple assets. A number of different rainbow option payoffs exist, for example, a *Best of Assets or Cash* option pays the maximum risky asset or cash at expiry. Presumably this payoff is positive and will always be exercised. A *Call on Max* gives the owner the right to purchase the maximum asset at strike on expiry, while a *Call on Min* gives the right to purchase the minimum asset.

| | |
|---|---|
| Best of Assets or Cash | $\max(s_1, s_2, ..., s_n, K)$ |
| Call on max | $\max(\max(s_1, s_2, ..., s_n) - K, 0)$ |
| Call on min | $\max(\min(s_1, s_2, ..., s_n) - K, 0)$ |

In this dissertation we price a *Call on max* option. The analytic formula for option price and Delta for any number of underlying assets in the Black-Scholes model is implicit in Ouwehand and West (2006). The formula for an N-asset rainbow option requires the evaluation of an N-dimensional normal cumulative distribution function. Functions computing this CDF are now more generally available,

but weren't at the time of writing the article. Those that existed were inaccurate, sometimes yielding negative option prices.

### Simulating a *call on max* rainbow option

To simulate the price of a *call on max* for N-many assets, let $S_{max} = \max(s_1(T), ..., s_n(T))$ be the maximum asset at time T and let M be the number of paths in the simulation.

$$C_{\max}(t) = e^{-r\tau}\mathbb{E}_{\mathbb{Q}}\left[\mathbb{I}_{\{S_{max}>K\}}\left(S_{max}-K\right)\right] \approx e^{-r\tau}\frac{1}{M}\sum_{i=1}^{M}\mathbb{I}_{\{S_{max,i}>K\}}\left(S_{max,i}-K\right) \tag{2.1}$$

The Delta is the sensitivity of the call price with respect to a change in the initial asset price. There is a Delta for each of the N assets underlying the option, and corresponding Vegas. The Vega is the sensitivity of the call price with respect to the initial variance (not the implied volatility). These are given for the $j^{th}$ asset by,

$$\begin{aligned}\frac{\partial C_{\max}(t)}{\partial S(t)^j} &\approx e^{-r\tau}\frac{1}{M}\sum_{i=1}^{N}\mathbb{I}_{\{S^i_{max}>K\}}\left(\frac{\partial S^i_{max}}{\partial S(t)^j}-K\right)\\ \frac{\partial C_{\max}(t)}{\partial V(t)^j} &\approx e^{-r\tau}\frac{1}{M}\sum_{i=1}^{N}\mathbb{I}_{\{S^i_{max}>K\}}\left(\frac{\partial S^i_{max}}{\partial V(t)^j}-K\right)\end{aligned} \tag{2.2}$$

We consider a simulation with 3 underlying assets, which amounts to a singe price with 3 Deltas and Vegas (6 sensitivities in total). These sensitivities are useful for dynamic hedging strategies, in which the portfolio is frequently rebalanced in an attempt to mitigate risk. These simulations tend to run for extended periods of time and may be infeasible, warranting acceleration.

## 2.2 The Heston Stochastic Volatility Model

The Heston Model is defined by a coupled two-dimensional SDE. Suppose the risk free rate is $r$, let $S_t$ be the asset price process at time $t$ and $V_t$ be the instantaneous variance of $dS_t$ which mean-reverts at $\theta$ with speed $\kappa$. Let $\sigma$ be the volatility of the variance. Also, let $dW_1 \cdot dW_2 = \rho dt$ for a constant correlation $\rho$. The Heston model is given by

$$\begin{aligned}dS_t &= rS_t dt + \sqrt{v_t}S_t dW_1\\ dV_t &= \kappa\left(\theta - V_t\right)dt + \sigma\sqrt{V_t}dW_2\end{aligned} \tag{2.3}$$

which is continuous and in order to simulate assets a discretization scheme needs to be chosen. Andersen (2007) detail a number schemes with their associated properties, including Euler and Milstein. We've chosen the Milstein discretization. The log stock process is used for $dS_t$ to ensure positivity. This is not the case for the volatility process $V_t$ which may become negative with positive probability so it is either truncated or reflected about 0 (see equation 2.4). Further, the volatility process is a square root diffusion so, in addition to possibly taking the square root of a negative, $\frac{\partial V_t}{\partial V_0}$ contains a $\frac{1}{\sqrt{V_t dt}}$ term which is a division by zero when $v_t = 0$. We use initial volatility parameters between 0.25 and 0.3 to help ensure $v_t > 0$ at all time steps. The Heston model reduces to Black-Scholes if $V_t$ is constant, which can be achieved by setting $\sigma = 0$ and $V_0 = \theta$.

The above Equation 2.3 describes the dynamics for a single asset. Multiple assets introduce asset-asset, asset-volatility and volatility-volatility correlations. For our purposes volatility-volatility correlation is ignored. Instead we use constant asset-volatility correlation where $\rho^i$ where $dW_1^i \cdot dW_2^i = \rho^i dt$ for asset $i$. We use asset-asset correlation between $dW_1^i$ and $dW_1^j$ which specified by a correlation matrix.

The Milstein discretisation using truncation is listed in equation 2.4. $Z_s$ and $Z_v$ are standard normal random numbers with correlation $\rho$. A concise derivation is provided in Rouah (2011) to which we refer readers. $V_t$ will remain non-negative, and even positive, if the Feller conditions are satisfied. However, discretization of the SDE may lead to negative $V_t$.

$$
\begin{aligned}
S_{t+dt} &= S_t \exp\left(\left(r - \frac{1}{2}V_t\right)dt + \sqrt{V_t dt} Z_s\right) \\
V_{t+dt} &= \left(V_t + \kappa\left(\theta - V_t\right)dt + \sigma\sqrt{V_t dt} Z_v + \frac{1}{4}\sigma^2 dt\left(Z_v^2 - 1\right)\right)^+
\end{aligned}
\tag{2.4}
$$

## 2.3 Adjoint Differentiation

Adjoint differentiation (AD) offers an alternate technique to finite differences for calculating sensitivities. AD reverses the chain rule known from elementary calculus to efficiently calculate the gradient of a mathematical function. In contrast, finite differences estimate sensitivities by independently bumping parameters. The novelty of using AD is that run time remains constant as the number of sensitivities increases. Griewank and Walther (2008) have proven the constant factor is between 2-4 times the time required for a single sensitivity. In addition, accuracy is to machine precision whereas, under certain conditions, finite difference may be subject to discretization errors. In this dissertation AD is used to calculate pathwise sensitivities.

### Forward and reverse modes of algorithmic differentiation

Adjoint differentiation is a subset of Algorithmic Differentiation containing two modes: forward and reverse, both of which use the chain rule. The forward mode calculates an input sensitivity with respect to all outputs simultaneously and is efficient for equations with more outputs than inputs. However, equations typically contain multiple inputs with a single output, such as a price, and in these cases reverse mode is more efficient. This is particularly beneficial in cases similar to Gremse *et al.* (2016) using 438 input parameters.

Reverse mode, commonly called adjoint mode, applies the chain rule in reverse to calculate the gradient of an output variable with a single pass. Both forward and adjoint methods use intermediate results to reduce the computational complexity. In Computer Science terminology this technique is called memoization. The equation is separated into constituent, elemental operations of multiplication, addition, etc. to which the chain rule is applied and its result is kept for proceeding calculations. In elementary calculus the chain rule is typically applied to symbolic expressions but in AD it is applied to numerical values.

To describe this slightly more formally, consider a function $f$ decomposed into elemental operations $g_i$. An elemental operation is either binary or unary and cannot be decomposed. The following assumes $f$ takes $M$ input parameters, is composed of $N$ elemental operations, and produces a single output $y$:

$$f(x_{\text{-1}}, x_{\text{-2}}, \ldots, x_{\text{-M}}) = (g_{\text{N}} \circ g_{\text{N-1}} \circ \ldots \circ g_0)(x_{\text{-1}}, x_{\text{-2}}, \ldots, x_{\text{-M}}) = y \tag{2.5}$$

The convention is to use negative indexing for input parameters. During an initial forward pass each $g_i$ result is stored in a variable $x_i$ which is used in the reverse pass to accumulate the gradient.

$$\nabla f(x_{\text{-}1}, x_{\text{-}2}, ..., x_{\text{-M}}) = \left(\frac{\partial}{\partial x_{\text{-}1}}, \frac{\partial}{\partial x_{\text{-}2}}, ..., \frac{\partial}{\partial x_{\text{-M}}}\right)(y)$$
$$= \left(\frac{\partial}{\partial x_{\text{-}1}}, \frac{\partial}{\partial x_{\text{-}2}}, ..., \frac{\partial}{\partial x_{\text{-M}}}\right)(g_{\text{N}} \circ g_{\text{N-1}} \circ ... \circ g_0) \quad (2.6)$$

Applying the chain rule to the composition results in the product

$$\left(\frac{\partial g_{\text{N}}}{\partial g_{\text{N-1}}} \cdot \frac{\partial}{\partial x_{\text{-}1}}, \frac{\partial g_{\text{N}}}{\partial g_{\text{N-1}}} \cdot \frac{\partial}{\partial x_{\text{-}2}}, ...\right)(g_{\text{N-1}} \circ g_{\text{N-2}} \circ ... \circ g_0) \quad (2.7)$$

$\frac{\partial g_{\text{N}}}{\partial g_{\text{N-1}}}$ is common to each element in the left vector so it is calculated once and stored in a variable $\bar{x}_{\text{N-1}}$. The process continues until the gradient is fully accumulated,

$$\nabla f(x_{\text{-}1}, x_{\text{-}2}, ..., x_{\text{-M}}) = \left(\frac{\partial y}{\partial x_{\text{-}1}}, \frac{\partial y}{\partial x_{\text{-}2}}, ..., \frac{\partial y}{\partial x_{\text{-M}}}\right) = (\bar{x}_{\text{-}1}, \bar{x}_{\text{-}2}, ..., \bar{x}_{\text{-M}}) \quad (2.8)$$

The intermediate $x_i$ storage reduces computational complexity at the expense of increased memory and may require excessive use of memory.

### Automatic differentiation

Software tools and techniques exist to automatically generate adjoint codes saving time and manual error. These solutions contain interesting challenges for Computer Science & Mathematics and are more challenging to accelerate.

Manual intervention is still generally required to accelerate AD. Gremse *et al.* (2016) use an algorithmic differentiation tool called *dco*. It generates a data structure called a *tape* which stores results from the adjoint process. These are manually copied to accelerated hardware such as the GPU for processing. Gremse *et al.* (2016) experience reductions in run time from 2 hours to 522ms with 66% of this run time spent constructing the *tape*. In this dissertation we implement adjoint calculations manually to remain flexible in memory layout and hardware optimisations. The disadvantage is that the implementation is tightly coupled with the discretization and is not generalizable.

## 2.4 Example Rainbow option with pathwise Adjoint Differentiation for GBM and Heston models

In this section we use two examples to contextualize the adjoint method with pathwise sensitivity calculations for a rainbow option. The first example uses Geometric Brownian Motion (GBM) with constant volatility. In this scenario we are able

to use analytic formulae to verify the simulation's price and Deltas. The second example uses stochastic volatility for the Heston Model. In this scenario we verify the price, Deltas and Vegas using analytic Fast Fourier Transform solutions, not for the rainbow option but for a vanilla call (essentially a rainbow call simplified to a single underlying asset). We then verify the multi-asset rainbow option using finite difference techniques.

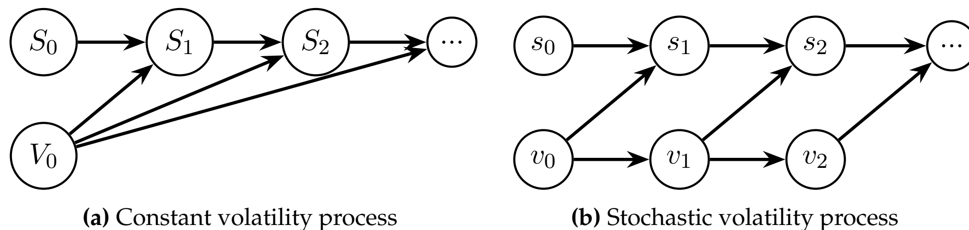
The rainbow option contains $3$ underlying assets whose properties are listed in table 2.1. The GBM example with constant volatility ignores Heston specific parameters. Each asset has a similar starting value with initial volatilities set to obscure a clear favourite. Assets 1&3 have asset-asset correlation $0.919$ while assets 1&2 is lower at $0.321$. Asset-volatility correlation is constant at $\rho = -0.4$ for all $3$ assets. Volatilities mean-revert around a realistically large value that ensures positivity. In all cases 360 time steps are used.

**Tab. 2.1:** Asset parameters

| # | $s_0$ | $v_0$ | $\theta$ | $\kappa$ | $\sigma$ | $\rho$ | r | T | K | Correlation matrix | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 100 | 0.25 | 0.25 | 9 | 0.5 | -0.4 | 0.05 | 1 | 150 | 1 | 0.321 | 0.919 |
| 1 | 95 | 0.3 | 0.3 | 9 | 0.5 | -0.4 | 0.05 | 1 | 150 | 0.321 | 1 | 0.54 |
| 2 | 97.5 | 0.28 | 0.28 | 9 | 0.5 | -0.4 | 0.05 | 1 | 150 | 0.919 | 0.54 | 1 |

Analytic equations commonly found in texts for Adjoint calculations are similar to simulations. The only real difference is that simulations introduce a layer of recursion because the output values are re-used as input parameters. A multi-time step simulation to calculate the price accumulates intermittent results between time steps to obtain a final price. In the adjoint process the gradient is also accumulated in a similar manner.

The constant volatility example produces a single output (the asset price) while Heston produces two outputs (asset price and volatility), illustrated in figure 2.1




**(a)** Constant volatility process

**(b)** Stochastic volatility process

**Fig. 2.1:** Progression of constant volatility and stochastic volatility processes

The dashed line in figure 2.2 illustrates the recursive connection between time steps for sensitivity outputs. This connection is particular to the simulation and separate from the Adjoint algorithm described in subsection 2.3.

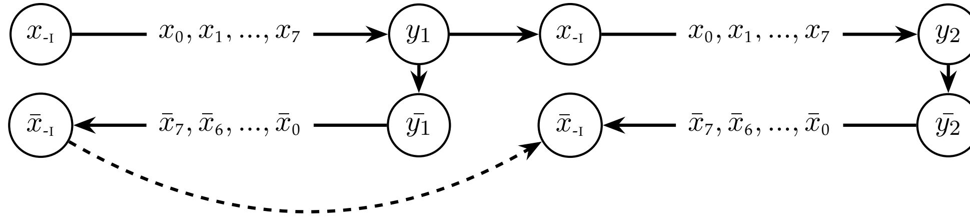


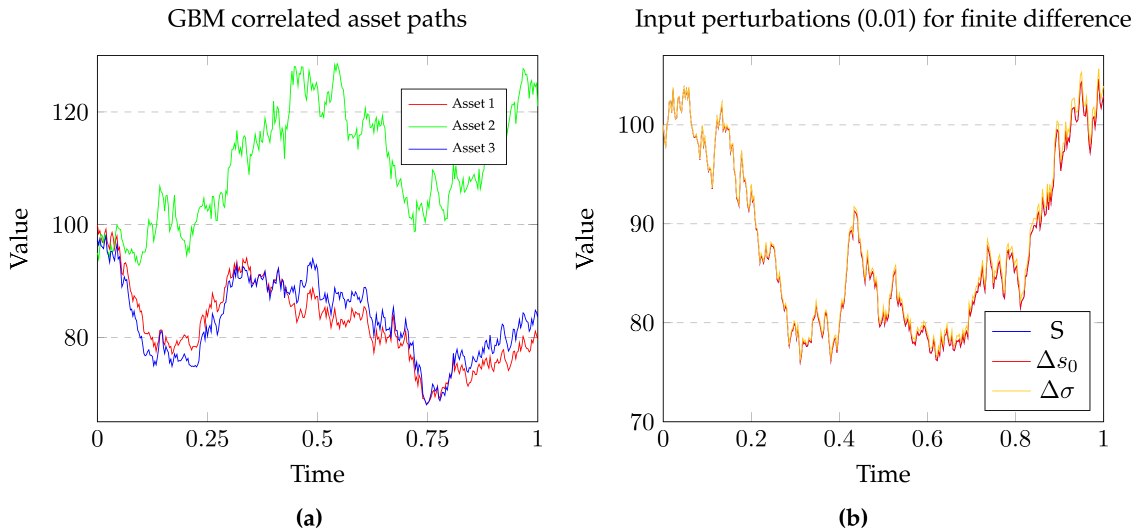

**Fig. 2.2:** Accumulation of adjoint results between time steps

### Pathwise adjoint sensitivities for GBM with constant volatility

Let the asset process at time $t$ be $S_t$ in time steps $dt$ and maturity $T$. Let the risk neutral measure be $\mathbb{Q}$ with associated risk free rate $r$. $Z$ is a normally distributed random number. Then,

$$S_{t+1} = S_t \exp\left(\left(r - \frac{1}{2}\sigma^2\right) dt + \sigma\sqrt{dt}Z\right) \tag{2.9}$$

This process is illustrated with three correlated paths in figure 2.3a using the assets from table 2.1.




We seek the Deltas and Vegas of the *Call on max* using adjoint techniques. To do so we simulate the above assets multiple times and average the results using equa-

tion 2.2. The first step is to decompose the GBM process into elemental operations, seen in the left column of Table 2.2 proceeding from top to bottom. Elemental results are populated to $x_i$ variables each dependent on those above themselves and produce asset price $y1$ for a single time step. The right column is the adjoint (reverse) method which executes from bottom to top accumulating the gradient in $\bar{x}_i$ variables using the chain rule where each variable depends on those below it.

**Tab. 2.2:** Forward and adjoint elemental operations for GBM.

| | |
|---|---|
| $x_{-4} = t$ | $\bar{x}_{-4} = \frac{\partial y}{\partial x_{-4}} = \bar{x}_3 \cdot \frac{\partial x_3}{\partial x_{-4}} = \bar{x}_3 \cdot x_2$ |
| $x_{-3} = r$ | $\bar{x}_{-3} = \frac{\partial y}{\partial x_{-3}} = \bar{x}_2 \cdot \frac{\partial x_2}{\partial x_{-3}} = \bar{x}_2 \cdot 1$ |
| $x_{-2} = \sigma$ | $\bar{x}_{-2} = \frac{\partial y}{\partial x_{-2}} = \bar{x}_4 \cdot \frac{\partial x_4}{\partial x_{-2}} + \bar{x}_0 \cdot \frac{\partial x_0}{\partial x_{-2}} = \bar{x}_4 \cdot W_t + \bar{x}_0 \cdot 2x_{-2}$ |
| $x_{-1} = S_0$ | $\bar{x}_{-1} = \frac{\partial y}{\partial x_{-1}} = \bar{x}_7 \cdot \frac{\partial x_7}{\partial x_{-1}} = \bar{x}_7 \cdot x_6$ |
| $x_0 = \sigma^2 = x_{-2}^2$ | $\bar{x}_0 = \frac{\partial y}{\partial x_0} = \bar{x}_1 \cdot \frac{\partial x_1}{\partial x_0} = \bar{x}_1 \cdot \frac{1}{2}$ |
| $x_1 = \frac{x_0}{2}$ | $\bar{x}_1 = \frac{\partial y}{\partial x_1} = \bar{x}_2 \cdot \frac{\partial x_2}{\partial x_1} = \bar{x}_2 \cdot -1$ |
| $x_2 = r - x_1 = x_{-3} - x_1$ | $\bar{x}_2 = \frac{\partial y}{\partial x_2} = \bar{x}_3 \cdot \frac{\partial x_3}{\partial x_2} = \bar{x}_3 \cdot x_{-4}$ |
| $x_3 = x_2 \times t = x_2 \times x_{-4}$ | $\bar{x}_3 = \frac{\partial y}{\partial x_3} = \bar{x}_5 \cdot \frac{\partial x_3}{\partial x_5} = \bar{x}_5 \cdot 1$ |
| $x_4 = \sigma \times W_t = x_{-2} \times W_t$ | $\bar{x}_4 = \frac{\partial y}{\partial x_4} = \bar{x}_5 \cdot \frac{\partial x_5}{\partial x_4} = \bar{x}_5 \cdot 1$ |
| $x_5 = x_3 + x_4$ | $\bar{x}_5 = \frac{\partial y}{\partial x_5} = \bar{x}_6 \cdot \frac{\partial x_6}{\partial x_5} = \bar{x}_6 \cdot x_6$ |
| $x_6 = \exp(x_5)$ | $\bar{x}_6 = \frac{\partial y}{\partial x_6} = \bar{x}_7 \cdot \frac{\partial x_7}{\partial x_6} = 1 \cdot x_{-1}$ |
| $x_7 = S_0 \times x_6 = x_{-1} \times x_6$ | $\bar{x}_7 = \frac{\partial y}{\partial x_7} = 1$ |
| $y_1 = x_7$ | $\bar{y}_1 = \frac{\partial y_1}{\partial y_1} = 1$ |

The adjoint operations in the above table do not include the calculations represented by the dashed line in figure 2.2. These calculations are described by Equations 2.10 & 2.11 for the Deltas and Vegas and need to be chained to the adjoints $\bar{x}_{-1}$ and $\bar{x}_{-2}$ respectively. The Deltas are accumulated with an application of the product rule,

$$\frac{\partial S_t}{\partial S_0} = \frac{\partial}{\partial S_0}\left(S_{t-1}\exp(...)\right) = \frac{\partial S_{t-1}}{\partial S_0}\exp(...) + S_{t-1}\frac{\partial}{\partial S_0}\exp(...) \tag{2.10}$$

In the above the right hand side of the summation is zero and the equation conveniently reduces to $\frac{\partial S_{t-1}}{\partial S_0}\exp(...)$ which simplifies to $\frac{\partial S_t}{\partial S_0} = \frac{S_T}{S_0}$ since ultimately $\frac{\partial S_0}{\partial S_0} = 1$.

The Vegas are accumulated again with an application of the product rule,

$$\begin{aligned} \frac{\partial S_t}{\partial V_0} &= \frac{\partial S_{t-1}}{\partial V_0} \exp(\ldots) + S_{t-1} \frac{\partial}{\partial V_0} \exp(\ldots) \\ &= \frac{\partial S_{t-1}}{\partial V_0} \exp(\ldots) + S_{t-1} \left(-\sigma dt + \sqrt{dt} Z\right) \end{aligned} \tag{2.11}$$

Table 2.3 contains the price and Deltas using adjoint differentiation for the simulated rainbow option. It includes results from both finite difference techniques and the analytic solution. In this simulation we used the Heston model manipulated as a GBM process by overriding the parameters. Specifically, we set $\kappa = 0 = \sigma$. Finite difference results were obtained with increments of $0.01$.

**Tab. 2.3:** *Call on max* rainbow option with $3$ correlated assets using the Heston model with overridden parameters to mimic Geometric Brownian Motion. The price using the analytic equation is 2.71634379091099. The price using finite differences is 2.71364952301177 and the price using adjoint differentiation is 2.71364952301177

| Method | Delta $S_1$ | Delta $S_2$ | Delta $S_3$ |
|---|---|---|---|
| Analytic | 0.054023113645 | 0.097674656054 | 0.06706777284 |
| Finite Difference | 0.0549076808059 | 0.0974200533840 | 0.0661378277104 |
| Adjoint Differentiation | 0.0548977154275 | 0.0974059348977 | 0.0661327767784 |

### Pathwise adjoint sensitivities for Heston with stochastic volatility

An example path for the 3 assets from table 2.1 using the Heston model is illustrated in figure 2.3. Note the asset prices appear scaled in comparison to the GBM paths due to larger volatility values from its stochastic nature. The stochastic volatilities are seen to mean-revert about their starting values. Asset 2 clearly shows the negative asset-volatility correlation around time period 0.75.

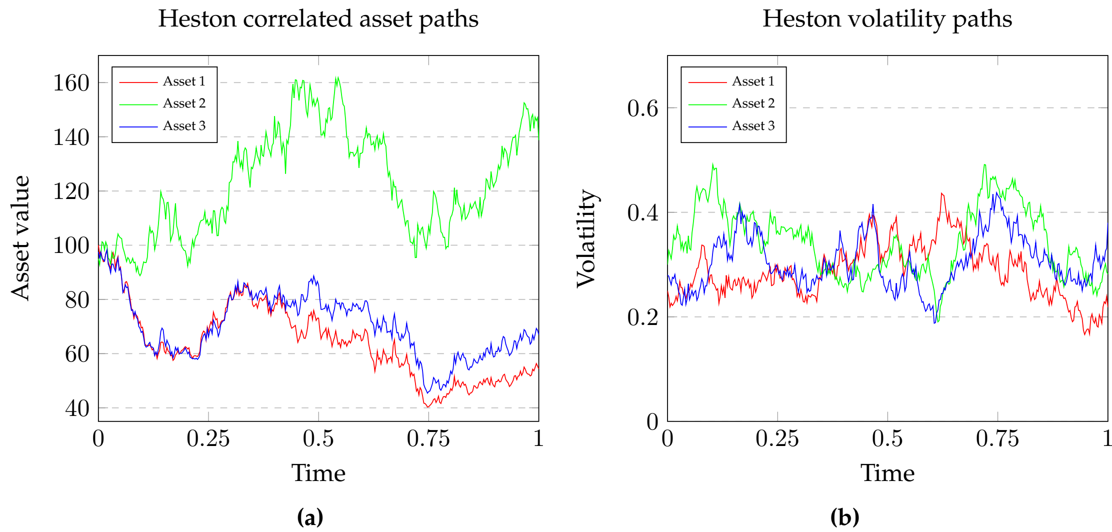



**Fig. 2.3:** Heston Asset and volatility curves for three correlated assets

Equation 2.3 contains two outputs to consider for adjoint calculations and 8 inputs. Table 2.6 lists the outputs as $y_1$ (asset value) and $y_2$ (volatility value). It also includes elemental operations from the Milstein discretization for the forward and adjoint pass. The table is intended to be read from top to bottom and operations are categorized by dependency in rows, with each row dependent on those above themselves.

The forward pass uses all four columns of the table while only two columns are reserved for the adjoint pass. This is to distinguish adjoint operations for the two outputs $y_1$ and $y_2$. Note that row 4 in the adjoint pass contains a chain of equalities but are kept in the same dependency layer since the assignment might as well occur in a single statement. Superscripts with $v$ denote sensitivities related to the volatility process $y_2$, such as $\bar{x}^v_{\text{-}2}$ for $\frac{\partial y2}{\partial v_0}$ whereas $s$ superscripts denote sensitivities related to the asset process $y_1$ such as $\bar{x}^s_{\text{-}2}$ for $\frac{\partial y1}{\partial v_0}$. These superscripts were added to avoid ambiguity between the two columns.

The final layer includes operations that accumulate sensitivities between time steps which are not part of the adjoint algorithm. $\bar{x}_{\text{-}1} = \frac{\partial S_t}{\partial S_0}$ for the Delta behaves

the same as in equation 2.10 and is not included in the equation below. The adjoint $\bar{x}_{\text{-}2} = \frac{\partial S_t}{\partial V_0}$ used for the Vega requires both $\frac{\partial S_t}{\partial V_0}$ and $\frac{\partial V_{t-1}}{\partial V_0}$ detailed in equations 2.12 & 2.13 respectively. Letting $L = \left(r - \frac{1}{2}V_{t-1}\right)dt + \sqrt{V_{t-1}dt}Z_s$ we have

$$\begin{aligned}
\frac{\partial S_t}{\partial V_0} &= \frac{\partial}{\partial V_0}\left(S_{t-1}\cdot\exp\left(L\right)\right) \\
&= \frac{\partial S_{t-1}}{\partial V_0}\cdot\exp\left(L\right) + S_{t-1}\cdot\frac{\partial\exp\left(L\right)}{\partial V_0} \\
\text{where } \frac{\partial\exp\left(L\right)}{\partial V_0} &= \exp\left(L\right)\cdot\frac{\partial L}{\partial V_0} \\
&= \exp\left(L\right)\cdot\frac{\partial V_{t-1}}{V_0}\cdot dt\cdot\frac{1}{2}\cdot\left(-1+\frac{Z_s}{\sqrt{V_{t-1}dt}}\right)
\end{aligned} \tag{2.12}$$

Similarly,

$$\begin{aligned}
\frac{\partial V_{t-1}}{\partial V_0} &= \frac{\partial}{\partial V_0}\left(\left(V_{t-2}-\kappa\left(\theta-V_{t-2}\right)dt+\sigma\sqrt{V_{t-2}dt}Z_v+\frac{1}{4}\sigma^2dt\left(Z_v^2-1\right)\right)^+\right) \\
&= \mathbb{I}_{\{V_{t-1}>0\}}\left(\frac{\partial V_{t-2}}{\partial V_0}-\kappa\frac{\partial V_{t-2}}{\partial V_0}dt+\sigma\frac{1}{2\sqrt{V_{t-2}dt}}\frac{\partial V_{t-2}}{\partial V_0}dtZ_v\right)
\end{aligned} \tag{2.13}$$

The rainbow option price, Deltas and Vegas are listed in Table 2.4 for forward difference and adjoint techniques. Although a closed form solution is not available in this setting the Heston model dynamics are verified with vanilla call options on each asset using the Fast Fourier Transform. Results for these are contained in table 2.5. Prices for the *Call on max* in the table below are as follows. The finite difference price is 16.0863852941686 and the Adjoint differentiation price is 16.0863852941686.

**Tab. 2.4:** *Call on max* rainbow option with 3 correlated assets using the Heston model with original parameters.

| Method | Delta $S_1$ | Delta $S_2$ | Delta $S_3$ |
|---|---|---|---|
| Finite Difference | 0.192291878681 | 0.251813882935 | 0.155906951066 |
| Adjoint Differentiation | 0.192221690500 | 0.251779381776 | 0.155819943406 |
| | Vega $S_1$ | Vega $S_2$ | Vega $S_3$ |
| Finite Difference | 2.46037476166 | 2.74329704531 | 1.93522805387 |
| Adjoint Differentiation | 2.44853959824 | 2.74396866117 | 1.91795273434 |

Prices for the three vanilla call options with respect to $S_1$, $S_2$, $S_3$ are,

$$\text{Fast Fourier Transform: } S_1 = 7.87474894, S_2 = 7.938299573, S_3 = 8.100948355$$
$$\text{Finite Difference: } S_1 = 7.84501521, S_2 = 7.911827615, S_3 = 8.071811243$$
$$\text{Adjoint Differentiation: } S_1 = 7.84501521, S_2 = 7.911827615, S_3 = 8.071811243$$

**Tab. 2.5:** Vanilla Call results for each asset using the Heston Model and original parameters.

| Method | Delta $S_1$ | Delta $S_2$ | Delta $S_3$ |
|---|---|---|---|
| Analytic | 0.321991501542 | 0.318558344525 | 0.323865333104 |
| Finite Difference | 0.320016635614 | 0.316812005760 | 0.321950313955 |
| Adjoint Differentiation | 0.31998305997 | 0.316784085350 | 0.321928411625 |
| | Vega $S_1$ | Vega $S_2$ | Vega $S_3$ |
| Analytic | 3.964307167163 | 3.422866357729 | 3.661480215957 |
| Finite Difference | 4.00257345292 | 3.46342567025 | 3.70441430833 |
| Adjoint Differentiation | 4.00030833111 | 3.4636078863 | 3.70514120772 |

Note in table 2.6 the forward pass contains 25 elemental operations over 7 layers, an average independence of 3.57. The adjoint pass is sparser after omitting unused operations, totalling 22 operations.

**Tab. 2.6:** Elemental operations for Heston using Milstein discretisation

| | | | | |
|---|---|---|---|---|
| | $x_{-7} = \kappa$ | $x_{-6} = \theta$ | $x_{-5} = \sigma$ | $x_{-4} = dt$ |
| | $x_{-3} = r$ | $x_{-2} = v_0$ | $x_{-1} = s_0$ | |
| 1 | $x_0 = x_{-2} \times x_{-4}$ | $x_1 = x_{-3} \times x_{-4}$ | $x_2 = x_{-6} \times x_{-4}$ | $x_3 = (x_{-5})^2$ |
| | $x_4 = z_v \times z_v$ | | | |
| 2 | $x_5 = \sqrt{x_0}$ | $x_6 = x_3 \times x_{-4}$ | $x_7 = x_{-7} \times x_2$ | $x_8 = x_{-7} \times x_0$ |
| | $x_9 = \frac{-1}{2} \times x_0$ | $x_{10} = x_4 - 1$ | | |
| 3 | $x_{11} = x_{-5} \times x_5$ | $x_{12} = z_s \times x_5$ | $x_{13} = x_{-2} + x_7$ | |
| | $x_{14} = \frac{1}{4} \times x_{10}$ | | | |
| 4 | $x_{15} = x_9 + x_{12}$ | $x_{16} = z_v \times x_{11}$ | $x_{17} = x_{13} - x_8$ | $x_{18} = x_{14} \times x_6$ |
| 5 | $x_{19} = x_1 + x_{15}$ | $x_{20} = x_{17} + x_{16}$ | | |
| 6 | $x_{21} = x_{20} + x_{18}$ | $x_{22} = \exp(x_{19})$ | | |
| 7 | $x_{23} = (x_{21})^+$ | $x_{24} = x_{-1} \times x_{22}$ | | |
| | $y_2 = x_{23}$ | $y_1 = x_{24}$ | | |
| | $\bar{y}_2 = 1$ | $\bar{y}_1 = 1$ | | |
| 7 | $\bar{x}_{23} = \bar{y}_2$ | $\bar{x}_{24} = \bar{y}_1$ | | |
| 6 | $\bar{x}_{21} = \mathbb{I}_{\{x_{23}>0\}} \bar{x}_{23}$ | $\bar{x}_{22} = x_{-1} \cdot \bar{x}_{24}$ | | |
| 5 | $\bar{x}_{20} = \bar{x}_{21}$ | $\bar{x}_{19} = \bar{x}_{22} \cdot x_{22}$ | | |
| 4 | $\bar{x}_{17} = \bar{x}_{20}$ | $\bar{x}_{15} = \bar{x}_{19}$ | | |
| | $\bar{x}_{16} = \bar{x}_{17}$ | $\bar{x}_{12} = \bar{x}_{15}$ | | |
| | $\bar{x}_{13} = \bar{x}_{16}$ | $\bar{x}_9 = \bar{x}_{15}$ | | |
| 3 | $\bar{x}_{11} = \bar{x}_{16} \cdot z_v$ | $\bar{x}_5 = \bar{x}_{12} \cdot z_s$ | | |
| | $\bar{x}_8 = \bar{x}_{17} \cdot -1$ | $\bar{x}_1 = \bar{x}_{19}$ | | |
| | | $\bar{x}_0^s = \bar{x}_9 \cdot -\frac{1}{2}$ | | |
| 2 | $\bar{x}_5 = \bar{x}_{11} \cdot \bar{x}_{-5}$ | | | |
| | $\bar{x}_0^v = \frac{1}{2} \cdot \frac{\bar{x}_5}{x_5} + \bar{x}_8 \cdot x_{-7}$ | | | |
| | $\bar{x}_{-2}^v = \bar{x}_{-2}^v \cdot (\bar{x}_{13} + \bar{x}_0 \cdot x_{-4})$ | $\bar{x}_{-1} = (\bar{x}_{24} \cdot x_{22}) \cdot \bar{x}_{-1}$ | | |
| | | $\bar{x}_{-2}^s = \bar{x}_{-2}^s \cdot x_{22} - (\bar{x}_0 \cdot x_{-4}) \cdot \bar{x}_2^v \cdot \left(-1 + \frac{Z_s}{x_5}\right)$ | | |

# Chapter 3

# Multithreading

In this chapter we introduce two multithreading models, the Graphics Processing Unit (GPU) and the three dimensions for achieving parallelism on the GPU. A program's run time can be reduced with one of two methods, either by upgrading the hardware on which it runs (vertical scaling) or distributing the workload among multiple processors (horizontal scaling). A thread in computing terms is an execution context for a set of instructions. A program may be composed of multiple sub contexts which are able to execute independently on separate processing units. Multithreading achieves horizontal scaling by concurrently executing such threads over multiple cores.

The two primary multithreading models are Fork/Join and the Message Passing Interface (MPI). Fork/Join implements multithreading at the block level such as *for* loops and *while* statements while MPI implements multithreading at the data structure level. In this dissertation we are concerned with the Fork/Join model as it is used to distribute the Monte Carlo simulation over multiple cores, particularly on the GPU. We will briefly consider an implementation on the Central Processing Unit (CPU) using the OpenMP library. The library extends compilers with directives that instructs the thread execution for a block of code using a single statement, as in the following snippet for a Monte Carlo simulation

```
double payoff\_sum = 0;
#pragma omp parallel for reduction(+:payoff\_sum)
for (int i = 0; i < sim_size; i++) {
    // ...
    payoff\_sum = payoff\_sum + max(s - K, 0.0);
}
```

**Fig. 3.1:** Example using OpenMP to multithread a Monte Carlo simulation. *payoff_sum* is shared by threads while block level variables are private.

the *parallel for* statement tells the compiler that the proceeding *for* loop must be executed in parallel over multiple threads. The *reduction(+:payoff_sum)* ensures the reduction phase of the Monte Carlo simulation accumulates the *payoff_sum* safely. This value is shared between threads which write to it with the + operator. If this mechanism is not in place the value will become corrupted. The mechanics of the thread's execution and life cycle is abstracted from the programmer and is the responsibility of the runtime environment.

## 3.1 Graphics Processing Units

A Graphics Processing Unit (GPU) is similar to a vector unit (Volkov and Demmel (2008)) and acts as a coprocessor to the CPU. Code is implemented at the thread level which correspond to an index in a grid of blocks, similar to a pixel coordinate in an image. The GPU procedure call specifies the grid and block sizes for up to 3 dimensions. In essence each thread executes a single iteration of the *for loop* in Figure 3.1, which is made redundant if the number of iterations is equivalent to the grid size.

Threads are assigned to cores on a CPU differently to a GPU. A multicore CPU is restricted to executing one thread per processing core, containing an arrangement of arithmetic units. In contrast, the GPU does not contain physical cores and is instead equipped with a large number of arithmetic units that is able to execute tens of thousands of concurrent threads. For example, the Nvidia K40 is capable of scheduling 30 720 threads per clock cycle in 1.14ns.

### GPU processing workflow

The first stage is generally to allocate memory on the host computer to hold data prior to executing on the GPU and to hold results afterwards. The CPU allocates corresponding memory on the GPU device through its Global memory layer, which is accessible by the host computer, and copies data into it. The GPU procedure, called a kernel, is given pointers to these global memory addresses so that threads know where to access data. The grid and block size is also specified with the kernel call.

Threads pull data from global memory into various memory layers closer to arithmetic units so that calculations execute at lower latencies. Results are stored back to global memory and when all threads finish executing the control is returned to the CPU which copies them from the device onto the host to be used with the rest of the application. The CPU then continues execution, possibly making additional kernel calls.

### Memory layout

The memory layers on a GPU provide finer grained parallelism than the CPU. Memory which is addressable between threads is shared but is otherwise thread local. Memory closer to arithmetic units has lower latency and smaller capacity, such as Shared memory, L1 cache and registers. These are physically arranged around arithmetic units into identical units called Streaming Multiprocessors (SM) (see Figure 3.2). Each SM is equipped with multiple warp schedulers so that it can schedule blocks of threads independently. Off chip memory lies outside the SMs and has higher capacity and larger latency, such as L2 cache, Local, Texture and Global memory.

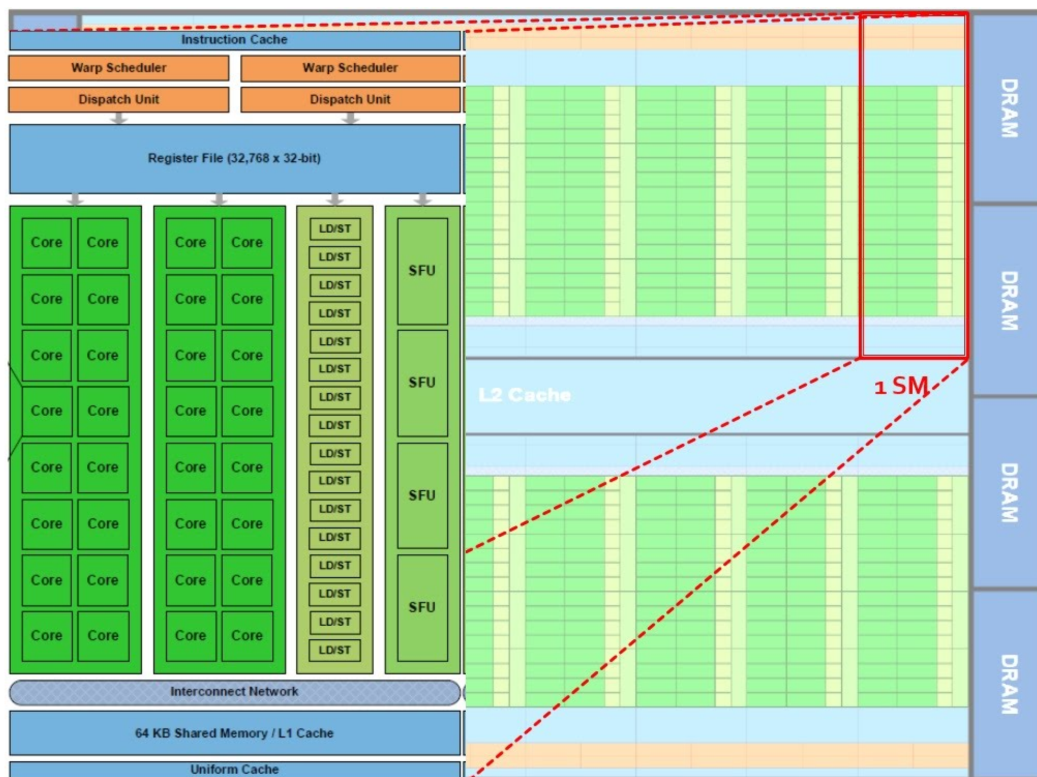



**Fig. 3.2:** The block on the left is an enhancement of a single streaming multiprocessor, it's layout is an arrangement of cores, arithmetic units surrounded by on-chip memory layers, and warp schedulers.

### Grid block layout and execution

The grid and block size is determined by the user to suite the applications needs. At run time each thread uses its grid index as a unique identifier which is mapped to a unique address in memory. There are a number of possible grid layout choices to

model a multi-asset Monte Carlo simulation. Three assets may be simulated with a 1-dimensional grid associating a path to each column. For performance reasons block sizes must be a multiple of 32, and 64, 128 or 256 are reliable block sizes for most applications. Choosing a block size of 64 the dimensions would be specified as $1\times64$. Each of the 64 threads in a block is then required to process all three assets through all the time steps.

An alternative approach is to process only a single asset per thread and use a grid with multiple rows representing each asset. Each thread within the block of 64 threads then needs to share asset-asset correlation data with the threads in their column. This is problematic with 3 assets and in fact any number of assets not a multiple of 32. If the block size does not divide the number of paths, i.e. $(50\,000 \bmod 64) \neq 0$, then the grid block size over specifies the number of threads with $782 \times 64 = 50\,048$. An *if* statement in code must be used to clip thread execution to within the 50 000 path boundary. Failing to do so results in threads reaching outside the allocated global memory ranges and causing exceptions to be thrown.

## Three dimensions of GPU parallelism

Successive generations of GPUs increase hardware capacity with the possibility of additional features. There are three aspects to the hardware consistent between generations which may be exploited to future proof GPU code and improve performance on existing GPUs. This subsection details these aspects: Thread level Parallelism (TLP), Instruction Level Parallelism (ILP) and Single Instruction Multiple Data (SIMD).

### Thread Level Parallelism

TLP relates to the concurrent execution of blocks. This may be increased in two ways, either with additional SMs in future GPUs or by tweaking grid block dimensions to facilitate hardware resources executing at capacity for the number of active blocks. Currently most SMs have a maximum active block count of $64$. At each clock cycle warps of $32$ threads are scheduled per SM from among active blocks.

### Single Instruction Multiple Data

These warps of $32$ threads execute identical instructions in lock-step through the instruction pipeline and each reference different addresses in memory. This is known as Single Instruction Multiple Data and is fixed by hardware to a width of $32$ instructions. SIMD may only be improved on future generations by increasing the

instruction pipeline width. Block sizes which are not multiples of 32 result in unused instruction units, wasting performance.

SIMD is implemented using pipelining to keep instruction units busy with multiple in flight instructions at different stages of execution. Instructions may stall on execution or memory dependencies and the scheduler will swap them for active threads to keep the pipe free of bubbles of inactivity. Latencies taking 400-600 clock cycles are hidden by pipelining. High occupancy is required to hide latencies and improve performance but Volkov and Demmel (2008) detail techniques to achieve better performance at lower occupancy.

#### Instruction Level Parallelism

The SIMD width of 32 has not improved between generations but arithmetic units increase continuously as transistor density improves. To utilise additional units warp schedulers improve to increase the number of instructions issued per thread per clock cycle. Historically, limited support for doubles could only couple independent double-floating-point instructions with single-floating-point instructions. This is no longer the case on some GPUs and future generations may improve scheduling further by increasing the number of independent instructions per thread.

## 3.2 Compute Unified Device Architecture

Compute Unified Device Architecture (CUDA) is a library released by NVIDIA facilitating general purpose processing on capable GPUs. It extends programming languages such as Fortran, C and C++ allowing native development. CUDA matches the evolution of NVIDIA GPUs with a programming interface to access new features. Table 3.1 lists the generations together with compute capabilities

**Tab. 3.1:** Generations of CUDA compute architectures

| Year | 2008 | 2011 | 2013 | 2015 |
|---|---|---|---|---|
| Generation | Tesla | Fermi | Kepler | Maxwell |
| Architecture | GT200 | GF104 | K40 | GTX980 |
| Compute capability | 1.2 | 2.1 | 3.5 | 5.2 |
| Multiprocessors | 30 | 7 | 15 | 16 |
| Cores/Multiprocessor | 8 | 48 | 192 | 128 |
| Number of Cores | 240 | 336 | 2880 | 2048 |

This section briefly describes Kepler and Maxwell and features useful to finance, technical details can be found in NVIDIA (2012). Chapter 5 discusses perfor-

mance results from the K40 (GK110 architecture) from Kepler, and the GT950 from Maxwell.

### Host and device Application Programming Interface

CUDA provides vectorized data structures compatible with host and device *float2*, *float4*, *dim3*. GPU procedures are annotated in code with the *__device__* prefix and called from host code with a set of parameters encased in triple angle bracket,

```
__device__ void monte_carlo_GPU() { ... } // declaration
monte_carlo_GPU<<<gridDim, blockDim>>>(); // procedure call
```

The NVCC compiler driver compiles device code to PTX assembly while the host compiles the rest of the application in its native build process. Compiled device code can either be linked to the application or built into a standalone binary containing the CUDA run time driver.

### Considerations for financial applications

Double floating point arithmetic has better support on Kepler devices and should be used for financial calculations requiring the additional accuracy. Double arithmetic is supported at $\frac{1}{3}$ the speed of single precision while performance on Maxwell is $\frac{1}{27}$ the speed. Kepler can schedule a pair of double arithmetic instructions per clock cycle while Maxwell supports only single/double and single/single instructions.

Dynamic Parallelism is a feature supported by Kepler and Maxwell. It enables threads to recursively spawn additional kernels at grid sizes specified during run time. Dynamic Parallelism may apply to multi-level Monte Carlo simulations achieving superior workflows for generating correlated random numbers prior to simulation. In this situation a kernel may generate any amount of correlated random numbers, execute the corresponding simulation kernel, and recurse until convergence is reached. A reduction kernel from the pathwise method may also be included. Unlike inline function calls dynamic parallelism does not affect register pressure nor concurrency. Prior to this feature the CPU was required to launch kernels individually at an overhead of 7-14ms.

### Massively multithreaded random number generation

Machine implementations of random number generators are deterministic with inherently serial algorithms. A state variable is all the configuration needed to

generate a deterministic random number and is updated after calls to generate sequences. In multithreaded environments each thread must hold a unique state variable and sub-sequences must not overlap. In massively multithreaded environments tens of thousands of state variables are held simultaneously and the state size and computational complexity affects concurrency and performance.

Manssen *et al.* (2012) give a broad review of CUDA random number generators including Lagged Fibonacci, XORShift and Counter Based generators. The Mersenne Twister is a lagged fibonacci generator but its large state is unsuitable for GPU use, even with versions of smaller state. Fibonacci generators generate random numbers in parallel sweeps. The Mersenne Twister allows 256 threads to generate numbers simultaneously but this dictates the block size. XORShift uses binary XOR and shift operations which are faster than addition, subtraction and division. However, the state requires $192$ bits per thread. Manssen *et al.* (2012) were able to find a generator with $1024$ bits of state and maximal length $2^{1024} - 1$ and implement it on the GPU by dividing the state into $32$ bit words. Each thread in a warp of 32 threads cooperates in updating $1$ bit. Using *skip-ahead* the state space is partitioned into sub-streams used by different warps. Counter based generators are used in secret-key cryptography more so than simulations and are developed to DES (Data Encryption Standard) and AES (Advanced Encryption Standards). Unlike the other types these are non-recursive and therefore do not rely on a state variable. The $n^{th}$ number in a sequence is directly determined by some function $f_k(n)$, where $k$ is some key of the parameter space. The Philox generator is counter based and uses $128$ bit state (period of $2^{128}$) but it does not need multiple loads and stores. Different keys $k$ generate independent random streams as a result of Kerckhoffs' principle [1] and a $64$ bit key results in $2^{64}$ independent sequences.

An important consideration for section 4.3 is that counter based generators, identified by Salmon *et al.* (2011), allow random numbers to be generated on the fly. They conclude that the Philox and Threefly families are among the fastest.

## 3.3 Measuring scalability with Strong and Weak scaling

This section introduces Strong and Weak scaling to analyse an applications potential acceleration to multithreading. Both provide expectations for an applications acceleration guiding the parallelization strategy. Strong scaling is synonymous with acceleration while weak scaling is synonymous with throughput. NVidia (2016) states, "The performance benefit of a GPU depends on the extent to which

[1] Kerckhoffs' principle states that *A cryptosystem should be secure even if everything about the system, except the key, is public knowledge*

the application can be parallelised."

### Strong Scaling

Strong scaling, equated with Amdahl's Law, measures how the time to solution decreases when additional processors are added to the system. Linear strong scaling implies run time reduces linearly to the number of processors. Let $P$ be the portion of run time from code which can be parallelised, $N$ the number of cores and $S$ the maximum speedup. An upper bound is obtained setting $N = \infty$ using equation 3.1. With $P = \frac{3}{4}$ we expect a maximum speedup of just 4.

$$S = \frac{1}{1 - P + \frac{P}{N}} \tag{3.1}$$

### Weak Scaling

NVidia (2016) states, "Weak scaling is a measure of how the time to solution changes as more processors are added to a system with a fixed problem size per processor". Equated with Gustafson's Law it measures how the problem size scales with the number of processors. Let $W$ be work done in serial and $W(s)$ work done having increased the number of cores by factor of $s$ then the speedup $S$ is,

$$\begin{aligned} W(s) &= (1 - P)W + sPW \\ S = \frac{W(s)}{W} &= (1 - P) + sP \end{aligned} \tag{3.2}$$

Weak scaling is appropriate for applications where the problem size is unknown, such as Monte Carlo simulations that execute until convergence is reached.

## Chapter 4

# Implementation

Ideally threads should be lightweight, containing a small number of computations and memory usage. This is not possible due to overhead from the adjoint procedure, and as a result the approach used here centres around compute intensive threads. The additional memory requirement from storing intermediate adjoints also reduces the amount of available storage. Since multiple simulation sizes need to be supported for testing run times, we are forced to use a constant amount of memory per thread. The following constraints are made from these two points: Firstly, we cannot run multiple kernels since doing so requires results to be persisted to global memory between calls. A common use case is to pre-generate random numbers to global memory in one kernel, and use them for calculations with another kernel. We are instead forced to generate random numbers on the fly. Another use case is to separate the forward and adjoint calculations into individual kernels. However, this requires intermediate results to be stored concurrently for all time steps.

Our approach is to develop a single kernel which generates random numbers during forward calculations, and stores intermittent results in local memory for adjoint calculations. A feature is that threads use a constant amount of memory irrespective of simulation or step size by recycling local variables. Since variables are thread local, threads must correlate and process each asset in a tuple of paths. This approach does not scale well with the number of assets, which are assumed to remain constant. To support any asset size we would need to be careful in choosing the grid/block dimensions to ensure blocks are always a multiple of 32.

Pseudocode is contained in Algorithm 1 for our implementation. Lines 4-16 contain the Monte Carlo simulation which is executed entirely on the GPU. Lines 17-21 comprise the reduction phase which is executed on the CPU (although this may also take place on the GPU). Line 3 allocates memory which also occurs on the CPU.

**Algorithm 1** Pseudo code for simulating a rainbow option

```
1: procedure BEGIN SIMULATION(simsize, stepsize, assets, correlation)
2:     /* p=payoff, d=Delta, v=Vega, n=size(assets) */
3:     psum = dsum[n] = vsum[n] = 0
4:     for i = [1, 2, ..., simsize] do
5:         z = generate_normal_rnds(correlation, n)
6:         pmax = 0
7:         for a = [1, 2, ..., n] do
8:             p, d, v = 0, 1, 0
9:             for t = [1, 2, ..., stepsize] do
10:                p, d, v = accumulate(p, d, v, z[a], assets[a])
11:            end for
12:            pmax = max(p, pmax)
13:            dsum[a] += d, vsum[a] += v
14:        end for
15:        psum += pmax
16:    end for
17:    option_price = exp(-r*τ) * psum / simsize
18:    for a = 1, 2, ..., n do
19:        delta[a] = exp(-r*τ) * dsum / simsize
20:        vega[a] = exp(-r*τ) * vsum / simsize
21:    end for
22: end procedure
```

## 4.1 Development Methodology

Strong and Weak scaling upper bounds are obtained from an initial, serial CPU implementation from which we identify regions of code to be multithreaded. As proof of concept for acceleration we use OpenMP to multithread the serial implementation, which requires minimal time investment prior to spending effort on the GPU implementation.

For GPU development NVidia (2016) recommends the *Assess, Parallelise, Optimise, Deploy* (APOD) cycle which provides "an evolutionary rather than revolutionary set of changes to the application." *Assess* consists of measuring serial run time to obtain scaling measurements. *Parallelise* implements a GPU kernel for parallel regions of code. *Optimise* is a cyclic sub-process which profiles run time for bottlenecks, alleviating them in code, and profiling again for subsequent bottlenecks. *Deploy* sends optimised code into production where it can be monitored before

starting a new APOD cycle. Nvidia's Visual Profiler tool is indispensable to the *Optimise* stage providing a guided analysis with performance metrics for compute and memory usage.

## 4.2 Testing Methodology

Accuracy is established with the serial CPU implementation and ensured during GPU development. However, our primary focus is GPU run time. We do not consider convergence reduction techniques and attempt to give simulations the same workload. Run time measurements are automated with a series of scripts to reproduce results across systems.

The *Call on max* option contains 3 correlated assets with parameters listed in table 2.1. Initially, we override three Heston input parameters to mimic Geometric Brownian motion to test the Rainbow's price and Delta against closed form solutions from Ouwehand and West (2006). Specifically, we set $\kappa = 0 = \sigma$. This provides the price and Deltas but not the Vegas. The Vegas are extracted from finite difference tests accompanying these analytic solutions. After the results are confirmed the parameter override is removed to test Heston dynamics. In the absence of an analytic solution for the Rainbow with Heston dynamics we resort to individual vanilla call options and Fast Fourier Transform solutions to verify their price, Deltas and Vegas. Putting everything together, the Rainbow option running Heston dynamics is compared to finite difference results.

### Hardware

Measurements are taken on the Intel Xeon CPU at UCT's High Performance Computing (HPC) Facility [1]. The CPU contains a total of 20 cores between 2 sockets (2 NUMA nodes) and is the fastest of the CPUs which were tested. GPU performance is measured on the K40 installed on the HPC cluster and GT950 installed on a desktop computer.

### Automating execution

The automation pipeline consists of four scripts. A macro level script iterates a number of simulation size and time step combinations. Each combination is input to a second script which executes the application binary five times and outputs five run times. These are input to a third script to calculate the mean and record

[1] Computations were performed using facilities provided by the University of Cape Town's ICTS High Performance Computing team: http://hpc.uct.ac.za

| Intel Xeon CPU | |
|---|---|
| Architecture | x86_64 |
| CPU(s) | 20 |
| Thread(s) per core | 1 |
| Core(s) per socket | 10 |
| Socket(s) | 2 |
| NUMA node(s) | 2 |
| Vendor ID | GenIntel |
| Model | 79 |
| CPU MHz | 2200.117 |

**Tab. 4.1:** CPU Hardware

| | GTX 950 | K40 |
|---|---|---|
| Capability | 5.2 | 3.5 |
| Global mem | 1996MB | 11520MB |
| SMs | 6 | 15 |
| Cores/MP | 128 | 192 |
| Cores | 768 | 2880 |
| Max Clock | 1329MHz | 745MHz |
| Memory Clock | 3305Mhz | 3004MHz |
| Shared memory | 49152 b | 49152 b |
| Registers/block: | 65536 | 65536 |

**Tab. 4.2:** GPU Hardware

it to a temporary file with the time step and simulation size. The temporary file is post processed by a fourth script which converts run times into acceleration and generates a multi-curve graph for each time step. These accelerations are based on serial CPU run times which are cached in a separate file.

### Measuring run time

Run time is measured with the C function *gettimeofday* which is thread safe and used for serial CPU, multithreaded CPU and GPU execution. The overhead of the *gettimeofday* function is subtracted from the result. Two run times are recorded, the entire application and the isolated Monte Carlo simulation.

## 4.3 The Application Design

The application supports varying combinations of simulation and step size. These are the only two input parameters provided by the user. Asset parameters are sourced from file which is piped through *stdin*. A key design decision is ensuring memory scales by a constant factor to the simulation and step size. This is achieved recycling variables in the forward and adjoint process while generating random numbers on the fly. Random numbers are not pre-generated to global memory as in most documented cases. The memory footprint using floats is around 256 bytes for forward and adjoint calculations of a single time step.

The GPU kernel exploits the large number of calculations inherited from the adjoint method. All data, including random number state, is stored in local memory and the compiler is instructed to allocate a large number of registers per thread.

This induces lower occupancy but the additional low latency register use improves performance for computations on the whole.

Random numbers are generated from the CURAND library. We do not implement a custom generator despite any performance opportunities there might be. As a result we analyse neither statistical nor convergence properties between generators. For inter asset correlation we attempt a number of strategies, one using Shared memory and the other using Shuffle Instructions. In addition we test executing multiple assets per thread against assigning a single asset to a thread.

## Chapter 5

# Results

Run times for the *Call on max* option using the three correlated asset parameters from Table 2.1 are listed below in Table 5.1. The simulation sizes used are multiples of the K40 block size (128) for optimal GPU usage and 360 time steps were used.

**Tab. 5.1:** *Call on max* run time in seconds (lower is better) with 360 time steps

| Processor | Simulation size | | | | |
|---|---|---|---|---|---|
| | 10 240 | 51 200 | 102 400 | 250 880 | 501 760 |
| Intel Xeon (1 core) | 1.012 | 5.001 | 9.989 | 24.408 | 48.801 |
| Intel Xeon (12 cores) | 0.134 | 0.68 | 1.346 | 3.3405 | 6.716 |
| GT950 (768 cores) | 0.0042 | 0.0192 | 0.0366 | 0.0893 | 0.115 |
| K40 (2880 cores | 0.0041 | 0.0171 | 0.0336 | 0.0807 | 0.158 |

The effect on run time for all devices is linear in simulation size (and also time steps). At 501 760 paths the CPU takes 48.801 seconds which reduces to 0.115 seconds on the GT950. We were not able to execute the most optimized kernel on the K40 due to time constraints. The results below are from a 12-core Intel Xeon CPU with a clock rate of 1200MHz, less than half as fast as the 20-core Intel Xeon with 2200MHz. All CPU based run times (1 core & 12 cores) have been halved to accommodate for this.

Using 12 cores OpenMP reduces run time by 7.266× increasing throughput from 30 845 to 224 133 paths per second. 60% of the theoretical maximum acceleration is achieved but obtaining 100% is unlikely even in highly optimal cases. The GT950 achieves a 424× speedup with 768 cores clocked at 1329MHz while the K40 contains 2880 cores clocked at 745MHz, and is likely to provide better performance. Note that GPU run times include the reduction phase executed on the CPU which adds 4ms to GPU run time.

Maxwell is competitive with its 135% performance per watt over Kepler. ECC (Error Correcting Codes) is enabled on the K40 which reduces performance by

around 8%. The GT950 achieves a throughput of 13 million paths per second. The remainder of this chapter is devoted to performance from the *Optimise* development sub-cycle illustrating the kernel's evolution in Figure 5.1. In particular acceleration measurements are used relative to single threaded run time which is still halved to mimic 2400MHz.

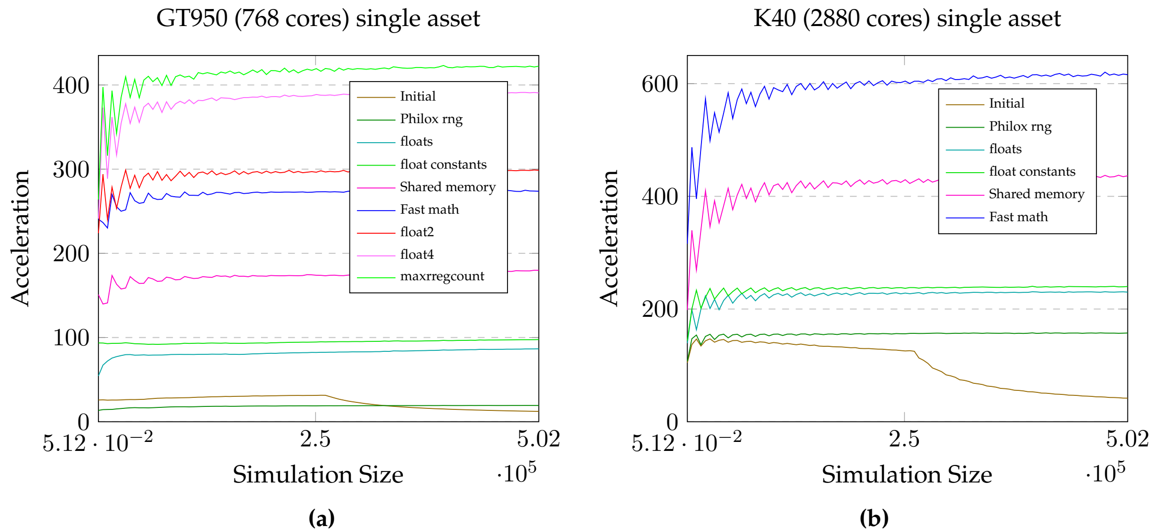



**Fig. 5.1:** Optimisation speedups on the GT950 (a) and K40 (b) graphics cards.

### The initial kernel

The initial kernel is a close port of CPU code using double floating point arithmetic and the Mersenne Twister semi random number generator from CURAND. The performance degradation witnessed around 250 000 paths is characteristic of inefficient memory access. NVIDIA's visual profiler reports the kernel is bound by memory latency using 32 transactions per access during random number generation whereas 4 or 8 is optimal.

### Philox quasi random number generation

In this optimization the Mersenne Twister is replaced with the Philox quasi-random number generator to mitigate the inefficient memory accesses. As can be seen from Figure 5.1, performance no longer degrades and a speedup of $19\times$ and $157\times$ is achieved on the GT950 and K40 respectively. Random numbers are generated to double precision, two at a time, using the *curand_normal2_double* method call. Each pair is used in the asset-volatility correlation process for individual assets. The

GT950 performs poorly with double calculations which are $\frac{1}{27}$ the speed of floating point arithmetic whereas doubles on the K40 perform better at only $\frac{1}{3}$ single-precision speed. NVIDIA's visual profiler reports that the kernel is now bound by compute, which indicates that currently performance is limited by computations and is the focus of the following optimization.

### Floating point arithmetic and *float* constants

In this optimization variables declared as *double* are converted to *float* and random numbers are also generated as *float* using *curand_normal2*. The GT950 experiences a significant gain in performance from 38× to 86× while the K40 only increases from 157× to 230×. This is expected as the GT950 is hamstrung by *double* calculations whereas the K40 is not.

The visual profiler still reports a high usage of double instruction units. Inspecting PTX assembly code reveals execution dependencies on *F2F.F64.F32* instructions contributing to 83.11% of stalls (fig 5.2). A stall occurs when an instruction waits on the result from another instruction, which wastes clock cycles, degrading performance. The instruction is injected by the compiler to convert 64bit doubles to 32bit floats in memory. The constants 0.5 and 0.25 used in the code from the Heston equation are compiled as *double*. By replacing constants with *0.5f* and *0.25f* performance increases on the GT950 to 97×, while the K40 reaches 230×.

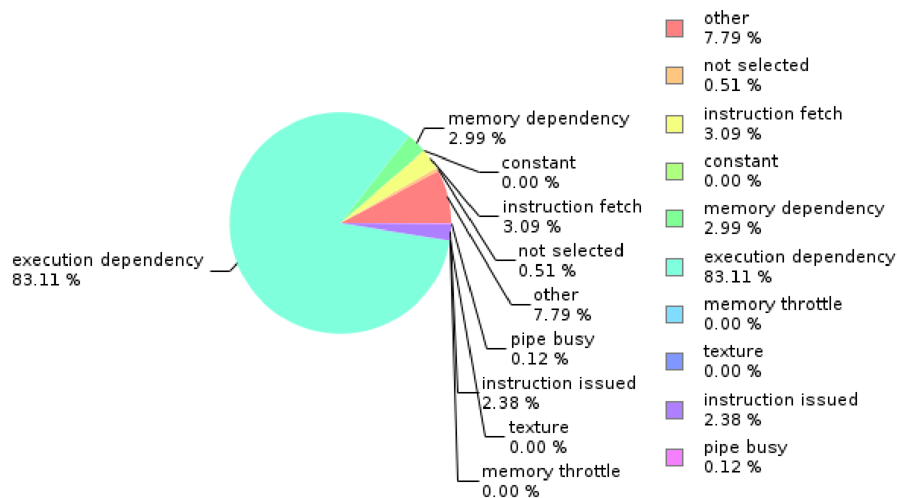



**Fig. 5.2:** Execution dependency stalls visualised from NVIDIA's visual profiler. Instructions are clearly being limited by execution dependencies incurred when doubles are converted to floats.

### Using *Shared memory*

Shared memory is located on-chip (on each SM) which results in low latency accesses. It is shared by threads in a block and limited to 48KB in total. Shared memory is ideal for storing values which are frequently accessed and required by each thread, such as the correlation matrix and asset parameters. In addition the random numbers are generated to shared memory. These are not shared between threads since each thread simulates all assets, however the lower latency is beneficial. At $128$ threads per block on the K40 we were able to store 1 564 floats to *shared memory* using 6 256 bytes per block. This nearly doubles performance on both GPUs, with the GT950 increasing from $97\times$ to $179\times$ and the K40 from $240\times$ to $436\times$. Throughput is now $5.5$ and $13.4$ million paths per second, respectively.

### The *-use_fast_math* compiler flag

The *-use_fast_math* compiler flag is a medium priority strategy listed in NVidia (2016) benefits compute intensive tasks. Exponentiation with *expf* is mapped to the hardware level providing quicker speed but less accuracy although Monte Carlo results were not affected to at least 4 decimal places. The reciprocal square root function *rsqrt* is also faster, present after differentiating the volatility's square root diffusion. As an aside, NVidia (2016) advises explicit multiplication is certainly faster than the $pow$ function to calculate $x^2$ or $x^3$. Performance increases considerably on the K40 from 597× to 615× and the GT950 increased from 179× to 273×. This

### *float2* and *float4* data structures

CUDA provides a number of vectorized data types which may be used on both the CPU and GPU. Two of these are the *float2* and *float4* data structures. They are useful for holding $2$ or $4$ float variables in memory which can be accessed with a single memory transaction. We can use this to double or quadruple the number of paths per thread without increasing the number of memory transactions. Using *float2* the simulation size is halved at the beginning of the application and each thread performs twice the number of calculations using twice the memory storage. This excessive memory requirement results in fewer concurrently executing blocks, lowering TLP. Although, the speedup increases on the GT950 and K40 to $298\times$ and $615\times$ respectively.

We were unable to test additional optimizations on the K40, but by using *float4* GT950 performance increased even further to a 390× speedup.

### Using a larger number of registers

As a final optimization strategy the *maxrregcount* compiler flag is used which tells the compiler the maximum number of registers to be allocated per thread. By providing a very large value of 255 the compiler is coerced into assigning a slightly higher number of registers per thread than per normal. The thread benefits from very low latency accesses and fewer memory transactions which improves performance slightly on the GT950 to 422$\times$.

### *Shuffle instructions* vs *Shared memory* for correlation

The asset-asset correlation calculations are currently performed and stored to shared memory. An alternate technique exists to share values between threads at the register level and doing so alleviates shared memory to be used by other variables. The Shuffle instruction works by instructing threads to read registers from neighbouring register locations instead of their own. Using this mechanism threads can generate random numbers, store them in registers, and read random numbers from neighbouring threads to correlate them. It is limited to the warp size of 32 threads and we require an asset count which is a multiple of 32. Unfortunately, initial tests were inferior to Shared memory.

### 5.0.1 Discussion

Acceleration is based on CPU performance and as such may contain a degree of subjectivity. In particular, clock frequency affects performance just as much as the number of cores. Maxwell's $135\%$ performance per watt gain on Kepler is likely to be a factor in the K40 achieving just less than half the acceleration of the GT950.

Good performance can be achieved on the GPU simply using *float* variables and constants. In addition the *use_fast_math* and *maxrregcount* compiler flags provide a clear performance improvement and are easy to include. These can be implemented without drastically effecting accuracy. Further, appropriate use of Shared memory results in significant gains, as does vectorized data types like *float2* and *float4*. However, with additional assets the extra computation and memory constraints may result in *float4* performing worse at lower simulation sizes.

Du Toit and Ehrlich (2013) note that calculations on the GPU perform twice as fast using single precision data values compared to double precision. This is our experience on the Maxwell device but not Kepler. Single precision accuracy is $10^{-6}$ which is an order of magnitude less than the standard deviation of Monte Carlo simulations and therefore negligible. If double floating point arithmetic is required then the K40 should be used.

The GT950 reaches peak performance at lower simulation sizes than the K40. This is witnessed in particular by the curves for Shared memory and Fast math. It is likely a result of the higher clock frequency in Maxwell devices. The K40 GPUs sacrifice clock frequency with the additional cores in order to keep relatively energy efficient.

The Philox random number generator uses the Box-Muller transform described in Howes and Thomas (2007). Box-Muller uses sine and cosine operations which are computationally expensive on the CPU. However, it does not involve table lookups and does not require a large number of constants making it favourable for compute intensive GPUs. The lack of branching or looping in Box-Muller is a further benefit. Based on our profiling results, 32% of threads were inactive generating random numbers with CURAND's Mersenne Twister, which was not the case using the Philox generator. In addition, The Philox generator is able to generate up to $4$ random numbers per call. We did not test statistical qualities but Salmon *et al.* (2011) vouch for its *crush resistance* and it contains periods in excess of $2^{128}$, and key spaces in excess of $2^{64}$.

Our manual adjoint implementation is tightly coupled with the discretization scheme making it harder to switch discretizations, whereas automatically generated adjoint codes are not. With domain knowledge manual implementations generally perform better and are easier to parallelize than automatically generated codes. Griewank and Walther (2008) state in section 6.3: "In practice, source transformation AD tools so far have limited coverage of parallel programming constructs."

The Milstein Discretisation scheme may be less efficient and offer lower convergence than other discretisations. Andersen (2007) mentions the Heston Model in Glasserman (2003) displays somewhat erratic convergence behaviour for European Call options. Andersen (2007) describe the Milstein scheme essentially breaks apart when using realistic input parameters. In addition, Glasserman (2003) mention it fails to satisfy smoothness conditions for the volatility process and lacks theoretical support. Andersen (2007) do not recommend the Milstein scheme to discretise the volatility process. Kahl and Jäckel (2006) state their alternative Balanced Milstein discretisation method performs efficiently and is stable over all parameter values.

The reduction process of the pathwise method is a further area for acceleration. At 501 760 paths the CPU completed the reduction in 4ms and we did not deem this long enough to warrant its own kernel. Parallel reduction is suitable for the GPU and performs in $O(\log_2 N)$. Nickolls *et al.* (2008) provides code samples and a description of parallel reduction on the GPU.

# Chapter 6

# Conclusion

The rainbow option with 3 underlying assets and pathwise adjoint differentiation is favourable for GPU processing. We witnessed a reduction in run time from 48.801 seconds to 0.115 seconds on the GT950 and better performance may be achieved on the K40. In total the GT950 experienced a 422$\times$ speedup reaching a throughput of 13 million paths per second. Run time scales linearly with the simulation and step size, as might be anticipated.

We found that instructions in the forward and adjoint passes of the Milstein discretization contain an Instruction Level Parallelism (ILP) of 3.57 which is beneficial for GPU processing. The GPU schedules two independent instructions per thread per clock cycle and if increased in future GPU architectures will result in better performance.

# Bibliography


Andersen, L. B. (2007). Efficient simulation of the heston stochastic volatility model, *Available at SSRN 946405* .

Bernemann, A., Schreyer, R. and Spanderen, K. (2011). Accelerating exotic option pricing and model calibration using GPUs, *Available at SSRN 1753596* .

Bloomberg (2009). Bloomberg Uses GPUs to Speed Up Bond Pricing, http://www.wallstreetandtech.com/infrastructure/bloomberg-uses-gpus-to-speed-up-bond-pricing/d/d-id/1262483. [Online; accessed 12-January-2016].

Brown, K. (2015). More Than 100 Accelerated Systems Now on TOP500 List, http://nvidianews.nvidia.com/news/accelerator-use-surges-in-world-s-top-supercomputers. [Online; accessed 16-February-2016].

Du Toit, J. and Ehrlich, I. (2013). Local volatility FX basket option on CPU and GPU, *The Numerical Algorithms Group Ltd, Tech. Rep* .

Giles, M. and Glasserman, P. (2005). Smoking adjoints: fast evaluation of greeks in monte carlo calculations.

Giles, M. and Xiaoke, S. (2008). Notes on using the NVIDIA 8800 GTX graphics card, https://people.maths.ox.ac.uk/gilesm/codes/libor_old/report.pdf. [Online; accessed 25-January-2017].

Glasserman, P. (2003). *Monte Carlo methods in financial engineering*, Vol. 53, Springer Science & Business Media.

Gremse, F., Höfter, A., Razik, L., Kiessling, F. and Naumann, U. (2016). GPU-accelerated adjoint algorithmic differentiation, *Computer physics communications* **200**: 300–311.

Griewank, A. and Walther, A. (2008). *Evaluating derivatives: principles and techniques of algorithmic differentiation*, Siam.

Howes, L. and Thomas, D. (2007). Efficient random number generation and application using CUDA, *GPU gems* **3**: 805–830.

Humber, A. (2008). Tokyo Tech Builds First Tesla GPU Based Heterogeneous Cluster To Reach Top 500, http://www.nvidia.com/object/io_1226945999108.html. [Online; accessed 16-February-2016].

Johnson, H. (1987). Options on the maximum or the minimum of several assets, *Journal of Financial and Quantitative Analysis* **22**(03): 277–283.

Kahl, C. and Jäckel, P. (2006). Fast strong approximation Monte Carlo schemes for stochastic volatility models, *Quantitative Finance* **6**(6): 513–536.

Kirk, D. (2007). NVIDIA CUDA software and GPU parallel computing architecture, *ISMM*, Vol. 7, pp. 103–104.

Manssen, M., Weigel, M. and Hartmann, A. K. (2012). Random number generators for massively parallel simulations on GPU, *The European Physical Journal Special Topics* **210**(1): 53–71.

Margrabe, W. (1978). The value of an option to exchange one asset for another, *The journal of finance* **33**(1): 177–186.

Nickolls, J., Buck, I., Garland, M. and Skadron, K. (2008). Scalable parallel programming with CUDA, *Queue* **6**(2): 40–53.

NVIDIA (2012). NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110/210, *Technical report*, NVIDIA.

Nvidia, C. (2010). Curand library, *NVIDIA Corporation, Santa Clara, CA* .

NVidia, C. (2016). CUDA C best practices guide, *NVIDIA, Santa Clara, CA* .

Ouwehand, P. and West, G. (2006). Pricing rainbow options, *Wilmott magazine* **5**: 74–80.

Rouah, F. D. (2011). Euler and Milstein discretization, *Documento de trabajo, Sapient Global Markets, Estados Unidos. Recuperado de www.frouah.com* .

Salmon, J. K., Moraes, M. A., Dror, R. O. and Shaw, D. E. (2011). Parallel random numbers: as easy as 1, 2, 3, *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, IEEE, pp. 1–12.

Stulz, R. (1982). Options on the minimum or the maximum of two risky assets: analysis and applications, *Journal of Financial Economics* **10**(2): 161–185.

The Risk Awards 2007 (2007). Quants of the year - Paul Glasserman and Michael Giles, http://www.risk.net/awards/1498251/quants-year-paul-glasserman-and-michael-giles. [Online; accessed 17-February-2016].

top500.org (2016). top500 2016 listing, https://www.top500.org/lists/2016/11/. [Online; accessed 12-January-2016].

Volkov, V. (2010). Better performance at lower occupancy, *Proceedings of the GPU technology conference, GTC*, Vol. 10, San Jose, CA, p. 16.

Volkov, V. and Demmel, J. W. (2008). Benchmarking GPUs to tune dense linear algebra, *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, IEEE, pp. 1–11.

# Appendix A

# Milstein Discretisation

The following derivation is provided by Fabrice Douglas Rouah which can be found in Rouah (2011). It is suggested to seek the author's reference. The proceeding content has been filtered slightly to include references to the Milstein discretisation pertinent to our application.

The stock price process follows the stochastic differential equation

$$dS_t = \mu\left(S_t, t\right) dt + \sigma\left(S_t, t\right) dW_t \tag{A.1}$$

Where $W_t$ is a brownian motion. $S_t$ is simulated over the time interval [0,T] in discretised time increments $0 < t_1 < t_2 < ... < t_m = T$ which are equally spaced. Integrating $dS_t$ from $t$ to $t + dt$ produces

$$S_{t+dt} = S_t + \int_t^{t+dt} \mu\left(S_u, u\right) du + \int_t^{t+dt} \sigma\left(S_u, u\right) dW_u \tag{A.2}$$

This integral marks the start of the discretisation scheme. Andersen (2007) consider several algorithms for time-discretisation and Monte Carlo simulation of Heston stochastic volatility models. We've used the Milstein scheme with truncation. The SDE coefficients using the Milstein scheme are $\mu\left(S_t\right)$ and $\sigma\left(S_t\right)$ which depend only on $S$, and not directly on $t$. The stock price is therefore driven by the SDE

$$\begin{aligned} dS_t &= \mu\left(S_t\right) dt + \sigma\left(S_t\right) dW_t \\ &= \mu_t dt + \sigma_t dW_t \end{aligned} \tag{A.3}$$

In integral form

$$S_{t+dt} = S_t + \int_t^{t+dt} \mu_s ds + \int_t^{t+dt} \sigma_s dW_s \tag{A.4}$$

The coefficients $\mu_t = \mu\left(S_t\right)$ and $\sigma_t = \sigma\left(S_t\right)$ are expanded using Ito's Lemma. The SDE's for the coefficients are

$$\begin{aligned} d\mu_t &= \left(\mu_t' \mu_t + \frac{1}{2}\mu_t'' \sigma_t^2\right) dt + \left(\mu_t' \sigma_t\right) dW_t \\ d\sigma_t &= \left(\sigma_t' \mu_t + \frac{1}{2}\sigma_t'' \sigma_t^2\right) dt + \left(\sigma_t' \sigma_t\right) dW_t \end{aligned} \tag{A.5}$$

Where prime represents differentiation with respect to S, and the derivatives with respect to t are zero. Through substitution we obtain

$$\begin{aligned} S_{t+dt} =& S_t + \int_t^{t+dt} \left( \mu_t + \int_t^s \left( \mu'_u \mu_u + \frac{1}{2} \mu''_u \sigma_u^2 \right) du + \int_t^s \left( \mu'_u \sigma_u \right) dW_u \right) ds \\ &+ \int_t^{t+dt} \left( \sigma_t + \int_t^s \left( \sigma'_u \mu_u + \frac{1}{2} \sigma''_u \sigma_u^2 \right) du + \int_t^s \left( \sigma'_u \sigma_u \right) dW_u \right) dW_t \end{aligned} \tag{A.6}$$

The cross multiplication $dW_u dW_s$ is order $dt$ while $duds = dsdu = dsdW_u$ are ignored. This simplifies to

$$S_{t+dt} = S_t + \mu_t \int_t^{t+dt} ds + \sigma_t \int_t^{t+dt} dW_s + \int_t^{t+dt} \int_t^s \left( \sigma'_u \sigma_u \right) dW_u dW_s \tag{A.7}$$

Applying Euler discretisation to the last term,

$$\begin{aligned} \int_t^{t+dt} \int_t^s \sigma'_u \sigma_u dW_u dW_s &\approx \sigma'_t \sigma_t \int_t^{t+dt} \int_t^s dW_u dW_s \\ &= \sigma'_t \sigma_t \int_t^{t+dt} \left( W_s - W_t \right) dW_s \\ &= \sigma'_t \sigma_t \left( \int_t^{t+dt} W_s dW_s - W_t W_{t+dt} + W_t^2 \right) \end{aligned} \tag{A.8}$$

Now define $dY_t = W_t dW_t$. Using Ito's Lemma $Y_t = \frac{1}{2} W_t^2 - \frac{1}{2} t$ so that

$$\int_t^{t+dt} W_s dW_s = Y_{t+dt} - Y_t = \frac{1}{2} W_{t+dt}^2 - \frac{1}{2} W_t^2 - \frac{1}{2} dt \tag{A.9}$$

Substituting back into A.8 to obtain

$$\begin{aligned} \int_t^{t+dt} \int_t^s \sigma'_u \sigma_u dW_u dW_s &\approx \frac{1}{2} \sigma'_u \sigma_u [(W_{t+dt} - W_t)^2 - dt] \\ &= \frac{1}{2} \sigma'_u \sigma_u [(\Delta W_t)^2 - dt] \end{aligned} \tag{A.10}$$

Where $\Delta W_t = W_{t+dt} - W_t$, which is equal in distribution to $\sqrt{dt} Z$ with $Z$ distributed as a standard normal. Combining equations A.7 and A.9 the general form of Milstein discretisation is

$$S_{t+dt} = S_t + \mu_t dt + \sigma_t \sqrt{dt} Z + \frac{1}{2} \sigma'_t \sigma_t dt \left( Z^2 - 1 \right) \tag{A.11}$$

## A.1 Milstein Discretisation of the Heston Model

The coefficients for the variance process are $\mu(v_t) = \kappa(\theta - v_t)$ and $\sigma(v_t) = \sigma\sqrt{v_t}$. An application of equation A.11 for $v_t$ produces

$$v_{t+dt} = v_t + \kappa(\theta - v_t)\,dt + \sigma\sqrt{v_t dt}Z_v + \frac{1}{4}\sigma^2 dt\left(Z_v^2 - 1\right) \tag{A.12}$$

The coefficients for the stock price process are $\mu(S_t) = rS_t$ and $\sigma(S_t) = \sqrt{v_t}S_t$ so equation A.11 for $v_t$ becomes

$$S_{t+dt} = S_t + rS_t dt + \sqrt{v_t dt}S_t Z_s + \frac{1}{4}S_t^2 dt\left(Z_s^2 - 1\right) \tag{A.13}$$

Discretising the log-stock process, which by Ito's Lemma is

$$d\ln S_t = \left(r - \frac{1}{2}v_t\right)dt + \sqrt{v_t}dW_{1,t} \tag{A.14}$$

The coefficients are $\mu(S_t) = \left(r - \frac{1}{2}v_t\right)$ and $\sigma(S_t) = \sqrt{v_t}$. $v_t$ is known at time t and may be treated as a constant. Once again using A.11 produces

$$\ln S_{t+dt} = \ln S_t + \left(r - \frac{1}{2}v_t\right)dt + \sqrt{v_t dt}dZ_s \tag{A.15}$$

$\ln S_t$ is always positive so its discretisation does not require correction. We obtain

$$S_{t+dt} = S_t \exp\left(\left(r - \frac{1}{2}v_t\right)dt + \sqrt{v_t dt}Z_s\right) \tag{A.16}$$