# Monte Carlo Evaluation of Financial Options using a GPU

**Claus Jespersen**

20093084

A thesis presented for the degree of
Master of Science

AARHUS UNIVERSITET

Computer Science Department
Aarhus University
Denmark
02-02-2015

Supervisor: Gerth Brodal

## Abstract

The financial sector has in the last decades introduced several new financial instruments. Among these instruments, are the financial options, which for some cases can be difficult if not impossible to evaluate analytically. In those cases the Monte Carlo method can be used for pricing these instruments. The Monte Carlo method is a computationally expensive algorithm for pricing options, but is at the same time an embarrassingly parallel algorithm. Modern Graphical Processing Units (GPU) can be used for general purpose parallel-computing, and the Monte Carlo method is an ideal candidate for GPU acceleration. In this thesis, we will evaluate the classical vanilla European option, an arithmetic Asian option, and an Up-and-out barrier option using the Monte Carlo method accelerated on a GPU. We consider two scenarios; a single option evaluation, and a sequence of a varying amount of option evaluations. We report performance speedups of up to 290$x$ versus a single threaded CPU implementation and up to 53$x$ versus a multi threaded CPU implementation.

# Contents

## Problem definition

The financial sector has in the last decades introduced several new financial instruments. Among these instruments, are the financial options, which for some cases can be difficult if not impossible to evaluate analytically. The Monte Carlo method can however easily evaluate prices for these options. In this thesis we will investigate whether the Monte Carlo method is useful for GPU acceleration. In particular, we will investigate the evaluation of vanilla European options, the arithmetic Asian option, and the Up-and-out barrier option using the Monte Carlo method on a GPU. We will benchmark using two different scenarios: the case for a single option evaluation as well as the case of multiple options evaluation.

# Part I

# Theoretical aspects of Computational Finance

## 1 Computational Finance

Financial institutions have always depended on arithmetics and algorithms/accounting methods for proper evaluation of financial products. By financial products, we can think of loans, obligations, stock options, interest swaps, etc. In the old days, before the modern mainstream computer became available, financial institutions were dependent on the human brain for assessing financial problems. While the human brain has many impressive features, its computing capabilities is not anywhere near the capabilities of modern computers.

However, in the last several decades a lot of new products have been developed within the financial sector. There are probably many valid reasons why these products have been introduced. However, one of the explanations that cannot be dismissed is that it is actually possible to evaluate them. With the introduction of the personal computer, and the limitations of Moore's law, we have increased the computational capabilities of modern computers to a stage where no human can compete. These developments have allowed the products we denote as financial derivatives to grow significantly the last several decades. Financial derivatives are defined as special kinds of contracts on underlying assets such as stocks, bonds, commodities. These assets will be denoted *underlying.* A simple example could be a contract of the *forward* type. A forward is a contract where two entities agree to trade some underlying at a fixed price and time, e.g., company A writing a forward to company B that requires company B to buy 100,000 Microsoft Inc. stocks at the fixed price of \$40 on the 1st of January 2015. The value of this contract depends on how the underlying performs. In the case where the underlying Microsoft stock is $\geq$ \$40 at expiration, the contract results in a profit and thus is valuable, and vice versa if the stock price is $<$ \$40 at expiration. Futures are almost identical to forwards but differ in that they are traded through exchanges. E.g., the first future ever was traded at the Chicago Board of Trade in 1864 and the underlying was grain. By being

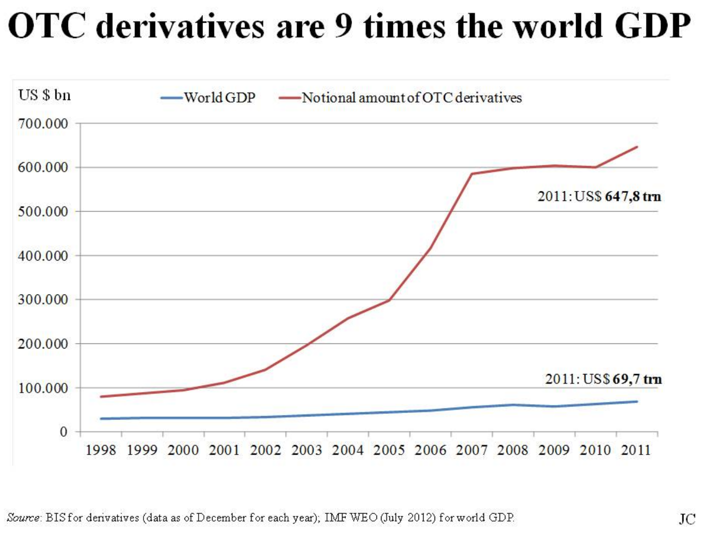

Figure 1: A graphic showing growth in over-the-counter financial derivative trading. Source: Wikipedia [19]

exchange traded contracts, futures carry no counterpart credit risk. The result of financial derivatives is that the values of the contracts are dependent on how the underlying assets perform.

The market for financial derivatives has largely been driven by globalization and the technological capabilities of modern computers. This is especially the case for some of the derivatives, namely those that only can be evaluated by computers due to their intensive computational requirements. However, far from all derivatives are computationally intensive, and thus other explanations must exist. From Figure 1 we can see that a lot of the derivatives are so-called Over-the-counter (OTC) derivatives. This means that they are not traded on exchanges, e.g., NASDAQ, but are created and traded bilaterally among financial institutions. The derivatives being OTC implies that the they are not regulated like regular exchange traded derivatives, and there has also been some speculation that OTC trading requires a more lax treatment for taxation and accounting purposes.

## 1.1 Options

In their simplest form, option contracts are closely related to forwards and futures in that they are also financial derivatives, i.e., contracts on underlying assets, but are markedly different from the others in that options give the owner a *right* but not the obligation to *exercise*, i.e., buy or sell, the underlying. Hence, the term *option*. This means that an option can be thought of as an insurance where one buys the right to exercise some pay-off that might or might not be beneficial, but if not exercised the contract will simply become void. Indeed, this instrument is often used to hedge risk – meaning minimizing the risks of asset fluctuations. Another noteworthy property of options is that they each carry an intrinsic value which is termed an option *premium*.

### 1.1.1 Types of options

There are basically two types of option contracts:

- Call options.
- Put options.

Later, we will distinguish even further since the computational complexity is not related to being neither a call nor a put option. Furthermore, options can be categorized by the types of the underlying assets and the trading markets since these are factors that determine the way the option contracts in particular are constructed.

A call option gives the bearer the right to purchase some of the contract's underlying assets at some fixed price. From a speculator's point of view, these contracts will be bought in the event that the speculator expects the asset to appreciate in value. We typically denote it a long call contract if it is purchased. Likewise, we denote it a short call contract if it is sold. A simple example: a trader purchases a call option on 100 Microsoft stocks, typically exchange traded stock options are denominated in 100 of the underlying stock per contract, and let the option have a exercise price ($E$) of \$100 and a premium ($P$) of \$10. Thus by a given spot price ($S$) the following formula one can calculate the pay-off $= S - E - P$. And thus if $S - E > P$, i.e. the spot price less the contracts exercise price is larger than the premium, then the speculator exercises the option and yield a net profit. If $0 \leq S - E \leq P$ then the speculator will exercise the contract will not yield a profit since the paid premium is larger than the

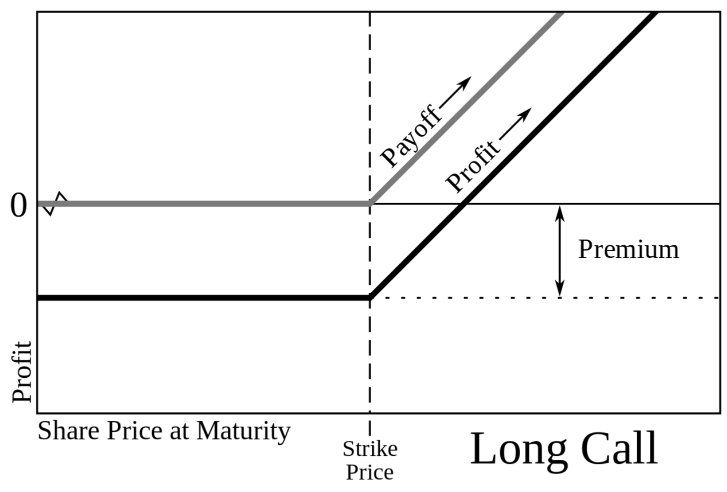

Figure 2: A graphic showing the pay-off of a long call option. Source: Wikipedia [19]

difference between the spot and exercise price. Otherwise the contract becomes void and will thus not be exercised. Figure 2 gives an overview of the pay-off of a long call option.

A put option is an option that gives the bearer the right to sell some of the contracts underlying assets at some fixed price, rather than just selling the assets up front. From a speculators point of view, then theses contracts will be bought in the event that the speculator expects the asset to depreciate in value. We also typically denote it a long put contract if one purchases the option contract otherwise we call it a short put contract if one instead sells a put option. A simple example: a trader purchases a put option on 100 Microsoft stocks. Let the option have a exercise price ($E$) of \$100 and a premium ($P$) of \$10. Thus by a given spot price ($S$) the following formula one can calculate the pay-off $= E - S - P$. And thus if $E - S > P$, i.e. the contracts exercise price less the spot price is larger than the premium, then the speculator exercises the option and yield a net profit. If $0 \leq E - S \leq P$ then the speculator will exercise the contract will not yield a profit since the paid premium is larger than the difference between the spot and exercise price. Otherwise the contract becomes void and will thus not be exercised. Figure 3 gives an overview of the pay-off of a long put option.

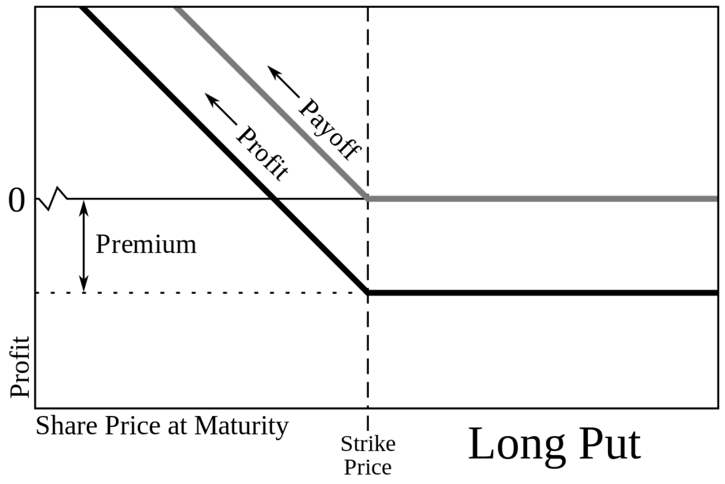

Figure 3: A graphic showing the pay-off of an long put option. Source: Wikipedia [19]

### 1.1.2 Exotic options

There are many different ways to distinguish option contracts: these examples are only a few of the categories:

- Which markets they trade: There exists financial options on bonds as well as stocks but also on commodities like grains and cattle.
- Basic type: whether it is a short or long contract, and whether it is a call or a put option as described in the previous chapter.
- Complexity of the option or option *style*. Whether it is a simple (vanilla) option or a complex (exotic) option. Typically, most exchange traded options are of the simple type whereas most OTC options are of exotic nature.
- Underlying asset type: The underlying asset can either be of the financial type (stocks, bonds, etc.) or of the *real* asset type (real estate, properties, etc.).
- Trade type: whether it is exchange traded or OTC traded.

As can be seen from above, options come in many shapes and sizes – from a mathematical point of view, there exist infinitely many different option contracts. However, for the purposes of this thesis, we will investigate the simple vanilla European option as well as the exotic option types: arithmetic Asian and the barrier options .

Typically, we distinguish between simple and exotic options by their payoff. Traditionally, we describe both European and American style options as vanilla options since they are much alike and the payoffs are identical. The only difference between American and European options is on which dates the options can be exercised. A European option may only be exercised at expiration time whereas an American option can be exercised at any point of time from its inception to its expiration. While this may sound as a minor, subtle, or trivial difference, it actually increases the complexity of evaluating an American option significantly compared to a European counterpart.

Exotic options on the other hand can vary much more than the vanilla options. The major difference between vanilla and exotic options is related to the payoff as mentioned earlier. However, since the exotic options can vary in many other regards, it would be to simple to only consider the payoff. Just to give an overview, here is an enumeration of a few complex options:

- Bermudan options: An option where one can exercise at specific dates and/or on expiration time. Essentially, this is an option that lies somewhere between an American and a European option.
- Asian options: An option where the the value of the underlying gets averaged till expiration.
- Barrier options: An option where a property of the underlying asset needs to reach some barrier before it can be exercised or become void.
- Basket options: An option that has more than one underlying asset.

The enumeration above is far from exhaustive, but it lists the most common examples of exotic option types. What is especially interesting and almost common to all of them, is that the valuation of them are often quite computationally intensive. The major reason for this computational complexity is that there rarely exists an analytical way of evaluating an option and thus one needs to simulate/approximate the value.

## 1.2 Pricing of options

The complexity of evaluating options can range from a trivial analytical solution to a path-dependent, very computationally intensive simulation. The standard European option, which is the easiest to price, is typically evaluated using the Black-Scholes model. The Black-Scholes model was first published in 1973 in the paper "*The Pricing of Options and Corporate Liabilities*" [4] and has since been developed further into more complex models, e.g., the Heston model [10]. Nevertheless, Robert Merton and Myron Scholes received the Nobel Prize in Economics in 1997 for the development of this model.

### 1.2.1 The Black-Scholes Partial Differential Equation

Like most economic models the Black-Scholes PDE has a lot of assumptions:

- The risk-free interest rate is constant throughout the contract.
- The movement of the underlying asset is assumed to be a random walk with a constant drift, i.e., a geometric Brownian motion.
- The stock does not pay dividends.
- There are no transaction costs or fees.
- There are no arbitrage opportunities in the market.
- It is possible to borrow and lend any amount of cash at the risk-free interest rate.
- It is possible to buy and sell any amount of the underlying asset.

However, before we discuss the model, let us look at the essential partial differential equation (PDE) that one needs to solve to price European options [20]:

$$\frac{\partial V}{\partial t} + \frac{\sigma^2 S^2}{2}\frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

where $V$ is the price of the option as a function of time $t$, $r$ is the risk-free interest rate, $\sigma$ is the volatility of the stock, and $S$ is the stock price. However, applying some simple arithmetics and one gets

$$\frac{\partial V}{\partial t} + \frac{\sigma^2 S^2}{2}\frac{\partial^2 V}{\partial S^2} = rV - rS\frac{\partial V}{\partial S}$$

The left hand side now consists of two terms: *theta* and *gamma*. Theta is the time decaying term $\frac{\partial V}{\partial t}$, i.e., the change in derivative value due to time increasing and gamma, $\frac{\sigma^2 S^2}{2}\frac{\partial^2 V}{\partial S^2}$, is the convexity of the derivative value with respect to the underlying asset. On the right hand side, we also have two terms: the first term $rV$ is the risk-free return from holding a long position in the derivative and the second term is a short position of $\frac{\partial V}{\partial S}$ shares in the underlying asset.

While the Black-Scholes equation can be solved analytically, it is in general not possible for all PDEs to be solved analytically. 'Analytically' in this domain means that there exists a non-stochastic formula or function that can easily be evaluated and that yields a deterministic result. Thus, if there does not exist an analytical formula for a given PDE, then one needs to simulate and approximate a solution. When one derives the Black-Scholes equation, there is an assumption (already mentioned above) that the movement of the underlying asset (most often a stock) adheres to a geometric Brownian motion. Thus, we note that a geometric Brownian motion in this domain can be formulated as

$$\frac{dS}{S} = \mu\ dt + \sigma\ dW$$

where $W$ is a stochastic variable, i.e., a Wiener process. One can then simply use Monte Carlo simulations to price the options. We will investigate this further in the next chapter.

### 1.2.2 Solving the PDE and pricing vanilla European options

As mentioned earlier, when one has applied the boundary and terminal conditions of a derivative, the PDE can be solved numerically with, for instance, the finite difference method or Monte Carlo simulations. However, in certain cases, it is possible to analytically derive a formula for pricing a given option, and in particular it is possible for a European call option. For a European call option we have the following boundary conditions:

$$\begin{aligned} C(0,t) &= 0 \text{ for all } t \\ C(S,t) &\to S \text{ as } S \to \infty \\ C(S,T) &= \max\{S-K, 0\} \end{aligned}$$

Here, $T$ denotes the expiration time and $t$ the time. The two first boundary conditions are a bit theoretical since they rarely occur. The first condition tells us that a stock worth 0$ will continue being worth 0$ (which is obvious since then the company is bankrupt). The second condition is simply the case where a stock heads for infinity, and one way to circumvent it is to ignore the gamma term - this case does not really happen in the real world anyway. Finally, one can derive and use the following formula [11]:

$$C(S,t) = N(d_1)S - N(d_2)Ke^{-r(T-t)}$$

where

$$\begin{aligned} d_1 &= \frac{1}{\sigma\sqrt{T-t}}[\ln(\frac{S}{K}) + (r + \frac{\sigma^2}{2})(T-t)] \\ d_2 &= \frac{1}{\sigma\sqrt{T-t}}[\ln(\frac{S}{K}) + (r - \frac{\sigma^2}{2})(T-t)] \\ &= d_1 - \sigma\sqrt{T-t} \end{aligned}$$

Here, $N(x)$ is the standard normal distribution function:

$$N(x) = \frac{e^{-\frac{1}{2}x^2}}{\sqrt{2\pi}}$$

However, if one were to evaluate a vanilla European call option using a non-analytical solution, then one would have to solve [11]

$$C\left(S,t,T\right) = e^{-rT}\mathbb{E}\left[\left(S(t_i) - K\right)^+\right]$$

where $r$ is the risk free interest rate, $S$ is the underlying asset price, $t$ is the time, $T$ is the expiration time, $\mathbb{E}$ is the expected value, and $(S(t_i) - K)^+$ denotes $\max\left(S(t_i) - K, 0\right)$.

## 1.3 Asian options

The Asian option (or average option) is an exotic option and differs quite a lot from the standard vanilla European option. The averaging of the Asian option is related to the underlying asset's or assets' movement from start to expiration, and thus peaks in either direction do not affect the option as much as they do for the vanilla European option. This gives the Asian option the following advantages:

- Asian options are usually somewhat cheaper and carry lower premiums due to its lower volatility of the underlying compared to other options.
- It is typically more in line with the accounting principle of averaging fluctuating costs over some time frame, and thus end users of options prefer these to other more volatile options.
- Asian options offer protection against manipulation of the underlying asset, e.g., a spike in price suddenly before expiration does not affect the Asian option significantly.

However, we distinguish between two fundamentally different Asian options, namely the arithmetic Asian option:

$$C\left(S,t,T\right) = \frac{1}{T}\int_t^T S(t)dt$$

as well as the geometric Asian option:

$$C\left(S,t,T\right) = \frac{1}{T}\int_t^T \ln\left(S(t)\right)dt$$

For the purposes of this thesis, we will limit the scope to the arithmetic case. The arithmetic Asian option does not have an analytical solution and is thus an excellent candidate for the Monte Carlo method. The discrete version of the arithmetic Asian option is

$$C\left(S,t,T\right) = \frac{1}{N}\sum_{i=1}^{N} S(t_i)$$

where there are $N$ time steps: $t = t_1, t_2, ..., t_N = T$. The solution to the discrete version of a call option is the following [11]:

$$C\left(S,t,T\right) = e^{-rT}\mathbb{E}\left[\left(\frac{1}{N}\sum_{i=1}^{N} S(t_i) - K\right)^+\right]$$

Here, $r$ is the risk free interest rate, $S$ is the underlying asset price, $t$ is the start time, $T$ is the expiration time, $\mathbb{E}$ is the expected value, and $(S(t_i) - K)^+$ denotes $\max\left(S(t_i) - K, 0\right)$.

## 1.4 Barrier options

Barrier options are options that feature at least one barrier. By barrier, we mean a certain value where *something* happens once the underlying stock hits this barrier. We typically define this *something* by the following:

- For knock-in barrier options, once the underlying reaches the (upper or lower) barrier, the option becomes non-void.
- For knock-out barrier options, once the underlying reaches the (upper or lower) barrier, the option becomes void.

Thus for the barrier option we get both a call and a put, and for each of these we get four variations:

- Up-and-out
- Up-and-in
- Down-and-out
- Down-and-in

This implies that we have eight variations in total. In this thesis, we will only look at the Up-and-out put barrier option, which can be priced as follows [11]:

$$P\left(S,t,T\right) = e^{-rT}\mathbb{E}\left[\left(\frac{1}{N}\sum_{i=1}^{N} K - S(t_i)\right)^{+} \mathbf{1}_{M_S<B}\right]$$

where $r$ is the risk free interest rate, $S$ is the underlying asset price, $t$ is the start time, $B$ is the barrier level, $T$ is expiration time, $\mathbb{E}$ is the expected value, and $\left(S(t_i) - K\right)^{+} \mathbf{1}_{M_S<B}$ denotes $\max\left(S(t_i) - K, 0\right)$ when $(S(t_i)) < B, \forall i$ and otherwise 0 (void).

## 1.5 Stochastic processes

Above, we used the term $S(t_i)$ to denote the price of some asset. However, we did not explicitly state how such a price evolution would be simulated. It turns out that there exist several valid Stochastic processes which can be used to model financial assets.

### 1.5.1 Geometric Brownian Motion

A time continuous stochastic process in which the logarithm of the randomly varying quantity follows a Brownian motion or more generally a Wiener process is denoted a geometric Brownian motion. This means that some stochastic process $S_t$ is said to follow a geometric Brownian motion if it satisfies the following stochastic differential equation [20]:

$$dS_t = \mu S_t dt + \sigma S_t dz$$

where $\mu$ is the drift, $\sigma$ is the volatility, and $dz = \epsilon(t)\sqrt{dt}$ is the Wiener process, where $\epsilon(t)$ is a standard normal distributed random variable. Now, to derive an analytical solution that can be implemented and thus used for simulation purposes, we use Ito's lemma that states that for some random process $s$ defined by the Ito process that

$$ds(t) = a(x,t)dt + b(x,t)dz$$

where $dz$ is the standard Wiener process as mentioned above. If we then let $y(t) = F(x,t)$ which satisfies the Ito equation

$$dy(t) = \left( a\frac{\partial F}{\partial x} + \frac{\partial F}{\partial t} + b^2 \frac{\partial F}{2\partial x} \right) dt + b\frac{\partial F}{\partial x}dz \tag{1}$$

then if we apply Ito's lemma to the process $F(S_t) = \ln S_t$, we can see that

$$a = \mu S,\ b = \sigma S,\ \frac{\partial F}{\partial S} = \frac{1}{S},\ \frac{\partial^2 F}{\partial S^2} = -\frac{1}{S^2}$$

Substituting these values into (1), we get that

$$d\ln S = \left( \frac{a}{S} - \frac{b^2}{2S^2} \right) dt + \frac{b}{S}dz = \left( \mu - \frac{\sigma^2}{2} \right) dt + \sigma dz$$

By Ito integrating on both sides, we obtain that

$$\ln(S_t) - \ln(S_0) = \left( \mu - \frac{\sigma^2}{2} \right)(t-0) + \sigma dz$$
$$\iff S_t = S_0 e^{\left( \mu - \frac{\sigma^2}{2} \right) dt + \sigma dz}$$

and, finally, by using the fact that $dz = \epsilon(t)\sqrt{dt}$ for a normal Wiener process, we conclude that [20]

$$S_t = S_0 e^{\left(\mu - \frac{\sigma^2}{2}\right)dt + \sigma\sqrt{dt}\epsilon(t)}$$

## 1.6 Related work

On the subject of Monte Carlo simulated option pricing performances in the literature, we get ranges from one extreme of a $900x$ speedup [14] to the opposite extreme of only a $2.5x$ speedup [12]. The speedup of course depends on what kind of option, e.g., vanilla European, Asian, etc., and what kind of pricing model, e.g., analytical, Monte Carlo, Binomial tree, etc. Hence, one has to be careful with the interpretations of results. However, the authors' backgrounds might also influence the results. In the paper titled "*Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU*" [12] all eight authors are employed by Intel. They deliberately chose to use double-precision on an Nvidia mainstream GPU that is known to have a crippled double-precision performance due to Nvidia marketing a much more expensive line of GPUs for double-precision computing. Furthermore, double-precision is not necessary for Monte Carlo simulations. Had they picked single-precision instead, they would have gotten at least a speedup of $15x$ or more.

On the other end of the spectrum, one does not really demonstrate anything by using a single-threaded Monte Carlo option pricer in Matlab versus a highly optimized GPU version even though one gets a speedup of $900x$ [14]. Others demonstrate a much more fair comparison using a multi-threaded optimized CPU version versus a GPU implementation showing a more modest speedup of $50x$ versus the multi-threaded CPU version and around $200x$ versus the single-threaded CPU version [1, 18, 21]. It is also popular to use energy efficiency as performance metric [7], i.e., the number of options evaluated per Watt usage. Whatever performance metric one uses, the "mathematization" of the financial sector necessitates the usage of high performance in the present as well as in the future [22].

# 2 The Monte Carlo Method

This chapter provides an overview over the practical as well as theoretical perspectives of the Monte Carlo method in general but also in particular its application within the domain of computational finance. The Monte Carlo method is used in general within the financial sector [6] – particularly within the domain of option pricing [5]. This chapter is a necessity to understand why the Monte Carlo method is not only necessary but also requires a considerable amount of computational power. Although the Monte Carlo method has been used since ancient times, it was formalized by Stanislaw Ulam and John von Neumann in the late 1940s [19].

## 2.1 A classical Monte Carlo method example

The Monte Carlo method, in general terms, can be summarized as follows:

- Define a domain of possible inputs.
- Find a suitable probability density function to make a suitable probability distribution for generating random inputs.
- Perform a deterministic computation on the random inputs.
- Aggregate/average the results.

Pi ($\pi$) is one of the classical constants that one meets in almost any scientific context from classical geometry to post modern physics, and yet the constant is only an approximation since it is an irrational number. There exist many ways to approximate $\pi$, but one in particular is a classical Monte Carlo application. The algorithm for the Monte Carlo method for estimating $\pi$ is

- Draw a square on the ground and then draw a circle touching every side of the square.
- Uniformly scatter some rice grains of uniform size over the square area.
- Count the rice grains that landed inside the circle as well as outside.
- Divide the inside count by the total rice grain count. This gives you an approximation of $\frac{\pi}{4}$. Multiply the result by 4 to get the approximated $\pi$.

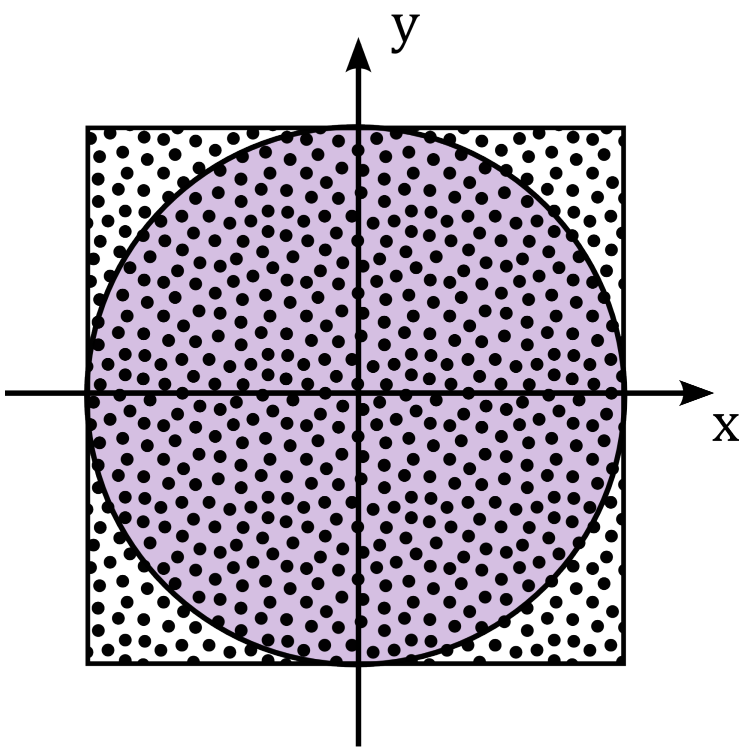

Figure 4: A graphic showing one way of estimating pi using the Monte Carlo method. Source: wikipedia.org [19].

An example of the $\pi$ approximation algorithm can be seen in Figure 4. However, as we also see from the figure, the circle and square are only vaguely covered with dots. Intuitively, one would expect that the approximation of $\pi$ would be more accurate the more rice grains are distributed over the area as described in the above algorithm, and in fact this happens to be the case for the Monte Carlo method in general. With the $\pi$ example in mind, consider the case where we only distributed 100 rice grains and 84 landed within the circle area. Then one gets an approximated $\pi$ of 3,36... which is a very crude approximation. However, if we distribute 10,000 rice grains, i.e., we increase the amount of simulations, and imagine that 7854 rice grains landed within the circle, we get an approximation of 3,1416. Thus by increasing the amount of simulations, the experiment converges to the true $\pi$: in this case we went from a completely unusable result to a still very crude result but at the cost of increasing the amount of simulations with two orders of magnitude (100x). At this point, it should be clear that the Monte Carlo method does converge to the true value the experiment seeks – although at the cost of increased compute requirements.

### 2.1.1 Formal specifications of the Monte Carlo method

Although no true formal definition exists of the Monte Carlo method, it can in general be considered a method for solving or modelling a subset of the stochastic simulations of probabilistic models. Ripley [15] limits the Monte Carlo method to Monte Carlo integration, whereas Sawilowsky [16] distinguishes further between simulation, Monte Carlo method, and Monte Carlo simulation. He defines them as follows:

- A simulation is a fictional/virtual representation of an event/object of the real world.
- The Monte Carlo method is a technique for solving a mathematical or, typically, a statistical problem.
- A Monte Carlo simulation uses repeated sampling to solve a problem or determine some properties of some phenomenon/problem.

To this end: while one might intuitively have a good idea of the Monte Carlo method, there exists no single formal definition. However, the typical definition would be that of Monte Carlo integration.

### 2.1.2 Monte Carlo integration

The problems that Monte Carlo integration addresses are the computation or evaluation of multidimensional definite integrals. Consider the following integral:

$$\alpha = \int_a^b f\left(x\right) dx$$

If one were to approximate the value of $\alpha$ in the above integral using Monte Carlo integration, one would draw the random samples $U_i, i = 1, ..., n$, from the uniform distribution over the continuous range of the integral $[a, b]$ and then finally compute $\hat{\alpha}$ as

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

where $\alpha$ approaches $\hat{\alpha}$ as $n$ goes to infinity as the law of large numbers ensures that

$$\lim_{n\to\infty} \hat{\alpha} = \alpha$$

The complexity of the Standard Error of the Monte Carlo method is $\mathcal{O}\left(\frac{1}{\sqrt{N}}\right)$ where $N$ is the amount of samples, i.e., the number of simulations. Running time therefore depends on the amount of precision needed for each individual problem that is solved using the Monte Carlo method.

### 2.1.3 Quasi-Monte Carlo

The Quasi-Monte Carlo method is a modified version of the regular Monte Carlo method. In the Quasi-Monte Carlo method we use quasi-random input, i.e., low-discrepancy sequences, instead of using regular pseudo-random input. Low-discrepancy sequences, such as Sobol [17] or Halton [9] sequences, are deterministic but yet "appear" random for their purpose, and thus the term quasi-random should be taken with a grain of salt. However, the primary advantage of low-discrepancy sequences is that they fill out the given domain of numbers much better than those of a pseudo-random generator. The Quasi-Monte Carlo method reduces the complexity of the Standard Error to $\mathcal{O}\left(\frac{(\log N)^d}{N}\right)$ where $d$ is the dimension of the integral. In general, however, the Quasi-Monte Carlo method performs closer to $\mathcal{O}\left(\frac{1}{N}\right)$ (when $d$ is small) which is why the Quasi-Monte Carlo method in the literature is often being denoted as $\mathcal{O}\left(\frac{1}{N}\right)$ [3]. There is currently an ongoing debate in the literature on whether or not Quasi-Monte Carlo is worthwhile for multi-dimensional integrals (when $d$ is large) [13]. For this thesis, the Quasi-Monte Carlo method will not be used.

### 2.1.4 Optimizations

The standard Monte Carlo method has a relatively slow rate of convergence, $\mathcal{O}\left(\frac{1}{\sqrt{N}}\right)$ as mentioned above, and, thus, to decrease the error by a factor of two, we need to increase the number of simulations with a factor of four. A number of optimizations exist to increase the rate of converge, and these techniques are often referred to as variance reduction techniques. Quasi-Monte Carlo is one of them, but others are the antithetic variates, the control variates, and the stratified sampling [15]. It is worth noting that these optimization techniques are limited to optimizing the input data to the Monte Carlo method rather than optimizing the Monte Carlo method itself.

Part II

# A review on Modern Computing Hardware

# 3 GPU computing

The purpose of this chapter is to give the reader a thorough introduction to general purpose GPU computing (GPGPU). GPGPU computing has developed from its very basic graphics pipeline purposes to a much more mature general purpose application framework. The following subsections highlight the history of GPGPU as well as discuss various GPGPU frameworks such as OpenCL and how these frameworks have allowed using GPUs for much broader purposes than just graphics rendering.

## 3.1 GPU vs. CPU

Before investigating the details of GPU computing, it is perhaps worthwhile to discuss the drivers behind the recent shift towards GPU computing specifically as well as towards massive parallelism in general.

Traditionally, the performance gains realised in computing have been attributed to increasing amounts of transistors on integrated circuits (ICs) as well as the frequency with which these transistors could switch. Moore's law states that the number of transistors doubles every 1.5-2 years, and thus increasing the logic and computing power has been true since its inception circa 1965[1]. Up until the last single-core super-scalar Netburst architecture, Intel's Pentium 4 CPU family, we could account on this law for an ever-increasing single-thread sequential performance.

The Prescott revision of the Pentium 4 CPUs where made using a lithography of 90 nm and reached a peak of 3,8 GHz and a thermal design power (TDP) of 115 Watts. At this point, however, Intel started experiencing extreme power requirements and thus accompanying heat dissipation problems, especially when

[1]Some interpret Moore's law as stating that the performance will double every 1.5-2 years rather than that the number of transistors will.

trying to increase performance, and thus at this point the law of diminishing returns became the final frontier. The reason for this diminishing returns barrier is that for each performance increment of the processor, the increased TDP and heat dissipation requirements increase more than proportionally to the performance gain.

Intel initially had a goal of reaching 10 GHz with its Netburst architecture. This was in hindsight far from achievable so instead they, including the rest of the industry, started pursuing multi-core CPUs with much better efficiency with regards to performance/TDP per core. Today, it is not uncommon to have 2-8 physical cores and 2-16 logical cores within a single CPU. Even low power CPUs for mobile devices often contain two or more cores.

## 3.2 OpenCL history

OpenCL is a framework that consists of a software library to write programs than can be executed on heterogeneous hardware platforms such as Central Processing Units (CPUs), Graphical Processing Units (GPUs), as well as field programmable gate arrays (FPGAs). OpenCL can be considered a superset of the C99 programming language that adds a few extra keywords. Although intended to run on virtually all mainstream processors, it is intended for massive parallelism and thus heavily targets GPUs and GPU-like hardware architectures, e.g., Intel's Xeon Phi coprocessors. It has been developed at the Khronos Group since 2008, a consortium consisting of major software and hardware companies, e.g., Apple, Intel, Nvidia, and AMD, although it was initially developed by Apple only.

The competing framework, Compute Unified Device Architecture (CUDA) by Nvidia, which only runs on Nvidia hardware, has historically been the most successful implementation measured by market share. OpenCL has nevertheless gained a lot of market share since AMD joined OpenCL and will probably be the dominating framework in time due to its support by most major software and hardware manufacturers including Nvidia – as well as the fact that it runs on virtually all modern CPUs and GPUs.

## 3.3 General-Purpose computing on GPU (GPGPU)

To understand OpenCL, one needs to understand GPGPU. GPUs have historically been developed to serve one need and one need only: to accelerate the graphics pipeline in games. This allowed for ever-increasing levels of detail in graphics rendering capabilities. Rendering graphics on a screen essentially consists of calculating the color value of each pixel. This can be done in parallel and thus graphics rendering is a perfect match for parallel hardware architectures.

Up until the late 90s, it was quite common to do graphics rendering in software, typically denoted "software rendering", which was executed by the machines' CPUs. It was quickly discovered that CPUs were not the best fit for graphics rendering due to the limiting processing power and the lack of accelerated parallel execution because of the sequential architecture. In the early 2000s, the graphics pipeline started to expand with the advent of programmable shaders. A shader is basically a small program executed on each primitive (fragment or vertex) that could be utilized to create much more realistic graphics with lighting and shadows – unfortunately with the cost of higher computational requirements.

Due to the ever-increasing performance requirements of graphics rendering as well as the mainstreams CPUs lack of parallel execution performance, discreet graphics cards were no longer merely nice to have for extra features but became a real requirement. It was quickly realised that these discreet graphics cards, which often offered an overwhelming amount of computing power, probably could be used for other purposes than pure gaming. One of the first alternative usages was to accelerate matrix multiplications. This was done by exploiting the programmable shaders in such a way that, when executed, it would use the whole graphics pipeline – as if it were real graphics being rendering in a game. This was clearly cumbersome since it required rewriting a problem into a another problem solvable by a limited graphics rendering pipeline. ATI (now AMD) introduced the ATI FireStream framework in 2006 and this was the first fully fledged framework for general purpose programming of GPUs directly supported by a hardware manufacturer. When AMD bought ATI and shortly thereafter introduced OpenCL, they decided to pursue OpenCL rather than FireStream. In the middle of the 2000s, Nvidia started developing CUDA as a competitor to ATI's FireStream and released it in 2007. CUDA has been the most successful GPGPU implementation to date by market share.

## 3.4 Hardware architectures

Due to the nature of graphics pipelines, GPUs are today a completely different piece of hardware than traditional CPUs. GPUs today are many-core devices that can execute thousands of threads in parallel.

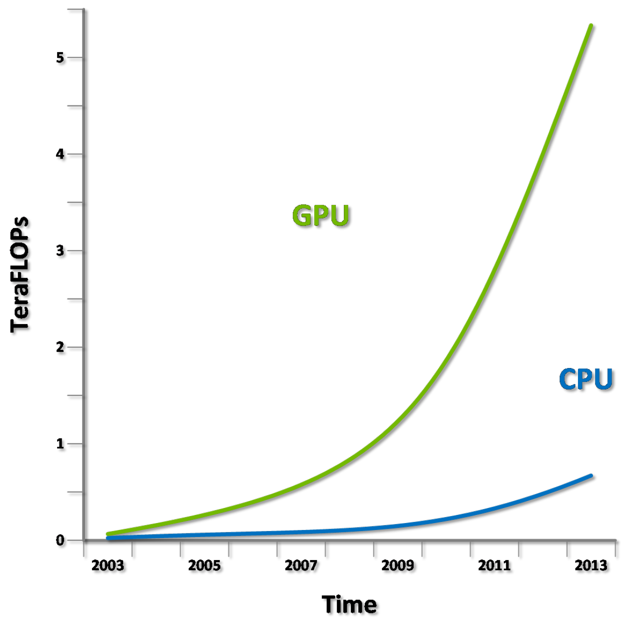

Figure 5: A graph that shows the development in raw single-precision compute performance in GPUs and CPUs. Source: Nvidia.

Since the graphics pipeline can be executed in parallel, it is no surprise that modern GPUs are designed to execute the graphics rendering in parallel to maximize efficiency. Historically, GPUs, however, have not been the only hardware available for parallel problems. Actually, it is only very recently that GPUs started being general purpose programmable: until then we had clusters and "supercomputers". If one peeks at the list of the top 500 supercomputers[2], one can see that supercomputers are massive computers with a huge amount of computing cores. Supercomputers are often very large clusters of mainstream CPUs

[2] `http://www.top500.org/`

"clustered" together with high speed networks. Thus, the distinction between a supercomputer and a mainstream consumer computer is often only limited to the amount of physical machines rather than the type of hardware. There do, however, also exist supercomputers built with "special" hardware designed for parallel processing. Intel's newly released Xeon Phi processor, for instance, is a co-compute unit that is designed for heavy parallel processing, and this is its only purpose, and thus it cannot be used for other general usages.

One of the major drawbacks with the cluster design is that it is in general very difficult to utilize within the parallel programming paradigm – namely synchronization between processes and states among cores/machines in the grid. This is especially cumbersome in clusters since communication through a network introduces higher latencies. These challenges are not limited to clusters since synchronization of cores within a single CPU can be challenging as well and can incur severe performance penalties if done wrong.

From a purely hardware-architectural standpoint, GPUs and CPUs are very different. In general CPUs are still optimized for sequential execution, even though they often have several cores available for execution. A GPU is on the other hand optimized for parallel execution. These major architectural differences manifest themselves in that the GPUs' die areas are primarily spent implementing logical cores (ALUs) whereas modern CPUs' die areas are used primarily for cache, conditional-branch prediction logic, out-of-order execution speculation, and other I/O logic. Paradoxically, today's CPUs often contain a small embedded GPU which, compared to a discreet desktop GPU, is severely limited but has its niche within mobile hardware platforms, e.g., laptops, where power usage is often more important than performance.

Taking a peek at Figure 5, we can see that the theoretical performance in single-precision floating operations has increased greatly in GPUs compared to CPUs. Interestingly, when comparing double-precision performance, we notice that the increased gain is not nearly as great. This is due to the fact that graphics rendering only requires single-precision. Conversely, the logic of CPUs for single-precision and double-precision are equivalent – especially since the introduction of 64-bit CPUs in mainstream PCs. Although modern GPUs offer double-precision, this feature has primarily been driven by recent GPGPU requirements. Among these are pressure from within the financial sector which sometimes need higher precision. Nvidia's mainstream GPUs typically deliver $\frac{1}{24}$ double-precision performance compared to single-precision whereas AMD's GPUs are typically limited to $\frac{1}{8}$ of the performance – this, however, varies among each specific GPU. Both Nvidia and AMD offer non-mainstream GPUs which are not nearly as limited regarding double-precision performance but come with a significant increase in price.

Another major difference between the hardware architectures of GPUs and CPUs is the memory subsystem. Caching, as already mentioned, is quite different among GPUs and CPUs. CPUs dedicate a lot of its die area to caching. For instance, my current desktop CPU has the stats given in Table 1. Here,

| | CPU specifications |
|---|---|
| Brand | Intel |
| Model | i7-3770 |
| Physical cores | 4 |
| Logical cores | 8 (hyper-threading) |
| Clock | 3.4 GHz base (3.9 GHz turbo) |
| Level-1 cache | $4 \cdot 32$ KB = 256 KB |
| Level-2 cache | $4 \cdot 256$ KB = 1,024 KB |
| Level-3 cache | 8,192 KB |
| Memory bandwidth | Dual-channel DDR3-1600 (25.6 GB/s) |
| Theoretical performance | 108.8/54.4 GFLOPS (single-/double-precision) |

Table 1: Specifications of my current desktop processor. Source: Intel.

we can see that on my CPU there is a lot of cache available for the few cores it has. The memory subsystem is a typical DDR3 memory setup, and the chip supports up to dual-channel with an effective frequency of 1600 MHz for a total theoretical memory-transfer capability of 25.6 GB/s.

## 3.5 Cost/benefit

Since the inception of CUDA, and later on OpenCL due to the rising interest in GPGPU, cost/benefit should not be neglected as a driving force behind the recent advancements in GPGPU frameworks such as OpenCL as well as new GPU hardware architectures optimized for not only graphics rendering but also general purpose computing.

If we look at the raw performance numbers between CPUs and GPUs, we see that the GPUs typically outperform the CPUs with at least one order of magnitude. For my personal setup, the numbers are

| | CPU | GPU | $\frac{\text{GPU}}{\text{CPU}}$ |
|---|---|---|---|
| Single-precision | 108.8 GFLOPS | 4,096 GFLOPS | 37.5$x$ |
| Double-precision | 54.4 GFLOPS | 1,024 GFLOPS | 18.8$x$ |

Table 2: Theoretical performance metrics of my CPU compared to my GPU. Sources: AMD, Wikipedia, and Intel.

Hence, the GPU, AMD Radeon HD 7970, has a raw performance advantage compared to the CPU, Intel i7-3770, by an impressive factor of 37.5$x$ for single-precision as seen in Table 2. Likewise, the GPU still has an impressive factor of 18.8$x$ of theoretical performance advantage over the CPU for double-precision – although it is only half as fast compared to its single-precision performance. The GPU's performance difference in double-precision is noteworthy but can be explained by the fact that graphics rendering requires only single-precision. This is, however, not a complete explanation since, typically, Nvidia's mainstream GPUs' double-precision performance are artificially limited to $\frac{1}{24}$ of their single-precision performance, but this is a deliberate strategy to market its higher double-precision performing hardware – the Nvidia Tesla series compute hardware – which is a business-made decision.

But what about performance in monetary terms? Often one can describe the rise and fall of products or trends by their utilities, e.g., the monetary cost savings they deliver compared to competing products. Above we noted that a mainstream GPU is often performing many times better (at least from a theoretical perspective), but what about performance relative to its cost? The most critical metrics are performance/initial price and performance/Watt (energy). In Table 3 we have an overview of the performance delivered compared

| | CPU | GPU | $\frac{\text{GPU}}{\text{CPU}}$ |
|---|---|---|---|
| Initial price | \$294 | \$500 | |
| $\frac{\text{Single-precision}}{\text{initial cost}}$ | 0.37 GFLOPS $\cdot$ \$$^{-1}$ | 8.19 GFLOPS $\cdot$ \$$^{-1}$ | 22.1$x$ |
| $\frac{\text{Double-precision}}{\text{initial cost}}$ | 0.185 GFLOPS $\cdot$ \$$^{-1}$ | 2.048 GFLOPS $\cdot$ \$$^{-1}$ | 11.05$x$ |

Table 3: Cost/benefit calculation of initial price versus theoretical performance. Sources: AMD, Wikipedia, and Intel.

to the initial cost. It is quite obvious that the GPU delivers a much better performance per \$ invested: for double-precision $> 10x$ and for single-precision $> 20x$. Although these numbers are calculated specifically for my hardware, it would in general be safe to assume that they approximately hold for most other combinations of CPU and GPU hardware – perhaps with the caveats of certain Nvidia GPUs.

| | CPU | GPU | $\frac{\text{GPU}}{\text{CPU}}$ |
|---|---|---|---|
| TDP[3] | 77 W | 250 W | |
| $\frac{\text{Single-precision}}{\text{TDP}}$ | 1.41 GFLOPS $\cdot W^{-1}$ | 16.38 GFLOPS $\cdot W^{-1}$ | $11.6x$ |
| $\frac{\text{Double-precision}}{\text{TDP}}$ | 0.7 GFLOPS $\cdot W^{-1}$ | 4.1 GFLOPS $\cdot W^{-1}$ | $5.85x$ |

Table 4: Cost/benefit calculation of energy usage versus theoretical performance. Sources: AMD, Wikipedia, and Intel.

Regarding continuously accruing costs, the most critical metric is performance per power consumption. Not surprisingly, the GPU again leads this race with a comfortable $> 10x$ for single-precision and $> 5x$ for double-precision. These numbers show that, for those applications where one can utilize the hardware fully, a tremendous increase in efficiency can be achieved. These numbers are quite significant since power usage is a continuing cost that will incur proportionally with frequent usage so a potential saving of 5-10$x$ for power consumption is a significant overall cost reduction.

To conclude, one can clearly see that from a purely economical point of view, a GPU can deliver better performance for each \$ invested. Although the above numbers are based on my personal equipment, it is safe to assume that the conclusion will hold for most likely any hardware configuration where the purchasing prices are roughly equivalent (between the CPU and the GPU). One can wonder, however, that if a GPU really is that much more efficient than a comparable CPU, then why would one use a CPU instead of a GPU? The answer to this is two-fold:

- Investment of writing custom code (OpenCL/CUDA) for the particular hardware.
- Far from all algorithms can be efficiently converted to utilize the GPU hardware.

[3]Thermal Design Power.

## 3.6 The AMD Tahiti GPU architecture

In 2012, AMD released a new GPU lineup consisting of the revisions Cape Verde, Pitcairn, and Tahiti – all codenamed southern islands. The overall architecture in the southern islands chips is called Graphics Core Next (GCN). This architecture is a deviation of the earlier Very Long Instruction Word (VLIW) architecture. AMD justifies this architectural shift in that it is much easier to deliver high performance – especially in GPGPU cases. The largest and fastest chip is the Tahiti chip equipped in the Radeon 7950 and 7970 graphics card models. I own an AMD Radeon 7970 GHz edition which is a slightly overclocked stock AMD Radeon 7970 graphics card. To understand the chip architecture,

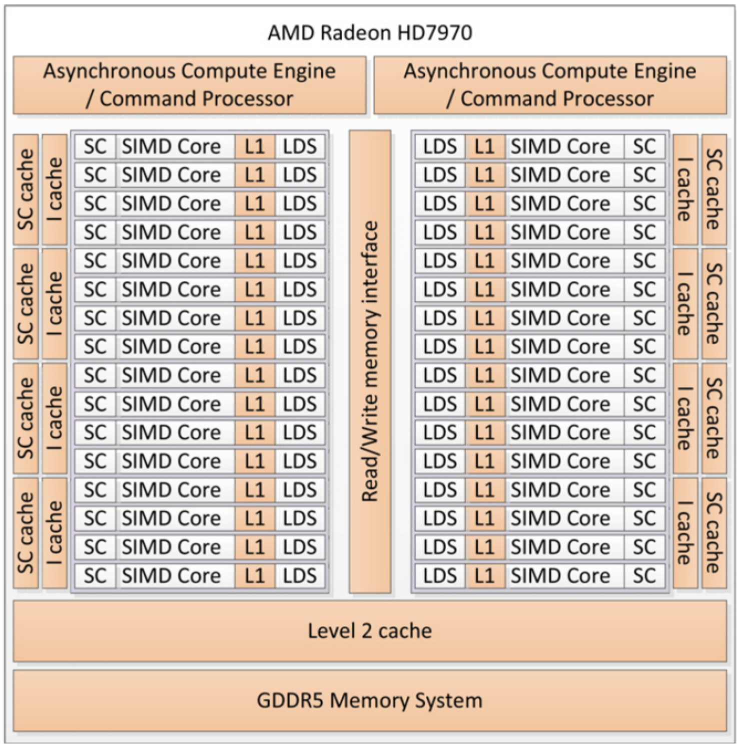
AMD Radeon HD7970
Asynchronous Compute Engine / Command Processor
Asynchronous Compute Engine / Command Processor
SC cache
I cache
SC
SIMD Core
L1
LDS
Read/Write memory interface
Level 2 cache
GDDR5 Memory System

Figure 6: A block diagram of AMD's Tahiti architecture. Source: "*Heterogeneous Computing with OpenCL*" [8].

we need to realize that the architecture compared to a CPU is very different. For instance, the term "Core" can be a bit deceptive when dealing with GPUs. One normally thinks of a core as a logical unit of execution capable of not only basic integer arithmetics and floating point operations but also other logical processing such as decoding instructions and controlling the execution flow – as one normally expects a core implemented in a CPU can do.

In Figure 6, we can see a simplified block diagram of the AMD Radeon 7970 GPU. From the diagram, we can see that the GPU has eight clusters consisting of four Compute Units (CUs) for a total of 32 CUs. Sometimes the term "core" is used instead of Compute Unit, however, they are still not the equivalent to normal CPU cores. Each Compute unit consists of a single-scalar unit (SC), four 16-wide SIMD units, registers, and a small cache. The single-scalar unit handles basic integer operations as well as branching while the SIMD units are where the true computing performance is hiding. Each SIMD unit can be considered an ALU which, instead of processing one element at a time as a typical ALU would do in a CPU, computes a single instruction on 16 elements – hence, it being a SIMD unit. This equals a total of 32 Compute Units · 4 SIMD units · 16-wide ALU = 2,048 single element ALUs or streaming processors as is the most commonly used term.

| GPU specification | |
|---|---|
| Brand | AMD |
| Model | Radeon HD 7970 GHz Edition |
| Physical cores | 32 (compute units) |
| Streaming processors | 2,048 |
| Clock | 1 GHz base (1.05 GHz turbo) |
| Level-1 cache | 16 KB per compute unit = 512 KB |
| Level-2 cache | 768 KB |
| Level-3 cache | 16 KB instruction cache and 32 KB scalar data cache (per compute unit) |
| Memory bandwidth | GDDR5 (288 GB/s) |
| Theoretical performance | 4,096/1,024 GFLOPS (single-/double-precision) |

Table 5: Specifications of my current graphics card. Source: AMD, "*Heterogeneous Computing with OpenCL*" [8].

It should be clear at this point that the GPU architecture of modern GPUs is drastically different than that of mainstream CPUs. The caches' primary purpose is to speed up often accessed data and instructions, but the registers – or local memory (LDS) as per OpenCL convention – are a rather small local memory of 32 KB available on a shared basis among the streaming processors. The local memory is very fast compared to the global memory with a latency equivalent of that of the L1-cache. Often when developing for GPUs, the typical bottleneck is an overgenerous use of this memory which, if memory allocation surpasses the locally available in the GPU, will use the vastly slower global memory. In the AMD Radeon HD 7970, the 32 KB will be shared among all of the streaming processors in each compute unit. This means that there will be a maximum of 32 KB / 64 streaming processors = 512 bytes per streaming processor.

## 3.7 OpenCL in action

### 3.7.1 The OpenCL specification

The OpenCL specification specifies several levels of how an OpenCL implementation should handle a kernel (an OpenCL program) and how the application programming interface (API) should look. The layers are:

1. Platform model: The *host* is defined as the processor coordinating the execution of the kernel on one or more capable processors (OpenCL *devices*: CPUs, GPUs, and other OpenCL-capable hardware). This is the abstract hardware model that the OpenCL application developers use to model and execute kernels.

2. Execution model: This is used for defining the OpenCL environment on the host as well as how kernels are executed on the device(s). In this step, the OpenCL *context* is set up which is necessary for the host-device communication as well defining the concurrency model used when executing the kernel(s).

3. Memory model: The OpenCL memory model defines the abstract memory hierarchy that kernels can use – regardless of the actual underlying memory architecture. The model is heavily inspired by how the typical memory hierarchy of a modern GPU is – although it is still general enough to adapt to other memory layouts of different architectures.

4. Programming model: This is the definition of how the chosen concurrency model actually maps to the physical hardware.

In the most typical case, we have a machine with a CPU, the *host*, executing some application where a part of this is implemented as a OpenCL kernel using the GPU as an accelerator. The platform model is the relationship between the CPU and GPU. The host then creates the kernels with some parameters and instantiates it with a specified degree of parallelism – this is the execution model. The data within the application necessary for the execution of the kernel is allocated within the abstract OpenCL memory hierarchies by the programmer, and the OpenCL runtime and driver for that particular hardware will map the abstract memory hierarchies to real physical memory hierarchies of the device – this is the memory model. Lastly, the hardware threading context that execute the kernel must be instantiated onto the actual GPU hardware – this is the programming model. However, the above is only one example of a typical OpenCL application flow and could deviate in several ways. The CPU, as the host, could also execute OpenCL kernels as well as the dedicated GPU located on most modern CPUs. One could also, for instance, have several GPUs and execute kernels in parallel distributed on all GPUs.

### 3.7.2 The OpenCL execution model

When one wants to execute code within the OpenCL specification, one needs to write a OpenCL kernel. Writing a OpenCL kernel is almost syntactically identical to any other CPU concurrency model, e.g., OpenMP and win32 thread API, in that it is basically a C function (though with some extra keywords). One major difference between OpenCL kernels and other typical CPU concurrency models is that in OpenCL one does not consider the overhead of creating threads and switching between these as well as the overhead of handling the physical resources, e.g., the CPU cores, as is necessary when using standard win32/posix threads, for instance. The benefit of using OpenCL is that the coarseness of typical CPU concurrency is mostly eliminated while still keeping the goal of representing parallelism programmatically at the finest granularity possible. The thread or unit of concurrent computation in OpenCL is called a *work item*.

When an OpenCL compatible device begins executing a kernel, it provides some OpenCL specific functions like `get_global_id()` which returns a number or index to the programmer so that he knows which work item is currently being executed. Since threads or work items are very lightweight in comparison to typical CPU concurrency models, we normally unroll loops and map each iteration to a work item. Consider a two-dimensional problem which in a sequential implementation would require two nested for loops. Now, we instead translate this into creating work items in two dimensions – one for each for loop – and then get the relevant indexes by calling the `get_global_id()` for each dimension. For instance, consider the following sequential pseudo algorithm:

```
int SomeCFunction(int x_length, int* x_arr[],
   int y_length, int* y_arr[]) {
   for(i = 0; i < x_length; i++)
   {
      for(j = 0; j < y_length; j++)
      {
         // algorithm...
      }
   }
}
```

Listing 1: Simple example of an arbitrary algorithm.

which can loosely be translated to the following OpenCL parallel kernel:

```
__kernel void SomeCKernel(int* input_array[],
   int* output_array[]) {
   int i = get_global_id(0);
   int j = get_global_id(1);

   // algorithm...
}
```

Listing 2: Simple illustration of an OpenCL kernel.

First, notice that kernels always return void. Thus, one always has to return a value by writing to an output array. Secondly, notice how we have transformed the two input arrays in the sequential solution to one input array in the OpenCL kernel. This is due to the nature of how a GPU reads from memory and thus determines what the memory layout for optimal performance is. Since a GPU is a SIMD device, it naturally executes one instruction on multiple data and thus it is obviously optimal to have a data layout such that when an ALU executes an instruction, it has all the necessary data available at hand. Otherwise, it will stall until the data arrives. A GPU will read memory in a coalescent fashion. This means that it can read a *wavefront* at a time. A wavefront is AMD's terminology for an execution unit whereas Nvidia uses the term *warp*. The wavefront for the AMD 7970 GPU is a 64-wide execution unit since, as noted earlier, the 7970 GPU's SIMD computation core contains four ALUs (each capable of executing one instruction on 16 data pieces at a time). When accessing and reading a memory location in the GPU's global memory, that location is read in an coalescent fashion, meaning that a multiple of 64 memory addresses are read at a time instead of just one single address at time as a CPU would do. This coalescent memory reading fashion requires special care from the programmer since exceptionally poor memory reading utilization can result in equally bad execution performance since the data is not available when needed, and thus the device is far from being fully utilized.

When a kernel is instantiated and executed, the programmer must specify the number of work items needed and create these as an n-dimensional range (NDRange). A NDRange is a one-to-three-dimensional vector which, as noted above, is the parametrization of the threads within the kernel. Using the `get_global_id()` function, one can, as shown earlier, get the current index of the thread executing the kernel. While one can pick arbitrary values for NDRange, it is advised for optimal performance that special care is given depending on the running hardware. This is because work items are grouped together in *workgroups*. In general though, the OpenCL runtime will, at least for a one-dimensional NDrange, optimize an arbitrary choice to make it optimal for proper performance. However, this does not hold for NDranges with more than one dimension. When executing a kernel, one might need to syn-

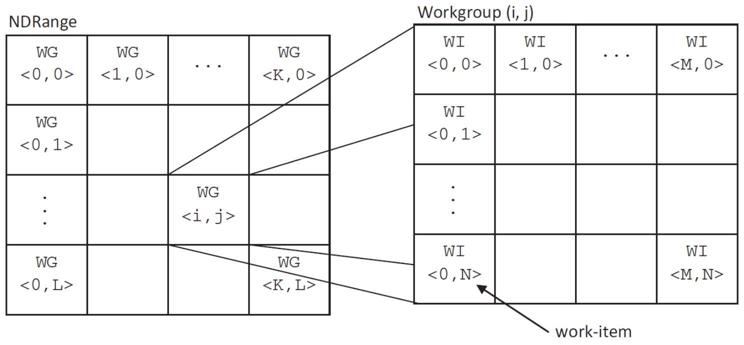

Figure 7: An example of a two-dimensional NDRange of work items in their respective workgroups. Source: "*Heterogeneous Computing with OpenCL*" [8].

chronize the work items, i.e., make sure that the kernel execution follows the algorithm. These synchronization methods are called barrier operations and are only available within workgroups as of OpenCL 1.2 [8]. There exists an upper limit on how large a workgroup can be, which for the AMD 7970 GPU is 256 work items. It is, however, exceedingly difficult to precisely calculate how the NDrange should be parameterized: this depends on each specific kernel for each specific OpenCL capable hardware since hardware, as noted earlier, can be very different. This does not mean that the kernel will deliver poor performance when executing on different hardware; it only means that it most likely will not deliver its maximum theoretically possible performance. For specific hardware implementations, a programmer should always tweak the settings using a profiler for benchmarks to maximize performance. Lastly, as noted earlier, local memory (or registers) is only available on a shared basis among each workgroup. Some kernels might require a large number of local variables while others do not. In the former case one might spill over with register usage which means that global memory will be used instead for variables. Since this memory is orders of magnitude slower, performance will degrade severely.

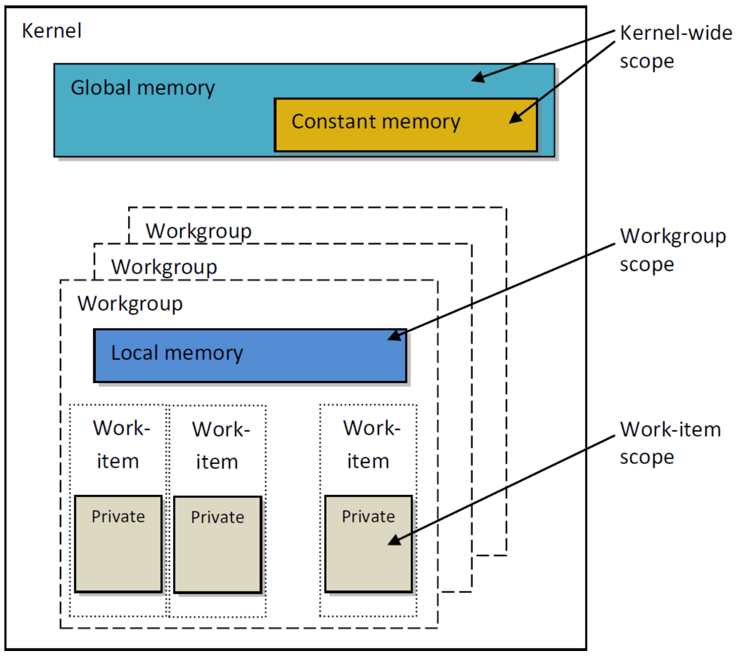

Figure 8: The OpenCL memory hierarchy. Source: "*Heterogeneous Computing with OpenCL*" [8].

## 3.8 The OpenCL memory model

As mentioned earlier, the architectures of OpenCL capable hardware are very different. We have traditional CPUs optimized for sequential performance, FPGA, and GPUs for parallel performance, and yet all of these hardware devices ought to run every OpenCL kernel – save for minor platform/hardware specific tweaks. However, with the abstract memory hierarchy available through the OpenCL API, one can wonder how this maps to the different architectures available. It is heavily inspired – if not identical – to the memory layout of modern GPUs.

The OpenCL memory model distinguishes between four different memory types, which are here itemized from Figure 8:

- Global memory.
- Constant memory.
- Local memory.
- Private memory.

Global memory is to the GPU what the main- or system-memory is for a CPU. This is the memory where 3D meshes and their textures reside when GPUs are rendering graphics, and since this kind of processing requires large amounts of RAM, it is today not unheard of having 2 to 6 GB of dedicated video RAM on a graphics card. The AMD 7970 GHz Edition graphics adapter, that is used for this thesis, is equipped with 3 GB of RAM.

For OpenCL computing purposes on GPUs, the video memory is the equivalent of the global memory. This memory is used to transfer and hold input data as well as output data from and to the host. If executing the kernel on a CPU, the main memory would be the global memory. Global memory is visible from all work units on the computing device. The OpenCL keyword `__global` is used in front of pointers to mark them as global.

The constant memory is, contrary to what the name would indicated, not located in a read-only memory. On GPUs, it is actually typically located within the global memory – whereas on CPUs it can be more complicated where exactly this data will reside. It is designed to hold data that is accessed simultaneously by all of the work units, e.g., the constant $\pi$ would be a very relevant candidate. Using the OpenCL keyword `__constant`, one creates a variable in the constant memory.

The local memory is a scratch pad memory which is dedicated to compute unit on the device. On a GPU such as the AMD 7970, this is, as mentioned earlier, a 32-KB address space that is available for all compute units in a workgroup. This address space is quite commonly dedicated on the chip, as is quite common with GPUs, but there is no absolute requirement for this. The idea with this memory space is that is should have a significantly lower latency and much higher bandwidth than that of the global memory. Using the OpenCL keyword `__local`, one creates a variable in the local memory.

Lastly, we have the private memory which is unique to each work item. Local variables in the kernel as well as kernel arguments are private by default. For most purposes, these variables are mapped to the private memory although, as mentioned earlier, care should be taken since variables might spill over from the small register memory to global memory which is much slower.

## 3.9 Theoretical limitations

To this end, GPUs can deliver an enormous amount of computing power, but there are unfortunately some limitations on utilizing these. When evaluating the performance of a larger software program, it is critical to the overall performance which parts of it that can be parallelized. These concerns have been conceptualized in Amdahl's law [2]:

$$T(n) = T(1)\left(B + \frac{1}{n}(1 - B)\right)$$

Here, $n$ is the number of threads of execution, $T(n)$ is the time of execution, and the $B \in [0, 1]$ is the fraction of the program that is strictly sequential. The law states that the limiting factor of the performance speedup of the program is determined by the amount of code that is strictly sequential. The theoretical speedup according to Amdahl's law can be calculated as follows:

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1)\left(B + \frac{1}{n}(1 - B)\right)} = \frac{1}{B + \frac{1}{n}(1 - B)}$$

Thus, for a program of which 10% is strictly sequentially executed and the remaining 90% parallelizable, with 1000 threads one gets a maximum speedup of

$$\frac{1}{0.1 + \frac{1}{1000}(1 - 0.1)} = 9.91x$$

## 3.10 Summary

In this chapter, the focus has been on the OpenCL parallel computing framework, GPU architecture, GPUs compared to CPUs, motivations for using a GPU, and, lastly, highlighting the execution and memory model in OpenCL. It should be clear at this point that GPUs can deliver tremendous amounts of raw computing power compared to mainstream CPUs – not only in raw numbers – but also relatively compared to electrical power usage or initial purchasing costs. However, these benefits come with the cost of extra programming complexity since one needs to know the underlying hardware for maximum performance.

### 3.10.1 Strengths of Graphics Processors

The main advantages of using GPUs for computational problems are

- Higher theoretical performance available compared to CPUs. Typically GPUs are more than an order of magnitude faster in single-precision performance and around a single order of magnitude faster in double-precision.
- Cheapest available compute performance measured in GFLOPS$\cdot\$^{-1}$ for initial cost.
- Best power efficiency compared to CPUs. It is not uncommon that a GPU can deliver $>$ 5x (GLFOPS$\cdot W^{-1}$) that of a CPU. Hence, both cooling requirements as well as electricity bills are much lower than those needed when using a CPU.

### 3.10.2 Weaknesses of Graphics Processors

Although GPUs can be beneficial regarding cost and compute performance, there are some disadvantages when using GPUs:

- Not all algorithms are suitable for massively parallel hardware (GPUs): GPUs perform abysmally for sequential programs.

- It is necessary to learn OpenCL/CUDA for programming GPUs. This means that there is a steep learning curve for a relative narrow domain of problems, and often you are at the mercy of the quality of the implementation of the OpenCL specification by the hardware manufacturer, e.g., lacking Linux drivers.

- There is a overhead of transferring back and forth between the host and GPU, and for smaller problems, this overhead might make it not worthwhile to execute on a GPU.

- The programmer needs to know the hardware running the kernels since a lot of optimization needs to be done for proper utilization of the GPU. Thus, while OpenCL kernels are intended to be "write once run everywhere", rarely it is the case for maximum performance.

# Part III

# Algorithms, implementations, and results

## 4 Testing methodologies

### 4.1 OpenCL execution measurements

Before we can actually benchmark and compare performance results of CPU vs. GPU executions, we need to step back and get an overview over what we are actually measuring. Given that this thesis is about accelerating option evaluations using a GPU, we need to understand what parts of the algorithm we are actually accelerating, and what overhead using a GPU incurs. Recall that there occur some overheads from using OpenCL in general but even more specifically when utilizing a GPU. Consider Figure 9 where an overview of the steps necessary for executing an OpenCL kernel is presented.
Figure 9 shows that two steps are necessary, but they only occur once. These events comprise the initialization of the OpenCL subsystem such as querying OpenCL for available OpenCL compatible hardware and then creating an OpenCL context as well as an execution queue for the selected hardware. This is necessary every time the application is run on the host, but only once. The next step actually allows for two possible scenarios: either Just-In-Time (JIT) compilation of the source code or using a pre-compiled binary − in which case there are really no overhead at all. However, the last feature has only recently become available, secondly, it is not the recommended way to include OpenCL code due to potential hardware compatibility/optimization issues that the JIT compiler can handle on the local machine. Nevertheless, I have decided that the benchmarks will refrain from including time spend on handling initializations including JIT compiling of the OpenCL source code.

From Figure 9, we can see that the last three steps involve memory transfers back and forth between the host and the GPU as well as the actual execution of the kernel on the GPU. These three steps are absolutely critical in measuring the performance and execution time for benchmarking against a CPU implementation since they are the complete round trip necessary for an OpenCL

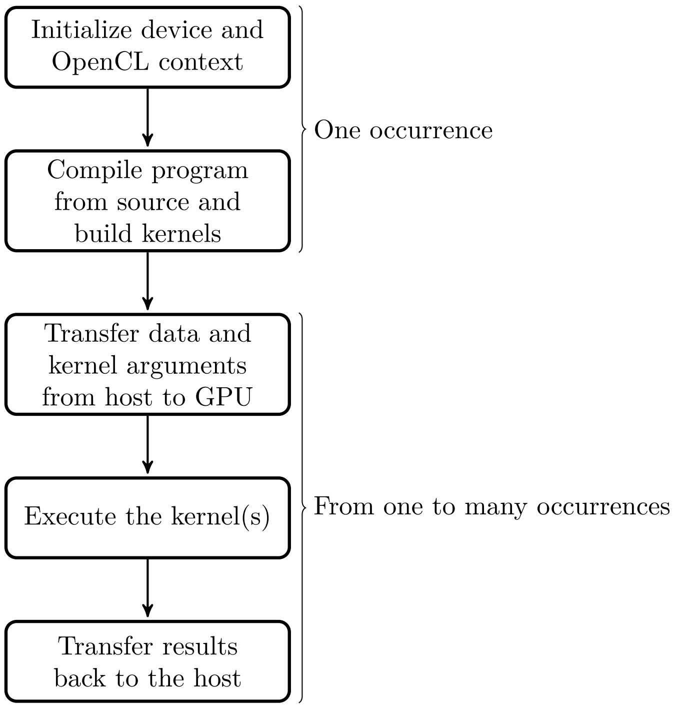
Initialize device and OpenCL context
Compile program from source and build kernels
One occurrence
Transfer data and kernel arguments from host to GPU
Execute the kernel(s)
From one to many occurrences
Transfer results back to the host

Figure 9: Overview of the steps for executing an OpenCL kernel.

execution on a GPU. These three events will be referred to as **Total execution time** ($\text{GPU}_{total}$) in benchmarks. However, I will also include timings for the actual kernel execution time, i.e., the three steps above except the memory transfer from the host to the GPU, and denote this **Kernel execution time** ($\text{GPU}_{kernel}$).

The justification for including the kernel execution time is several-fold. First and foremost, it represents an apple-to-apple comparison to the CPU in pure execution performance. Secondly, it is sometimes the case that the memory transfers only consume an insignificant amount of total computing time which actually is the case in some of the scenarios that are benchmarked in this thesis. Also, there can be some hardware differences that might render the results obsolete, e.g., my PCI Express bus is of version 3.0 which is capable of transferring around 16 GB/s; however, there is a new version 4.0 just around the corner capable of transferring around 32 GB/s which of course will reduce OpenCL memory transfers by half. Consider the following example: we have a case where the initial memory transfer from the host to the GPU takes 500ms, the kernel execution takes 50ms, and the memory transfer from GPU takes 15ms for a total of 550ms + 50ms + 15ms = 615ms. Then, if we upgraded the PCI Express bus to 4.0, we could double the memory transfer and thus get a total of $\frac{550\text{ms}}{2} + 50\text{ms} + \frac{15\text{ms}}{2} = 332.5\text{ms}$ reducing the total execution time by $\frac{332.5\text{ms}}{615\text{ms}} = 45\%$. Lastly, $\text{GPU}_{total}$ will converge to $\text{GPU}_{kernel}$ as we increase the workload enabling us to use $\text{GPU}_{kernel}$ as a metric for upper bound performance.

## 4.2 Benchmark scenarios

When benchmarking software, it is often worthwhile to consider the contexts in which said software will run. In this thesis, we have implemented several Monte Carlo simulations for pricing various options. However, option evaluations can take place in several different scenarios so to take these into account regarding benchmarking, we will enumerate and discuss them and the motivation behind them.

The motivation for this thesis occurred at my old job where I once, at random, was discussing financial matters with a colleague where he mentioned that he implemented a Monte Carlo simulation of an option in Excel that took more than 11 minutes to execute. More than 11 minutes is a long time to get an answer for a quantitative analyst which was his role in the company. Actually, it really was too long time to wait since he completely abandoned the project and focused his attention elsewhere. Clearly, if one could significantly reduce the execution time of the algorithm, the waiting time for an employee as my colleague could be reduced dramatically and actually make him productive in his efforts to solve his real problem. This is therefore the first scenario we will benchmark: a single evaluation of one option.

When studying the literature for this thesis, I noted that a lot of literature was measuring performance in hundreds or thousands of option evaluations rather than a single one. This was the case since the papers were analysing performance from a point of view of a large financial institution. Clearly, people like my colleague as mentioned above could work in a financial institution, but rather than evaluating his need for performance, the authors instead took the entire portfolio of assets of the institution into account for pricing options. Modern financial institutions, e.g., hedge funds, investment banks, and proprietary trading firms, often have very large portfolios consisting of many different kinds of assets: stocks, bonds, and financial derivatives. Due to current accounting rules, daily assessments of these portfolios' values need to be recalculated every day – the mark-to-market principle. This is not only necessary for abiding to national as well as international accounting/financial laws for being solvent, but also internally to verify that the company is not partaking in trades which are too risky. The method currently used – the mark-to-market method – simply dictates that today's new developments in prices should be reflected in booked values. Therefore, e.g., all options need to be recalculated every day. Clearly, this generates a lot of evaluations depending on what kind of assets the institution has booked. This is the second scenario: evaluation of a large number of options.

## 4.3 CPU, GPU, and software setup

While investigating the current scientific literature on GPU acceleration, there were some premises that did not really make sense when benchmarking a CPU vs. GPU. As mentioned earlier on page 27, a modern CPU has several cores. This has been predicted by many observers for some time since increasing single core performance has been difficult given that there seems to be a limit on how high the frequency of a CPU can be clocked. This means that CPUs have instead increased their amounts of cores for multi-threading purposes where true parallel execution can be realised. However, this fact rarely gets mentioned in the research papers so these often do not include the multi-threaded performance of the CPU vs. the GPU. Clearly, this should be unacceptable giving that most – if not all – modern CPUs have more than a single core. My CPU, as mentioned earlier in Table 1, has four physical cores, but features hyper-threading. Hyper-threading means that each physical core can execute two threads meaning that the CPU has eight logical cores.
The benchmarks in this thesis will therefore include not only single core executions for the CPU, but also multi-threaded executions to maximize the performance of the CPU in an effort to make a fair assessment. The multi-threaded version will utilize eight threads for a full 100% utilization of the CPU.

| Software/hardware specifications | |
|---|---|
| OS | Microsoft Windows 8.1 |
| IDE | Visual Studio 2013 |
| C/C++ compiler | Visual C++ 2013 |
| Compiler flags | `/O2 /openmp` |
| OpenCL version | OpenCL 2.0 AMD-APP (1642.5) |
| OpenCL profiling | `CL_PROFILING_COMMAND_START`<br>`CL_PROFILING_COMMAND_END` |
| OpenCL flags | `-cl-unsafe-math-optimizations`<br>`-cl-mad-enable -cl-fast-relaxed-math` |
| AMD driver | AMD Catalyst 14.12 |
| PCI Express version | 3.0 |

## 4.4 Considerations of Monte Carlo algorithm implementations

To evaluate a financial option properly, there are some initial considerations that need to be taken into account. For the Monte Carlo method, in general one needs a relatively high amount of high quality random numbers. These random numbers are critical to the Monte Carlo method's ability to converge to the true answer. When we have the proper amount of random numbers, we need to execute the actual Monte Carlo method and, lastly, aggregate and discount the results of the simulation.

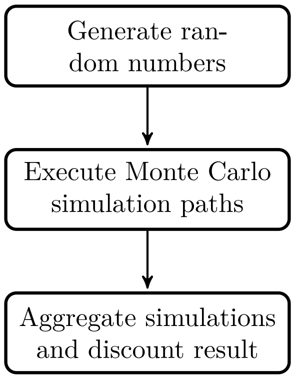
Generate random numbers
Execute Monte Carlo simulation paths
Aggregate simulations and discount result

Figure 10: Overview of the steps necessary for applying the Monte Carlo method to financial option evaluation.

In Figure 10, we can see these steps illustrated. However, while the Monte Carlo method typically calls for random numbers, one can also use quasi-random numbers – or low-discrepancy sequences – such as Sobol or Halton sequences. Taking this into account, I have decided to refrain from including random number generation in timings of execution time. One can argue that random or quasi-random numbers are crucial to the Monte Carlo method, but since we can choose between generating random numbers or using static sequences for the input to the simulations, one can argue how valuable it is to include the random number generation step. Secondly, pseudo-random number generation is a very complex subject on its own: a subject that one easily could write a thesis on is CPU vs. GPU random number generation performance. Complete new algorithms have been developed for utilizing the parallel nature of a GPU, but they are still not to be found in standard libraries. These new parallel pseudo-random number generators, while very fast, show different characteristics than typical pseudo-random generators such as the Mersenne twister. The Mersenne twister has actually been implemented for GPU execution, however, it is a poorly performing algorithm on GPUs due to its very large internal state that takes up more than the locally available memory. Given all this, it is obvious that the OpenCL API does not include a random function. Please see Figure 11 for an overview of these considerations.

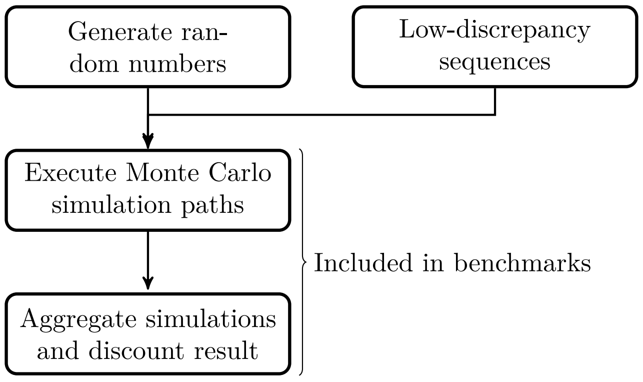
Generate random numbers
Low-discrepancy sequences
Execute Monte Carlo simulation paths
Aggregate simulations and discount result
Included in benchmarks

Figure 11: Overview of the possible steps for applying the Monte Carlo method to financial option evaluation and which parts that are included in the benchmarks.

## 5 Results

Before we review the results for the different algorithms, we first investigate how the algorithm is split into kernels and which parts are executed on the

CPU rather than the GPU.

## 5.1 Reduction sum

Reduction summing is the relatively trivial problem of aggregating some function: in this case addition over some numbers. Recall from Figure 10 that evaluating a financial option superficially consists of two parts:

- Executing Monte Carlo simulations.
- Aggregating simulations and discounting the result.

Hence, before we review the overall results, we need to consider the aggregating part of the algorithm. Recall from Section 3, where we discussed the major differences in CPU versus GPUs, that CPUs are excellent at sequential performance since a lot of research has been done to increase single-thread performance. Consider the following code snippet:

```c
int sum = 0;
for(size_t i = 0; i < array_length; i++) {
   sum += array[i];
}
sum /= array_length;
```

Listing 3: Simple C routine for aggregating an array and computing the mean.

Listing 3 shows how we normally would write a standard routine for aggregating all values in the array and finding the average by dividing by the length of the array. This code will execute very fast on modern CPUs because it is a sequential operation, and since the read from memory is linear, it can pre-fetch values quickly from the main memory and the sum variable will be cached/saved locally on the CPU, i.e., in a register.

Intuitively, one would think that this operation is trivial to parallelize because addition is not only associative, but also commutative. Therefore, if one splits the array in, e.g., 100 buckets, then each thread could aggregate $\frac{1}{100}$ of the total items splitting up the work equally. One thread could then add the last 100 aggregates and subsequently divide by the arrays total size. This, however, is easier said than done using a GPU.

Actually, it is impossible to implement this efficiently using a GPU. The reason is that the GPUs are typically memory bound for most problems and that is definitely also the case for the reduction sum problem. Consider the following example of implementing a naive reduction in an OpenCL kernel:

```
__kernel void ReductionSum(global float* input,
   volatile global float* output) {
   float inc = input[get_global_id(0)]; // get value
   atomic_xchg(&(output[0]), inc); // increment atomically
}
```

Listing 4: Simple naive reduction sum kernel.

In listing 4, each thread loads a value from the input array and adds this value to the global sum "variable". However, the code above performs poorly by only utilizing a single digit percentage of the theoretical performance. Assume the input array has a size of $2^{19}$ =524,288 floats. Then we will launch a kernel with the corresponding 524,288 threads. Each thread will read from the input array (which is fast), but each thread will have to wait in line for incrementing the global sum value in `output[0]`. This is by definition a sequential operation since each thread has to wait in a queue, and thus we get no parallelism. Secondly, the threads are executed in wavefronts of 64 threads in each workgroup so scheduling will limit execution as well since it might reschedule a wavefront that is idling due to all threads waiting in queue for execution. Therefore, thread scheduling will consume a significant time slice as well.

A much better solution that performs acceptably requires that we first and foremost accept the limitations of the GPU hardware. Since the work items on the GPU are organized in workgroups of up to 256 work items, and since there exists local memory available to these work items, it would be better to split the aggregation up in buckets of 256 work items at a time. When a complete workgroup is done, we will write the value to a unique global memory cell that is only available to this workgroup – and thus we limit the pressure on the global memory.

There are several limitations to the optimized version that need to be taken into consideration. The first one is that it actually does not really do a complete sum reduction; instead it sums only in chucks each as large as the work size – which on modern AMD GPUs at maximum can be 256 – and, thus, only reduces the complete sum reduction by a factor of 256. Hence, one has to either run another sum reduction kernel or, if that is not possible, transfer the results back and let the CPU aggregate the rest efficiently making this an algorithm that combines both the usage of the CPU and the GPU. The second limitation is that the global work size of the kernel needs to be a multiple of 256 and thus the input buffer as well since each workgroup will be of a size of 256 work items. However, this limitation could be circumvented by either including some conditionals in the kernel for testing if we are out of bounds or one could make a fixed-size input buffer and pad it with dummy values that then naturally need handling after the kernel has been executed. Lastly, the input buffer size needs to be at least of a *reasonable* size, e.g., using an input buffer size of 256 will only spawn one workgroup on the GPU and thus utilize a maximum of $\frac{1}{32}$ of the GPU's 32 compute units. Either way, for the purposes of this thesis, I have decided not to include these features since all sizes of the inputs to the reduction sum kernel will be divisible by 256.

| Elements | Buffer size | CPU | GPU$_{total}$ | GPU$_{total} > 1x$ |
|---|---|---|---|---|
| $2^{16}$ =65,536 | 0.26 MB | 0.050 ms | 0.150 ms | N/A |
| $2^{21}$ =2,097,152 | 8.39 MB | 1.650 ms | 3.562 ms | 3.476 ms |
| $2^{27}$ =134,217,728 | 536.87 MB | 108.4 ms | 133.4 ms | 130.9 ms |

Table 6: Benchmark results of reduction sum where GPU time is total execution time including aggregating on the CPU.

For benchmarking purposes, I have included the possibility of several kernel executions (GPU$_{total} > 1x$) versus only one kernel execution and then let the CPU aggregate the rest. The idea is to find the sweet spot where we gain most from the GPU and do the rest using the CPU.

In Table 6, we can see the preliminary benchmark results. The CPU execution is faster for any sizes of inputs to the reduction sum algorithm than that of the GPU. This might be surprising, but in reality it is expected since the GPU is at a huge disadvantage due to memory transfers. Recall from the hardware analysis in Section 3 that the GPU has the disadvantage of having to first transfer kernel arguments and data over the PCI Express bus before beginning the kernel execution. However, if we analyze the execution timing data more thoroughly, we can get a deeper overview over the individual steps. Table 7 clearly show the overhead of transferring input data to the GPU from the host.

| Step | Time | Relative time |
|---|---|---|
| Memory transfer to GPU | 113.144 ms | 86.30% |
| Kernel execution #1 | 17.690 ms | 13.51% |
| Kernel execution #2 | 0.025 ms | 0.01% |
| Kernel execution (total) | 17.715 ms | 13.52% |
| Memory transfer to host | 0.003 ms | $< 0.01\%$ |
| CPU execution on host | 0.117 ms | 0.08% |
| $\text{GPU}_{total}$ + CPU | 130.9 ms | 100% |

Table 7: A detailed overview of every step in the execution process of $2^{27}$ elements where the reduction sum kernel is executed twice.

This process consumes ~86.5% of the total execution time. Time spent on actual execution amounts to $17.690\text{ms} + 0.025\text{ms} + 0.117\text{ms} = 17.832\text{ms}$ including the execution spent on the CPU. 17.832ms amounts to a total of ~13.6%: we can clearly see how little time is actually spent on the computations in the $\text{GPU}_{total}$. Recall from Figure 11, that we need to execute the reduction sum kernel as the last step of evaluating an option.

| | | Absolute time in ms | | |
|---|---|---|---|---|
| Elements | Buffer size | CPU | $\text{GPU}_{kernel}$ | $\text{GPU}_{kernel} > 1x$ |
| $2^{16}$ =65,536 | 0.26 MB | 0.054 ms | 0.013 ms | N/A |
| $2^{21}$ =2,097,152 | 8.39 MB | 1.650 ms | 0.369 ms | 0.368 ms |
| $2^{27}$ =134,217,728 | 536.87 MB | 109.1 ms | 20.87 ms | 17.53 ms |
| | | Relative performance (Speedup) | | |
| Elements | Buffer size | CPU | $\text{GPU}_{kernel}$ | $\text{GPU}_{kernel} > 1x$ |
| $2^{16}$ =65,536 | 0.26 MB | $1x$ | $4.15x$ | N/A |
| $2^{21}$ = 2,097,152 | 8.39 MB | $1x$ | $4.47x$ | $4.48x$ |
| $2^{27}$ =134,217,728 | 536.87 MB | $1x$ | $5.23x$ | $6.22x$ |

Table 8: Benchmark results of the reduction sum algorithm where $\text{GPU}_{kernel}$ is the total execution time on the GPU including aggregating the rest on the CPU.

This means that the input data for the reduction is already available on the GPU which implies that the memory transfer overhead from the benchmark results in Table 6 is irrelevant. Let us conduct a new benchmark where we exclude the initial memory transfer from host to the GPU, but include all kernel executions, memory transfer back to the host, and the remaining reduction sum execution conducted by the CPU. The benchmark results from Table 8 clearly shows the performance gain of using the GPU for reduction sum of the output from the Monte Carlo simulations. Also, note that the performance increases proportionally as we increase the amount of elements that need to be aggregated.

## 5.2 Vanilla European call option

```
Algorithm 1: Monte Carlo vanilla European call option.
  Data: expiration time $T$, initial stock price $S$, strike price $K$, risk-free
        rate $r$, volatility $\sigma$, paths (sample size) $M$.
  Result: Call value of an vanilla European option.
  begin
      // Step 1: Initialization
      Sum $\longleftarrow 0$
      Paths $\longleftarrow$ S * Array[M]
      $dt \longleftarrow T$
      // Step 2: Compute paths
      for $i = 1$ to $M$ do
          $\epsilon_i \longleftarrow$ Compute a random N(0,1) sample
          Paths[i] $\longleftarrow S \cdot e^{\left(\mu - \frac{1}{2}\sigma^2\right)dt + \sigma\sqrt{dt}\epsilon_i}$
      // Step 3: Calculate option value
      for $i = 1$ to $M$ do
          Paths[i] $\longleftarrow e^{-rt} \cdot$ Max(Paths[i] $- K, 0)$
          Sum $\longleftarrow$ Sum + Paths[i]
      Sum $\longleftarrow \frac{\text{Sum}}{M}$
      return Sum
```

The vanilla European call option is not a path dependent option like the Asian and barrier options. Instead, it can be solved directly using the analytical solution to the Black-Scholes equation which can be verified in Section 1.2.2. However, it can of course still be evaluated using the Monte Carlo method just like any other option can. In general though, one would not want to use Monte Carlo methods for analytically solvable options like the vanilla European option, but for verification of the correctness of the algorithm I have included it. Also, the barrier and Asian option can be thought of as extensions to the vanilla European option. One major difference between the vanilla European option and the exotics is the lack of time steps required. The exotic options require some kind of time steps since the very evaluations of them depend on the daily movements of the underlying assets.

This is reflected in Algorithm 1 where we can see that it only features one for loop in the second step where it computes the simulation paths. This means that the algorithm is not iterating over the time steps and, therefore, computation-wise makes it much lighter than the exotic options. However, a single time step is still necessary so we simply set $dt = T$, i.e., the expiration of the option is simply the time step. Also note, that in Algorithm 1 we need a random variable, $\epsilon_i$; we do not generate this random variable in either of the GPU or CPU versions of the algorithm. Instead, as discussed earlier, we generate the random values before the execution of the algorithm simply to limit the variability in execution speed of the Monte Carlo part of the algorithm.

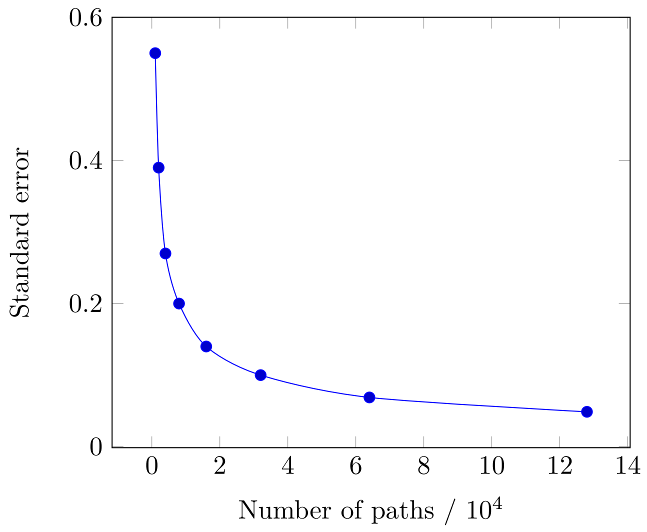

Figure 12: Plot showing the relationship between the number of simulation paths and the Standard Error.

In Figure 12, we see the relationship of the Standard Error of the Monte Carlo simulated values as a function of the number of simulation paths. As we would have expected, we see a nice decrease in the standard error as we increase the number of paths. It can be observed that the Standard Error approximately halves every time we double the amount of simulation paths.

In Figure 13, we clearly see that the Monte Carlo method's expected value converges to the true value as we increase the amount of simulation paths. The plot starts with 1,000 simulation paths and doubles the amount of paths for every point on the plot and ends with 128,000 simulation paths. It is worth noting that we actually sometimes see a worsening in the expected result by doubling the amount of paths. For example, observe that going from 8,000 to 16,000 paths actually makes the Monte Carlo estimate deviate more from the exact result. This is, however, the nature of the Monte Carlo method and is

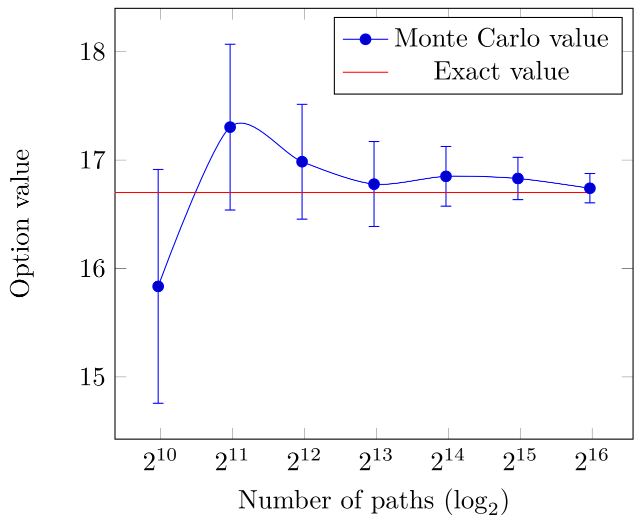

Figure 13: Plot showing that the results of Monte Carlo simulating converge to the true value as we increase the number of simulation paths.

simply due to the fact that we are using too few simulation paths to begin with which makes the standard error too large. In the figure, we can also see the 95% confidence interval brackets, i.e., $\text{MC}_{value} \pm 1.96 \cdot \text{StdErr}$. The confidence interval of course explains why it is possible to estimate a worse result by going from 8,000 to 16,000 paths since both confidence intervals are still including the exact value. Clearly, it can also be observed that a lot of paths is necessary for the estimate being within the exact value. Using, for instance, only 1,000 paths could with a 95% probability yield anything in the interval $[15.83 - 1.96 \cdot \text{StdErr}, 15.83 + 1.96 \cdot \text{StdErr}] = [14.75, 16.91]$ which clearly is too wide a range considering that the true value of the option is 16.69.

For the benchmarks, I have decided to go with $2^{21}$ =2,097,152 paths (unless otherwise stated) since this yields a standard error of 0.012 which generates a much narrower confidence interval and, hence, an estimation much closer to the exact result.
In Table 9, we see an overview of the performance gained utilizing a GPU for varying numbers of simulation paths. For $\text{GPU}_{total}$, we get a speedup of $11x - 23x$ compared to single-thread performance of the CPU depending on the amount of simulation paths we use. However, for the multi-threaded version of the CPU, we get a more limited speedup of $2.5x - 6.9x$ using a GPU for evaluat-

|  | Absolute time in ms |  |  |  |
|---|---|---|---|---|
| Simulation paths | CPU$_{single}$ | CPU$_{multi}$ | GPU$_{total}$ | GPU$_{kernel}$ |
| $2^{16}$ =65,536 | 1.885 ms | 0.424 ms | 0.162 ms | 0.025 ms |
| $2^{21}$ =2,097,152 | 51.31 ms | 11.20 ms | 3.629 ms | 0.727 ms |
| $2^{27}$ =134,217,728 | 1646.8 ms | 492.8 ms[4] | 71.7 3ms | 12.49 ms |
|  | Relative performance (GPU$_{total}$) |  |  |  |
| Simulation paths | CPU$_{single}$ | CPU$_{multi}$ | GPU$_{total}$ | $\frac{\text{GPU}_{total}}{\text{CPU}_{multi}}$ |
| $2^{16}$ =65,536 | $1x$ | $4.45x$ | $11.63x$ | $2.61x$ |
| $2^{21}$ =2,097,152 | $1x$ | $4.58x$ | $14.14x$ | $3.09x$ |
| $2^{27}$ =134,217,728 | $1x$ | $3.34x$[4] | $22.96x$ | $6.87x$ |
|  | Relative performance (GPU$_{kernel}$) |  |  |  |
| Simulation paths | CPU$_{single}$ | CPU$_{multi}$ | GPU$_{kernel}$ | $\frac{\text{GPU}_{kernel}}{\text{CPU}_{multi}}$ |
| $2^{16}$ =65,536 | $1x$ | $4.45x$ | $75.4x$ | $16.96x$ |
| $2^{21}$ =2,097,152 | $1x$ | $4.58x$ | $70.58x$ | $15.41x$ |
| $2^{27}$ =134,217,728 | $1x$ | $3.34x$[4] | $131.85x$ | $39.46x$ |

Table 9: Benchmark results of the reduction sum algorithm where GPU$_{kernel}$ is the total execution time on the GPU including aggregating the rest on the CPU.

ing a single option. In both cases, we clearly see that the performance increases as a function of the number of simulation paths. Just like in the reduction sum case, we have a large overhead of transferring random data to the GPU before execution and this overhead significantly reduces the total performance when using the GPU. If we reviewing the results for GPU$_{kernel}$, we observe a much greater speedup in the area of $70x$-$132x$ versus the single-thread CPU version and $15x$-$40x$ versus the multi-threaded CPU version.

These results clearly suggest that the GPU is excellent for accelerating the computations, but unfortunately is severely limited by the memory transferring speeds of the PCI Express bus. Also, it is worth noting that hyper-threading, that Intel incorporates in some of its processors, are actually not just a marketing fad since the multi-threaded CPU performance is around $4.5x$ that of the single-thread version. Recall from Table 1, that the Intel i7-3770 only has four physical cores: we would expect a speedup of a maximum of $4x$ compared to

[4]For some reason, I was unable to execute this one with more than four threads. It might be an issue with Microsoft's implementation of OpenMP or it might be that Intel's Hyper-Threading is causing troubles. Needless to say, performance suffered greatly due to this error.

the single-thread version. However, hyper-threading actually gives an extra 25% performance boost in these Monte Carlo simulations. To continue the benchmarks, we will also evaluate scenario two where we evaluate many independent options in one go.

| Options | $\text{GPU}_{total}$ | $\text{GPU}_{kernel}$ | $\frac{\text{GPU}_{total}}{\text{options}}$ | $\frac{\text{GPU}_{kernel}}{\text{options}}$ |
|---|---|---|---|---|
| 5 | 7.487 ms | 4.368 ms | 1.497 ms | 0.873 ms |
| 10 | 8.702 ms | 5.386 ms | 0.870 ms | 0.539 ms |
| 20 | 12.38 ms | 9.196 ms | 0.619 ms | 0.459 ms |
| 40 | 16.52 ms | 13.56 ms | 0.413 ms | 0.339 ms |
| 80 | 31.84 ms | 28.76 ms | 0.398 ms | 0.359 ms |
| 160 | 56.39 ms | 53.48 ms | 0.352 ms | 0.334 ms |
| 320 | 86.47 ms | 83.55 ms | 0.270 ms | 0.261 ms |

Table 10: Benchmark results of evaluating more than one option. The number of simulation paths is fixed at $2^{21}$ =2,097,152.

In OpenCL terms, we simply enqueue the kernels necessary for execution and then execute them all one at a time. In Table 10, we can observe the benchmark results. Notice how increasing the amount of options decreases the amount of time necessary per option evaluation. This is expected as we reuse the random data initially transferred from the host to the GPU so that this initial overhead gets averaged out by the amount of options being evaluated. Secondly, it is worth noticing that the difference in the evolution of times per option ($\frac{\text{GPU}_{total}}{\text{options}}$ vs. $\frac{\text{GPU}_{kernel}}{\text{options}}$) are quite different, but this is also to be expected since the $\frac{\text{GPU}_{total}}{\text{options}}$ includes the penalty of the initial random data transfer. Another observation worth noticing is that the more options we evaluate, the faster the evaluation gets per option so there does not seem to be a plateau. However, at 320 options with 2,097,152 paths and four bytes per path (float), we generate $2{,}097{,}152 \cdot 4 \cdot 320 = 2.7$ GB of simulation output that needs to be aggregated by the reduction sum kernel. Increasing the amount of options from 320 to say 640 would again clearly double the output amount to $2 \cdot 2.7 = 5.4$ GB, and since my GPU is only equipped with 3 GB of RAM, 320 options is the limit of what we can execute at maximum simultaneously. It is possible to execute even more, but then it would simply be a multiple of the numbers in Table 10.

In Table 11, we have an overview of the relative performances we get by increasing the amount of options that need to be evaluated. The results are based on those from Table 10 and Table 9. It should by now be clear that the amount of options evaluated has a large impact on what performance one can expect. The

| Options | $\frac{\text{CPU}_{single}}{\text{GPU}_{total}}$ | $\frac{\text{CPU}_{multi}}{\text{GPU}_{total}}$ | $\frac{\text{CPU}_{single}}{\text{GPU}_{kernel}}$ | $\frac{\text{CPU}_{multi}}{\text{GPU}_{kernel}}$ |
|---|---|---|---|---|
| 5 | $35.28x$ | $7.48x$ | $58.77x$ | $12.83x$ |
| 10 | $58.98x$ | $12.87x$ | $95.19x$ | $20.78x$ |
| 20 | $82.89x$ | $18.09x$ | $111.79x$ | $24.40x$ |
| 40 | $124.24x$ | $27.12x$ | $151.36x$ | $33.04x$ |
| 80 | $128.92x$ | $28.14x$ | $142.93x$ | $31.20x$ |
| 160 | $145.77x$ | $31.82x$ | $153.62x$ | $33.53x$ |
| 320 | $190.03x$ | $41.48x$ | $196.59x$ | $42.91x$ |

Table 11: Benchmark results of evaluating more than one option. The number of simulation paths is fixed at $2^{21} = 2,097,152$.

GPU$_{total}$ versus the CPU for single-thread performance ranges from $35x$-$190x$ in speedup going from 5 to 320 options, and likewise for CPU multi-threaded performance, we also see a very large range from $7.5x$-$41.5x$ speedup. What might be difficult to observe from Table 11 is that there seems to be a plateau at 40-160 options where no significant speedup is observed – or at least not the same speedup increase as in the other option ranges. Secondly, performance actually decreases when going from 40 to 80 options in the GPU$_{kernel}$ case. I have not been able to explain this behavior other than it might have something to do with the scheduling mechanism on the GPU. The phenomena is easier to see when looking at Figure 14. From the figure, we can also easily see how GPU$_{total}$ and GPU$_{kernel}$ converge as a function of amount of options that need to be evaluated.

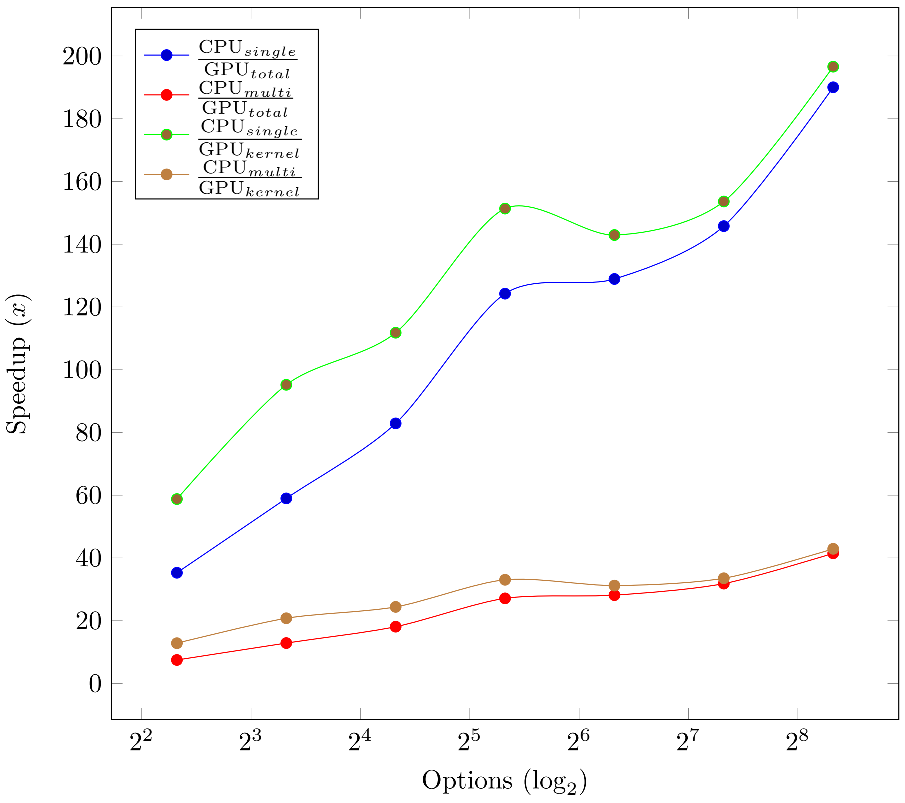

Figure 14: Plot showing the relative performances of GPU$_{total}$, GPU$_{kernel}$ versus the CPU performance for single-thread and multi-threaded.

## 5.3 Arithmetic Asian option

The arithmetic Asian option is a path dependent option meaning that no known analytical solution exists, and, thus, the Monte Carlo method should be applied. The primary difference between the Vanilla European option and the arithmetic Asian option is that we increase the number of time steps from a single timestep to many timesteps. This is necessary since the Asian option uses the arithmetic average of every value generated between the time of writing the option and its expiration date.

**Algorithm 2:** Monte Carlo arithmetic Asian call option.

```
Data: expiration time $T$, initial stock price $S$, strike price $K$, risk-free
      rate $r$, volatility $\sigma$, paths (sample size) $M$, time steps $N$.
Result: Call value of an arithmetic Asian option.
begin
    // Step 1: Initialization
    Sum $\longleftarrow 0$
    Incs $\longleftarrow$ S * Array[M]
    Paths $\longleftarrow$ S * Array[M]
    $dt \longleftarrow \frac{T}{N}$
    // Step 2: Compute paths
    for $j = 1$ to $N$ do
        for $i = 1$ to $M$ do
            $\epsilon_i \longleftarrow$ Compute a random N(0,1) sample
            Incs[i] $\longleftarrow$ Incs[i] $\cdot e^{\left(\mu - \frac{1}{2}\sigma^2\right)dt + \sigma\sqrt{dt}\epsilon_i}$
            Paths[i] $\longleftarrow$ Paths[i] $+$ Incs[i]
    // Step 3: Calculate option value
    for $i = 1$ to $M$ do
        Paths[i] $\longleftarrow \frac{\texttt{Paths[i]}}{N}$
        Paths[i] $\longleftarrow e^{-rt} \cdot$ Max(Paths[i] $- K, 0)$
        Sum $\longleftarrow$ Sum $+$ Paths[i]
    Sum $\longleftarrow \frac{\texttt{Sum}}{M}$
    return Sum
```

This can be observed by noticing the differences between Algorithm 2 and Algorithm 1. Notice that in Algorithm 2 we have an extra argument, the number of time steps $N$, which is used in the new for loop in step 2 as we discussed above. How one chooses to implement the time steps can vary depending on what kind of asset we are modelling. For the purposes of this thesis, I have implemented the time step mechanism such that there exists one time step for each working day in a given period. Other possibilities could be to count every day as a time step in any given time period.

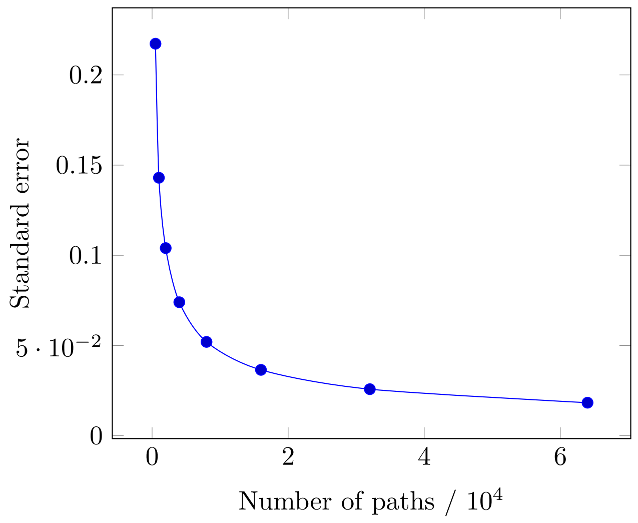

Figure 15: Plot showing the relationship between the number of simulation paths and the standard error.

In Figure 15, we can see the relationship between the number of paths and the corresponding standard error of the estimate. It follows that we have the same asymptotic relationship for the standard error as in Figure 12, although the absolute standard errors are somewhat smaller than that of the vanilla European option. However this can easily be explained with the averaging nature of the Asian option's time steps, which *smooths* each pricing simulation path.
In Figure 16, we again see a plot of the exact value of the option as well as the Monte Carlo estimated value as functions of the number of paths. Again, the data points are supplied with the 95% confidence interval, i.e., $\text{MC}_{value} \pm 1.96 \cdot \text{StdErr}$, which again clearly shows that the standard error is declining as we increase the amount of simulation paths – as in Figure 15. What is noteworthy again is that the convergence is faster for the arithmetic Asian option than for the vanilla European option which we already discussed in relation to Figure 15. However, this allows us to change the amount of paths necessary for getting the same precision as in the vanilla European option – i.e., we can use fewer simulation paths to generate the same relative 95% confidence interval. To get the same relative 95% confidence as in the vanilla European option, it is only necessary to have $2^{17}$ =131,072 simulation paths. Thus, unless otherwise stated,

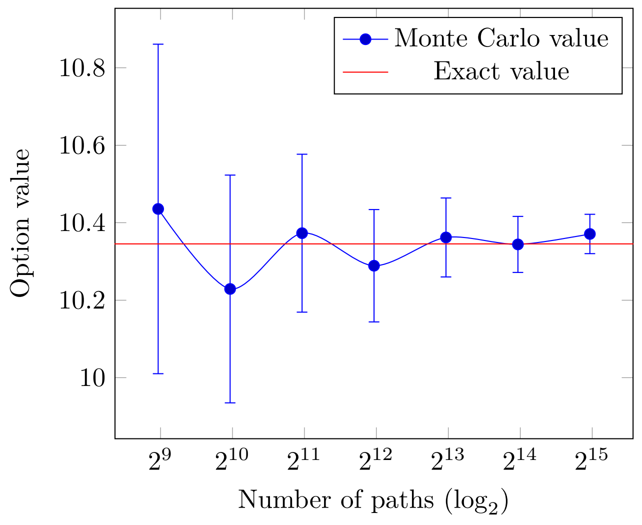

Figure 16: Plot showing that the results of the Monte Carlo simulation processes converge to the true value as we increase the number of simulation paths.

131,072 will be the default amount of paths used for the arithmetic Asian option for benchmarking purposes.
In Table 12, we see a performance overview of using varying amounts of time steps. Recall that the amount of time steps should reflect the expiration time in that 20-22 time steps would be the equivalent of an expiration of one month, and 40-44 time steps would be the equivalent of two months and so on. Clearly, one would expect that an increase in time steps would result in a slower performance vis-a-vis dealing with fewer time steps since more simulations need to be done. This adds another dimension to the performance evaluation of the arithmetic Asian option than for, e.g., the vanilla European option.

The implementation strategy for the GPU version is that we spawn a work item for each simulation path $M$ and then iterate through a small for loop over the $N$ time steps. This means that we have eliminated one of the for loops in Algorithm 2. Another strategy, more like the one used for vanilla European options, could be to use one work item for each path $M$ and time step $N$ and, thus, unwinding both for loops in step 2 of Algorithm 2. However, it turns out that the former solution in general is faster that the latter; therefore I have chosen the latter for further benchmarks. The primary reason that this one is faster than the latter has to do with memory optimizations.

| | Absolute time in ms | | | |
|---|---|---|---|---|
| Paths $M$; Steps $N$ | CPU$_{single}$ | CPU$_{multi}$ | GPU$_{total}$ | GPU$_{kernel}$ |
| $2^{17}$ =131,072; 10 | 26.252 ms | 4.623 ms | 2.079 ms | 0.072 ms |
| $2^{17}$ =131,072; 40 | 112.11 ms | 20.566 ms | 6.610 ms | 0.352 ms |
| $2^{17}$ =131,072; 160 | 589.64 ms | 101.67 ms | 19.824 ms | 1.823 ms |
| | Relative performance (GPU$_{total}$) | | | |
| Paths $M$; Steps $N$ | CPU$_{single}$ | CPU$_{multi}$ | GPU$_{total}$ | $\frac{\text{GPU}_{total}}{\text{CPU}_{multi}}$ |
| $2^{17}$ =131,072; 10 | $1x$ | $5.68x$ | $12.63x$ | $2.22x$ |
| $2^{17}$ =131,072; 40 | $1x$ | $5.45x$ | $16.96x$ | $3.11x$ |
| $2^{17}$ =131,072; 160 | $1x$ | $5.79x$ | $29.74x$ | $5.13x$ |
| | Relative performance (GPU$_{kernel}$) | | | |
| Paths $M$; Steps $N$ | CPU$_{single}$ | CPU$_{multi}$ | GPU$_{kernel}$ | $\frac{\text{GPU}_{kernel}}{\text{CPU}_{multi}}$ |
| $2^{17}$ =131,072; 10 | $1x$ | $5.68x$ | $364.61x$ | $64.21x$ |
| $2^{17}$ =131,072; 40 | $1x$ | $5.45x$ | $318.49x$ | $58.43x$ |
| $2^{17}$ =131,072; 160 | $1x$ | $5.79x$ | $323.44x$ | $55.77x$ |

Table 12: Benchmark results of the reduction sum algorithm where GPU$_{kernel}$ is the total execution time on the GPU including aggregating the rest on the CPU.

Assume we are using $2^{17}$ =131,072 paths and 40 time steps. Evaluating one option, we would need 131,072·40 =5,248,880 random input values, but would only generate $\frac{5{,}248{,}880}{40}$ =131,072 output values (before reduction sum) due to the averaging of the 40 time steps. This yields a total of $5{,}248{,}880 + 131{,}072 = 5{,}379{,}952$ read/writes to the GPU's main memory. This means that the write requirements to the GPU's main memory generally is much lower, inversely proportional to the time step factor, than that for, e.g., the vanilla European option.

Consider the vanilla European option with $2^{21} = 2{,}097{,}152$ simulation paths. The Algorithm 1 maps exactly one input to one output and, hence, we get $2{,}097{,}152 \cdot 2 = 4{,}194{,}304$ read/writes to main memory. Thus, the arithmetic Asian option with 40 time steps only uses $\frac{5{,}379{,}952}{4{,}194{,}304} - 1 = 28,3\%$ more memory read/writes than the vanilla European option – however, the performance of the arithmetic Asian option is still more than twice that of the European option $\left(\frac{0.727\text{ms}}{0.352\text{ms}} = 207\%\right)$ for the GPU$_{kernel}$ running times. There are, however, two explanations for this result: the first one is that we have fewer work items for the arithmetic Asian option, but each work item does considerably more

| Options | $\text{GPU}_{total}$ | $\text{GPU}_{kernel}$ | $\frac{\text{GPU}_{total}}{\text{options}}$ | $\frac{\text{GPU}_{kernel}}{\text{options}}$ |
|---|---|---|---|---|
| 2 | 6.645 ms | 0.680 ms | 3.323 ms | 0.340 ms |
| 4 | 7.353 ms | 1.601 ms | 1.838 ms | 0.400 ms |
| 8 | 9.093 ms | 2.984 ms | 1.137 ms | 0.373 ms |
| 16 | 11.51 ms | 5.63 ms | 0.719 ms | 0.352 ms |
| 32 | 17.30 ms | 10.94 ms | 0.541 ms | 0.342 ms |
| 64 | 27.91 ms | 21.87 ms | 0.436 ms | 0.342 ms |
| 128 | 49.60 ms | 43.662 ms | 0.387 ms | 0.341 ms |

Table 13: Benchmark results of evaluating more than one option. The number of simulation paths is fixed at $2^{17} = 131,072$, and the number of time steps is fixed at 40.

work per work item than those for the vanilla European option, and, thus, the scheduler on the GPU can more easily hide memory reading latencies. Secondly, we know from Section 5.1 that the reduction sum kernel is rather expensive so the smaller the inputs, the faster it runs. The arithmetic Asian option outputs only 131,072 values for the reduction sum kernel meaning that the reduction sum kernel runs with a significant lower amount of inputs $\frac{2{,}097{,}152}{131{,}072} = 16$ than that of the vanilla European option.

| Options | $\frac{\text{CPU}_{single}}{\text{GPU}_{total}}$ | $\frac{\text{CPU}_{multi}}{\text{GPU}_{total}}$ | $\frac{\text{CPU}_{single}}{\text{GPU}_{kernel}}$ | $\frac{\text{CPU}_{multi}}{\text{GPU}_{kernel}}$ |
|---|---|---|---|---|
| 2 | $33.74x$ | $6.19x$ | $329.74x$ | $60.49x$ |
| 4 | $61.00x$ | $11.19x$ | $280.28x$ | $51.42x$ |
| 8 | $98.60x$ | $18.09x$ | $300.56x$ | $55.14x$ |
| 16 | $155.92x$ | $28.60x$ | $318.49x$ | $58.43x$ |
| 32 | $207.23x$ | $38.01x$ | $327.81x$ | $60.13x$ |
| 64 | $257.13x$ | $47.17x$ | $327.81x$ | $60.13x$ |
| 128 | $289.69x$ | $53.14x$ | $328.77x$ | $60.31x$ |

Table 14: Benchmark results of evaluating more than one option. The number of simulation paths is fixed at $2^{17} = 131,072$, and the number of time steps is fixed at 40.

It is also noteworthy that the multi-threaded version ($\text{CPU}_{multi}$) is approximately 5.5 times faster than the single-thread version ($\text{CPU}_{single}$): Intel's Hyper-Threading is generating a $\frac{5.5}{4} - 1 = 37.5\%$ performance boost. In Table 13 and Table 14, we have an overview of the benchmark results of running multiple consecutive option evaluations. Notice that $\frac{\text{CPU}_{multi}}{\text{GPU}_{total}}$ and $\frac{\text{CPU}_{multi}}{\text{GPU}_{kernel}}$ converge as we would expect due to averaging of the initial memory transfer with the total amount options evaluated. For the $\frac{\text{CPU}_{single}}{\text{GPU}_{kernel}}$ and $\frac{\text{CPU}_{multi}}{\text{GPU}_{kernel}}$, the performance is relatively stable independently of the amount of options being evaluated although we do observe a minor *speed bump* around evaluating 4 to 16 options, and I have not found a reasonable explanation other than it might be due to some scheduling or the reduction sum kernel. In Figure 17, we see the results from Table 14 and can clearly see visualized that the $\text{GPU}_{total}$ converges to $\text{GPU}_{kernel}$ as we increase the total amount of options being evaluated.

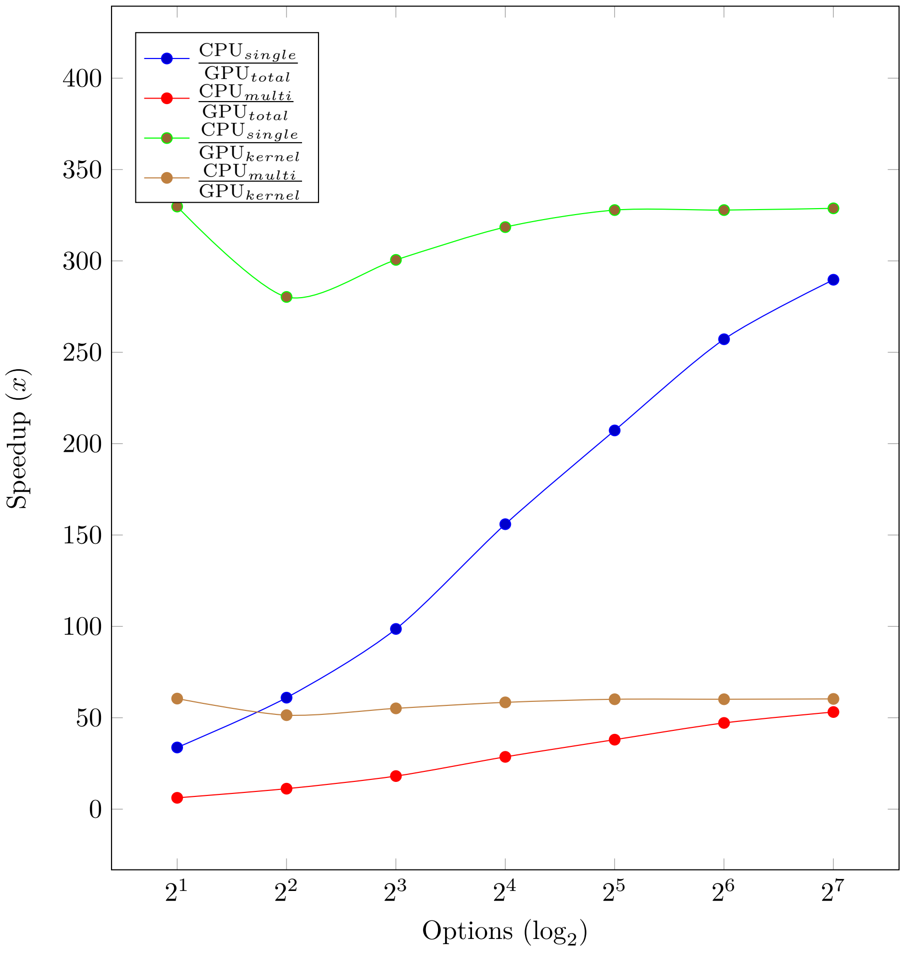

Figure 17: Plot showing the relative performances of $\text{GPU}_{total}$, $\text{GPU}_{kernel}$ versus the CPU performances for single-thread and multi-threaded.

## 5.4 Monte Carlo barrier Up-and-Out put option

```
Algorithm 3: Monte Carlo barrier Up-and-Out put option.
  Data: expiration time $T$, initial stock price $S$, strike price $K$, risk-free
        rate $r$, volatility $\sigma$, paths (sample size) $M$, time steps $N$, barrier
        $B$.
  Result: Put value of a barrier Up-and-Out option.
  begin
      // Step 1: Initialization
      Sum $\longleftarrow 0$
      Paths $\longleftarrow$ S * Array[M]
      BarriersBroken $\longleftarrow$ 0 * Array[M]
      $dt \longleftarrow \frac{T}{N}$
      // Step 2: Compute paths
      for $j = 1$ to $N$ do
          for $i = 1$ to $M$ do
              if BarriersBroken[i] $!= 1$ then
                  $\epsilon_i \longleftarrow$ Compute a random N(0,1) sample
                  Paths[i] $\longleftarrow$ Paths[i] $\cdot e^{\left(r-\frac{1}{2}\sigma^2\right)dt+\sigma\sqrt{dt}\epsilon_i}$
                  if Paths[i] $\geq B$ then
                      BarriersBroken[i] $\longleftarrow 1$
                      Paths[i] $\longleftarrow 0$
                      break
      // Step 3: Calculate option value
      for $i = 1$ to $M$ do
          if BarriersBroken[i] $= 0$ then
              Paths[i] $\longleftarrow e^{-rt} \cdot$ Max(K - Paths[i], 0)
              Sum $\longleftarrow$ Sum + Paths[i]
      Sum $\longleftarrow \frac{\text{Sum}}{M}$
      return Sum
```

The barrier option looks on the surface a lot like the arithmetic Asian option, however, it deviates on two noteworthy points: first, the barrier option has another parameter used for the barrier, i.e., the value for which the option value (in this case) is *out*. Secondly, we do not average any of the values corresponding to the time steps. With the introduction of the barrier, we also introduce necessary conditional if statements that can be a challenge for the GPU.

Regarding the precision of the algorithm, a few aspects need to be considered.

With the introduction of the barrier, we get a challenge if we pick a barrier very close to the current spot price. In Figure 18, we can clearly see that a barrier very close to the spot price reduces the standard error significantly, especially with a lower amount of simulation paths.

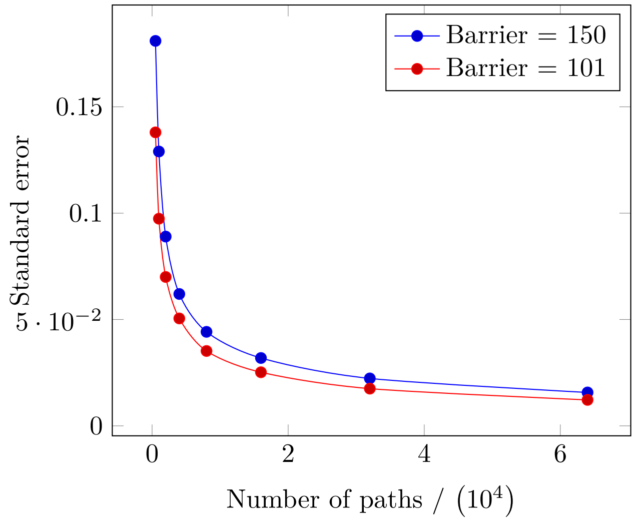

Figure 18: Plot showing the relationship between the number of simulation paths and the standard error.

Since the complexity of the standard error is the same as in all Monte Carlo simulations, we would expect that the errors converge as we increase the amount of simulation paths – which we clearly can see in Figure 18. The reason for the standard error difference in relation to the barrier level is that if the barrier is very close to the spot price, then close to half of all simulation paths will reach the barrier and, thus, become void. If all upwards movement in the underlying price is permitted due to the narrow barrier, we reduce the possible variance and, ultimately, the standard error as well.

In Figure 19, we can easily see the narrowing of the 95% confidence interval as we increase the amount of simulations paths and also that the Monte Carlo simulated value converges to the exact value. As in Figure 18, we could also have made a plot of the convergence with the 95% confidence interval for the same parameters except the barrier set at 101. This would yield the same convergence, but with a narrower 95% confidence interval.
For the purposes of benchmarking, the same amount of simulation paths, $2^{17} =$

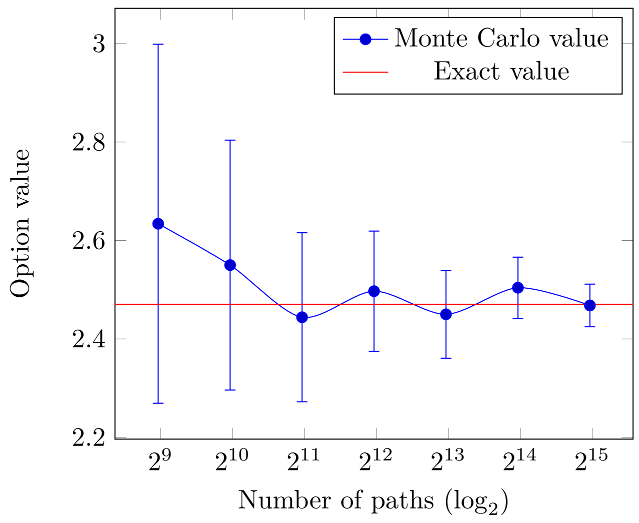

Figure 19: Plot showing that the results of the Monte Carlo simulation processes converge to the true value as we increase the number of simulation paths. In this case the barrier is 150.

$131,072$ as in the arithmetic Asian option case, is used since this amount of simulations paths yield approximately the same level of confidence. However, the benchmarks this time reflect a change in the barrier level instead of the amount of time steps as in the arithmetic Asian option benchmark. As is expected, as we increase the barrier level away from the spot price, the absolute time to evaluate the option increases. This is related to the same argument as in the case for the standard error in that increasing the barrier from the spot price decreases the amount of simulations reaching the barrier, and therefore more calculations are needed.

The results of the benchmarks are somewhat discouraging as can be seen in Figure 15. We can see that the results are much more modest than those of the arithmetic Asian option. For GPU$_{total}$, the increase in absolute running time is purely related to the increase of the GPU$_{kernel}$ execution time since we do not change the amount of data transferred from the host to the GPU. However, GPU$_{total}$ ranges between $6.5x - 15.5x$ versus the single-thread CPU version and only $1.26x - 3.03x$ versus the multi-threaded CPU version. In the GPU$_{kernel}$ case, we obviously see somewhat faster performance than in the GPU$_{total}$ case due to the exclusion of the initial memory transfer. However, it is quite discouraging that the relative performance is decreasing as we increase the barrier level.

| | Absolute time in ms | | | |
|---|---|---|---|---|
| Paths $M$; Barrier $B$ | CPU$_{single}$ | CPU$_{multi}$ | GPU$_{total}$ | GPU$_{kernel}$ |
| $2^{17} = 131,072$; 101 | 41.218 ms | 7.908 ms | 6.266 ms | 0.589 ms |
| $2^{17} = 131,072$; 104 | 77.595 ms | 16.042 ms | 7.599 ms | 1.311 ms |
| $2^{17} = 131,072$; 150 | 128.636 ms | 25.978 ms | 8.255 ms | 2.231 ms |
| | Relative performance (GPU$_{total}$) | | | |
| Paths $M$; Barrier $B$ | CPU$_{single}$ | CPU$_{multi}$ | GPU$_{total}$ | $\frac{\text{GPU}_{total}}{\text{CPU}_{multi}}$ |
| $2^{17} = 131,072$; 101 | $1x$ | $5.21x$ | $6.58x$ | $1.26x$ |
| $2^{17} = 131,072$; 110 | $1x$ | $4.84x$ | $10.26x$ | $2.14x$ |
| $2^{17} = 131,072$; 150 | $1x$ | $4.95x$ | $15.49x$ | $3.03x$ |
| | Relative performance (GPU$_{kernel}$) | | | |
| Paths $M$; Barrier $B$ | CPU$_{single}$ | CPU$_{multi}$ | GPU$_{kernel}$ | $\frac{\text{GPU}_{kernel}}{\text{CPU}_{multi}}$ |
| $2^{17} = 131,072$; 101 | $1x$ | $5.21x$ | $69.98x$ | $13.43x$ |
| $2^{17} = 131,072$; 110 | $1x$ | $4.84x$ | $59.49x$ | $12.41x$ |
| $2^{17} = 131,072$; 150 | $1x$ | $4.95x$ | $57.33x$ | $11.21x$ |

Table 15: Benchmark results of the Monte Carlo simulated barrier algorithm where GPU$_{kernel}$ is the total execution time of the GPU including aggregating the rest on the CPU.

These poor performance metrics should not be surprising at all. The OpenCL kernel implementation of the algorithm is implemented as a scalar kernel rather than a vector kernel as in the arithmetic Asian option. The reason for using a scalar kernel is out of necessity rather than a preference. In the barrier option case, we have conditionals, and for each conditional we change the program flow and terminate the loop skipping the rest of the time steps – i.e., we hit the barrier. However, if we used a vector kernel, it would not be possible since vectorized versions of booleans do not exist in OpenCL. Thus, we get hit on the performance from two sides: first, the expensive conditionals that in this case cause thread divergence when *a lot* of threads hit the barrier. If we set the barrier high enough so that only few work items hit the barrier, thread divergence is only a limited issue. Secondly, using a scalar kernel severely limits the usage of SIMD cores on the GPU device which limits the performance gains one could expect.

In Table 16 and Table 17, we see the benchmark results of evaluating more

| Options | $\text{GPU}_{total}$ | $\text{GPU}_{kernel}$ | $\frac{\text{GPU}_{total}}{\text{options}}$ | $\frac{\text{GPU}_{kernel}}{\text{options}}$ |
|---|---|---|---|---|
| 2 | 10.467 ms | 4.394 ms | 5.234 ms | 2.197 ms |
| 4 | 14.926 ms | 8.974 ms | 3.732 ms | 2.244 ms |
| 8 | 23.772 ms | 17.820 ms | 2.972 ms | 2.228 ms |
| 16 | 41.519 ms | 35.567 ms | 2.595 ms | 2.223 ms |
| 32 | 77.282 ms | 71.330 ms | 2.415 ms | 2.229 ms |
| 64 | 147.363 ms | 141.411 ms | 2.303 ms | 2.210 ms |
| 128 | 289.777 ms | 283.825 ms | 2.264 ms | 2.217 ms |

Table 16: Benchmark results of evaluating more than one option. The number of simulation paths is fixed at $2^{17} = 131,072$, and the number of time steps is fixed at 40.

than one option at a time. The results reflect the discussion above in that performance metrics are not that great compared to those of the arithmetic Asian option as well as those of the vanilla European option. We still get close to or above one order of magnitude for the $\frac{\text{CPU}_{multi}}{\text{GPU}_{total}}$ so for the second benchmark scenario it would still generate a nice speedup by using a GPU rather that a CPU. Note also that since the barrier is fixed at 150, we do not see any noticeable performance increments or decrements of the $\text{GPU}_{kernel}$ as we increase the amount of options being evaluated – just like the arithmetic Asian option. We can see that $\frac{\text{CPU}_{multi}}{\text{GPU}_{kernel}}$ sets the upper bound at around $11.5x - 12x$.

| Options | $\frac{\text{CPU}_{single}}{\text{GPU}_{total}}$ | $\frac{\text{CPU}_{multi}}{\text{GPU}_{total}}$ | $\frac{\text{CPU}_{single}}{\text{GPU}_{kernel}}$ | $\frac{\text{CPU}_{multi}}{\text{GPU}_{kernel}}$ |
|---|---|---|---|---|
| 2 | $24.58x$ | $4.96x$ | $58.55x$ | $11.82x$ |
| 4 | $34.47x$ | $6.96x$ | $57.32x$ | $11.58x$ |
| 8 | $43.28x$ | $8.74x$ | $57.74x$ | $11.66x$ |
| 16 | $49.57x$ | $10.01x$ | $57.87x$ | $11.69x$ |
| 32 | $53.27x$ | $10.76x$ | $57.71x$ | $11.66x$ |
| 64 | $55.86x$ | $11.28x$ | $58.21x$ | $11.75x$ |
| 128 | $56.82x$ | $11.47x$ | $58.02x$ | $11.72x$ |

Table 17: Benchmark results of evaluating more than one option. The numbers of simulations paths and time steps are fixed at $2^{17} = 131,072$ respectively 40.

The convergence of $\frac{\text{CPU}_{multi}}{\text{GPU}_{total}}$ and $\frac{\text{CPU}_{multi}}{\text{GPU}_{kernel}}$ is relatively fast since $\frac{\text{CPU}_{multi}}{\text{GPU}_{total}}$ already yields a speedup of $> 10x$ at 16 options. For comparison, the arithmetic Asian option converges much slower as a function of the amount of options being evaluated. These results can easily be identified in Figure 20.

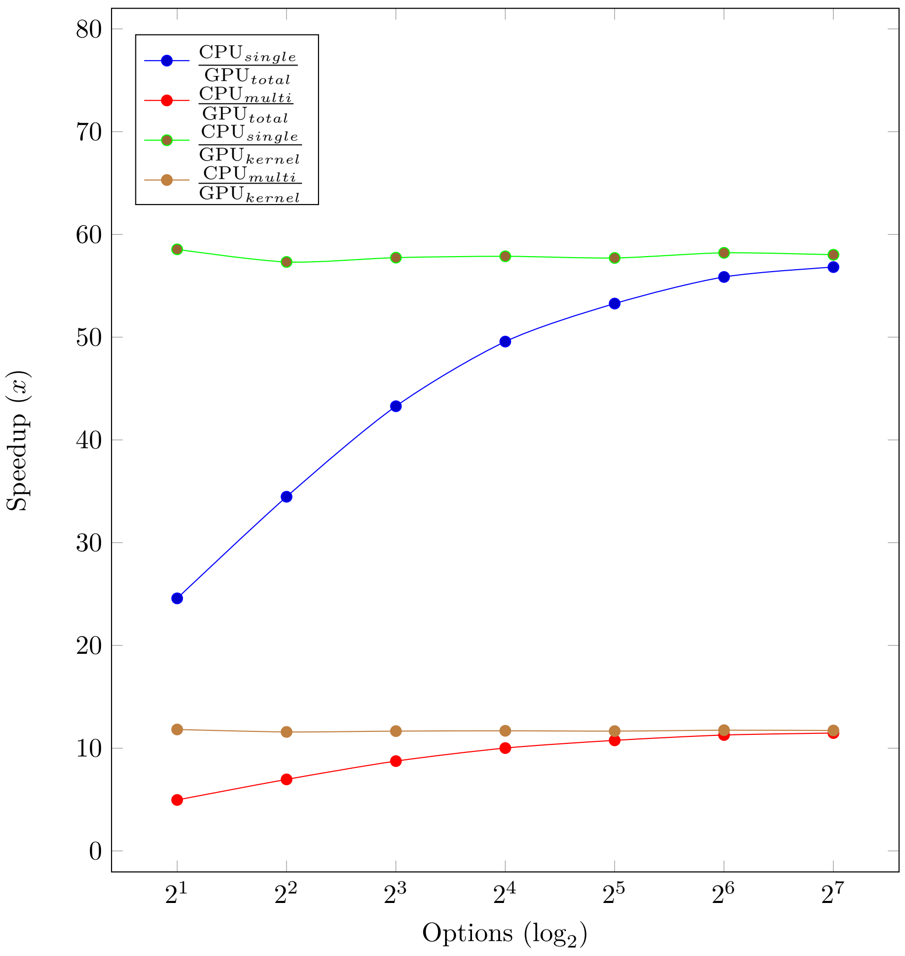

Figure 20: Plot showing the relative performances of GPU$_{total}$ and GPU$_{kernel}$ versus the CPU performances for single-thread and multi-threaded versions.

## 5.5 Implementing the kernels

Implementing and optimizing OpenCL kernels can be a challenging task. One needs to understand the hardware for a proper performance optimized result. Consider the reduction sum case, where the primary bottleneck was the global memory on the GPU. If we develop this algorithm naively, then the performance would be abysmal. However, in the case for this thesis, we decided on using the maximum workgroup size allowed on the AMD 7970 GPU. This choice, was however, specifically related to the particular hardware executing the kernel. Nvidia GPUs are known for allowing a workgroup size of 512 work items, and thus on a Nvidia GPU we could aggregate in buckets of 512 rather than 256 on the AMD GPU.

For the evaluation of the options, several considerations needs to be taken into account for achieving maxmimum performance. First is the consideration of work item configuration, e.g., how should we map the for loops in the arithmetic Asian option work items. Should we unwind both loops or just the outer loop? We found that leaving the inner loop iterating over the time steps intact, resulted in greater performance than unwinding it. Compiled code by the OpenCL compiler is already quite optimized, since adding compiler flags, e.g., `-cl-fast-relaxed-math`, does not increase execution speed significantly. This is in contrast with my earlier experiences with Nvidias CUDA platform, where compiler optimization flags were necessary for achieving adequate performance.

Whether one should use vector kernels or scalar kernels depends on the algorithm. Ideally, the kernel should be vectorized but for some algorithms, e.g., the barrier option, it is not possible. However, the degree of vectorization one should use has to be determined through tweaking and simply trying different kernels. The vanilla European option kernel achieves best performance, when using 4-wide float vectors, whereas the arithmetic Asian option performs best using 16-wide float vectors.

When implementing OpenCL kernels and performance optimizing for AMD GPUs, then one should use AMDs CodeXL tools. These tools allows for kernel profiling which can show numerous statistics, e.g., register usage, kernel occupancy, SIMD utilization, etc. With these tools I discovered the so called kernel "warm-up", i.e., the first time a kernel is executed it performs 25% worse than the following executions. This phenomena might explain why some benchmarks might show *odd* results, especially when executing a kernel a single time.

Also having a analytical solution at hand to verify the algorithms result is handy. It is very difficult to debug on a GPU, so having an analytical solution at hand, or a reference implementation will greatly reduce uncertainty of the correctness of the algorithm. In this thesis the CPU versions was implemented first and verified to be correct, before implementing the algorithm in a OpenCL kernel.

# 6 Conclusions and Future Work

In this thesis, we have implemented the following financial options: vanilla European Option, arithmetic Asian Option, and the barrier Option. The purpose was to utilize a GPU for evaluation and maximizing performance compared to a high quality single-thread and multi-threaded CPU implementation. We tested two scenarios with the following results:

For the scenario of a single option evaluation, the vanilla European option achieved a speedup of $14x$ and $3x$, the arithmetic Asian option achieved a speedup of $17x$ and $3x$, and the barrier option achieved a speedup of $10x$ and $2x$ versus the single-thread CPU and multi-threaded CPU implementation respectively.

For the scenario of multiple options evaluation, the vanilla European option achieved a speedup of $190x$ and $41x$ (320 options), the arithmetic Asian option achieved a speedup of $290x$ and $53x$ (128 options), and the barrier option achieved a speedup of $56x$ and $11x$ (128 options) versus the single-thread CPU and multi-threaded CPU implementation respectively.

These results clearly shows that the Monte Carlo method is an excellent candidate for GPU acceleration within the domain of financial options – and possible other domains as well. However, care must be taken on what kind of option is evaluated, since results can vary dramatically, e.g., the speedup of the arithmetic Asian option is much greater than that of the barrier option, which can be explained by the barriers' conditionals as well as lack of vectorization. In this thesis we did not include random number generation in the benchmarks, which should be investigated further in future work, since random data is so critical for the Monte Carlo method. The Quasi-Monte Carlo method could possibly reduce the absolute running times drastically, thus researching its' application in relation to evaluation of financial options is obvious. Also, there exists several optimizations techniques that can reduce the variance of the input data, theses techniques could reduce the running times as well. Lastly, a review of different options and their applicability for GPU accelerated Monte Carlo simulations needs further investigation.

# 7 Bibliography

[1] L.A. Abbas-Turki, S. Vialle, B. Lapeyre, and P. Mercier. Pricing derivatives on graphics processing units using monte carlo simulation. *Concurrency and Computation: Practice and Experience*, 26(9):1679–1697, 2014.

[2] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[3] Søren Asmussen and Peter W. Glynn. *Stochastic Simulation: Algorithms and Analysis*, volume 57 of *Stochastic Modelling and Applied Probability*. Springer New York, 2007.

[4] Fishcer Black and Myron Scholes. The pricing of options and corporate liabilities. *Journal of political economy*, 81(3):637, 1973.

[5] P. Boyle. Options: a Monte Carlo approach. *Journal of Financial Economics*, 4:323–338, 1977.

[6] P. Boyle, M. Broadie, and P. Glasserman. Monte Carlo methods for security pricing. *Journal of Economic Dynamics and Control*, 21(8-9):1267–1321, 1997.

[7] C. de Schryver, I. Shcherbakov, F. Kienle, N. Wehn, H. Marxen, A. Kostiuk, and R. Korn. An energy efficient fpga accelerator for monte carlo option pricing with the heston model. In *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, pages 468–474, Nov 2011.

[8] Benedict Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous Computing with OpenCL: Revised OpenCL 1.2 Edition.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2 edition, 2013.

[9] J. H. Halton. Algorithm 247: Radical-inverse quasi-random point sequence. *Commun. ACM*, 7(12):701–702, December 1964.

[10] SL Heston. A closed-form solution for options with stochastic volatility with applications to bond and currency options. *Review of Financial Studies*, 6(2):327–343, 1993.

[11] J. C. Hull. *Options, Futures, and Other Derivatives.* Prentice-Hall, Upper Saddle River, N.J., sixth edition, 2006.

[12] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu. *SIGARCH Comput. Archit. News*, 38(3):451–460, June 2010.

[13] William J. Morokoff and Russel E. Caflisch. Quasi-monte carlo integration. *JOURNAL OF COMPUTATIONAL PHYSICS*, 122:218–230, 1995.

[14] C. Niramarnsakul, P. Chongstitvatana, and M. Curtis. Parallelization of european monte-carlo options pricing on graphics processing units. In *Computer Science and Software Engineering (JCSSE), 2011 Eighth International Joint Conference on*, pages 247–249, May 2011.

[15] Brian D. Ripley. *Stochastic Models.* John Wiley and Sons, Inc., 2008.

[16] S. Sawilowsky and G. Fahoome. *Statistics Through Monte Carlo Simulation with Fortran.* JMASM, 2002.

[17] I. M. Sobol'. On the distribution of points in a cube and the approximate evaluation of integrals. 7(4):86–112, 1967. English translation of Russian original published in Zh. vychisl. Mat. mat. Fiz. **7**(4), 784–802, 1967.

[18] Xiang Tian and Khaled Benkrid. High-performance quasi-monte carlo financial simulation: Fpga vs. gpp vs. gpu. *ACM Trans. Reconfigurable Technol. Syst.*, 3(4):26:1–26:22, November 2010.

[19] Wikipedia. Wikipedia, the free encyclopedia, 2015.

[20] Paul Wilmott. *Paul Wilmott Introduces Quantitative Finance.* Wiley-Interscience, New York, NY, USA, 2 edition, 2007.

[21] N.A. Woods and T. VanCourt. Fpga acceleration of quasi-monte carlo in finance. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 335–340, Sept 2008.

[22] Stavros A. Zenios. High-performance computing in finance: The last 10 years and the next. *Parallel Computing*, 25(13-14):2149–2175, December 1999.

# Appendices

## A OpenCL kernels

```
__kernel void MC_EuropeanVanillaScalar(float t, float s, float k,↩
    float r, float sigma, int paths_M,
  const __global float *randoms, __global float *results) {

  int posX = get_global_id(0);
  float St = 0.0f;
  float discount = exp(-r * t);

  St = s * exp((r - (sigma*sigma / 2.0))*t + (sigma*sqrt(t) * ↩
      randoms[posX]));
  St = discount * max(St - k, 0.0f);
  results[posX] = St;
}
```

Monte Carlo pricer for vanilla European option scalar kernel.

```
__kernel void MC_EuropeanVanillaVector(float t, float s, float k,↩
    float r, float sigma, int paths_M,
  const __global float4 *randoms, __global float4 *results) {

  int posX = get_global_id(0);
  float4 St;
  float discount = exp(-r * t);

  St = s * exp((r - (sigma*sigma / 2.0))*t + (sigma*sqrt(t) * ↩
      randoms[posX]));
  St = discount * max(St - k, 0.0f);
  results[posX] = St;
}
```

Monte Carlo pricer for vanilla European option 4-wide vector kernel.

```
__kernel void MC_Asian_Scalar(float t, float s, float k, float r,↩
    float sigma, int paths_M, int timesteps_N,
  const __global float *randoms, __global float *results) {

```

```
  int posX = get_global_id(0);
  float xinc = s;
  float xpath = s;
  float delta = t / ((float) timesteps_N - 1.0);
  float discount = exp(-r * t);

  for (int i = 1; i < timesteps_N; i++)
  {
     xinc = xinc * exp((r - (sigma*sigma / 2.0))*delta + (sigma*↩
         sqrt(delta) * randoms[posX * timesteps_N + i]));
     xpath = xpath + xinc;
  }

  xpath = xpath / (float) timesteps_N;
  xpath = discount * max(xpath - k, 0.0f);
  results[posX] = xpath;
}
```

Monte Carlo pricer for arithmetic Asian option scalar kernel.

```
__kernel void MC_Asian_Vector(float t, float s, float k, float r,↩
    float sigma, int paths_M, int timesteps_N,
  const __global float16 *randoms, __global float16 *results) {

  int posX = get_global_id(0);
  float16 xinc = s;
  float16 xpath = s;
  float16 delta = t / ((float) timesteps_N - 1.0);
  float16 discount = exp(-r * t);

  for (int i = 1; i < timesteps_N; i++)
  {
     xinc = xinc * exp((r - (sigma*sigma / 2.0))*delta + (sigma*↩
         sqrt(delta) * randoms[posX * timesteps_N + i]));
     xpath = xpath + xinc;
  }

  xpath = xpath / (float) timesteps_N;
  xpath = discount * max(xpath - k, 0.0f);
  results[posX] = xpath;
}
```

Monte Carlo pricer for arithmetic Asian option 16-wide vector kernel.

```
__kernel void MC_Barrier_Scalar(float t, float s, float k, float ↩
    r, float sigma, int paths_M, int timesteps_N,
   float barrier, const __global float *randoms, __global float *↩
       results) {

   int posX = get_global_id(0);
   float xpath = s;
   float delta = t / ((float) timesteps_N );
   float discount = exp(-r * t);
   bool barrierBroken = false;

   for (int i = 0; i < timesteps_N; i++)
   {

      if(!barrierBroken)
      {
         xpath = xpath * exp((r - (sigma*sigma / 2.0))*delta + (↩
             sigma*sqrt(delta) * randoms[posX * timesteps_N + i])↩
             );
         if(xpath >= barrier)
         {
            xpath = 0.0f;
            barrierBroken = true;
            break;
         }

      }
   }

   xpath = discount * max(k - xpath, 0.0f);
   results[posX] = xpath;
}
```

Monte Carlo pricer for Up-and-out barrier option scalar kernel.

```
__kernel void reduce(__global float4* input, __global float4* ↩
    output, __local float4* sdata)
{
    unsigned int tid = get_local_id(0);
    unsigned int bid = get_group_id(0);
    unsigned int gid = get_global_id(0);

    unsigned int localSize = get_local_size(0);
    unsigned int stride = gid * 2;
    sdata[tid] = input[stride] + input[stride + 1];

    barrier(CLK_LOCAL_MEM_FENCE);
    for(unsigned int s = localSize >> 1; s > 0; s >>= 1)
    {
        if(tid < s)
        {
            sdata[tid] += sdata[tid + s];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    if(tid == 0) output[bid] = sdata[0];
}
```

Reduction sum kernel.