KTH Computer Science
and Communication

Monte Carlo Option Pricing with Graphics Processing Units

Optionsprissättning med Monte Carlo på grafikkort

FREDRIK NORD
FNORDH@KTH.SE

Master's Thesis in Computer Science at NADA
Comissioned by Handelsbanken Capital Markets
Supervisors: Dilian Gurov, Erwin Laure
Examiner: Stefan Arnborg
2010-10-25

TRITA xxx yyyy-nn

Abstract

Monte Carlo methods are common practice in financial engineering for a wide variety of problems, one being option pricing. Large clusters of computers are used to run these calculations. Growing volumes and complexity of work that needs to be performed as well as strict requirements for fast responses makes for a pressing demand for high performance computing.

We present a prototype implementation of an option pricer running on graphics cards. The prototype supports various exotic option types, quasi Monte Carlo and support for custom models for the evolution of stock prices. We conclude that graphics cards can outperform CPUs given certain conditions and for reasonable problem sizes we find a 12x improvement over sequential code when pricing options in a production system.

Referat

Optionsprissättning med Monte Carlo på grafikkort

Monte Carlo är en vanlig metod inom finansbranschen och används till många olika problem, ett av dem är optionsprissättning. Stora beräkningskluster används för att köra dessa beräkningar. Allt större mängder och mer komplexa beräkningar som måste utföras samt strikta krav på snabba svar borgar för ett stort behov av snabba beräkningstekniker.

Vi presenterar en prototyp för optionsprissättning på grafikkort. Prototypen stöder olika exotiska optionstyper, quasi Monte Carlo och stöd för användardefinierade modeller för aktiers prisutveckling. Vår slutsats är att grafikkort kan överträffa vanliga processorer givet att vissa krav är uppfyllda och vi finner att för rimliga problemstorlekar är prototypen 12 gånger snabbare än sekventiell kod vid optionsprissättning i ett produktionssystem.

Acknowledgements

I owe my deepest gratitude to Mattias Karlsson and Tor Nordqvist at Handelsbanken Capital Markets for making this project possible and for taking the time to provide support, guidance and enthusiasm throughout the whole project.

I would also like to thank Michael Schliephake at PDC for our discussions on CUDA programming and optimization and for his work on making a Linux version possible.

I am grateful to Erik Örnberg at HP for lending us the Quadro card for an extended period of time.

Lastly I would like to thank Alberz Akrawi, Sten Bergqvist, Erik Edin, Dilian Gurov, Leif Jansson, Erwin Laure, Per Lindstrand, Erik Skogby and Daniel Walz for their valuable contributions during the whole process.

Contents

1 Introduction **1**

I Background and Theory **3**

2 Background and Problem **5**
2.1 Problem Statement 5
2.2 Related Work 5
2.2.1 Goals 6

3 Monte Carlo Methods **7**
3.1 Monte Carlo 7
3.2 Approximating Integrals with Monte Carlo 7
3.3 Option Pricing with Monte Carlo 8
3.4 Quasi Monte Carlo 9
3.4.1 Low Discrepancy Sequences 10
3.4.2 Construction of Sobol Sequences 10
3.4.3 Option Pricing with Quasi Monte Carlo 11

4 GPU Computing **13**
4.1 Introduction 13
4.1.1 Hardware overview 13
4.1.2 Parallel Processing 16
4.1.3 Strengths of Graphics Processors 17
4.1.4 Weaknesses of Graphics Processors 17
4.2 The Fermi Architecture 18
4.2.1 Background 18
4.2.2 Main Innovations 18
4.2.3 Architectural Overview 19
4.3 CUDA 20
4.3.1 Thread Model 20
4.3.2 Memory Model 22
4.3.3 Programming Model 24
4.3.4 Optimization and Performance Considerations 24

4.3.5 Numerical Accuracy . . . 26

5 Method **29**
5.1 Measurements - How and Why . . . 29
5.1.1 Total Time . . . 29
5.1.2 Computation Time . . . 30
5.1.3 Time as Seen by a User . . . 30
5.1.4 Selection Criteria . . . 30
5.1.5 Threads . . . 30
5.2 General . . . 31
5.2.1 Program Layout . . . 31
5.2.2 Program Features . . . 31
5.2.3 Compilation . . . 32
5.2.4 Environments . . . 32
5.2.5 Accuracy . . . 32

II Results **35**

6 GPU Implementation **37**

7 Results **41**
7.1 Pricing a Digital Basket Option . . . 41
7.1.1 Digital Basket Options . . . 41
7.1.2 Results in a Test Environment . . . 41
7.2 Pricing a Capped Basket Option . . . 44
7.2.1 Capped Basket Options . . . 44
7.2.2 Results in a Test Environment . . . 45
7.3 Comparison of CPUs and GPUs in a Production System . . . 45
7.3.1 The System . . . 45
7.3.2 Results . . . 47
7.3.3 Accuracy . . . 47

8 Discussion and Conclusions **51**
8.1 Limitations . . . 51
8.1.1 Floating Point Types and Precision . . . 51
8.1.2 Multi-Core Processing and Optimizations . . . 51
8.1.3 Problem Size . . . 52
8.1.4 Optimality of the Program . . . 52
8.2 Other Approaches . . . 53
8.2.1 Field-Programmable Gate Array . . . 53
8.2.2 More Computers . . . 53
8.2.3 The Cell Broadband Engine Architecture . . . 53
8.3 Discussion . . . 53

8.4 Conclusions 54
8.5 Further Work 55

Bibliography **57**

Appendices **58**

A Accuracy **60**

Chapter 1

Introduction

Options in their simplest form are financial contracts in which the issuer agrees to buy or sell an underlying instrument (a stock for example) at a certain date (called the exercise date) at a specified price (called the strike price.) The buyer of the contract then chooses whether or not to exercise the contract at expiration. The problem for the issuer is to set a fair price at which to sell this contract. The option described is a variant of what is commonly called a "vanilla" option but there are many variations on the same theme, for example there may be many underlying instruments (basket option) or the payoff may be dependent on the mean of the prices at certain intervals (asian option.)

Pricing options involves solving large integrals, sometimes ranging into thousands of dimensions. For small dimensional problems and certain types of options there are exact analytical solutions but at higher dimensions this approach becomes intractable.

One common approach is to do Monte Carlo simulations, essentially simulating the evolution of the underlying stock prices for discrete time steps (for example daily) and seeing what the option would be worth, doing this sufficiently many times yields a fair value of the option given the model that was used.

Monte Carlo is efficient for these large problems but several improvements in the theory has further improved the efficiency, for example variance reduction techniques and the use of low discrepancy sequences can speed up convergence.

Even with these improvements Monte Carlo remains computationally expensive, running tens of thousands simulations of problem dimensions ranging into the thousands require much from computer systems. The stringent demands of the financial world further compounds this problem, real time pricing and risk analysis requires results to be made available as quickly as possible.

Computer graphics cards, widely used to power modern computer games, has recently seen an expansion of use to the scientific arena as cheap high performance parallel computing devices. Providing a theoretical speed several times that of modern CPUs the GPUs has found usages in scientific computing, image analysis, cryptography and more.

This project focuses on exploring the use of graphics processors as a complement to using traditional hardware when pricing options using Monte Carlo. To achieve this goal we implement a prototype for option pricing on a graphics cards and measure the speedup obtained compared to central processing units, we also incorporate the prototype into a real option pricing system to see if speedups are sustainable and which conditions are necessary for graphics cards to excel. We will also explore different system configurations and highlight the differences between newer and older architectures.

Part I

Background and Theory

Chapter 2

Background and Problem

This chapter describes the problem and related work that has been done.

2.1 Problem Statement

The quantitative analysts at Handelsbanken Capital Markets have a custom C++ framework for pricing exotic derivatives, the calculations are made on a grid of computers where the portfolio to be valued is split into smaller units and each part is valued on a single node. Performance and speed is of crucial importance, which is why new methods and technology is always of interest.

The nodes in the grid spend a large part of their time doing Monte Carlo simulations. This has been identified as one of the main hot spots for computational activity and any improvement would be of great benefit.

The aim of the project is to evaluate the use of graphics cards for pricing options with Monte Carlo, and to determine if they are a viable complement to existing infrastructure, especially in terms of whether graphics cards can speed up computations and thus reduce the overhead in pricing and risk analysis.

To achieve this goal we have implemented a prototype option pricer for graphics cards and compared it to an equivalent version not using graphics cards. The prototype supports most of what one would expect from an option pricer and has sufficient features to be directly compared to system in use at Handelsbanken.

2.2 Related Work

Lee Howes and David Thomas [6] used graphics cards to achieve a 59x speedup when pricing a simple asian option with two underlying instruments. They also priced a variant of a lookback option in which they obtained a speedup of 23x. All these results were relative to using all four cores of a Quad Opteron running at 2.2 GHz. They used a hybrid Tausworthe generator combined with the Wallace Gaussian generator to produce random numbers.

Podlozhnyuk and Harris [17] implemented a Monte Carlo option pricer for simple European call and put options. Since there are analytical solutions to these options they were able to evaluate the accuracy of the results compared to the theoretically correct results.

Bennemann et al. [14] used graphics cards to price a more sophisticated basket option with local volatility models. They managed to achieve a speedup of 25-50x. Details are somewhat sparse but the results are against a "high-end CPU" and they used the Intel Math Kernel Library (a maths library that is optimized for Intel processors.)

Lee et al. [15] implemented a Monte Carlo option pricer using optimized code on a modern Core i7 CPU and a GeForce GTX280. They achieved a speedup of 1.75x. This paper is especially interesting since it uses highly optimized CPU code on a very modern processor. The work was based on the implementation from [17].

Mike Giles and Su Xiaoke[16] assessed the performance of NVIDIA GPUs in a LIBOR market model using Monte Carlo. They achieved a speedup of over 100x.

There are plenty more examples of graphics cards being used in a wide variety of application see for example the NVIDIA GPU Computing SDK code samples[1].

2.2.1 Goals

Besides what is addressed in these works we want to explore

- What conditions are necessary for exploiting the performance possibilities of GPUs.
- Whether the theoretical advantages can be translated into actual advantages when using graphics cards in real systems.
- Whether the program can be kept sufficiently general and still maintain an edge over traditional solutions.
- If it is possible to, with a reasonable amount of work, match current systems in terms of features and accuracy.

[1] http://developer.nvidia.com/object/cuda_3_1_downloads.html

Chapter 3

Monte Carlo Methods

This chapter provides a brief overview of Monte Carlo methods and their application to financial engineering. This chapter is not strictly necessary for the understanding of the rest of the thesis but provides the theoretical foundation of what is implemented in later chapters.

3.1 Monte Carlo

Monte Carlo methods are algorithms that produce results based on random sampling. An example of a Monte Carlo method is an algorithm for approximating π; we know that the ratio between the area of a circle inscribed in a square and the square is $\frac{\pi}{4}$ so to find the value of π we randomly place m points inside the square and count the number, n, that were placed inside the circle. If the points are chosen from a uniform distribution we know that $\frac{\pi}{4}$ will fall inside the circle and hence $\pi = 4\frac{n}{m}$.

In the example we see that the probability of an event (in this case hitting the circle) is approximated by taking the fraction of successful outcomes and total attempts, the law of large numbers ensures that this approximation is asymptotically correct and the central limit theorem shows us that the convergence is of the order $1/\sqrt{m}$.

3.2 Approximating Integrals with Monte Carlo

Consider the integral

$$\alpha = \int_a^b f(x)\,dx \tag{3.1}$$

If we were to use Monte Carlo to approximate the value of α we would draw U_i $i = 1, \ldots, n$ samples from the uniform distribution in the (continuous) range $a, \ldots, b$ and then compute

$$\hat{\alpha} = \frac{1}{n}\sum_{1}^{n} f(U_i) \tag{3.2}$$

as n goes to infinity we have $\hat{\alpha} = \alpha$.

As stated before the error in Monte Carlo is bound by $O(1/\sqrt{n})$, so in the one dimensional case presented above we would clearly benefit from using the trapezoidal rule which converges quicker, on the order of n^{-2}.

Now consider the case of a multi-dimensional integral:

$$\alpha = \int_D f(\vec{x})\, d\vec{x} \tag{3.3}$$

We can solve this by replacing the U_i generated above by draws from the d-dimensional space defined by D, the same computation still holds and we still get $O(1/\sqrt{n})$ convergence. If we instead were to use the trapezoidal rule the convergence would be $O(n^{-2/d})$ clearly Monte Carlo is preferable in large dimensional problems.

It is the relationship between integrals and expected values that make Monte Carlo useful in pricing options, for example in the case of a European call option with strike price K the expected present value of the payoff would be defined by

$$p = E[e^{-rT} max(0, S(T) - K)] \tag{3.4}$$

3.3 Option Pricing with Monte Carlo

Let's say we want to set a fair price for an option with N_U underlying assets with initial prices $S_{i,0}$ for $i = 1, \ldots N_U$ and an expiration date T divided into N_T time steps of uniform length $\Delta T = T/N_t$.

We use a geometric Brownian motion model to describe the evolution of the price of the underlying assets

$$\frac{dS_i(t)}{S_i(t)} = rdt + \sigma dW(t) \tag{3.5}$$

where r is the interest rate and σ is the volatility, both assumed to be constant for simplicity, and W is a standard Brownian motion. The solution of (3.5) is given by

$$S_i(T) = S_{i,0} e^{[r - \frac{1}{2}\sigma^2]T + \sigma W(T)} \tag{3.6}$$

Since $W(T) \sim N(0, T)$ we may write (3.6) as

$$S_i(T) = S_{i,0} e^{[r - \frac{1}{2}\sigma^2]T + \sigma\sqrt{T}Z} \tag{3.7}$$

Where $Z \sim N(0, 1)$.

The basic idea of pricing options with Monte Carlo is to generate several paths following a discretized version of the log-normal process given in (3.7) and evaluate

Table 3.1. Total time and standard deviation for different number of Monte Carlo simulations for option pricing implemented in MATLAB.

N_{MC}	Time (seconds)	Standard deviation
500	0.22	0.12647
5000	2.2	0.044881
50000	22.2	0.011681
500000	221	0.0045949
5000000	2214	0.0012964

the payoff for each path. These payoffs are averaged and discounted to get a fair price of the option.

The following algorithm shows one possible strategy to compute the price.

for i = 1 to N_{MC} **do**
  Generate N = $N_U \times N_T$ matrix of Gaussian random numbers
  Take N = N times the Cholesky decomposition of C to correlate the numbers
  for j = 1 to N_U **do**
    for k = 1 to N_T **do**
      Set $S_{j,k} = S_{j,k-1} e^{(r-\frac{1}{2}\sigma^2)*\Delta T+\sigma\sqrt{\Delta T}N_{j,k}}$
    end for
  end for
  Let v_i =payoff(S, strike)
end for
Then price = $e^{-rT}\frac{\sum_{i=0}^{N_{MC}} v_i}{N_{MC}}$

Where the payoff depends on the option we want to use.

From this very general example we can see that we have an approximate complexity of $O(N_{MC}N_TN_U)$, of course this can be optimized depending on the option we want to price, for example in the case of a European call option we may use $N_T = 1$.

The code above was implemented in MATLAB to price a simple option, all cases $N_U = 10$ and $N_T = 20$ was used.

Table 3.1 clearly shows the $\frac{1}{\sqrt{N}}$ convergence of Monte Carlo and if we were to need four decimal places of precision the computation would, by extrapolating from the table, take on the order of 26 days, to be useful in a financial setting it would have to run on the order of 1 millisecond. Obviously the need for computational power is great.

3.4 Quasi Monte Carlo

Quasi Monte Carlo methods parallel regular Monte Carlo methods with the difference that the "random" numbers are produced by a Quasi Random Number

Generator (QRNG), also called low discrepancy sequences. The difference between pseudo-random and low discrepancy sequences is illustrated in figure 3.1.

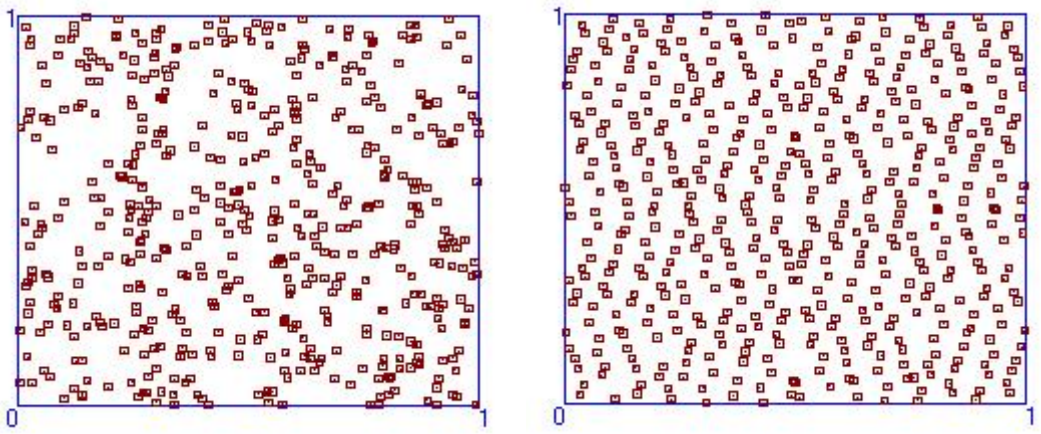

Figure 3.1. Uniform and LDS random numbers

3.4.1 Low Discrepancy Sequences

Low discrepancy sequences are sequences of numbers that together uniformly fills space. Discrepancy is the formal notion of deviation from uniformity and is described by equation (3.8) where A is a collection of subsets of the space, and vol(a) denotes the volume of a.

$$D(x_1, \ldots, x_n; A) = \sup_{a \in A} |\frac{\#\{x_i \in a\}}{n} - \text{vol}(a)| \tag{3.8}$$

When the discrepancy is low (see [2] for a precise definition of low) the sequence of numbers is called a low discrepancy sequence. There are several low discrepancy sequences available, for example Halton, Hammersley, Faure and Sobol sequences. Sobol sequences are usually used in finance since they provide more accurate results and can be generated very quickly.

3.4.2 Construction of Sobol Sequences

Generating Sobol sequences is quite straightforward and can be efficiently done on a computer. We will describe the Gray code construction due to Antanov and Saleev [1].

The construction begins with a primitive polynomial in $GF(2)$, the Galois field of two elements.

$$x^q + a_1 x^{q-1} + \ldots + a_{q-1} x + 1 \tag{3.9}$$

Where (3.9) has to be irreducible.

We then define the direction numbers v_j

$$v_j = m_j/2^j \tag{3.10}$$

where m_j $j > q$ is defined by the recurrence

$$m_j = 2a_1 m_{j-1} \oplus 2^2 a_2 m_{j-2} \oplus \ldots \oplus 2^{q-1} a_{q-1} m_{j-q+1} \oplus 2^q m_{j-q} \oplus m_{j-q} \tag{3.11}$$

and m_j $j \leq q$ are given as initialization numbers.
The sequence is then defined by

$$x_k = g_0(k)v_1 \oplus g_1(k)v_2 \oplus \ldots \oplus g_{r-1}(k)v_r \tag{3.12}$$

where $g_{r-1}(k) \ldots g_1(k)g_0(k)$ is the binary representation of the Gray code of k. More efficiently (3.12) can be stated as in (3.13)

$$x_{k+1} = x_k \oplus v_l \tag{3.13}$$

This describes the one-dimensional case, for more dimensions we simply choose a different primitive polynomial and do the above calculations for each dimension.

3.4.3 Option Pricing with Quasi Monte Carlo

Pricing an option with quasi Monte Carlo exactly parallels using regular Monte Carlo with the exception that the random numbers are taken from a low discrepancy sequence. These numbers are treated as being uniformly distributed and are converted to the Normal distribution by using standard methods.

Quasi Monte Carlo methods offer faster convergence than traditional Monte Carlo methods and are therefore often a preferred method in financial engineering.

Since higher dimensional numbers in the sequence are usually of lower quality (i.e. not as uniform) one can use a Brownian Bridge, essentially this approach uses the first number to determine the overall motion and successive numbers to determine smaller and smaller parts. See [2] for more details on this.

Chapter 4

GPU Computing

This chapter provides a brief overview of graphics processing units and how they are programmed using CUDA. For a deeper understanding see the references, especially [4], [5] and [7].

4.1 Introduction

4.1.1 Hardware overview

History of GPGPU

The application of Graphics Processing Units in fields not primarily concerned with graphics is commonly called general-purpose computing on graphics processing units (often abbreviated GPGPU.) The origins of GPGPU is in programmable shaders, most often used in graphics applications to give the programmer additional control of the rendering process. In the early stages of GPGPU the programmer, regardless of the field she was working in, had to painstakingly map the code she wanted to run to the domain of computer graphics. For example data would be input as textures which could be transformed by the programmable shaders.

NVIDIA realized that there was potential for these techniques and released CUDA in 2006. CUDA aimed to make programming the GPU for general applications easier and more similar to programming that would be common in scientific circles. Due to the relative simplicity of CUDA there has been a drastic rise in interest and applications of GPGPU.

NVIDIAs largest (and arguably only) competitor AMD also supports GPGPU via the Stream SDK (successor of the now defunct Close to Metal, CTM.) Although much less popular, the Stream SDK provides support for AMD graphics cards.

Both Microsoft and the Khronos group [1] have released standards in an attempt to provide a standardized interface to computational resources (graphics cards from any producer as well as CPUs.) The attempt by the Khronos group, called OpenCL, was released in late 2008 and is gaining popularity but is still in early development.

[1] Amongst other things the maintainers of the OpenGL standard.

The main advantage of OpenCL is that both AMD and NVIDIA provide drivers for their cards which means that the same code will run independently of the hardware, something neither CUDA nor Stream does.

Some Notation

We will often refer to device code, device memory, host code and host memory. With device we basically mean a graphics card that is attached to the host by which we mean a computer, see 4.1. Since device and host have separate memory this is an important distinction, we can't reference device memory from the host or vice versa but have to copy between the two.

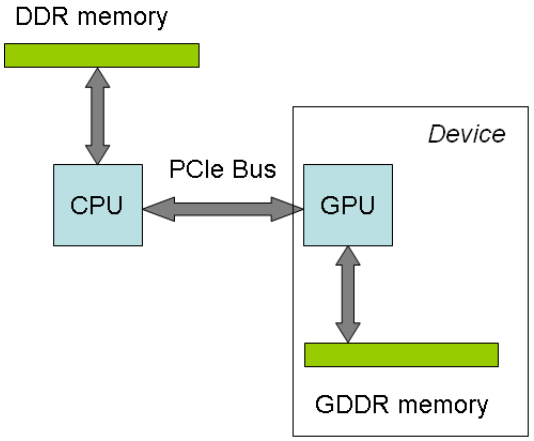

Figure 4.1. Overview of a computer system with an attached graphics card. DDR and GDDR refers to the RAM memory types.

We will also use the term device capabilities, which divide graphics cards into classes based on what they can do. Currently the device capabilites are 1.0,1.1, 1.2, 1.3 and 2.0. Typically lower capabilities means older cards and less functions available, for example devices of capability less than 1.3 do not support double precision arithmetic.

GPUs Compared to CPUs

One of the reasons that GPUs provide superior performance in pure computational power can be seen in figure 4.2. The percentage of the die size occupied by ALUs is much greater, which means that more of the chip is used for mathematical computations.

Figure 4.3 show the theoretical peak performance of CPUs and GPUs, we clearly see the fast development of graphics cards.

Figure 4.2. Layout of CPU and GPU architecture. Graph from [4].

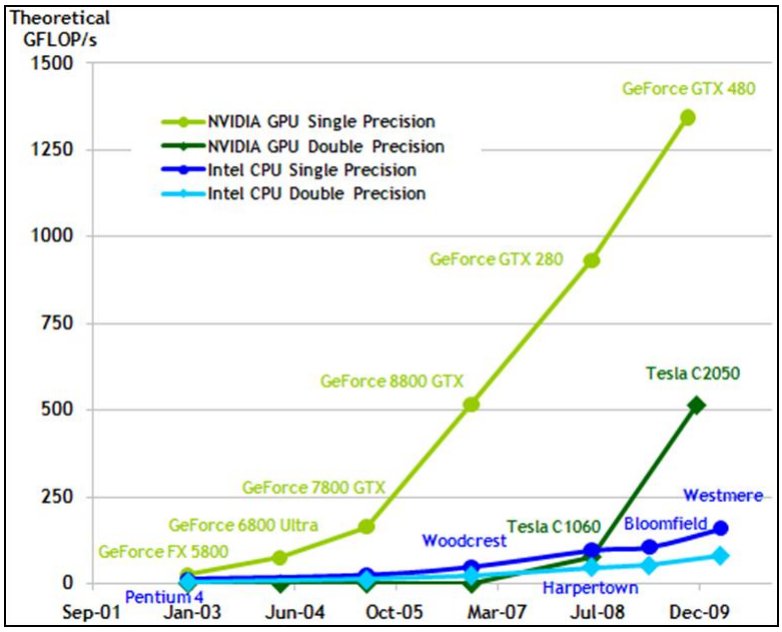

Figure 4.3. Theoretical FLoating point Operations Per Second, CPU and GPU. Graph from [4]. Note that this figure is slightly misleading, the GeForce GTX 480 and Tesla C2050 were delayed and release in March 2010.

Graphics cards

Currently there are three product lines from NVIDIA that are interesting from a GPGPU perspective: GeForce, Quadro and Tesla. The GeForce graphics cards are primarily aimed at accelerating computer games and as such are a consumer commodity. Graphics cards in the Quadro product line are mainly aimed at business users and provide additional performance for Computer Aided Design and Digital Content Creation.

The Tesla product line is exclusively aimed at High Performance Computing

and as such provide no way of displaying images on computer screens. The Tesla line generally provides the highest performance and the largest amount of available memory but is also the most expensive of the three.

4.1.2 Parallel Processing

Moore's law predicts that the number of transistors in commonly available integrated circuits will double every two years. Until recently this has meant drastic improvements in clock speed and subsequently performance of single core processors [10]. More transistors means that more power is consumed and more heat is generated, to overcome this the current trend is to have multi-core processors. Multi-core processors are increasingly common in CPUs but is also the main feature of GPUs, current CPUs can run four to eight threads at the same time while GPUs commonly run thousands of threads in parallel.

Amdahl's Law

Amdahl's law states the possible speedup given how much of a program can be parallelized and the number of processing elements that are available. Amdahl's law applies to strong scaling, which means that the problem size is fixed as the number of processors are varied.

Amdahl's law [11] is stated in (4.1) where P is the proportion of the computation that is parallelized, N is the number of processors and S_p is the overall speedup.

$$S_p = \frac{1}{(1-P) + \frac{P}{N}} \tag{4.1}$$

For example, if 50% of the program has to be sequential we could get a speedup of 1.6 using four processors. Even if we had infinitely many processors we could only get twice the performance (since $\frac{P}{N}$ disappears as N tends to infinity we are left with $\frac{1}{1-P}$ which is two.)

Intuitively this example says that if the sequential part of the program takes 100ms to execute and the parallel part runs in 100ms (since they were assumed equal) then even if we could run the parallel part in 0ms we still would have a total runtime of 100ms which is only half of what we started with.

Amdahl's law resulted in skepticism [12] on the applicability of parallel processing, since it states that the potential speedup is diminishing even as more processors are made available.

Gustafson's Law

Amdahl's law implicitly assumes that the parallel proportion of the program is fixed regardless of the problem size. In some cases this may be a sound assumption but for scientific computing problem size often scale up and as a result the parallel proportion increases when compared to sequential code. This is known as weak scaling.

Assume that the sequential portion of a program run on a parallel computer with N processors is s' and p' is the parallel portion. Then, obviously $s' + p' = 1$. On a sequential computer the same program would take $s' + p'N$ the speedup would then be $S_p = \frac{s'+p'N}{s'+p'} = N + (1 - N)s'$ this is Gustafson's law [12]. If the sequential proportion (s') is diminishing with problem size then, for sufficiently large problem sizes, S_p would tend to infinity as N does.

We will see that many of the results we obtain are directly predicted by Amdahl and Gustafson's law.

4.1.3 Strengths of Graphics Processors

The main advantages of graphics cards are:

- High throughput. When counting mathematical operations per second (often FLOPS - FLoating point Operations Per Second) graphics cards are at the top of most charts, especially when compared to CPUs.
- Cheapest computational resource available (GFLOPS/$.) Since graphics cards are a consumer commodity the economy of scale has made prices accessible compared to special solutions.
- Low energy consumption per GFLOPS compared to other solutions. (See for example the Green500 list [2])

4.1.4 Weaknesses of Graphics Processors

Although graphics cards offer many benefits there are of course drawbacks, the main of which are:

- They require a different skill-set to program. Code has to be designed to run on a GPU and it will be slightly different from it's CPU counterpart which means that time must be spent on redesigning algorithms and additional maintenance.
- They offer slow sequential execution (that is, they are inefficient for non-parallelizable problems.)
- They are inefficient for memory intensive programs (transferring data to and from the card can significantly reduce performance.)

Although, for example, OpenCL can run the same code on graphics cards as well as processors it must still be written for the least common denominator so writing a program that takes full advantage of both requires some expertise in GPU programming.

[2] http://www.green500.org/, a list of the most energy efficient supercomputers where the Tesla based Nebulae and Mole 8.5 currently place well.

4.2 The Fermi Architecture

4.2.1 Background

The Fermi architecture is the latest architecture developed by NVIDIA, it offers several new innovations that makes it especially appealing to the HPC crowd. The GeForce 400-series as well as the Tesla 2000-series are based on Fermi, however the GeForce series is artificially limited (especially with respect to double precision performance) so as to not compete with the higher-end cards.

Fermi was preceded by the GT200 (the GeForce 200 series) architecture which in turn was preceded by the G80 (GeForce 8 series) architecture.

4.2.2 Main Innovations

The main advantages of Fermi over previous architectures are:

- Native support for ECC for register files, shared memories, the L1 and L2 cache and the DRAM memory. Fermi supports Single-Error Correct Double-Error Detect (SECDED) ECC codes which means that single bit errors are corrected and errors in more than one bit can be detected. This feature is only available on Tesla cards and can be activated/deactivated as needed. Having it activated means a 5-20% performance penalty depending on application.

- More and faster atomic memory operations.

- Better support for C++. Fermi adds support for virtual functions, function pointers, new and delete operators and try-catch exception handling.

- A 40-bit unified address space. Fermi merges local, shared and global memory into a unified address space. With 40-bits there is support for one terabyte of memory, up from four gigabytes previously. There is also support for 64-bit addressing in the Instruction Set Architecture.

- Concurrent kernel execution. Kernels are the functions that are run in each thread that is spawned. On earlier architectures separate kernels had to be run sequentially but Fermi supports kernels to be run in parallel. This can be beneficial if a kernel does not fully utilize the resources available.

- Support for the IEEE 754-2008 standard. Earlier generations flushed subnormal [3] numbers to zero but in Fermi they are supported in hardware. Fermi also supports Fused Multiply Adds (FMA) which means that higher precision is kept in the intermediate steps when evaluating expressions of the form $A * B + C$.

[3] Numbers that are between zero and the smallest representable number

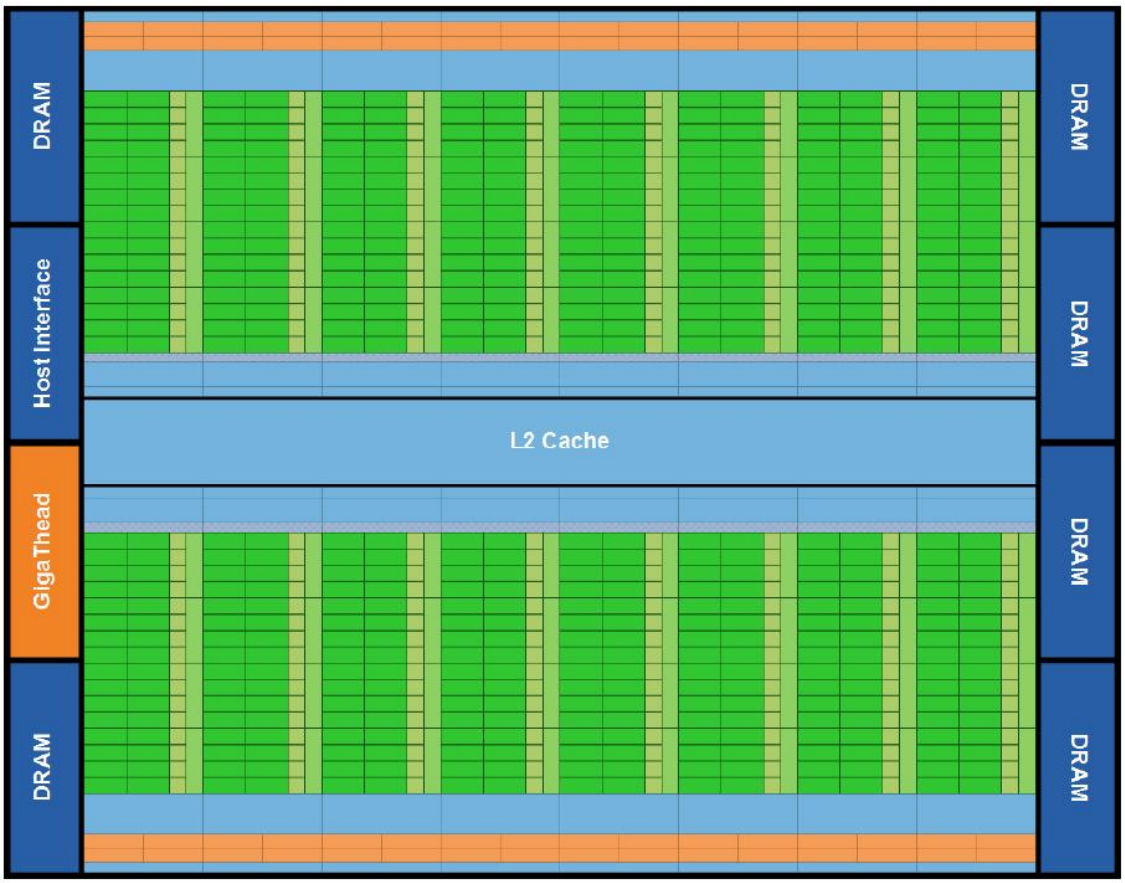

Figure 4.4. High level view of the Fermi architecture. From [8]

- Faster double precision mathematics. Fermi can execute 8 times as many double precision operations per clock as compared to previous generations and the gap between single and double precision throughput is only $\frac{1}{2}$ from previously $\frac{1}{8}$.

- Fast hardware caches. Fermi introduces an L1 and an L2 cache. The L1 cache and the shared memory share 64KB that is split into 16 and 48 KB this splitting allows programs to use more shared memory if needed and otherwise get the benefit of having more cache space. The L2 cache is a 768KB memory area that is in common for the entire device.

4.2.3 Architectural Overview

Figure 4.4 show a high level view of the Fermi architecture. There are 16 Streaming Multiprocessors (show in fig 4.5) each containing 32 CUDA cores.

The GigaThread engine is the top-level scheduler distributes threads, regardless of which kernel they are from, to SMs for execution. At the next level the warp scheduler decides which threads to run in each SM.

When a CUDA kernel is executed the GigaThread engine distributes blocks of threads to each SM, the SM in turn splits the threads into groups of 32 called warps. Each SM can handle 48 warps for a total of 24576 active threads. The warp schedulers in each SM chooses one warp each to execute, one instruction is then dispatched to an execution block containing 16 cores, 16 load/store units or four Special Function Units (used for transcendental operations.) Since there are two schedulers two half-warps are executed each clock-cycle. This translates to 32 single precision floating operations, 32 integer operations, 16 double precision operations, 16 load/store operations or 4 transcendental operations per clock per SM.

For double precision operations both execution blocks of 16 cores are seen as one 16 core double precision execution block this in effect makes it impossible to double-issue such instructions with the other types.

The same instruction is executed for each thread in a warp, this is commonly denoted by NVIDIA as SIMT (Single Instruction Multiple Thread) and is a generalization of SIMD (Single Instruction Multiple Data). The drawback of this system is that if a thread in a warp diverges (for example if a branching instruction evaluates to true for one thread and to false for the rest) the entire warp, except for the non-divergent threads, is stalled until the threads converge again.

4.3 CUDA

The Compute Unified Device Architecture or CUDA is the underlying architecture of NVIDIA GPUs. In this section we will describe the different models that are important to understand GPU programming, we will also describe the various types of memory available, what kind of optimizations can be done as well as a brief discussion about accuracy in a CUDA environment.

4.3.1 Thread Model

The Kernel

The basic atom of the CUDA world is the kernel. A kernel is simply a function, with a void return value, in CUDA C prefixed with the keyword `__global__`. When a kernel is called from host code N threads are spawned that execute the kernel code in parallel. Each thread has access to a unique identification number which enables us to do different work in different threads.

Grids and Blocks

Threads are organized in a hierarchy (illustrated in figure 4.6,) the top level is called the grid which contains two dimensions of several blocks. Each block in turn

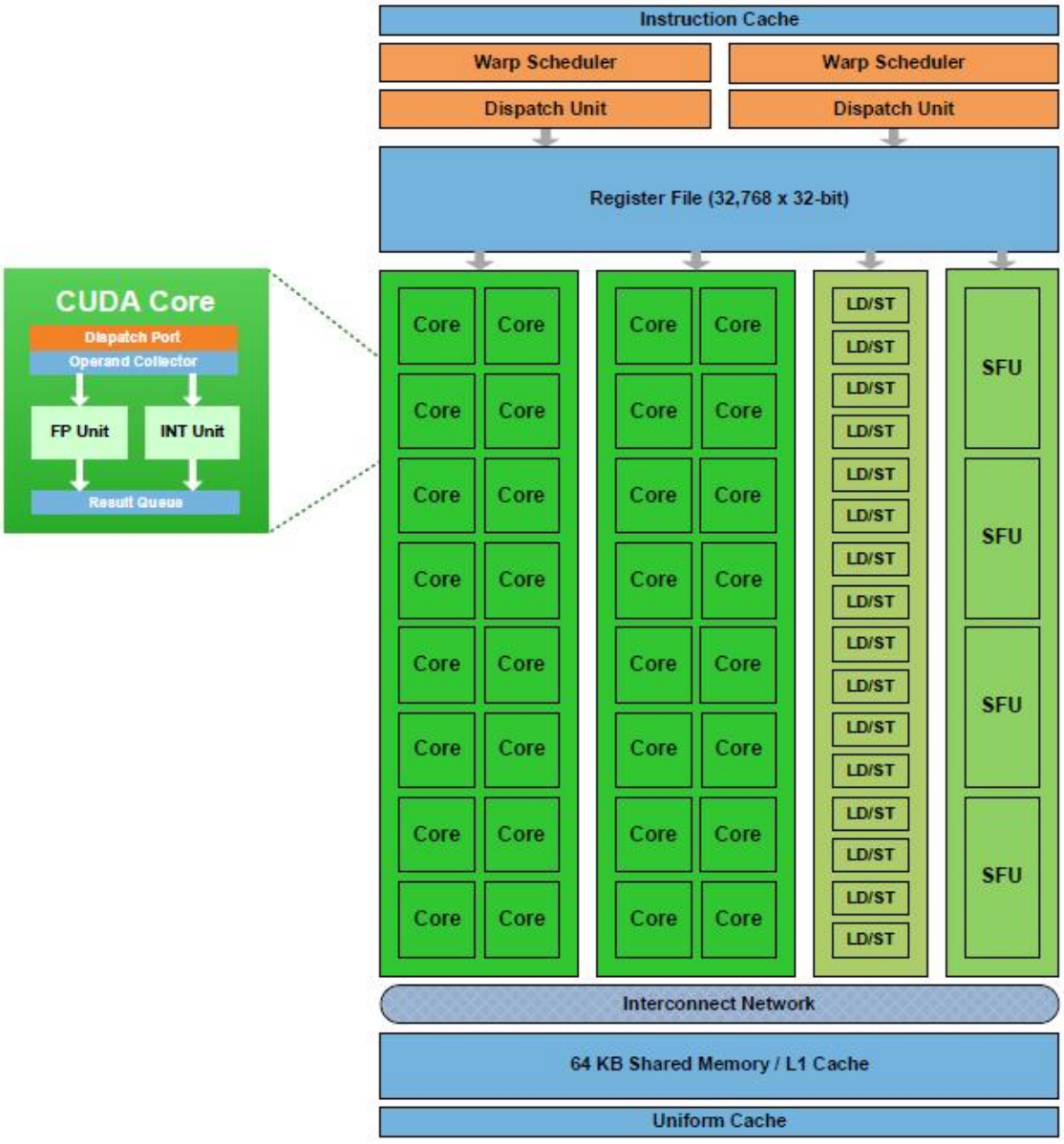

Figure 4.5. Closer view of the Streaming Mutiprocessor (SM). From [8]

consists of a three dimensional structure of threads. The size of the grid and the blocks are specified by the user when invoking a kernel, and the maximum numbers depend on the compute capability of the device.

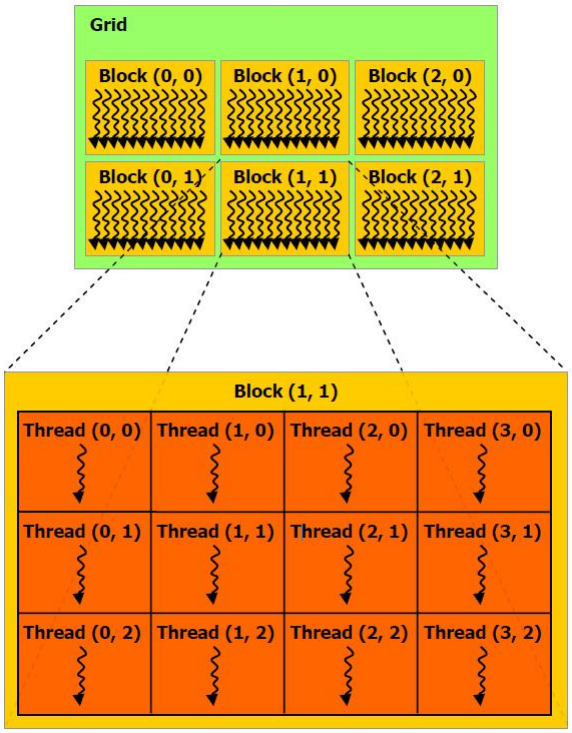

Figure 4.6. A Grid of Blocks. From [4]

Selecting the dimensions of the grid and of the blocks is largely problem dependent, in an image analysis application selecting two dimensional structures might make the problem easier to map to an algorithm than using only one dimension, which might be more beneficial in a program that accesses data in a linear fashion.

4.3.2 Memory Model

There are several types of memory available to a CUDA application programmer each provides different possibilities and constraints. The memory types are:

- Global memory
- Per thread memory
- Shared memory
- Registers

- Constant memory
- Texture memory
- Page-locked host memory
- Surface memory

The relevant memory types are described in short in the following sections a full description of all memory types can be found in [4] and [7].

Global Memory

Global memory is the main storage of the card consisting of 768 MB to 4GB (depending on model) of GDDR RAM. Accesses from global memory are in general slow and should be avoided when possible.

Global memory is controlled by the programmer by similar methods as the heap in C and is accessible by all threads. If two threads write to the same location the results are undefined. One uses `cudaMalloc` to allocate and `cudaFree` to deallocate memory. Data is then accessed through pointers in device code.

Global memory can not be allocated from device code and can not be accessed by host code, except by using special memory copying functions.

Per Thread Memory

Each thread has access to 16 or 512 KB (depending on compute capability) of local memory. Local memory is used to store large arrays, structures and for register spilling.

Local memory is stored in global memory so it shares the same drawbacks but it is easier to keep memory accesses coalesced when reading local memory.

Shared memory

Each thread block has access to a shared memory area that is 16KB or 48KB depending on device capability. All threads in the block can read and write memory in this area but if two threads write to the same memory location the results are undefined.

Shared memory is divided into banks, which can be accessed simultaneously as long as each memory location is in separate banks. If two memory accesses are in the same bank they have to be serialized and performance is reduced. A memory request where at most n requests are in the same bank is said to be a n-way bank conflict.

Shared memory can be very powerful in optimizing performance since it is on-chip and therefore vastly faster than global memory as long as bank conflicts are kept to a minimum.

Registers

Each multiprocessor has access to 8K,16K or 32K registers (depending on device capability,) registers store local variables in the same fashion as on a CPU. Register usage can influence occupancy (by reducing the number of threads that can be run on each multiprocessor) if too many are used in a device function.

Constant Memory

Constant memory is a part of global memory, however there is an 8KB cache that can speed up reads.

Constant memory can only be written to from host code, device code can not modify the contents of the memory but can read from it.

Constant memory can be a good alternative to using global memory as long as the amount of data is limited to 64KB.

4.3.3 Programming Model

There are two common ways to use CUDA in a program, either through CUDA C or through the CUDA driver API.

CUDA C is a set of extensions to the C language that support native keywords for data structures and kernel handling. CUDA C files have to be compiled with the nvcc compiler.

The nvcc compiler translates CUDA C specific extensions into calls to the CUDA C runtime functions and device code into PTX (a form of assembly for graphics cards) and/or cubin objects (binary objects) that it stores in a global initialized array. The code can then be compiled using a regular compiler (such as g++.) When the application is run the contained device code is loaded by the device driver and compiled to binary code (just-in-time compilation.)

The CUDA driver API offers a stand-alone API for compilation using any compiler. The resulting code is significantly more verbose but gives the programmer more control of the work flow and is language independent.

4.3.4 Optimization and Performance Considerations

Several guidelines for optimizing CUDA program performance are described in [5] this section describes the ones that proved the most useful in the context of this work.

Parallelize Sequential Code

Code running on a GPU should be parallel to the highest extent possible. Only running a single thread is very inefficient and similarly special casing code for certain threads leads to warp divergence and significantly reduced performance. This means that when writing an if-statement or a loop that considers the thread identification number we run the risk of causing warp divergence.

Additionally a running a larger part of the necessary computations on the device can reduce the need to transfer data to and from the device, for example generating random numbers on the device instead of on the host.

Minimize Data Transfer Between Device and Host

Transferring data to and from the device can be very costly in terms of performance since the PCIe bus is slow compared to native memory operations (177 GBps for a GeForce GTX 480 compared to 8 GBps on a PCIe x16 Gen2 bus.)

Data transfer can be an easy way to reuse code but one should keep in mind that large transfers can slow down the program and rewriting code in CUDA might be worthwhile.

Ensure Global Memory Accesses are Coalesced

According to [5] the single most important optimization is ensuring that memory operations are coalesced.

One way to make memory coalescing demands easier to meet is to use local memory (such as using `int a[2000]` instead of `int* a`.) This memory, however, is limited and cannot be used outside of the kernel.

Maintain 25% Occupancy

Occupancy is the number of threads running on a SM relative to the maximum number allowed. Maintaining a ratio over 25% allows the processor to exchange idle threads (due to, for example, waiting for arithmetic results to become available in a register or waiting for a memory transfer) for active threads.

Certain parameters, such as number of registers used and amount of shared memory used, can affect the maximum occupancy that can be achieved.

The Number of Threads per Block Should Be a Multiple of 32

Since warps currently are defined to be 32 threads, not running multiples of 32 threads results in inefficient use of resources.

[5] recommends using 128 to 256 threads per block as a starting point for evaluating performance.

Use the Fast Math Library

Using for example `__sinf()` instead of `sin()` usually performs better but provides less accurate results. See section 4.3.5 or [4] for a full exposition of the various functions that can be used and their precision.

Minimize the Use of Global Memory

Accessing global memory is slow compared to the arithmetic throughput of graphics cards. Avoiding unnecessary memory accesses and storing frequently accessed data in registers or shared memory can give a boost in performance.

4.3.5 Numerical Accuracy

Floating Point Representation

Since computer memories are inherently discrete we can never have an exact representation of a real number (such as $\frac{1}{3} = 0.3333\ldots$.) Instead we have to resort to a representation that is as good as possible given a set number of bits. The most commonly used standard for floating point arithmetic is the IEEE 754.

The IEEE 754 standard describes the single (or binary32) and double (or binary64) precision floating-point formats. Single precision numbers reserve 8 bits for the exponent and 23 bits for the significand while doubles have 11 bit exponents and 52 bits significands and both have 1 bit representing sign. For a more thorough review of how floating-point numbers are stored and used see for example [7].

CUDA is mostly IEEE 754 (2008) compliant, the differences are described in [4] pp 149-150.

Error bounds and ULP

ULP (commonly Unit in the Last Place) was defined by Kahan [13] as

> Ulp(x) is the gap between the two floating-point numbers nearest x , even if x is one of them.

ULP as defined by Kahan or with slight modifications are usually used to quantify errors of mathematical functions implemented in computer hardware. For example, the IEEE 754 requires all arithmetic operations to be within 0.5 ULP of the exact result.

Table 4.1 gives an excerpt of relevant CUDA mathematical functions and their error bounds. Especially note the error bounds of the intrinsic functions (starting with a ___.) The table is an excerpt from [4]

ECC

Background radiation can in some cases cause bits to flip in DRAM chips. Single bit flips ("soft errors") can be corrected through the use of an Error Correcting Code.

Devices of compute capability 1.3 or less do not support ECC natively. In computer games this does not matter much, it is probably not even noticable if for example a single pixel is off color in one frame. For scientific computations this can however have an impact, even producing wrong results.

Table 4.1. The error bounds for various mathematical functions in CUDA.

Function (Precision)	Error Bound
expf (single)	2 ULP
exp (double)	1 ULP
___expf (single)	2+`floor`(`abs`(1.16*x)) ULP
logf (single)	1 ULP
log (double)	1 ULP
___logf(single)	For $0.5 \leq x \leq 2$ the maximum absolute error is $2^{-21.41}$ otherwise it is 3 ULP.
sinf (single)	2 ULP
sin (double)	2 ULP
___sinf(single)	For $-\pi \leq x \leq \pi$ the maximum absolute error is $2^{-21.41}$ and larger otherwise.
cosf (single)	2 ULP
cos (double)	2 ULP
___cosf(single)	For $-\pi \leq x \leq \pi$ the maximum absolute error is $2^{-21.19}$ and larger otherwise.
erfinvf (single)	3 ULP
erfinv (double)	8 ULP

A software solution for ECC is described in [9], it does however have a significant impact on performance especially for memory intensive programs. The latest architecture from NVIDIA, Fermi, supports hardware ECC so in those cases this is a non-issue. Another approach is to run the calculations twice, if the results do not agree we can discard both results and start over.

Even though the main development has been on a platform that does not support ECC, no care has been taken to counteract soft errors. The averaging effect of Monte Carlo methods will cause single errors to have a very slight impact on the final result. But if, for example, an unfortunate error causes a high-order bit in one of results to flip this could significantly impact the final price.

These errors, however, seem rare, when run for repeatedly for 62 hours none of the almost 500000 samples differed.

Chapter 5

Method

This chapter describes the various measurements, design considerations and computer systems that were used in the evaluation of the prototype.

5.1 Measurements - How and Why

We used two main measures of the performance of host vs device code. Both measurements were made on the code as run on a CPU as well as on a GPU.

5.1.1 Total Time

The first measurement that was made was the total time, that is, the time it takes from wanting to get a result until the result is at hand. This measure includes memory allocation and copying both for host and device code.

The total time is of interest since it shows how much the overhead of using graphics cards (that of allocating the extra memory, copying parameters to and from the card and time spent setting up the device) can impact the speedup that is achievable. As we demonstrated in the section about Amdahl's law such an overhead can severely limit the speedup but as we showed in the section about Gustafson's law we will see that as the problem size increases this overhead diminishes and we can achieve significant speedups.

The total time can be improved upon in real-life applications, the measurement includes time spent setting up the card, which is only necessary to do once. Also memory allocations can be reduced if memory is kept between runs, this also avoids the time necessary to deallocate memory.

This measurement is close to how long a user will have to wait for a result and is in that sense superior as a measure for users as compared to computation time.

If T_c denotes the total time on a CPU and T_g the total time on a GPU we will denote $S = \frac{T_c}{T_g}$ as the speedup factor obtained with GPU.

5.1.2 Computation Time

The second measurement was computation time, which includes the time spent actually calculating results. This time does not include any overhead incurred by either using a CPU or a GPU. This measurement is subsumed by the total time measurement.

This measure is of a more theoretical interest since, when comparing host and device code, it shows the difference in power between a CPU and a GPU when doing the same amount of work.

If C_c denotes the total time on a CPU and C_g the total time on a GPU we will denote $S_c = \frac{C_c}{C_g}$ as the speedup factor obtained with GPU.

5.1.3 Time as Seen by a User

This measurement is only used when we introduce device code into a real system for pricing options. This is the true time as seen by a user since it is measured from a function call to when a result is obtained.

Note that this includes the improvements mentioned earlier, that is avoiding startup cost and memory allocations, so total time and time as seen by a user are not necessarily the same.

5.1.4 Selection Criteria

The timing measurements of ten consecutive runs were obtained, from this we calculated the minimum, maximum and median runs which formed the basis of the figures on the following pages.

Standard deviation was observed to be low in all cases except for total GPU time.

Several different problem sizes were tried, to keep the results concise we present two. The problem size representing small problems was chosen to be 10000 iterations, this number was selected because it is equivalent to the size that was run in the system in use at Handelsbanken. In practice we round the number of iterations to the nearest multiple of 256 to be able to easier map it to thread blocks so 10000 iterations were 10240 in the system. We chose 870400 iterations to represent large problems, the number might seem odd at first glance but comes from earlier versions where the number of iterations was limited by the available memory of the graphics card, this number was kept since it is large enough to represent large problems (that many iterations would rarely if ever be used in practice) and small enough not to cause memory issues.

5.1.5 Threads

All CPU results presented in the following sections are based on single-threaded code. The main reason for this is that the option pricing code in the production system was single-threaded, so single-threading was chosen to keep results consistent.

Since all processors used in the experiments had four cores, running single-threaded code does not utilize their full capacity.

Theoretically using all four cores would effect a four time speedup, hence dividing the resulting GPU speedups by four. However, initial experimentation using a naive OpenMP implementation showed a mere 20-30% speedup when using four threads, even when computing large problems. Such effects can be caused by programs being limited by the memory bandwidth but is in this case probably a result of inefficient programming.

Producing a fully optimized parallel CPU version (perhaps using SSE[1] as well) was deemed to be outside the scope of this project.

5.2 General

5.2.1 Program Layout

A single program (executable) contained both CPU and GPU code. Parameters were hard-coded in the case of the digital basket[2] and read from a file in the case of the capped basket. The program was written from scratch both for CPU and GPU except for the Sobol generators.

The same methods were applied to CPU and GPU code to the largest extent possible, the only exception being the Sobol sequence generator which was slightly different between the two.

The code written from scratch was used as a benchmark in time measurements made on CPU code in the test environment. When using the production system CPU time measurements were made on code that already existed in the code-base, developed by Handelsbanken.

5.2.2 Program Features

The main requirement of the program was that it should be general in the sense that valuing different option types and varying the number of time steps and underlying instruments should be easy. This was achieved by producing the trajectories, storing them and then supplying a payoff function with them. To implement a different option type one would simply swap the payoff function to the new one.

The program was written to support all features supported by a real system, including Sobol sequences, the Brownian bridge construction and the geometric Brownian motion as well as support for custom models for the evolution of stock prices.

[1] Streaming SIMD Extensions, certain instructions that utilize the Single Instruction Multiple Data capabilities of certain processors to achieve higher performance

[2] Also known as binary options.

Table 5.1. A description of the components in the computers that were used in the tests.

Property	HP xw4600	HP z400	Dell R5400
CPU Type	Core 2 Quad	Xeon	Xeon
CPU Model	Q9450	W3565	E5430
CPU Clock	2.67 GHz	3.20 GHz	2.66 GHz
Disk Type	HDD	SSD	HDD
Available RAM	3.23 GB	12 GB	3.00 GB
Compute Graphics Card	Quadro FX 4800	None	GTX 470
Display Graphics Card	Quadro NVS 290	Quadro NVS 295	Quadro NVS 295
OS	Windows XP	Windows 7	Windows XP
OS Type	32 bit	64 bit	32 bit
Graphics Slot	PCIe Gen2 x16	PCIe Gen2 x16	PCIe Gen2 x16
PSU	475W	475W	750W

5.2.3 Compilation

All CPU code was written and compiled in Visual Studio 2008 Professional Edition. Code was compiled using the options: /Ox, /Ob2, /Oi, /Ot, /Oy, /GT, /GL. GPU code was compiled using the nvcc compiler (toolkit version 3.1) configured for GPU architecture 1.3 and 2.0.

5.2.4 Environments

Code was compiled and ran with the latest[3] toolkit version (CUDA toolkit 3.1) and the latest drivers (Developer Drivers for WinXP 257.21.)

There were two cards used, the Quadro FX 4800 with 1.5GB of RAM and the GeForce GTX470 with 1280MB MB of GDDR5 RAM. The Quadro card was placed in a HP xw4600 Workstation and the GeForce card in a Dell R5400. The HP z400 system was used to provide a CPU benchmark for the production system.

See table 5.1 for a detailed description of the systems that were used.

The HP workstations were used for normal desktop activities at the time of measurements. The Dell machine was dedicated to running the code (except for a tightVNC server and connection.)

5.2.5 Accuracy

For the digital basket option results from the CPU and from the GPU were checked against each other to ascertain that they were in reasonable agreement, results were allowed to vary a bit (depending on the number of Monte Carlo simulations that were run) since slightly different Sobol generators were used for CPU and GPU.

[3] At the time of writing.

For the capped basket the results were also compared against the valuation produced by the production code.

Part II

Results

Chapter 6

GPU Implementation

The overall program flow is shown in figure 6.1. The host code runs a single thread, when the device is called multiple threads are spawned, indicated by arrows.

The GPU option pricing routine is described in more detail below:

1: Let N_{MC} be the number of Monte Carlo iterations the user wishes to perform.
2: Set `nThreads = 256`.
3: Set `nBlocks = N` $_{MC}$ `/nThreads`, the number of blocks as defined in the section on Grids and Blocks.
4: Set `grid = (nBlocks,1,1)`.
5: Set `threads = (nThreads,1,1)`.
6: Copy distribution functions to constant memory if they fit otherwise copy to global memory.
7: Allocate global memory to `res`, `directions` and `sobol` using `cudaMalloc`.
8: Init Sobol sequences.
9: Init Brownian bridge.
10: Load initial values, Cholesky matrix, Brownian bridge parameters and caps into constant memory.
11: Start computation timer.
12: Call `sobolGPU` to generate N_{MC} quasi-random numbers.
13: This is done on the GPU, each thread computes a separate dimension.
14: Store in sobol in global memory on the device.
15: Call `cuMCStepSobolFunc<<<grid, threads>>>()`.
16: Each thread computes a single Monte Carlo iteration in parallel and computes the payoff.
17: This is stored in `res` based on thread id.
18: Call `cudaSynchronize` to synchronize state.
19: Copy `res` back to host memory and perform summation and averaging there.
20: End computation timer.
21: Deallocate global memory.

The layout of the thread code is similar to the algorithm described in chapter 3. The variable `nThreads` was hard-coded into the program, and was chosen in

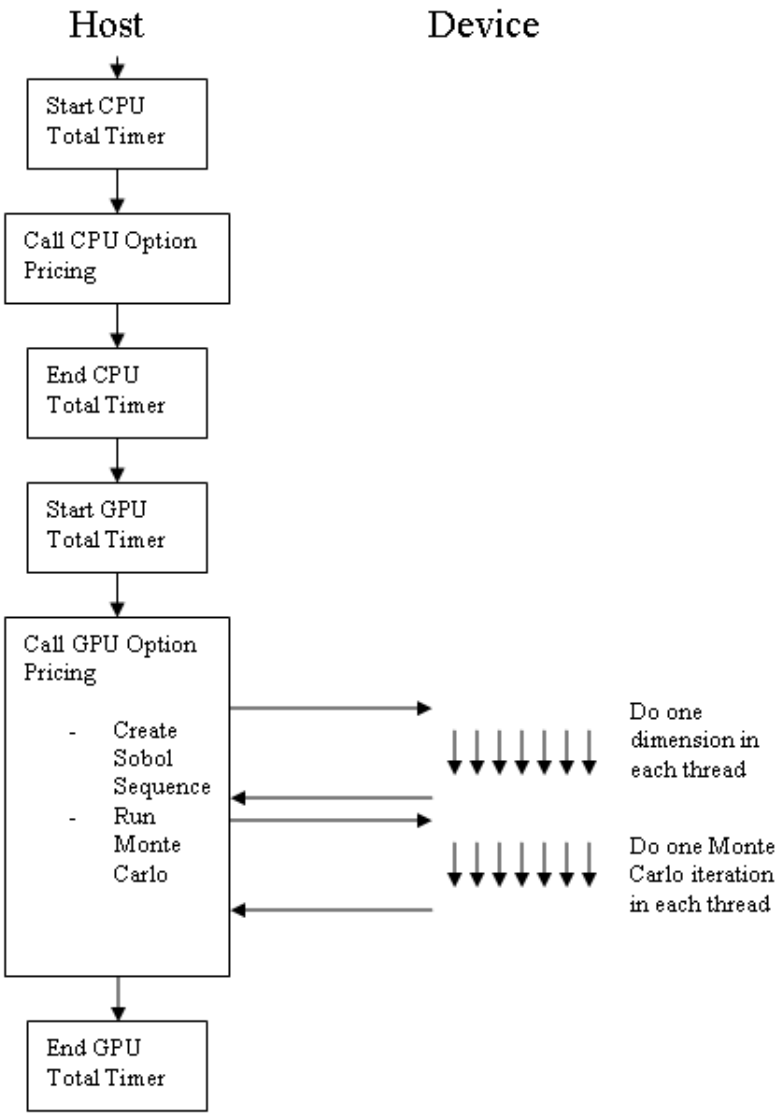

Figure 6.1. The overall architectural layout of the main program, shows the timer steps and the calculation steps.

part based on the CUDA Occupancy Calculator [1] and in part by experimentation of values using the CUDA Profiler. Choosing this value is a trade-off between the used resources (registers and shared memory) and the occupancy that is achieved.

The distribution functions mentioned on line 5 are the functions that define the evolution of the stock prices, each underlying stock has a separate function that is defined by x,y-coordinate pairs. A value from the Sobol sequence is looked up in this table or interpolated between the closest points if necessary.

In line 14 we use the CUDA specific syntax to call a kernel, this is basically a normal function call with the appended brackets to indicate the dimensions of the blocks and the grid.

Since we wanted to keep the routines general we computed each trajectory, stored it and supplied it to a payoff function. The payoff function is easily replaced if another option type is to be priced. This generality, however, comes at a cost: several option-type specific optimizations cannot be done.

[1] developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

Chapter 7

Results

This chapter describes the results of the prototype in several test cases. Pricing of a digital basket option and a capped basket option is described and the resulting speedups compared to CPU-code is illustrated.

7.1 Pricing a Digital Basket Option

7.1.1 Digital Basket Options

A digital basket option is an option in which we obtain a payoff of 0 or 1 based on certain criteria on the underlying instruments. In the test case we used a payoff of 1 if all underlying instruments had appreciated in value by at least 10% and a payoff of 0 otherwise.

This example is somewhat artificial but similar digital options are available. It was chosen since it illustrated the concepts well.

In contrast to other tests this one used the standard geometric Brownian motion model. This results in more arithmetic operations and fewer memory accesses which is beneficial to device performance.

This test used eight underlying instruments and twenty time steps.

7.1.2 Results in a Test Environment

10000 Simulations

Figure 7.1 illustrates the speedup obtained with 10000 Monte Carlo simulations. Especially note the large discrepancy between the speedup in total time and the speedup in computation time. Also note that the GTX470 is almost three times faster than the FX4800 with regard to computation time but actually slower with regard to total time. Just viewing the GFLOPS for the two cards this speedup is not quite expected (FX4800 does about 693 GFLOPS and the GTX470 about 1088 GFLOPS) but it shows that the Fermi architecture is better suited for these kinds of problems, even the GeForce cards.

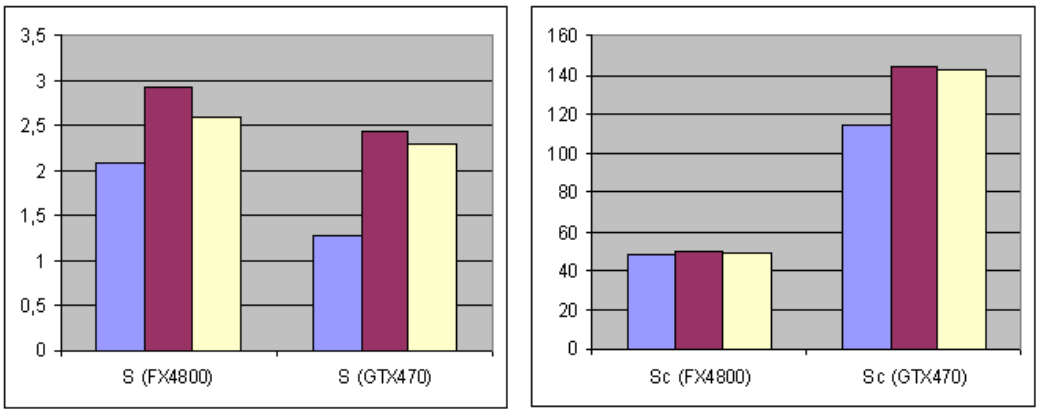

Figure 7.1. Speedup of digital basket in total time, S (left), and in computation time, S_c (right), versus Q9450. For 10000 Monte Carlo simulations. The bars represent the min, max and median of ten runs.

870400 Simulations

Figure 7.2 shows the speedup when using 870400 Monte Carlo simulations. We see that the speedups are much larger with respect to total time, this is reasonable since the overhead consumes less of the total time spent producing results.

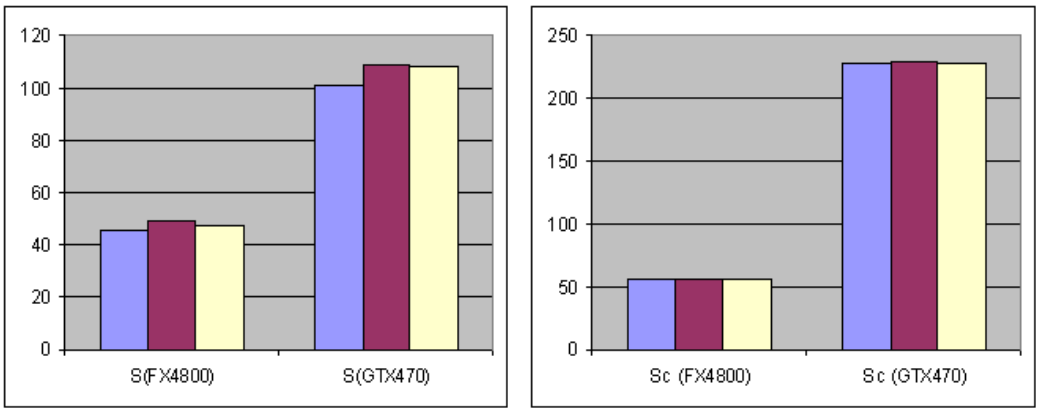

Figure 7.2. Speedup of digital basket in total time, S (left), and in computation time, S_c (right), versus Q9450. For 870400 Monte Carlo simulations. The bars represent the min, max and median of ten runs.

Varying the Number of Monte Carlo Simulations

Figure 7.3 shows the total and computation times of different hardware. In the total time graph we can clearly see the variance of the GPU total time measurement,

producing occasional spikes, especially for the GTX470.

In figure 7.4 we see the speedup of the graphics cards as compared to the Q9450 processor as the number of Monte Carlo simulations varies. It is interesting to note that the two factors seem to converge (for the Quadro FX4800 card) which is not surprising as we have discussed.

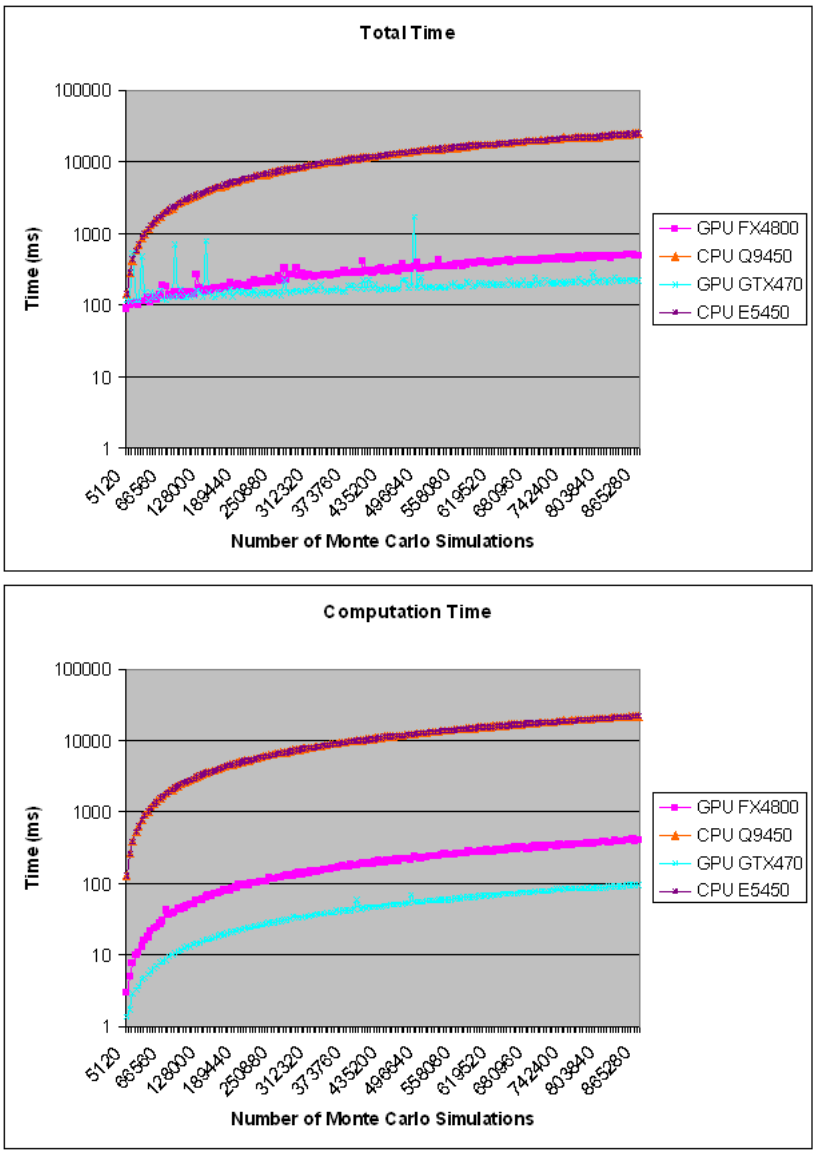

Figure 7.3. Total execution time (top) and computation time (bottom) as the number of Monte Carlo simulations was varied. Note that the time axis is in log-scale.

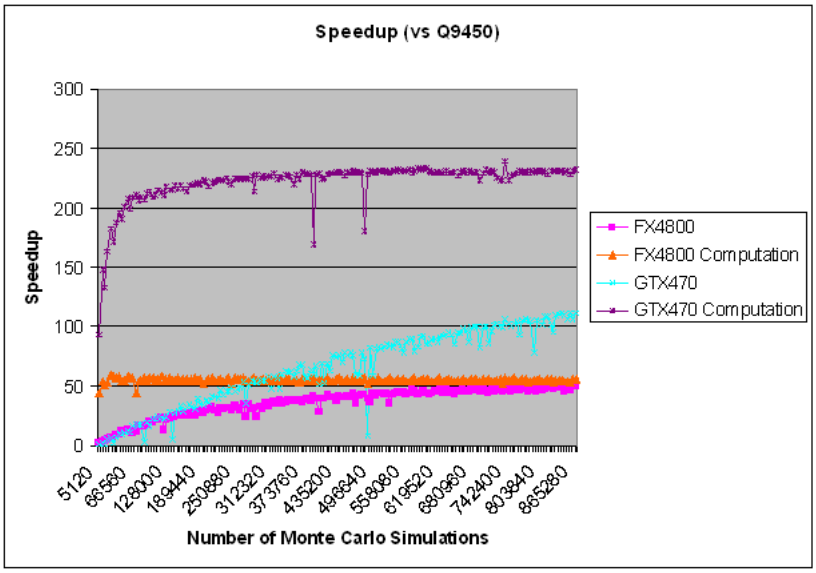

Figure 7.4. Speedup in total time, S, and computation time, S_c when varying the number of Monte Carlo simulations.

Double Precision Performance

Figure 7.5 shows the corresponding speedups when using double precision. Note that there is only data for the FX4800.

The achieved speedups are significantly higher than expected, double precision performance should be about 1/8 of single precision performance on this card. These results are probably due to single precision not being completely optimal.

7.2 Pricing a Capped Basket Option

7.2.1 Capped Basket Options

To price a capped basket option we evaluate the value for each underlying instrument at times $t_1, \ldots, t_n$, call these values $s_{i,1}, \ldots, s_{i,n}$. For each underlying instrument we have a cap $c_1, \ldots, c_u$. We then take the mean of s for each instrument $\bar{s_j} = \frac{1}{n}\sum_{i=0}^{n} s_{j,i}$ for $0 \leq j \leq u$. Then the final price of the option is $\frac{1}{u}\sum_{i=0}^{u} \min(\bar{s_i}, c_i)$.

Capped basket options are used in a real setting and the examples are picked from Handelsbankens portfolio of options.

The capped basket code also used custom functions for the evolution of each individual stock price, these were stored in global memory thus causing more memory accesses and decreased performance. In this particular case we were able to fit these functions into constant memory so the cache helped improve performance a bit. A better solution would be to store them in texture memory, in addition to a

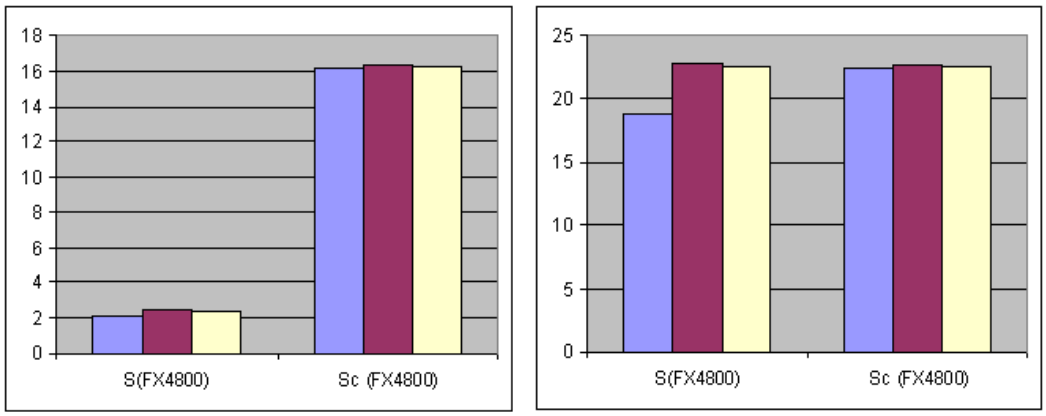

Figure 7.5. Double precision performance on the FX4800 vs the Q9450. Speedup in total time, S, and computation time, S_c. Shown are 10000 Monte Carlo simulations on the left and 870400 on the right.

fast cache texture memory offers hardware interpolation of the data. In this case we used linear interpolation so the texture memory would be sufficient, however, in future version other interpolation types will have to be supported.

7.2.2 Results in a Test Environment

The resulting speedups are shown in 7.6 and 7.7. In the first case we can see that the GPU code is actually slower than the CPU code, this is due to this particular capped basket option being slightly smaller than the digital basket (seven underlying instruments and five time steps) thus producing quicker code which causes the overhead to be even more overpowering. In the second case it is also interesting to note that, even though it is almost twice as quick in computation time, the GTX470 shows almost the same speedup as the FX4800 in total computation time.

7.3 Comparison of CPUs and GPUs in a Production System

7.3.1 The System

The valuation system used at Handelsbanken is a custom C++ framework that among other things prices a wide variety of exotic options. The system communicates with the trading platform Prime through a COM interface. Prime is very versatile and one can extend and implement modules using Python.

The calls to pricing capped basket options were redirected to the corresponding GPU function. The drawback with this approach was that everything was tailored to the existing system, for example vectors and matrices were wrapped in vector

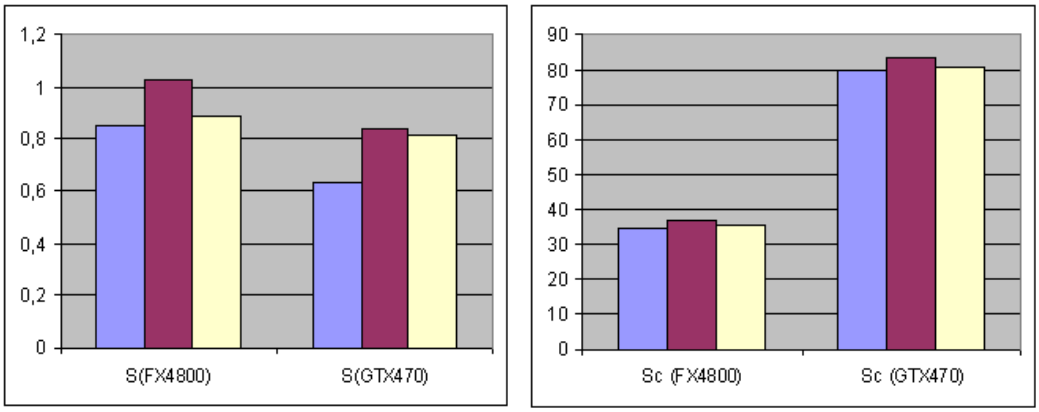

Figure 7.6. Speedup of capped basket in total time, S (left), and in computation time, S_c (right), versus Q9450. For 10000 Monte Carlo simulations. The bars represent the min, max and median of ten runs.

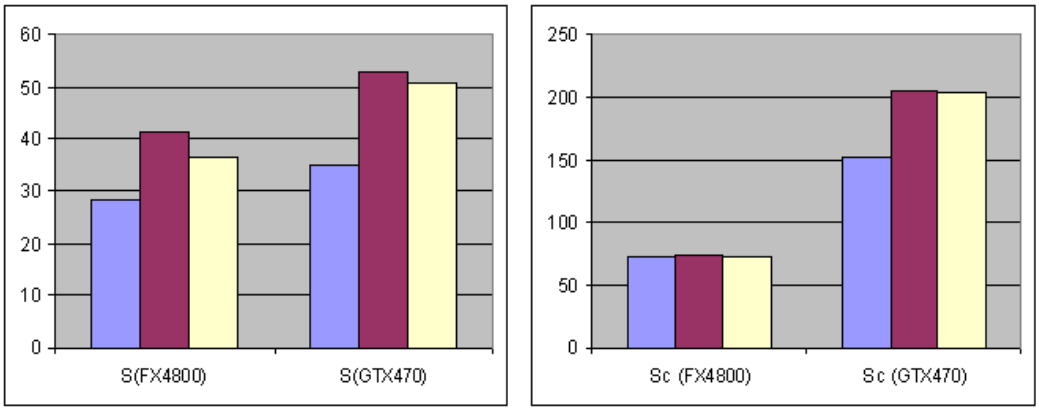

Figure 7.7. Speedup of digital basket in total time, S (left), and in computation time, S_c (right), versus Q9450. For 870400 Monte Carlo simulations. The bars represent the min, max and median of ten runs.

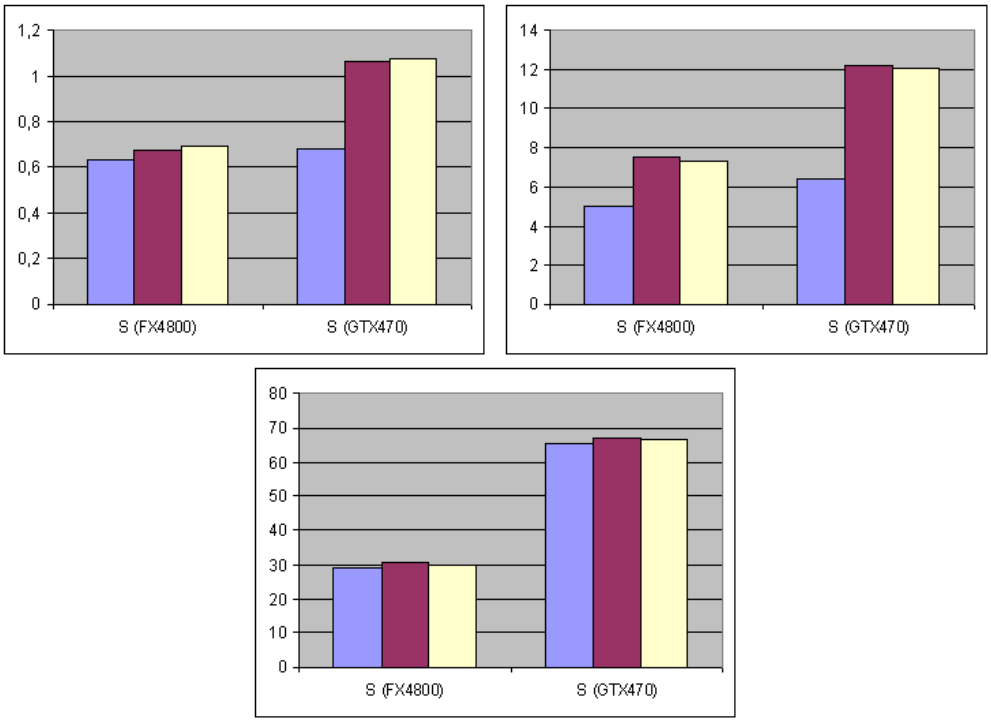

Figure 7.8. Speedup in time as seen by a user in a production system. Shown are 10000, 100000 and 870400 Monte Carlo simulations respectively.

and matrix classes which had first to be constructed in the Python code and then had to be deconstructed and moved into new array based structures in the C++ code. All data was passed as doubles which had to be converted into single precision floats which further added to the overhead.

It was noted that the capped basket pricing was slightly faster than the code used in the test environment but they were in the same order of magnitude.

7.3.2 Results

Figure 7.8 shows the speedup in time as seen by the user. We now compare to the Xeon W3565 (the business equivalent of the Core i7 960.)

Note that this comparison is not entirely fair since the CPU uses double precision and the GPU single precision, using single precision on the CPU would not significantly impact the results but using double precision on the GPU would.

7.3.3 Accuracy

To measure the accuracy and numerical stability of the GPU code we computed the delta. The delta is the derivative of the option price with respect to a underlying

instrument's spot price, as such it measures the rate of change of the option price when the underlying instrument changes value.

There were seven instruments in the basket, amongst which were ABB and SKF B. When changing the spot price (figure 7.9) of the underlying ABB the GPU closely followed the CPU both in the valuation and in the delta. There are no major spikes or discontinuities in the delta graph of neither the CPU nor the GPU.

When perturbing the spot price of SKF B (figure 7.10) we see clear spikes of both the GPU and the CPU with the valuation produced by the GPU experiencing significantly more ringing in the delta.

More instruments and the relative difference in their valuations can be found in appendix A.

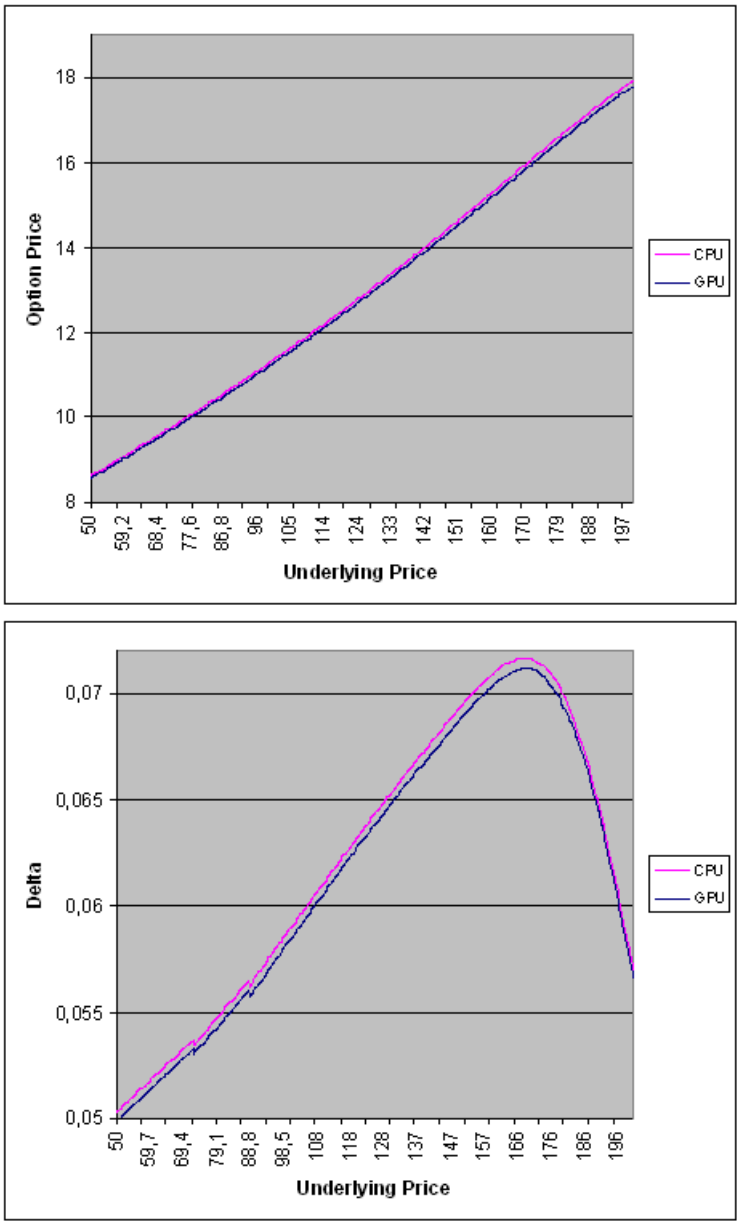

Figure 7.9. Option value (top) and delta function (bottom) as ABB changes value.

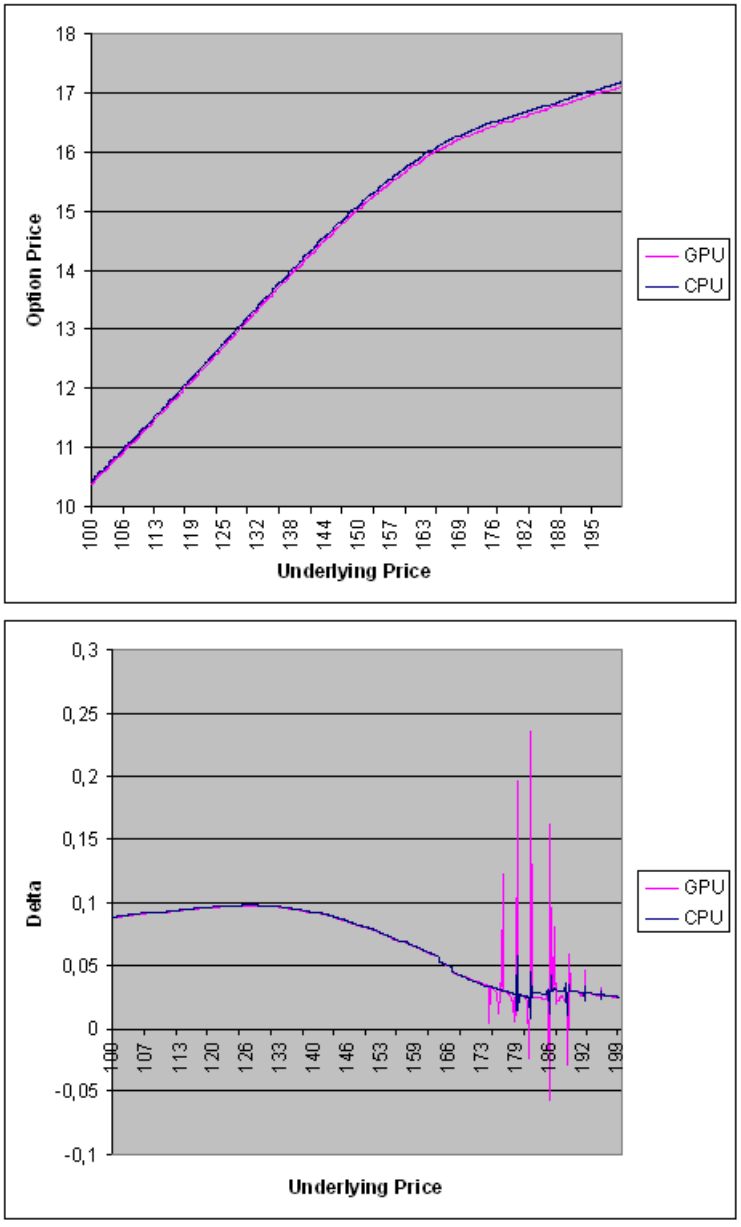

Figure 7.10. Option value (top) and delta function (bottom) as SKF B changes value.

Chapter 8

Discussion and Conclusions

In this chapter we discuss the results and draw some general conclusions about the use of graphics cards.

8.1 Limitations

8.1.1 Floating Point Types and Precision

As noted all results were obtained by using single precision floating-point types. Certain problems in for example linear algebra and quantum chemistry must use double precision or the results would be too inaccurate to be useful.

Earlier architectures had little or no focus on double precision since it is of little use in computer games but this has changed with the Fermi architecture. Double precision performance is no longer lagging behind as severely and graphics cards can provide a significant boost in performance. Unfortunately this feature is disabled in the cheaper GeForce cards and only enabled in the more expensive Tesla cards.

In the context of this problem (i.e. pricing options) single precision was assessed to be sufficient, however when computing the Greeks (basically numbers that convey how sensitive the option price is to different parameters, for example the volatility) by finite difference perturbation higher precision is desired.

8.1.2 Multi-Core Processing and Optimizations

Most modern CPUs are based on multi-core architectures, exploiting this in the program would have lead to lesser run-times for the CPU based programs and smaller factors of speedups as compared to the GPU implementations.

Producing a fully optimized parallel version would have been very interesting since it would harness the full power of the CPU, but since the final comparison systems was not parallel such a program was not implemented.

Such an implementation was made in [15] and it significantly impacted the speedup that was attained. However, the implementation was based on double precision which would affect the result.

In [15] Lee et al. argued that (they use the lattice Boltzmann method as a specific example, which claims as high as 114x speedup on a GPU) "with careful multi-threading, reorganization of memory access patterns, and SIMD optimizations, the performance on both CPUs and GPUs is limited by memory bandwidth and the gap is reduced to only 5X." Very similar considerations have to be made to produce an optimal GPU program and a 5x improvement still means that for every graphics card we need [1] 5 processors to match it, at a cost of 3-10 times that of a modern GeForce graphics card (not including additional required hardware such as motherboard, RAM etc.)

A drawback of using all cores is of course that it would impact the user using the system at the same time. While not relevant for a normal cluster, this could be a design consideration when using a grid of other's idle computers. Adding a graphics card to a user's computer would not impact her in any significant way and then running the program would not even be noticeable for a business system user.

8.1.3 Problem Size

As we have shown GPU programs scale extremely well with larger problem sizes. However, using larger problems might not be desired or necessary. The production system that was tested against previously used 10000 Monte Carlo iterations this provided a good balance between run-time and numerical accuracy. Our testing showed that while increasing the number of iterations was beneficial to accuracy there was not a major difference.

8.1.4 Optimality of the Program

There are several common measures of optimality for CUDA programs two of them are, achieved bandwidth and instruction throughput. Achieved bandwidth is measured in GB/second, this is computed as the total number of bytes read or written from global memory divided by the total runtime of the kernel. This can then be compared to the maximum obtainable of the device.

A cursory examination showed an approximate bandwidth of 20-25 GB/s when run on the GTX470 and 5-10GB/s when run on the FX4800[2] in the capped basket case. This can be compared to the maximum of 133.9 GB/s for the GTX470 and 76.8 GB/s for the FX4800. Obviously the achieved throughput is not optimal, this could be due to a couple of reasons. One reason could be that the program is not that memory intensive. Another could be that memory access patterns still are suboptimal.

Instruction throughput is the number of instructions performed divided by the number of single point arithmetic instructions that could have been executed in

[1] At the very least. Depending on the problem message passing between the additional computers that will be needed can probably lessen the speedup

[2] These numbers are highly approximate, in part based on examination of the program and in part with help from the CUDA profiler.

the same time. This number is usually between zero and one, higher being better. The program achieved an instruction throughput of 0.48 according to the CUDA profiler. This number is also suboptimal but is less telling than achieved bandwidth.

If it's possible to reach the highest figures while maintaining the generality of the program remains unknown, but more work can definitively be done in optimizing the performance.

8.2 Other Approaches

8.2.1 Field-Programmable Gate Array

FPGAs are integrated circuits that can be re-configured after they have been manufactured.

FPGAs have been used successfully in similar applications [18] to the ones described here with great success (over 50x the performance of a CPU in double precision.) [19] describes a FPGA base Monte Carlo option pricer that is 340x faster than the corresponding CPU implementation.

8.2.2 More Computers

Another approach would be to add additional computers to the grid, if we need a 5x speedup we could use 5x the number of computers. This would theoretically be a good solution but in practice it might not work out as well. For one we would have to introduce communication between the computers to transfer partial results which will be compiled, this might add significant overhead depending on the problem. There is also the issue of cost and space, computers have to be bought and maintained and space might be constrained as it is.

8.2.3 The Cell Broadband Engine Architecture

The Cell Broadband Engine is a multiprocessor architecture quite different from the standard x86 architecture. It is primarily aimed at gaming consoles (the Playstation 3 features one for example) and high-performance computing. One of the fastest supercomputers, the IBM Roadrunner, features a hybrid design of Cell and x86 based processors.

There is some evidence that the Cell processor could be useful for financial computations [20], [21].

8.3 Discussion

Although graphics cards can provide a major boost in performance we have seen that much of this performance is dependent on how measurements are made. If one uses small problems and include all overhead the results are disappointing. If on the other hand one uses large problems the results are impressive to say the least. But

this performance also comes at a cost, some applications are too sensitive to use single precision arithmetic and the faster but less accurate mathematical functions.

Another interesting issue is optimized CPU code. [15] showed that, with properly optimized CPU code, the performance gap between graphics cards and CPUs can be minimized. This has several advantages over adding a graphics card for example one could maintain high precision and it would work equally well for small problems since there is significantly less overhead.

If we were to be able to achieve a four time speedup using all cores of the Xeon W3565 the resulting speedup compared to the GTX470 would be about 3-4, while not quite as impressive it is still a major improvement.

There are a couple of factors that are not taken into account in [15], for one the Monte Carlo code is using double precision floating-point math, which is not optimal on the GTX280 they used. Another factor is that they compared a graphics card and a processor that are more than a year apart when considering release date. Current generation graphics cards (or even the most powerful of that generation, the GTX295 or the Tesla C1060) would probably have performed better.

Optimizing code for the CPU entails more than activating a compiler flag or adding an OpenMP pragma to the code, a careful analysis of thread behavior and memory accesses must be made as well as an analysis of the problem and choosing algorithms and data structures that fit into this paradigm. Also one must take full advantage of the rich set of Streaming SIMD Extensions.

Even when considering these things GPU computing can be worth while, one can, with a small additional cost, improve the total computational capacity of a single PC by a significant factor. This is probably where solutions such as OpenCL will shine, if they can truly harness the power of both the CPU and of the GPU.

Incorporating GPU code into an existing code base was surprisingly easy, the code was more or less copy-pasted into a new file in the existing project, the call to the GPU code could then be wrapped in a new class which one could instantiate and the new file was set to be compiled with nvcc.

The largest issue at the moment is the significant ringing in the delta function that was observed in figure 7.10 this behavior was also observed for other underlyings and is most likely due to precision problems. This problem has to be investigated further.

8.4 Conclusions

We conclude that GPU computing is well worth the hassle if

- The problem is parallelizable.
- The problem is sufficiently large.
- The problem can use single precision floating-point arithmetic.
- The problem is specific enough.

- Most of the problem can be computed on a GPU.
- The problem is not that memory intensive.
- The cost of development is less than the cost of extending existing infrastructure.

Before investing in GPU computing one should make sure that the existing systems are used to their full capacity, or that doing so would be more expensive than utilizing GPUs.

8.5 Further Work

The most pressing extension of this work would be to implement parallel, optimized code for option pricing on a CPU. This would give a better and fairer comparison.

Other than that, porting more code, for example finite difference methods, to GPUs would be a natural extension.

Bibliography

[1] Antonov, I. A. and Saleev, V.M. (1979.) An economic method of computing LPt-sequences. USSR Comput. Math. Math. Phys. 19, pp. 252-256.

[2] P. Glasserman, Monte Carlo Methods in Financial Engineering (Stochastic Modelling and Applied Probability) (Springer, 2003). ISBN 0387004513.

[3] P. Jaeckel, Monte Carlo Methods in Finance (Wiley, 2002). ISBN 047149741X.

[4] NVIDIA, NVIDIA CUDA Programming Guide 3.0 (2010). From `http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf` Last retreived 2010-11-18.

[5] NVIDIA, NVIDIA CUDA Best Practices Guide 3.0 (2010). From `http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf` Last retreived 2010-11-18.

[6] H. Nguyen, GPU Gems 3 (Addison-Wesley Professional, 2007). ISBN 0321515269.

[7] Kirk, David and Hwu, Wen-mei. Programming Massively Parallel Processors: A Hands-on Approach (Morgan Kaufmann, 2010), first edn. ISBN 0123814723.

[8] NVIDIA, NVIDIA Fermi Compute Architecture Whitepaper. From `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf` Last retreived 2010-11-18.

[9] Maruyama, Naoya and Nukada, Akira and Matsuoka, Satoshi. (Jul 2009.) "Software-Based ECC for GPUs," Symposium on Application Accelerators in High Performance Computing (SAAHPC'09).

[10] Sutter, Herb. (March 2005.) The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal, 30(3),

[11] Amdahl, Gene. (1967) "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities." AFIPS Conference Proceedings (30). pp. 483-485.

[12] Gustafson, John L. Reevaluating Amdahl's Law. (1988.) Communications of the ACM 31(5). pp. 532-533

[13] W. Kahan (2004). A logarithm too clever by half. From http://http.cs.berkeley.edu/~wkahan/LOG10HAF.TXT Last retreived 2010-11-18.

[14] Bennemann, Christoph and Beinker, Mark W. and Egloff, Daniel, and Gauckler, Michael. Teraflops for games and derivatives pricing. From http://quantcatalyst.com/download.php? file=DerivativesPricing.pdf Last retreived 2010-11-18.

[15] Lee, Victor W. and Kim, Changkyu and Chhugani, Jatin and Deisher, Michael and Kim, Daehyun and Nguyen, Anthony D. and Satish, Nadathur and Smelyanskiy, Mikhail and Chennupaty, Srinivas and Hammarlund, Per and Singhal, Ronak and Dubey, Pradeep (2010). Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture. pp. 451–460.

[16] Giles, Mike and Xiaoke, Su. Notes on using the nVidia 8800 GTX graphics card. From http://people.maths.ox.ac.uk/gilesm/codes/libor/report.pdf Last retreived 2010-11-18.

[17] Podlozhnyuk, Victor and Harris, Mark. Monte Carlo Option Pricing. From http://developer.download.nvidia.com/compute/cuda/sdk/website/C/src/MonteCarlo/doc/MonteCarlo.pdf Last retreived 2010-11-18.

[18] Woods, Nathan A. and VanCourt, Tom (2008). FPGA acceleration of quasi-Monte Carlo in finance. International Conference on Field Programmable Logic and Applications. FPL 2008: 335-340. ISBN 978-1-4244-1960-9.

[19] Tian, Xiang and Benkrid, Khaled and Gu, Xiaochen. High Performance Monte-Carlo Based Option Pricing on FPGAs http://www.engineeringletters.com/issues_v16/issue_3/EL_16_3_24.pdf Last retreived 2010-11-18.

[20] Agarwal, Virat and Liu, Lurng-Kuo and Bader, David A. (April 2008.) Financial Modeling on the Cell Broadband Engine. 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS), Miami, FL.

[21] Rotaru, T. and Dalheimer, M. and Pfreundt, F.-J. (2010.) Service-oriented middleware for financial Monte Carlo simulations on the cell broadband engine. Concurr. Comput. : Pract. Exper. pp 643-657.

Appendix A

Accuracy

Instrument	GPU	CPU	Diff	Diff%
AA	24,78	24,93	-0,15	-0,605326877
AB	53,73	54,2	-0,47	-0,874744091
AC	54,1	54,57	-0,47	-0,868761553
AD	9,54	9,74	-0,2	-2,096436059
AE	29,02	29,24	-0,22	-0,758097864
AF	116,2	117,88	-1,68	-1,445783133
AG	20,85	21,03	-0,18	-0,863309353
AH	21,38	21,56	-0,18	-0,841908326
AI	16,46	16,61	-0,15	-0,911300122
AJ	27,24	27,43	-0,19	-0,697503671
AK	10,09	10,29	-0,2	-1,982160555
AL	192,54	193,34	-0,8	-0,415498078
AM	14,86	14,99	-0,13	-0,874831763
AO	27,79	27,99	-0,2	-0,719683339
AP	36,26	36,19	0,07	0,193050193
AQ	-0,35	-0,37	0,02	-5,714285714
AR	16,61	16,73	-0,12	-0,722456352
AS	23,3	23,48	-0,18	-0,772532189
AT	21,82	22	-0,18	-0,824931256
AU	29,86	29,95	-0,09	-0,301406564
AV	27,79	27,99	-0,2	-0,719683339
AW	24,75	24,86	-0,11	-0,444444444
AX	22,29	22,48	-0,19	-0,852400179
AY	27,24	27,43	-0,19	-0,697503671
AZ	99,18	99,18	0	0
BA	12,89	12,84	0,05	0,387897595
BB	77,42	79,16	-1,74	-2,247481271
BC	9,76	9,91	-0,15	-1,536885246
BD	25,25	25,37	-0,12	-0,475247525
BE	29,02	29,24	-0,22	-0,758097864

Instrument	GPU	CPU	Diff	Diff%
BF	99,7	99,7	0	0
BG	125,69	126,15	-0,46	-0,365979792
BH	115,12	115,4	-0,28	-0,243224461
BI	99,9	99,9	0	0
BJ	14,05	13,95	0,1	0,711743772
BK	18,57	18,43	0,14	0,753904146
BL	98,84	98,81	0,03	0,030352084
BM	1,47	1,45	0,02	1,360544218
BN	142,16	142,23	-0,07	-0,049240293
BO	106,72	107,01	-0,29	-0,27173913
BP	16,52	16,81	-0,29	-1,755447942
BQ	97,54	97,55	-0,01	-0,010252204
BR	99,39	99,39	0	0
BS	106,51	106,39	0,12	0,112665477
BT	99,02	99,02	0	0
BU	96,77	96,84	-0,07	-0,072336468
BV	97,55	97,56	-0,01	-0,010251153
BW	28,08	28,2	-0,12	-0,427350427
BX	98,94	98,94	0	0
BY	100,75	97,78	2,97	2,947890819
BZ	99,95	99,93	0,02	0,020010005
CA	97,78	97,78	0	0
CB	114,84	114,92	-0,08	-0,069662139
CC	18,93	19,22	-0,29	-1,531959852
CD	99,93	99,93	0	0
CE	102,74	97,78	4,96	4,827720459
CF	9,64	9,89	-0,25	-2,593360996
CG	286,6	287,36	-0,76	-0,265177948
CH	112,09	112,15	-0,06	-0,053528415
CI	231,74	232,41	-0,67	-0,289117114
CJ	99,37	99,37	0	0
CK	139,87	139,99	-0,12	-0,085793952
CL	98,02	98,02	0	0
CM	21,05	21,15	-0,1	-0,475059382
CN	99,02	99,02	0	0
CO	33,23	33,28	-0,05	-0,150466446
CP	2,56	2,57	-0,01	-0,390625
CQ	99,39	99,39	0	0
CR	97,78	97,78	0	0
CS	105,68	105,79	-0,11	-0,104087812
CT	147,36	148,8	-1,44	-0,977198697
CU	117,22	117,43	-0,21	-0,179150316

Instrument	GPU	CPU	Diff	Diff%
CV	119,34	119,35	-0,01	-0,00837942
CW	98,95	98,95	0	0
CX	99,9	99,9	0	0
CY	98,58	98,64	-0,06	-0,060864273
CZ	124,26	124,42	-0,16	-0,128762273
DA	98,5	98,47	0,03	0,030456853
DB	108	108,17	-0,17	-0,157407407
DC	140,96	140,99	-0,03	-0,021282633
DD	16,64	16,64	0	0
DE	102,38	102,46	-0,08	-0,078140262
DF	19,74	20,18	-0,44	-2,228976697
DG	8,9	8,99	-0,09	-1,011235955
DI	114,32	114,33	-0,01	-0,008747376
DJ	97,22	97,22	0	0
DK	29,83	30	-0,17	-0,569896078
DM	32,24	32,19	0,05	0,155086849
DO	100,45	100,7	-0,25	-0,24888004
DP	98,98	99,08	-0,1	-0,101030511
DQ	101,93	102,08	-0,15	-0,147159816