GPU-Accelerated Option Pricing Algorithms

Daniel Michaeli

School of Science

Bachelor's thesis
Espoo 16.2.2025

Thesis supervisor:

M.Sc. Henrik Lievonen

Thesis advisor:

Prof. Lauri Savioja?

Aalto University
School of Science

AALTO UNIVERSITY SCHOOL OF SCIENCE	ABSTRACT OF THE BACHELOR'S THESIS

Author: Daniel Michaeli

Title: GPU-Accelerated Option Pricing Algorithms

Date: 16.2.2025 Language: English Number of pages: 4+32

Degree programme: Computer Science

Supervisor: M.Sc. Henrik Lievonen

Advisor: Prof. Lauri Savioja?

The digital transformation of financial markets has created unprecedented demand for speed in asset pricing, trading and hedging. This becomes particularly relevant for options and other complex derivatives that are often priced using computationally intensive numerical methods.
This thesis examines GPU acceleration techniques for two fundamental option pricing algorithms: the Cox-Ross-Rubinstein (CRR) binomial model and Monte Carlo (MC) methods. For each approach, the analysis includes a naive GPU mapping with minimal algorithm redesign and an optimized implementation targeting maximum performance. The thesis examines how the algorithms' structure impacts parallelism, compares relative GPU speedups against baseline implementations, and identifies bottlenecks with corresponding optimization strategies.
The results indicate substantial performance improvements for both algorithms, with naive implementations achieving 10x and 23-59x speedups for CRR and MC respectively. The CRR model benefits primarily from algorithmic restructuring to mitigate synchronization overhead, whereas MC methods, already being easy to parallelize, gain additional performance improvements from hardware-conscious optimization. The reviewed literature suggests that GPU acceleration is highly effective for both CRR and MC option pricing, and provides practical guidance for such implementations.

Keywords: options, pricing, GPU, parallel algorithms

AALTO-UNIVERSITETET HÖGSKOLAN FÖR TEKNIKVETENSKAPER	SAMMANDRAG AV KANDIDATARBETET

Författare: Daniel Michaeli

Titel: GPU-accelererade optionsprissättningsalgoritmer

Datum: 16.2.2025 Språk: Engelska Sidantal: 4+32

Utbildningsprogram: Datateknik

Ansvarslärare: M.Sc. Henrik Lievonen

Handledare: M.Sc. Henrik Lievonen

Digitaliseringen av värdepappersmarknader har skapat en enorm efterfråga på snabbhet inom prissättning, handel och hedging. Detta är speciellt relevant för optioner och andra derivatinstrument som ofta värderas genom beräkningsintensiva numeriska metoder.
Detta examensarbete analyserar GPU-accelerationstekniker för två grundläggande optionsprissättningsmetoder: Cox-Ross-Rubinstein (CRR)-modellen och Monte Carlo (MC) simulering. För varje metod presenteras en naiv GPU-implementation med minimal omarbetning, och en optimerad implementation för maximal prestanda. Arbetet analyserar hur algoritmernas struktur påverkar parallelberäkning, jämför prestanda gentemot basversioner, och identifierar flaskhalsar med motsvarande optimeringsstrategier.
Resultaten tyder på en avsevärd prestandaförbättring för båda algoritmerna. Den naiva GPU-implementationen av CRR-modellen uppnådde 10x snabbare beräkningar, medan MC simulering uppnådde 23-59x förbättring. CRR gynnas huvudsakligen av algoritmisk omdesign för att motverka synkroniseringsoverhead. MC simulering är ursprungligen lätt att parallellisera, och försnabbas maximalt genom hårdvarumedveten optimering. Litteraturanalysen visar att GPU-acceleration är mycket effektivt för både CRR- och MC-baserad prissättning, och erbjuder praktisk vägledning för sådana implementeringar.

Nyckelord: Nyckelord på svenska,
optioner, prissättning, GPU, grafikkort, parallella algoritmer

Contents

1 Introduction

Like many other parts of modern society, financial markets have undergone a radical digital transformation. Where traders once shouted orders across crowded exchange floors, computer algorithms on powerful servers now execute trades with minimal direct human interaction [1, 2]. This shift has made computational speed a critical competitive advantage. For example, high-frequency trading strategies exploit transient price discrepancies through millions of rapid transactions, where mere millisecond delays can determine profitability. These strategies are adopted not only by proprietary trading firms but also by large mutual and pension funds that need to minimize the market impact of large trades [1].

This algorithmic trading landscape extends to derivatives markets, where advances in financial mathematics [3] and the exponential increase in computing power [4] have enabled increasingly sophisticated pricing models. Derivatives denote financial instruments whose value is derived from the price of an underlying asset, such as a share, commodity or currency. Options are derivative contracts between two parties that grant the holder the right, but not the obligation, to buy or sell an asset at a predetermined price by a future date [5]. These instruments enable both speculative trading and risk management strategies, making them valuable even to non-financial institutions [6].

Modern option pricing often employs numerical and algorithmic solutions whose computational intensity conflicts with the response times demanded by algorithmically driven markets. Furthermore, real-world applications rarely involve single options in isolation. Portfolio-wide stress testing and hedging require simultaneous pricing of many options across multiple scenarios, demanding extraordinary processing power. These performance requirements has motivated research into adapting option pricing algorithms to leverage the GPU's parallel computing capabilities.

This thesis reviews GPU acceleration for two common option pricing algorithms: the Cox-Ross-Rubinstein (CRR) binomial model and Monte Carlo (MC) methods. The objectives are threefold: understanding how algorithm structure impacts parallelization, measuring relative performance gains, and identifying computational bottlenecks. The Black-Scholes-Merton (BSM) model is omitted as it is an analytical formula rather than an algorithm, providing no meaningful parallelization opportunities beyond arithmetic operations. PDE-based numerical methods, while offering substantial parallelization challenges, require mathematical foundations beyond the scope of this thesis.

Section 2 establishes formal definitions and fundamental option pricing theory. Section 3 examines GPU architecture and introduces parallel computing concepts for algorithm analysis. Section 4 and Section 5 analyze GPU implementations of the CRR and MC algorithms. Section 6 summarizes the findings and presents conclusions.

For scope restriction purposes, the following choices have been made:

- The analysis mainly examines European and American options, collectively termed *vanilla options*, with the exception of Monte Carlo methods that naturally handle *exotic options* with complex payoff structures. All underlying assets are assumed to be non-dividend-paying.

- The implementations are evaluated solely on computational performance — specifically execution time and parallelization efficiency. Accuracy, numerical stability, and other properties are not considered.
- The reviewed research uses varying architectures, optimization levels, and performance metrics, making direct quantitative comparison infeasible. Analysis therefore focuses on relative speedups of GPU implementations over their respective baselines.

2 Fundamentals of Option Pricing

The following section presents the theoretical background necessary to understand option pricing and the algorithms that will be GPU-accelerated. First, general definitions are established. The section then analyzes vanilla option payoffs and the key determinants of option prices. Two essential principles anchor the theoretical framework: *no-arbitrage pricing* and *risk-neutral valuation.* Together, these concepts form the mathematical foundation underlying modern option pricing models.

2.1 Definitions

The following terminology is adapted from Hull's "Options, Futures, and Other Derivatives" [5]:

Call Option. A contract granting the holder the right, but not the obligation, to purchase an underlying asset at a predetermined price by a specified date.

Put Option. A contract granting the holder the right, but not the obligation, to sell an underlying asset at a predetermined price by a specified date.

Strike Price. The predetermined price at which the underlying asset may be bought (call) or sold (put) when the option is exercised.

Maturity. The expiration date of the option contract, after which it becomes worthless if not exercised.

The phrase "by a specified date" warrants clarification, as exercise rules vary by option style. *European options* may be exercised only at maturity, while *American options* can be exercised at any time before maturity. Throughout this thesis, the underlying asset is generally referred to as "the stock," with its current market value denoted as the *spot price.*

Every option contract involves two parties with opposing positions. The buyer assumes the *long position*, paying an upfront premium for the right to exercise the option. The seller — also known as the option writer — takes the corresponding *short position*, receiving the premium in exchange for accepting the obligation to fulfill the contract if exercised.

2.2 Vanilla Option Payoffs

Options are zero-sum contracts: gains to one party exactly offset losses to the other. The *payoff* of an option represents its intrinsic value at exercise, whereas the profit accounts for the premium paid or received: profit = payoff − premium for long positions. The payoff is mechanically determined by the option's characteristics and market conditions at exercise, while the premium is the upfront cost determined by market participants. Option pricing models aim to determine the theoretical fair value of this premium.

The following presentation of option payoffs draws from Hull [5]. It presents diagrams that visualize how profit varies with the spot price at exercise. Each vanilla option type is examined through both long and short positions, supplemented by practical examples. For simplicity, all options are written on one share of the underlying stock, transaction costs and taxes are ignored, and markets are assumed sufficiently liquid to permit instantaneous trading.

A call option buyer pays a premium upfront to secure the right to purchase stock at the strike price. This position profits when the stock price rises sufficiently to offset the premium paid. Figure 1 illustrates this long position. Let S_T denote the spot price at exercise and K the strike price. When $S_T < K$, the option is said to be *out of the money (OTM)* and has zero payoff. At $S_T = K$, the option is *at the money (ATM)*, and when $S_T > K$, the option becomes *in the money (ITM)* with payoff $S_T - K$ [5]. However, the position only becomes profitable once the payoff offsets the premium paid. The "moneyness" terminology always reflects the perspective of the long position. Rational investors exercise ITM options to capture the payoff, even if it only partially offsets the premium paid. In liquid markets, this involves buying stock at the strike price and immediately selling at the higher spot price.

To demonstrate the use of options for risk management, consider a manufacturer dependent on crude oil anticipating a price increase. By purchasing call options, they fix a price ceiling for future oil purchases, effectively buying insurance against adverse price movements. The premium represents the cost of this protection, with the strike price chosen to align with their risk tolerance and budget constraints.

The short call position has an inverse profit diagram. When the option is OTM, it expires worthless and the writer retains the full premium as profit. When it is ITM, the buyer exercises, forcing the writer to sell the stock at the strike price K despite the higher spot price S_T [5]. The position remains profitable as long as the premium offsets this loss, with break-even at $S_T - K = p$. Figure 2 illustrates the corresponding profit diagram. Continuing with the previous example, this position could be taken by a crude oil producer expecting prices to remain below or near the strike price at maturity. The risk taken by the writer commands the premium.

The payoff from the long position in a call option is defined as

$$\max(S_T - K, 0) \tag{1}$$

as the buyer will only exercise when ITM to either profit or offset losses. Consequently, the payoff from the short position is defined as

$$-\max(S_T - K, 0) = \min(K - S_T, 0)$$

due to the zero-sum nature of the contract [5].[1]

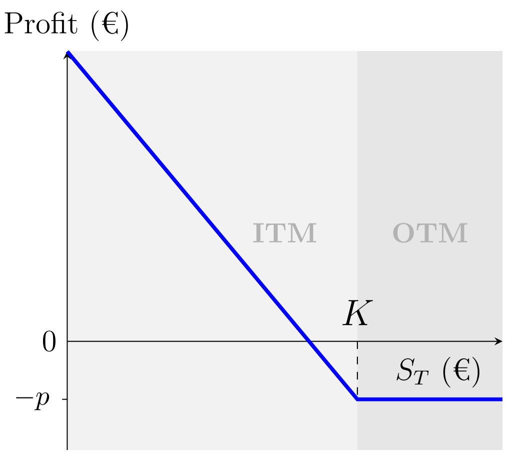
A put option buyer pays a premium upfront to secure the right to sell stock at the strike price. This position profits when the stock price falls sufficiently to offset the premium paid. Figure 3 illustrates both long and short positions. When $S_T > K$, the put option is OTM and expires worthless — no rational investor would sell at

[1]For American options with the right to early-exercise, the payoff is calculated using S_τ instead of S_T, where $\tau \leq T$ is the chosen point of exercise.

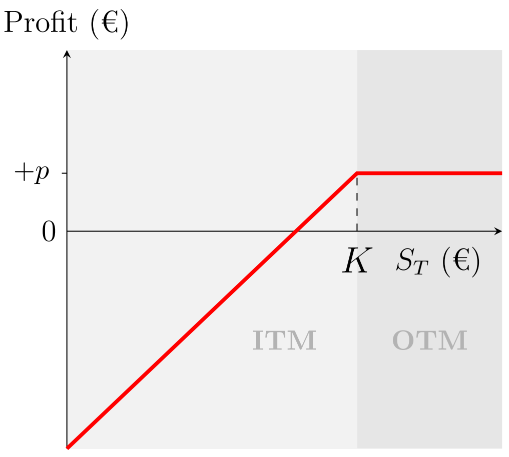
the strike price K when they could sell at the higher spot price S_T. When $S_T < K$, the option becomes ITM with payoff $K - S_T$ [5]. The buyer profits only when this payoff exceeds the premium paid. At exercise, the buyer can buy stock at the spot price and immediately sell it at the higher strike price.

The short put position profits under opposite conditions. When the option is OTM, it expires worthless and the writer retains the full premium as profit. When it is ITM, the buyer exercises, forcing the writer to purchase stock at the strike price K despite the lower spot price S_T [5]. The position remains profitable as long as the premium offsets this loss, with break-even at $K - S_T = p$.

An example case would be a crude oil supplier forecasting low prices in a year's time and taking the long position to secure a minimum revenue. The supplier pays a premium to hedge against unfavorable price movements. The short position would be taken by a market participant who anticipates crude oil prices not to fall enough to offset the premium.



(a) Long put position



(b) Short put position

Figure 3: Profit diagrams for long (a) and short (b) positions in a put option. K = strike price, S_T = spot price at exercise, p = premium. (a) The buyer pays the premium p for the right to sell stock at strike price K. When $S_T < K$, the option is in the money (ITM) and is exercised. The position profits when $K - S_T > p$. When $S_T > K$, the option is out of the money (OTM) and expires worthless with loss $-p$. (b) The writer receives premium p for the obligation to buy at price K. When ITM, the writer must purchase at the strike price K despite the lower spot price S_T. When OTM, the option expires worthless and the writer keeps the premium p. The position becomes unprofitable when $K - S_T > p$.

The payoff from the long position in a put option is defined as

$$\max(K - S_T, 0) \tag{2}$$

as being ITM now means $K > S_T$ [5]. Conversely, the payoff in the corresponding short position is defined as

$$-\max(K - S_T, 0) = \min(S_T - K, 0).$$

The buyer of an option is never obligated to exercise, and the loss is therefore limited to the premium paid. On the other hand, the writer is always obligated to potentially engage in a disadvantageous trade with the buyer. Additionally, since underlying asset prices have no theoretical upper bound but are limited to non-negative values, the short call position faces potentially unlimited losses. Table 1 aggregates information about the extreme-case profit and loss incurred from the different option types and positions.

Table 1: Extreme-case upside and downside potential for different option positions. p = premium, K = strike price. The long position in a call option has unlimited upside potential due to theoretically unbounded underlying asset prices, with losses limited to the premium paid. The corresponding short position inverts this behavior with unlimited downside risk. Put options have bounded payoffs as asset prices cannot fall below zero. The maximum gain for a long put position is $K - p$ when the spot price reaches zero, and the maximum loss for the corresponding short position is $-(K - p)$.

	Call Option	**Put Option**
Long	**Maximum Gain:** $+\infty$ **Maximum Loss:** $-p$	**Maximum Gain:** $K - p$ **Maximum Loss:** $-p$
Short	**Maximum Gain:** $+p$ **Maximum Loss:** $-\infty$	**Maximum Gain:** $+p$ **Maximum Loss:** $-(K-p)$

Pricing models aim to determine the theoretical fair value of the option premium p. While market prices form through supply and demand dynamics, they remain anchored by mathematical and economic principles. Short-term deviations occur, but prices tend to converge toward theoretically justified values over time as mispricing creates correcting trading opportunities.

2.3 Price Determinants

Hull [5] identifies the following primary factors that affect an option's value. While these serve as parameters in pricing models, their directional effects can be reasoned through without formal mathematics. The analysis examines each factor in isolation (ceteris paribus), although they often interact with each other in practice.

Stock Price. The relationship between stock and option prices follows directly from the payoff definitions. Call option values increase with stock price, while put option values decrease, as demonstrated in Section 2.2.

Volatility. Higher volatility increases both call and put option values. The asymmetric risk-reward profile — losses limited to premium paid versus potentially large gains — means increased price variability improves expected outcomes. This effect is most pronounced for OTM options that require significant price movements to gain value.

Maturity. Following the asymmetric risk-reward logic, longer-dated options generally command higher premiums due to increased opportunity for favorable price movements. American options particularly benefit through early-exercise flexibility. Expected dividends, however, can reverse this relationship for call options — an option expiring before the ex-dividend date may be worth more than one expiring after, since dividends reduce stock prices without benefiting option holders [5].

Risk-Free Rate. The risk-free rate represents the return on a theoretically risk-free investment, such as a government bond, and serves as a baseline for valuing riskier assets. In option pricing, it affects value by discounting future cash flows. Higher interest rates reduce the present value of the strike price. For call options, this makes exercising cheaper in present terms, increasing the option's value. For puts, it reduces the value of the payoff received, lowering the option's price. In real markets, interest rate changes also influence stock prices, complicating this relationship.

Dividends. Expected dividend payments decrease the value of calls and increase the value of puts, as stock prices typically drop on ex-dividend dates without benefiting option holders. This effect is most significant for longer-dated options that span multiple dividend payments.

2.4 No-Arbitrage Pricing and Risk-Neutral Valuation

This section introduces ideas from the work of Fischer Black, Myron Scholes, and Robert Merton that revolutionized option pricing and led to the 1997 Nobel Prize in Economic Sciences[2]. Their breakthrough contribution was demonstrating how a portfolio could be constructed to replicate and thus price an option. This replicating portfolio approach, and its elegant reformulation using risk-neutral probabilities, form the foundation for all modern option pricing models, including the CRR binomial model and MC methods. The presentation emphasizes intuition over rigor, and avid readers are encouraged to explore the references in greater detail [5, 7, 8]. The assumptions presented in Section 2.2 continue to hold.

Traditional asset valuation discounts expected future cash flows to present value. While theoretically sound, these approaches require subjective inputs that produce differing valuations. Expected cash flow calculations require probabilities of future outcomes — an inherently speculative exercise. Even if this value were known, no consistent methodology exists to determine the appropriate discount rate, as investors naturally have differing risk preferences [9]. The key insight of Black, Scholes, and Merton was recognizing that for options, these subjective estimates could be eliminated entirely by making a reasonable assumption about market behavior — the absence of arbitrage.

Arbitrage denotes simultaneous buying and selling of an asset at different prices to make risk-free profit. Imagine tomato prices being lower in town A than town B due to an exceptionally good harvest. An astute tradesman could buy tomatoes in

[2]The prize was awarded to Scholes and Merton; Black had passed away in 1995.

town A and immediately sell them in town B, pocketing the difference without taking any position on future price movements. This opportunity is self-eliminating: as the tradesman introduces more tomatoes from town A into town B, the corresponding prices adjust to the changes in supply until an equilibrium is reached [5, 10]. Other traders observing this opportunity might adopt the same strategy, eliminating it even faster. While arbitrage opportunities do arise, they are typically rare and short-lived in modern markets, as sophisticated traders quickly eliminate them. Hence, the no-arbitrage assumption is reasonable, at least over longer time periods.

It logically follows that in absence of arbitrage opportunities, two assets producing identical cash flows must be equally valued [5, 10]. This conclusion suggests a new approach: construct a portfolio that matches the option payoff in all scenarios. The option value must then equal the price of constructing such a portfolio.

Hull [5] demonstrates this approach using a one-period binomial model. The current stock price S_0 can move to either S_0u or S_0d $(u > 1, d < 1)$ after time T. These prices result in option payoffs f_u and f_d respectively. Figure 4 illustrates the setup. Restricting the stock price to two future outcomes makes it possible to construct a *replicating portfolio*, consisting of Δ shares of the stock and B of a risk-free asset growing at the risk-free rate r. Equating the portfolio value with the option payoff in each future state gives:

$$\begin{cases} \Delta(S_0u) + B(1+r) = f_u \\ \Delta(S_0d) + B(1+r) = f_d \end{cases}.$$

Solving for Δ and B yields

$$\begin{cases} \Delta = \frac{f_u - f_d}{S_0(u-d)} \\ B = \frac{f_d u - f_u d}{(u-d)(1+r)} \end{cases}.$$

It is evidently possible to construct a portfolio with identical payoff as the option itself, according to

$$\frac{f_u - f_d}{S_0(u-d)}S_0 + \frac{f_d u - f_u d}{(u-d)(1+r)}. \tag{3}$$

Under the no-arbitrage assumption, the value of the option must equal the price of constructing this portfolio, that is buying Δ shares and investing B in a risk-free asset. A negative Δ implies shorting the stock[3], and a negative B implies borrowing money at the risk-free rate instead of investing it. The derivation assumed neither a call nor a put option, and therefore works as a pricing model for any European (and in a one-period setting, American) option. An interesting observation is that the approach utilized no information about the expected return of the stock, or the probability of its price movements. The authors motivate this intuitively by the fact that such expectations are already reflected in the stock price itself, and need not be explicitly accounted for when pricing the derivative. The option is priced relative to the underlying stock [5].

[3]Short selling involves borrowing shares from a broker, selling them immediately at the current price, and later repurchasing them to return to the lender. If the price falls, the short seller profits from the difference.

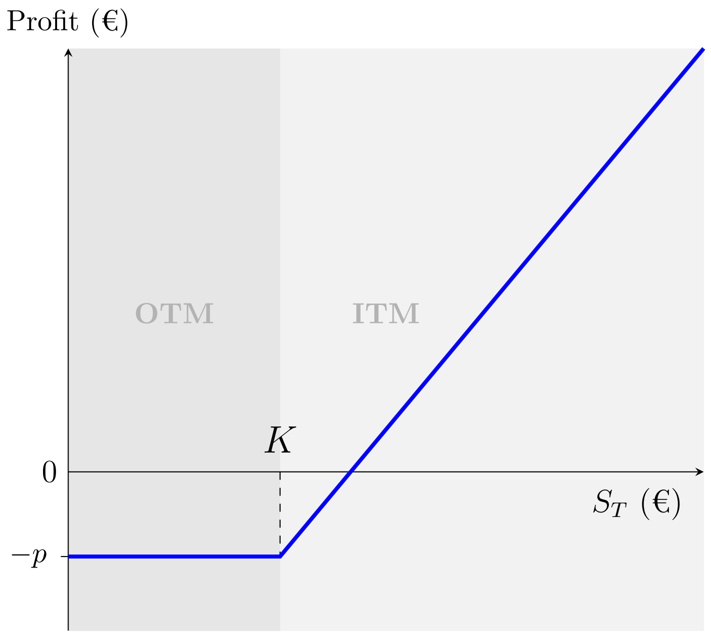


Figure 1: Profit diagram for the long position in a call option. K = strike price, S_T = spot price at exercise, p = premium. The buyer pays a premium to secure the right to purchase stock at price K. When $S_T < K$, the option is out of the money (OTM) and expires worthless with profit $-p$. When $S_T > K$, the option is in the money (ITM) and is exercised: the buyer purchases stock for K to sell at S_T. The position becomes profitable only when $S_T - K > p$.

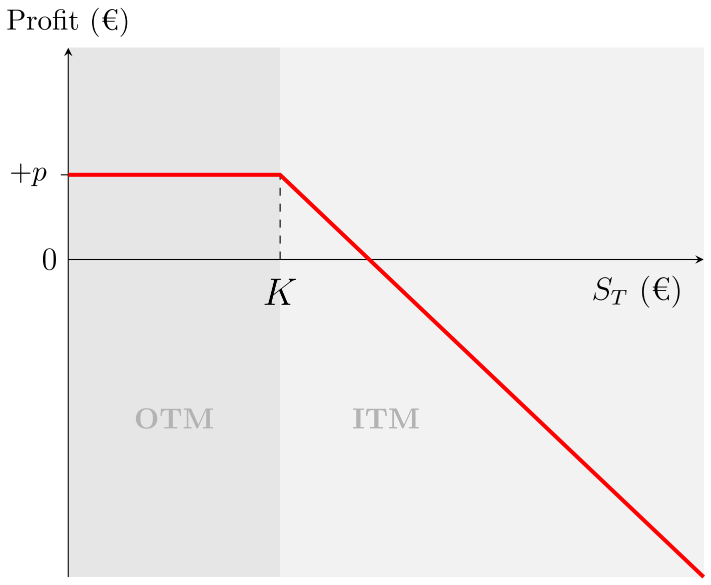


Figure 2: Profit diagram for the short position in a call option. K = strike price, S_T = spot price at exercise, p = premium. The writer receives a premium for the obligation to sell stock at price K. When $S_T < K$, the option is out of the money (OTM) and expires worthless. As it is not exercised, the writer retains the full premium p as profit. When $S_T > K$, the option is in the money (ITM) and is exercised: the writer is forced to sell the stock at the strike price K despite the higher spot price S_T. The profit decreases linearly with S_T, becoming negative once $S_T - K > p$.

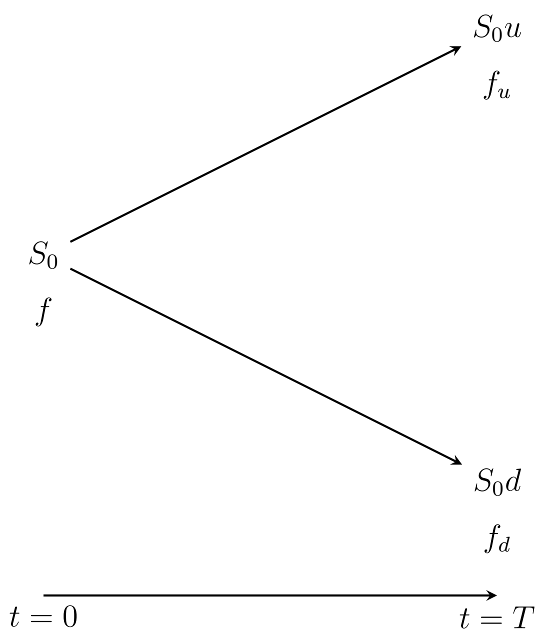


Figure 4: One-period binomial model. The stock price moves from S_0 to either S_0u or S_0d over time T, with corresponding option values f, f_u and f_d.

Equation (3) can be rewritten as

$$\frac{1}{1+r}\left(f_u\frac{(1+r)-d}{u-d}+f_d\frac{u-(1+r)}{u-d}\right).$$

Defining $\tilde{p} := \frac{(1+r)-d}{u-d}$, this weight and its complement $1-\tilde{p} = \frac{u-(1+r)}{u-d}$ sum to one and lie in $[0,1]$ under the no-arbitrage condition $d \leq 1+r \leq u$, thus behaving like probabilities. These are known as *risk-neutral probabilities*, and the calculation can therefore be interpreted as an expected value of the option payoff, discounted at the risk-free rate:

$$\frac{1}{1+r}(f_u \cdot \tilde{p} + f_d \cdot (1-\tilde{p})). \tag{4}$$

While risk-neutral probabilities often cause confusion, they can simply be understood as a mathematical reformulation into a more familiar approach for pricing assets — discounting future cash flows. They do not reflect actual stock price movements, but rather form an artificial probability measure that yields the same price as the replicating portfolio. The term risk-neutral stems from the fact that the risk-free rate is used for discounting, resembling how a risk-neutral investor would not demand a premium for uncertain cash flows. This elegant reformulation enables tools from probability theory, and elements of it will be found in all of the models to be introduced later. Gisiger [7] and Tham [8] provide more formal explanations of risk-neutral valuation. The full derivation of Equation (4) is provided in Appendix A.

3 GPU and Parallel Computing

The graphics processing unit (GPU) is a specialized processor originally designed to accelerate the rendering of images and video. The rapid growth of video games and graphics-intensive applications in the 1980s created unprecedented demand for capable hardware [11, 12]. Graphics-related tasks like shading, rasterization, and texture mapping require a massive number of floating-point operations per second (FLOPS), far exceeding what general-purpose CPUs could deliver. Companies like NVIDIA and ATI Technologies (later acquired by AMD) emerged as front-runners in the GPU industry [11].

Graphics-related tasks presented a unique computational challenge: they required massive parallelism rather than fast sequential processing. This led GPU designers to optimize for throughput rather than latency. *Latency* denotes the time between the start and completion of a single event, while *throughput* measures the total amount of work performed in a given time [13]. These fundamental metrics are widely used in computing performance-related contexts. Hennessy and Patterson [13] demonstrate that while both CPUs and GPUs have improved in both dimensions, throughput improvements have significantly outpaced latency improvements. This disparity reflects fundamental constraints: throughput can be increased by adding more processing units. Reducing latency, however, requires making individual processing units faster, which is approaching theoretical limits imposed by the laws of physics [13, 11]. Suomela's analysis of clock speed and instruction latency trends corroborate these observations [14]. This, along with the parallel nature of graphics-related computation, led to the exceptionally throughput-optimized GPU architectures of today.

As GPU architectures evolved, researchers and developers increasingly sought to utilize their parallel computing power for non-graphics applications. Early GPUs were, however, not designed for general-purpose computing. The hardware lacked flexibility, and no suitable programming tools existed. Developers were forced to mask their problems as graphics operations [11, 12]. NVIDIA transformed this landscape in 2007 when they released CUDA (Compute Unified Device Architecture), a software abstraction layer that exposed GPU hardware capabilities through familiar C/C++ extensions. Developers could now write parallel programs directly without disguising them as pixel operations. Today, general-purpose GPU (GPGPU) applications span diverse fields, including computational finance [11, 12].

3.1 CPU and GPU Architecture

Figure 5 presents a simple architectural diagram of a modern CPU and GPU. The CPU diagram is based on the writings of Stallings [15], while the GPU diagram is derived from both the work of Kirk and Hwu [12] and the CUDA C++ Programming Guide by NVIDIA [16].

The CPU contains several key components working together. The arithmetic logic unit is responsible for performing computational operations on data. Registers (not visible in the figure) store instructions, data, and memory addresses. Additional control units manage program flow and coordination with external components. The CPU executes programs in a two-step fetch-execute cycle. First, an instruction is fetched from memory and loaded into a register. The instruction is then executed, which in practice means either data processing, data transfer, or control flow operations. This cycle repeats until the program terminates, either by completing successfully, encountering an error, or being interrupted by another component.

CPUs have evolved to leverage multiple forms of parallelism. Instruction-level parallelism refers to a set of techniques to improve performance by executing multiple independent instructions simultaneously or in overlapping stages [13, 14]. These techniques range from instruction pipelining and superscalar execution, which process multiple instructions concurrently, to out-of-order execution and branch prediction, which maximize pipeline utilization and anticipate control flow.

Modern CPUs also incorporate *single instruction multiple data* (SIMD) capabilities through vector registers that store multiple data elements of the same type. This allows for executing a single instruction on all elements in the register simultaneously. Furthermore, most modern processors are multi-core, containing multiple instances of the core computational components. This enables thread-level parallelism, where a *thread* — an independent sequence of instructions that can be scheduled and executed [15] — can run on each core simultaneously [13, 14]. Nonetheless, CPUs remain fundamentally optimized for low-latency sequential execution and control flow rather than maximizing throughput [13, 14].

In contrast, while lacking the sophistication that makes CPUs excel at sequential and control-heavy tasks, GPUs dedicate most of their silicon area to processing units, creating massively parallel architectures optimized for numerical computation. GPU execution follows a *single instruction multiple thread* (SIMT) paradigm: thousands of threads execute the same instruction in parallel [16]. This increases throughput far beyond single- or multithreading on comparatively few CPU cores.

A modern GPU contains multiple independent *streaming multiprocessors* (SMs), each comprising numerous processing units that share control logic and local memory resources [12, 16]. Within each multiprocessor, threads execute in groups of 32 called *warps*. These warps generally operate in lockstep — all threads in a warp execute the same instruction simultaneously on their respective data.[4] Similarly to CPUs, all SMs share access to high-capacity but high-latency global memory (RAM), while each SM contains faster local caches. Additionally, each SM contains a programmer-managed memory region called *shared memory*, enabling inter-thread communication within the same multiprocessor [11, 16].

3.2 Parallel Computing Fundamentals

A theoretical framework helps to analyze parallelism opportunities and limitations. This section introduces key concepts for assessing algorithm structure, theoretical speedup limits, and common pitfalls.

[4]Since the NVIDIA Volta architecture (2017), threads within a warp can follow independent control paths, enabling more flexible algorithms. However, divergent execution reduces efficiency as different paths must be serialized [16].

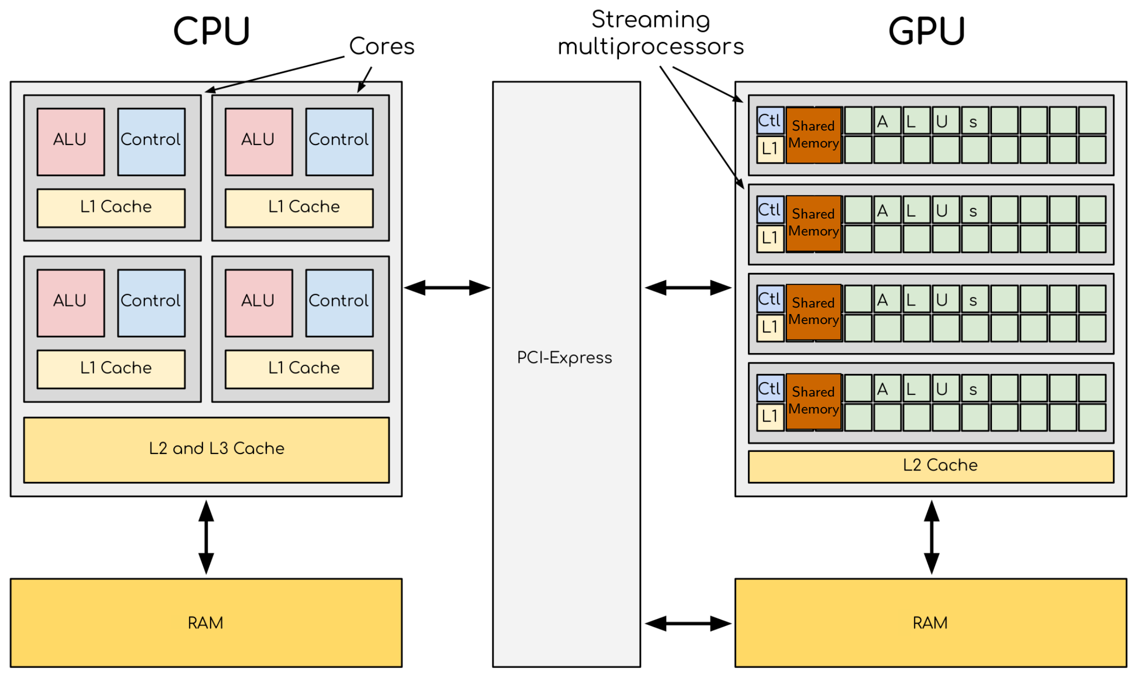


Figure 5: Comparison of modern multi-core CPU and GPU architectures. The CPU features four cores, each containing an arithmetic logic unit (ALU) and control unit, supported by hierarchical cache memory (L1-L3) to minimize data transfer bottlenecks between the random access memory (RAM) and the cores. The GPU contains multiple streaming multiprocessors, each containing numerous processing units and shared memory for inter-thread communication, but minimal control logic. This architectural difference reflects their design priorities: CPUs optimize for sequential performance and complex control flow, while GPUs maximize parallel throughput. The PCI-Express serves as a communication interface between them. Adapted from material by ENCCS, licensed under CC-BY 4.0 [17].

Kirk and Hwu [12] distinguish between two forms of parallelism: data parallelism and task parallelism. The former denotes performing the same operations in parallel on different data, while the latter means performing multiple different tasks simultaneously. An example of data parallelism is vector addition, where each respective component in vector A is added to its corresponding component in vector B. The authors argue that data parallelism is more abundant in modern computing as programs utilize increasingly large data sets. The SIMT-based architecture of GPUs is naturally highly optimized for data parallelism. This originates from graphics rendering, where identical operations must update many thousands of pixels simultaneously [11]. While task parallelism is less suited to GPU architectures, it can still be exploited by partitioning programs into independent sub-tasks.

Acar [18] provides a framework for formal analysis of parallel algorithms. Algorithms are represented as directed acyclic graphs (DAGs), where nodes represent executions of individual instructions and edges represent dependencies between them. The DAG can be viewed as a Hasse diagram that depicts a partial ordering of instructions under the relation "must execute before". Acar restricts the maximum node *out-degree* and *in-degree*[5] to two, corresponding to spawning a new thread and synchronizing the result from two threads. Figure 6 depicts a DAG representation of an algorithm.

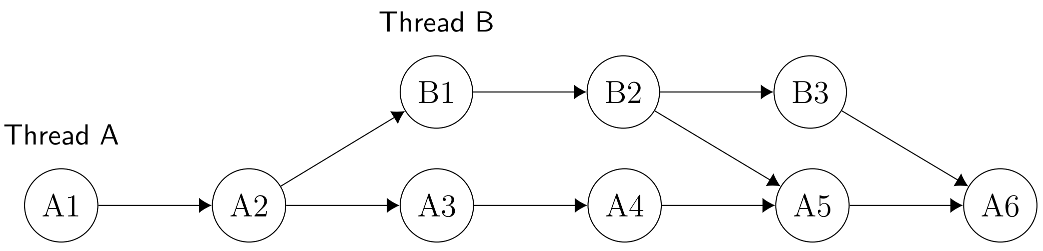


Figure 6: Directed acyclic graph (DAG) representation of a parallel algorithm. Nodes represent individual instructions and edges represent instruction dependencies that determine execution order. At node A2, a new thread is spawned, after which both threads execute independent instruction sequences in parallel. The threads synchronize at nodes A5 and A6, but thread B terminates only at the latter. DAG representations help visualize the algorithm structure and identify opportunities for parallel execution on independent paths.

[5]The number of edges extending out from or into a node.

Using these graphs, Acar [18] defines the *work* of an algorithm as the total number of nodes in the DAG, and *span* as the length of its longest path. Alternatively, assuming each instruction takes a unit of time and no overhead cost of thread management, one can think of the work as the sequential run-time on a single processor T_1, and span as the run-time on an idealized infinite-processor machine T_∞. For instance, the DAG presented in Figure 6 contains nine units of work but a span of only six units. It is now possible to define two lower bounds on algorithm run-time on a p-processor machine T_p:

Span-limited lower bound $T_p \geq T_\infty$. No matter the number of available processors, the execution cannot be faster than the longest dependency chain in the DAG. This illustrates the role of algorithm structure in parallelization potential, and motivates the effort to restructure algorithms to minimize span and other dependency chains.

Work-limited lower bound $T_p \geq T_1/p$. p processors can perform at most p operations in parallel, even if there are no dependencies. This highlights hardware constraints.

Furthermore, Acar demonstrates that for both level-by-level and greedy scheduling[6] the upper bound is within a factor 2 of the lower bound, again highlighting the importance of minimizing span and providing sufficient resources [18]. In practice, runtime analysis is often more complex due to factors such as varying instruction latencies, parallelization overhead, and synchronization and shared resource contention.

[6]Level-by-level scheduling assigns nodes to processors by a level-order traversal of the DAG. This can leave processors idle if the number of nodes per level does not evenly divide the number of processors. In contrast, greedy scheduling minimizes idle time by immediately assigning available nodes to idle processors.

4 GPU Acceleration of the Cox-Ross-Rubinstein Binomial Model

The Cox-Ross-Rubinstein (CRR) model [19] extends the single-period setting into its multi-period counterpart, providing a more realistic model of asset price evolution. The model maintains the same assumptions as before: discrete periods and constant multiplicative factors u and d for price movements. Over multiple periods, this creates a binomial lattice of possible price paths. This lattice expands outward from the initial stock price, generating a set of final states for the stock price and the corresponding option payoff.

The CRR model solves for the initial option value using backward induction. Starting at the terminal nodes, where option payoffs are calculated using Equations (1) and (2), the algorithm works backward through the lattice. Each interior node is treated as a single-period problem, with option values calculated using Equation (4). Figure 7 visualizes a two-period lattice, though the model extends to any number of periods.

Cox, Ross, and Rubinstein defined $u = e^{\sigma\sqrt{T/n}}$ and $d = e^{-\sigma\sqrt{T/n}}$, where σ is the

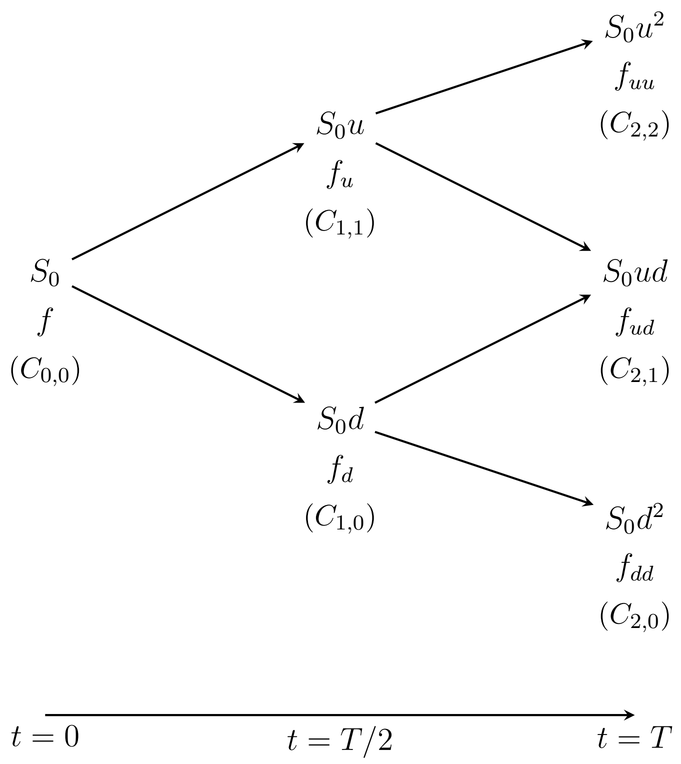


Figure 7: Two-period binomial lattice. Stock prices evolve over time, creating an expanding set of possible outcomes. Option payoffs are calculated at the terminal nodes, after which option values at interior nodes can be calculated using Equation (4). The symbols $C_{i,j}$ represent the algorithmic notation for the option values, where i indexes time and j the number of up-ticks the stock price has moved.

annualized volatility of the asset's log-returns, T is the total time in years, and n is the number of periods [19]. They show this to be useful for several reasons. First, it ensures the model converges to the famous Black-Scholes-Merton (BSM) model as $n \to \infty$. Second, the lattice recombines — permutations of a price path reach the same node. This maintains computational feasibility as the number of states grows linearly with n instead of exponentially. Additionally, the discount rate must be adjusted for n such that it matches that of the one-period example. The continuous rate $e^{rT/n}$ is typically used instead of discrete compounding [5].

The following pseudocode is based on the implementation by Clewlow and Strickland [20]. Let S_0 be the initial stock price and n the number of periods. The nodes of the recombining lattice are indexed by (i, j), where $0 \leq i \leq n$ indexes time[7] and $0 \leq j \leq i$ the number of up-ticks j the stock price has moved. The stock price at each node is defined as $S_{i,j} = S_0 u^j d^{i-j}$, and $C_{i,j}$ denotes the corresponding option value. Algorithm 1 calculates the option value at current time $C_{0,0}$.

Algorithm 1: CRR European Option Pricing

```
    Input: $S_0$ (initial stock price), $K$ (strike price), $T$ (time to maturity), $r$
           (risk-free rate), $n$ (periods), $\sigma$ (volatility), option type
    Output: Option value at $t = 0$
 1  $\Delta t \leftarrow T/n$;
 2  $u \leftarrow e^{\sigma\sqrt{\Delta t}}$;
 3  $d \leftarrow 1/u$;
 4  $disc \leftarrow e^{-r\Delta t}$;
 5  $p \leftarrow (e^{r\Delta t} - d)/(u - d)$;
    // Forward phase: compute stock prices at terminal nodes
 6  for $j \leftarrow 0$ to $n$ do
 7      $S_j \leftarrow S_0 \cdot u^j \cdot d^{n-j}$;
 8  end
    // Compute option payoffs at terminal nodes
 9  for $j \leftarrow 0$ to $n$ do
10      $C_j \leftarrow \max(0, S_j - K)$ for call option;
11      $C_j \leftarrow \max(0, K - S_j)$ for put option;
12  end
    // Backward phase: compute option values through lattice
13  for $i \leftarrow n - 1$ downto $0$ do
14      for $j \leftarrow 0$ to $i$ do
15          $C_j \leftarrow disc \cdot (p \cdot C_{j+1} + (1 - p) \cdot C_j)$;
16      end
17  end
18  return $C_0$;
```

[7] A period denotes an interval between two discrete points in time, hence for n periods we have $n + 1$ time points from 0 to n.

The core of the algorithm lies in line 15, which directly implements the risk-neutral valuation principle shown in Equation (4). A key advantage of the CRR and other lattice-based models is their natural ability to price American options, which generally cannot be valued using closed-form solutions like the BSM model [10]. For American options, the option value calculation on line 15 is modified to include the early-exercise possibility:

$$C[j] \leftarrow \max(\text{payoff}(S[j]), disc \cdot (p \cdot C[j+1] + (1-p) \cdot C[j])),$$

where $\text{payoff}(S[j])$ equals $\max(K - S[j], 0)$ for puts or $\max(S[j] - K, 0)$ for calls, representing the immediate exercise value.

4.1 Naive GPU-acceleration

The sequential algorithm has asymptotic time complexity $O(n^2)$ due to the nested loop in the backward induction phase. This poses a significant computational challenge as the number of periods increases. While nodes at successive time points form a clear dependency chain, nodes at the same time point are independent and can be computed in parallel. This structure suggests substantial parallelization opportunities.

Kolb and Pharr [21] implemented a GPU-accelerated version using this approach. At each time point, they spawned one thread per node, letting each thread compute a single node from its two child nodes. Threads synchronized after each time point to prevent data races. The implementation read and wrote intermediate results into two alternating arrays. To maximize GPU utilization, the algorithm was run in parallel for “a thousand or so independent options”. The full source code is found in the original article[8].

The parallel implementation does not redesign the algorithm in any way, but simply utilizes more resources for parallel computation. The work of the algorithm is thus $O(n^2)$, identical to the sequential time complexity. On the other hand, the span is only $O(n)$, so linear run-time can be achieved with sufficient processors. Figure 8 illustrates this parallel execution. Using $n = 1024$ periods, the authors reported throughput as a function of the number of American options processed. The single-threaded CPU implementation achieved approximately 110 options/s. The GPU version initially underperformed for batches of fewer than 5 options, likely due to parallelization overhead, but scaled to approximately 1150 options/s with larger batches — a 10x speedup. The throughput plateau suggests either full GPU utilization or other bottlenecks.

For large lattices, the naive parallelization encounters synchronization bottlenecks. Threads must synchronize after each time point, creating overhead that grows with n. This overhead is particularly severe when the lattice exceeds the capacity of a single streaming multiprocessor, forcing communication via slower global memory rather than shared memory. Suo et al. [22] explored several strategies to mitigate this. A

[8]The implementation from 2005 predates CUDA, using the Cg language common for GPU programming at the time.

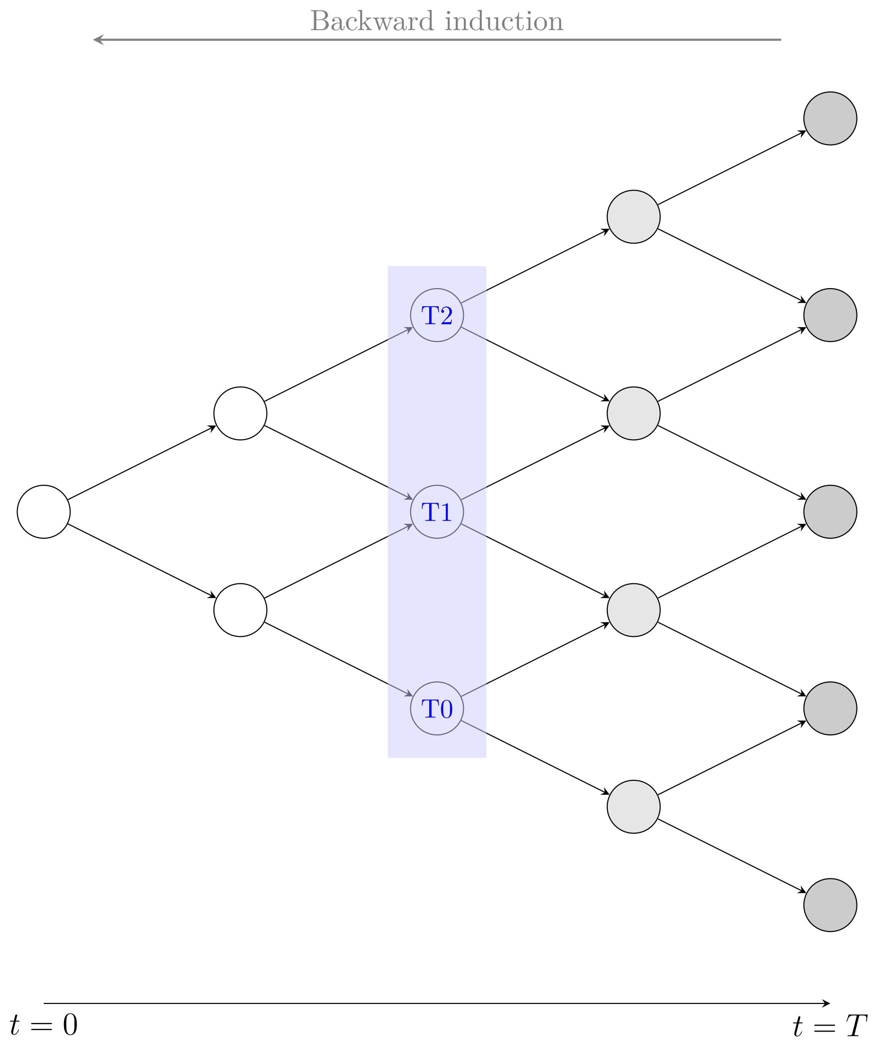


Figure 8: Naive GPU acceleration of the Cox-Ross-Rubinstein (CRR) algorithm. The binomial lattice structure reveals instruction dependencies across time. Nodes at each time point can be computed in parallel by independent threads (T0, T1, T2) during backward induction. Filled nodes are already computed, with the terminal nodes in dark gray.

naive attempt sacrificed parallelism by having a thread compute multiple nodes over several periods, thereby reducing synchronization frequency. This, however, led to redundant computation as nodes at intermediate time points must be calculated multiple times by different threads. More sophisticated approaches avoid this redundancy — the authors proposed a "triangle method" that carefully partitions the lattice to minimize both synchronization overhead and redundant calculations.

4.2 Additional Parallelization Across Time

Ganesan et al. [23] presented another approach that exploits parallelism across time, not just at individual time points. This is achieved by a reformulation of the problem that establishes direct relationships between non-adjacent nodes. Equation (4) relates the option values at time i to the values at time $i+1$. By repeatedly substituting this formula into itself and expanding, one obtains a general relationship between option values at time $i-m$ and those at time i:

$$F_{i-m}(j) = e^{-mr\Delta t} \sum_{k=0}^{m} c_k F_i(j+k) \tag{5}$$

where j indexes the nodes at time $i-m$, and the coefficients $c_k, 0 \leq k \leq m$ represent the (risk-neutral) probabilities of all paths from node $(i-m, j)$ to node $(i, j+k)$. These are simply binomial coefficients weighted by the appropriate powers of the probabilities: $c_k = \binom{m}{k} p^k (1-p)^{m-k}$. The key insight is that while the coefficients in this reformulation still exhibit the same sequential dependencies between successive time points, they can be computed independently for different segments of the lattice. Unlike the actual option values, there is no strict requirement for starting at the terminal nodes when calculating these coefficients.

Based on this insight, the authors presented the following parallel algorithm [23]: First, divide the n-period lattice into p partitions of length n/p. Each partition then computes coefficients relating the nodes at its left boundary to the ones at its right boundary. These independent operations proceed in parallel across all partitions. Once the coefficients are computed, the option values can be propagated from right to left across partition boundaries using Equation (5). This requires only p sequential steps rather than n[9]. Figure 9 illustrates the idea.

The total work remains $O(n^2)$ since every node must still be processed, either directly calculating the option value or the corresponding coefficient. The span, however, is further reduced from $O(n)$ to $O(n/p+p)$ due to the algorithmic reformulation. The corresponding DAG transforms from a tall, narrow lattice into a shorter, wider one. In the GPU implementation, partitions are computed on independent streaming multiprocessors. Communication between partitions is done through global memory, introducing latency L at each propagation step [23]. Increasing p improves execution time up to a certain point, where the communication latency and boundary calculation overhead outweighs the benefit of additional parallelism. The

[9]While the authors claimed this method extends to American options, they provided insufficient detail on how early exercise opportunities are evaluated when intermediate nodes are computed after the main propagation phase.

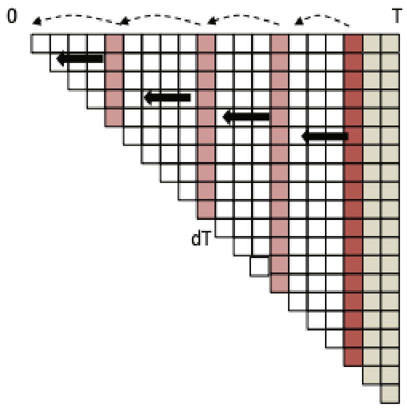


Figure 9: Partitioned Cox-Ross-Rubinstein (CRR) algorithm for parallel computation across time. The binomial lattice is divided into partitions. Within each partition, coefficients relating nodes at left and right boundaries are computed in parallel. Once coefficients are computed, option values propagate sequentially from right to left across partition boundaries, skipping many intermediate values. The rightmost partition can alternatively compute option values directly using standard backward induction, as is done in the figure. Reproduced with permission from the authors [23].

authors derived the optimal number of partitions as $p = \sqrt{2n/L}$ [23]. In practice, they achieve a 2x speedup over a naive GPU implementation. In clear contrast to bulk pricing, this approach dedicates all computing resources to an individual option, making it suitable for pricing of a single or small group of key portfolio assets.

5 GPU Acceleration of Monte Carlo Methods

Monte Carlo (MC) methods are a class of computational algorithms that approximate solutions to problems through random sampling. Unlike analytical solutions and deterministic numerical methods, both of which rely on domain-specific mathematical theory, MC methods are relatively simple and broadly applicable across many domains. The theoretical foundation is the *law of large numbers*, which states that the sample average tends toward the true expected value with increasing sample size [24]. Therefore, with a well-chosen sampling distribution whose expectation matches the target quantity, the average of a large number of simulated outcomes provides a reliable approximation of the solution.

While true random number generation is difficult, there exist many so-called pseudorandom number generators (RNGs) whose deterministic sequences are statistically indistinguishable from true randomness. Most RNGs first generate samples from a uniform distribution $U(0, 1)$, and then apply a transformation to map these to the desired distribution. The Box-Muller transform, for instance, converts uniform samples to normally distributed values: $U(0, 1) \rightarrow N(0, 1)$ [25].

Many software libraries provide RNGs for various distributions. Combined with affordable computing power, this has driven widespread adoption of MC methods across different domains. However, their generality comes at a significant computational cost. While analytical and deterministic methods exploit problem-specific structure for faster convergence, MC methods typically require many iterations to achieve accuracy. Nevertheless, MC methods excel at high-dimensional problems where traditional approaches fail or become exceedingly slow [25]. In derivatives pricing, MC methods are mostly applied to exotic options with complex payoff structures, for example basket options whose payoffs depend on numerous underlying assets [5, 10].

MC option pricing relies on the same risk-neutral valuation framework presented in Section 2.4. While the BSM model solves a partial differential equation analytically for the expected discounted payoff, the MC approach estimates this value by averaging many simulated option payoffs, each computed from a risk-neutral stock price path [26]. This estimate converges toward the theoretical value as the number of simulations increases.

MC simulation typically models stock price dynamics using a discretized version of the *geometric Brownian motion* process assumed by the BSM model. This is also the stock price dynamics of the CRR model in the limit as $n \rightarrow \infty$ [19]. Under these assumptions, one can solve for the exact distribution of the stock price at any future time point, and thus the discretization does not introduce an error [20]. After each period, the stock price is updated according to

$$S_{t+\Delta t} = S_t \cdot \exp\left(\nu \Delta t + \sigma \sqrt{\Delta t} \cdot \epsilon_i\right),$$

where $\nu = r - \frac{1}{2}\sigma^2$ is a deterministic growth component and $\epsilon_i \sim N(0, 1)$ is a standard normal variable for random price fluctuations [20]. Hence, there is no need for more than one period unless the payoff is path dependent. Asian options, for example, have payoffs based on mean stock prices over the holding period. In this case, much like the CRR model, a better approximation is achieved with a smaller time step. Algorithm 2, based on Clewlow and Strickland [20], demonstrates a generic implementation of MC option pricing. Simulating m paths of length n and evaluating a payoff with cost p yields a time complexity of $O(m \cdot (n + p))$. For most payoffs $p = O(1)$ as even path-dependent payoffs can often be computed incrementally during path generation. For European options where $n = 1$, this reduces to $O(m)$.

Algorithm 2: MC Option Pricing

```
Input: $S_0$ (initial stock price), $K$ (strike price), $T$ (time to maturity), $r$
       (risk-free rate), $n$ (periods), $\sigma$ (volatility), $M$ (paths), option type,
       payoff function
Output: Option value at $t = 0$
1  $\Delta t \leftarrow T/n$;
2  $\nu \leftarrow r - 0.5 \cdot \sigma^2$;
3  nudt $\leftarrow \nu \cdot \Delta t$;
4  sigsqrt $\leftarrow \sigma \cdot \sqrt{\Delta t}$;
5  $C_{\text{sum}} \leftarrow 0$;
   // Simulate $M$ stock price paths
6  for $m \leftarrow 1$ to $M$ do
7      $S[0] \leftarrow S_0$;
       // Generate price path over $n$ periods
8      for $i \leftarrow 1$ to $n$ do
9          Sample $\epsilon \sim \mathcal{N}(0, 1)$;
10         $\ln S[i] \leftarrow \ln S[i-1] + \text{nudt} + \text{sigsqrt} \cdot \epsilon$;
11         $S[i] \leftarrow \exp(\ln S[i])$;
12     end
       // Compute payoff for this path
13     $C_m \leftarrow \text{payoff}(S[0..n], K)$;
14     $C_{\text{sum}} \leftarrow C_{\text{sum}} + C_m$;
15 end
   // Average and discount payoffs
16 $C_0 \leftarrow e^{-rT} \cdot \frac{C_{\text{sum}}}{M}$;
17 return $C_0$;
```

5.1 Naive GPU-acceleration

MC simulation is considered an *embarrassingly parallel* problem — each path is independent, allowing straightforward parallelization without further redesign. By a similar argument as earlier, the work of the algorithm therefore equals the sequential time complexity $O(m \cdot (n + p))$. The span is $O(n + p)$, assuming an infinite-processor machine that can simulate all m paths in parallel.

Instead, the primary challenge becomes generating independent random numbers across threads. Naive parallelization can produce correlated number sequences between threads, violating the independence assumption of MC simulation. Effective implementations must therefore partition the RNG state space to ensure statistical independence across all threads [25]. Early GPU implementations, like the one by Howes and Thomas [27], had to manually construct such generators. Today, GPU libraries like cuRAND [28] provide parallel RNGs as standard functionality, eliminating the need for manual implementation.

Howes and Thomas [27] benchmarked GPU-accelerated MC option pricing using an Nvidia GeForce 8 GPU and quad-core AMD Opteron 2.2 GHz CPU. They achieved a 26x speedup for random number generation alone, with overall speedups of 59x for Asian options and 23x for a variant of lookback options. These results show that even fairly naive GPU implementations of the early days (2007) achieved substantial speedups over multi-core CPU versions, despite the inherent challenges of parallel random number generation. When isolating the simulation-related computations (using constants instead of random numbers), speedups reached 118x and 45x respectively. This indicates that random number generation becomes the dominant bottleneck in GPU implementations.

The choice of random number generator often depends on the option type and memory constraints, as some option payoffs require storing intermediate values in memory. In this example, the sum for the mean calculation in the Asian option can be accumulated in a single variable, leading to minimal memory requirements. This enabled the use of the faster Wallace RNG, which could not be used in the pricing of more memory-intensive variant of lookback options that needed to store entire price paths.

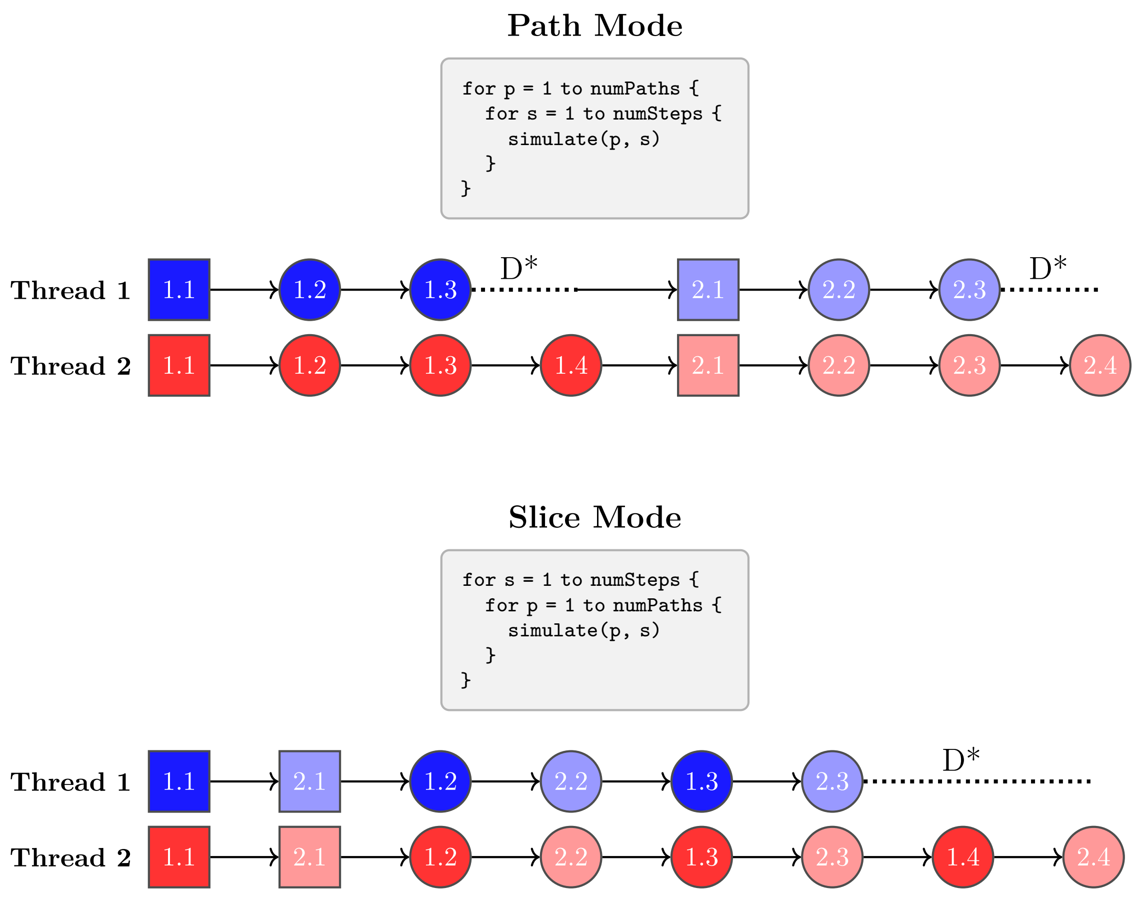
5.2 Further Hardware Optimization

Unlike the CRR algorithm, where reformulation was necessary to expose hidden parallelism, MC simulation presents a fundamentally different optimization challenge. The parallelism is already fully exposed at the algorithmic level — each simulation path is independent, requiring no communication or synchronization with other paths. This embarrassingly parallel structure means that further speedups must instead come from efficiently mapping this parallelism onto GPU hardware. While variance reduction techniques and quasi-random number generation can improve convergence rates [20], they benefit all implementations equally and are thus not relevant. Instead, GPU-specific optimizations focus on mitigating hardware constraints: efficient random number generation, memory access patterns and resource utilization.

Liu et al. [29] demonstrated several hardware-specific optimizations for GPU-accelerated MC option pricing, focusing on memory access inefficiencies and resource underutilization. The first optimization was to compress the working set, that is the amount of memory each thread must keep track of during simulation, to minimize expensive global memory access. The extent to which this was achievable heavily depended on the option type and choice of RNG. Memory bandwidth bottlenecks were addressed through coalesced access patterns, where adjacent threads access contiguous memory locations that can be fetched simultaneously. Register pressure presented another challenge. GPUs have limited registers per streaming multiprocessor, so implementations requiring many registers per thread reduce the number of concurrent threads. The authors proposed splitting the work into smaller GPU kernels[10] responsible for separate sub-tasks. While this reduces register pressure, it comes at a cost of additional global memory transfers between kernels. Together, these optimizations achieved a 43x speedup for exotic option pricing on an Nvidia Tesla C1060 GPU compared to an Intel 8-core Xeon 2.0 GHz CPU.

The authors also examined two different execution strategies for large workloads, where a single thread computed multiple stock price paths. In "path mode", the thread computed its assigned paths sequentially from start to finish. This minimized memory usage and suited homogeneous workloads, like simulating paths for the same option. In "slice mode", the thread instead simulated one step at a time for all its price paths. This was more memory intensive as intermediary data was stored for all the paths. On the other hand, slice mode handled heterogeneous workloads better, for example simultaneous pricing of different options. If these options have different maturities, some path simulations will finish earlier than others, leading to warp divergence [29]. Slice mode organized the computations to minimize the associated overhead cost. Figure 10 depicts the different execution modes and their effect on divergence. While GPU architectures have evolved significantly since the paper was published in 2010, the underlying tradeoffs between memory usage and thread utilization remain central to high-performance MC implementations.

[10]A GPU kernel is a function executed in parallel by many threads



```
for p = 1 to numPaths {
  for s = 1 to numSteps {
    simulate(p, s)
  }
}
```

```
for s = 1 to numSteps {
  for p = 1 to numPaths {
    simulate(p, s)
  }
}
```

Figure 10: GPU warp divergence under heterogeneous workloads in "path" versus "slice" execution modes, as described by Liu et al. [29]. Nodes representing simulation steps are labeled (p, s) where p is the path index and s is the step index. D^* denotes warp divergence. In path mode, threads complete entire paths sequentially, causing divergence when the paths differ in length. In slice mode, threads execute the same step across all paths before proceeding, resulting in a single divergence section at the end of the workload. While total idle time is theoretically the same, path mode's multiple divergence sections create more overhead, making slice mode more efficient for heterogeneous workloads.

6 Summary and Conclusions

This thesis evaluates the effectiveness of GPU acceleration for two widely used option pricing approaches: The Cox-Ross-Rubinstein (CRR) algorithm and Monte Carlo (MC) simulation. The analysis examines their parallelization potential, implementation strategies, achievable speedups and performance bottlenecks. The data confirm that both algorithms benefit significantly from GPU acceleration, though their inherent characteristics dictate different optimization approaches and performance gains. The results of the reviewed research are aggregated in Table 2.

The algorithms' structures explain their initial GPU performance. The CRR model's backward induction creates dependencies across time points, limiting parallelization to computations within each time point. By contrast, MC simulation computes independent stock price paths, making it ideally suited for parallel execution. While direct comparison is complicated by differences in GPU hardware, the reported results are consistent with this distinction: the naive CRR-GPU implementation by Kolb and Pharr achieved an approximately 10x speedup over a single-core CPU [21], whereas the MC-GPU implementation by Howes and Thomas reported 23-59x speedups despite being benchmarked against a quad-core CPU [27]. The disparity can likely be attributed to synchronization overhead affecting the CRR algorithm. Nonetheless, both naive implementations yield substantial improvements relative to implementation effort, making GPU acceleration worthwhile for either algorithm.

Given these characteristics, further optimization strategies take different forms. The CRR model benefits from algorithmic reformulation to expose hidden parallelism, as demonstrated by Ganesan et al. [23] who partitioned the time axis to achieve a 2x improvement over a naive GPU implementation. MC methods already expose the available parallelism and benefit more from hardware-specific optimizations. Liu et al. [29] achieved a 43x speedup over a CPU baseline through techniques including memory coalescing and minimizing warp divergence. Ultimately, both algorithms face scalability limits: CRR implementations contend with synchronization or communication overhead, while MC simulation remains constrained by memory bandwidth, particularly when using memory-intensive random number generators.

While the results suggest MC methods may achieve larger GPU speedups, this does not make them the optimal solution for option pricing. In practice, the choice of algorithm depends on multiple factors: option type, available hardware, and other case-specific requirements and constraints. For instance, vanilla options are rarely priced using MC methods since analytical or lattice-based models often provide faster solutions. The practical approach is to first select an appropriate pricing model and then apply GPU acceleration. This becomes particularly valuable for high-throughput applications like portfolio-wide risk calculations or bulk pricing operations.

An important caveat is that the analyzed research spans 2005-2010. Cutting-edge research often remains proprietary within financial institutions that safeguard their competitive advantage, making it difficult to review the most recent developments in the field. GPU architectures have also evolved dramatically since then, largely driven by the massive demand for AI computing. Modern GPUs feature improved memory hierarchies, better divergence handling, and enhanced support for general-purpose computing, mitigating several of the mentioned bottlenecks while perhaps introducing new ones. Likewise, modern multi-core CPUs provide faster baselines than those in the original studies. Despite these hardware advances, the fundamental principle persists: effective GPU utilization requires both clever algorithmic design and hardware-conscious optimization. Future work could benchmark these algorithms on current architectures to enable direct comparisons. Additional research opportunities include GPU acceleration of other models, such as finite difference or machine learning-based methods.

Table 2: Empirically measured GPU speedups for selected option pricing models. Reported results are based on real implementations with varying baseline architecture and research methodologies. While not directly comparable, they provide indicative insight into performance improvements when interpreted with appropriate caution.

Algorithm	**Reported Speedup**	**Strategy**	**Bottleneck**
CRR			
Naive	10x[1]	Parallelization at time points	Synchronization latency between nodes
Partitioned	2x[2]	Additional parallelization across time periods	Inter-partition communication
MC			
Naive	23–59x[3]	Parallel simulation paths	Random number generation, memory constraints
Optimized	43x[4]	Memory coalescing, register reuse, warp divergence minimization	Memory bandwidth

[1] NVIDIA GeForce 6800 Ultra vs single-core 2.0GHz AMD Athlon 64 3200+ CPU (2005) [21].
[2] NVIDIA Tesla C1060 vs CRR Naive on the same architecture (2009) [23].
[3] NVIDIA GeForce 8 vs 4-core 2.2GHz AMD Opteron CPU (2007) [27].
[4] NVIDIA Tesla C1060 vs 8-core Intel Xeon 2.0GHz CPU (2010) [29].

References

[1] M. J. McGowan, “The rise of computerized high frequency trading: Use and controversy,” *Duke L. & Tech. Rev.*, vol. 9, p. 1, 2010.

[2] D. MacKenzie, “Material signals: A historical sociology of high-frequency trading,” *American Journal of Sociology*, vol. 123, no. 6, pp. 1635–1683, 2018. doi: 10.1086/697318 .

[3] R. C. Merton, “Influence of mathematical models in finance on practice: past, present and future,” *Philosophical Transactions of the Royal Society of London. Series A: Physical and Engineering Sciences*, vol. 347, no. 1684, pp. 451–463, 1994. doi: 10.1098/rsta.1994.0055 .

[4] W. D. Nordhaus, “Two centuries of productivity growth in computing,” *The Journal of Economic History*, vol. 67, no. 1, pp. 128–159, 2007. doi: 10.1017/S0022050707000058 .

[5] J. C. Hull, *Options, futures, and other derivatives /*, ninth edition. ed. Harlow:: Pearson Education Limited, c2018., ISBN:978-1-292-21289-0.

[6] S. M. Bartram, G. W. Brown, and F. R. Fehle, “International evidence on financial derivatives usage,” *Financial management*, vol. 38, no. 1, pp. 185–206, 2009. doi: 10.1111/j.1755-053X.2009.01033.x .

[7] N. Gisiger, “Risk-neutral probabilities explained,” 2010. doi: 10.2139/ssrn.1395390 .

[8] J. Tham, “Risk-neutral valuation: A gentle introduction (1),” *Available at SSRN 290044*, 2001. doi: 10.2139/ssrn.290044 .

[9] The Royal Swedish Academy of Sciences, “The Sveriges Riksbank Prize in Economic Sciences in Memory of Alfred Nobel 1997 — Press Release,” Oct. 1997, accessed: July 25, 2025. [Online]. Available: https://www.nobelprize.org/prizes/economic-sciences/1997/press-release/

[10] P. Wilmott, *Paul Wilmott on quantitative finance.* John Wiley & Sons, 2013, ISBN:978-1-118-83683-5.

[11] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming.* Addison-Wesley Professional, 2010, ISBN:978-0131387683.

[12] D. B. Kirk and W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach.* Morgan Kaufmann, 2016, ISBN:9780128119860.

[13] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach.* Elsevier, 2011, ISBN:978-0123704900.

[14] J. Suomela, “Programming parallel computers,” Course lecture slides, Aalto University, n.d., accessed: July 2, 2025. [Online]. Available: https://ppc.cs.aalto.fi/index/

[15] W. Stallings, *Operating systems: internals and design principles.* Prentice Hall Press, 2011, ISBN:9780132309981.

[16] NVIDIA Corporation, *CUDA C++ Programming Guide*, NVIDIA, 2025, version 12.x, accessed July 2, 2025. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[17] ENCCS and individual contributors, “Gpu programming: Ecosystem and architecture,” https://enccs.github.io/gpu-programming/2-gpu-ecosystem/, 2025, licensed under CC-BY 4.0.

[18] U. A. Acar, “Parallel computing: Theory and practice,” https://www.cs.cmu.edu/afs/cs/academic/class/15210-f15/www/tapp.html, 2016, version 1.0, Accessed: July 20 2025.

[19] J. C. Cox, S. A. Ross, and M. Rubinstein, “Option pricing: A simplified approach,” *Journal of financial Economics*, vol. 7, no. 3, pp. 229–263, 1979. doi: 10.1016/0304-405X(79)90015-1 .

[20] L. Clewlow and C. Strickland, “Implementing derivative models,” *Jonh Wiley & Sons, Nonsei*, 1998, ISBN:9780470328965.

[21] M. Pharr and R. Fernando, *GPU Gems 2: Programming techniques for high-performance graphics and general-purpose computation (gpu gems).* Addison-Wesley Professional, 2005, ISBN:0321335597.

[22] S. Suo, R. Zhu, R. Attridge, and J. Wan, “Gpu option pricing,” in *Proceedings of the 8th Workshop on High Performance Computational Finance*, 2015, pp. 1–6. doi: 10.1145/2830556.2830564 .

[23] N. Ganesan, R. D. Chamberlain, and J. Buhler, “Acceleration of binomial options pricing via parallelizing along time-axis on a gpu,” in *Proceedings of Symposium on Application Accelerators in High Performance Computing*, 2009.

[24] S. M. Ross, *Introduction to Probability and Statistics for Engineers and Scientists (Sixth Edition)*, 6th ed. Academic Press, 2020. doi: 10.1016/C2018-0-02166-0 .

[25] J. E. Gentle, *Random number generation and Monte Carlo methods.* Springer, 2003, vol. 381, ISBN:978-0387001784.

[26] P. P. Boyle, “Options: A monte carlo approach,” *Journal of financial economics*, vol. 4, no. 3, pp. 323–338, 1977. doi: 10.1016/0304-405X(77)90005-8 .

[27] H. Nguyen, *Gpu gems 3.* Addison-Wesley Professional, 2007, ISBN:9780321515261.

[28] NVIDIA Corporation. (2025) cuRAND documentation. [Online]. Available: https://docs.nvidia.com/cuda/curand/index.html

[29] L. Liu, Y. Zhang, L. Liu, G. Yang, and W. Zheng, "Efficient monte carlo-based options pricing on graphics processors and its optimizations," *Science China Information Sciences*, vol. 53, pp. 1703–1712, 2010. doi: 10.1007/s11432-010-3109-7 .

A Derivation of Risk-Neutral Probability Form

In this appendix, I derive the risk-neutral probability form of the one-period binomial model option pricing formula.

Starting from (3):

$$\begin{aligned}
&\frac{f_u - f_d}{S_0(u-d)} S_0 + \frac{f_d \cdot u - f_u \cdot d}{(u-d)(1+r)} \\
&= \frac{f_u - f_d}{u-d} + \frac{f_d \cdot u - f_u \cdot d}{(u-d)(1+r)} \\
&= \frac{(f_u - f_d)(1+r)}{(u-d)(1+r)} + \frac{f_d \cdot u - f_u \cdot d}{(u-d)(1+r)} \\
&= \frac{f_u(1+r) - f_d(1+r) + f_d \cdot u - f_u \cdot d}{(u-d)(1+r)} \\
&= \frac{f_u[(1+r)-d] + f_d[u-(1+r)]}{(u-d)(1+r)} \\
&= \frac{1}{1+r} \cdot \frac{f_u[(1+r)-d] + f_d[u-(1+r)]}{u-d} \\
&= \frac{1}{1+r} \left(f_u \cdot \frac{(1+r)-d}{u-d} + f_d \cdot \frac{u-(1+r)}{u-d} \right)
\end{aligned}$$

This reveals the risk-neutral probabilities: $\tilde{p} = \frac{(1+r)-d}{u-d}$ and $\tilde{q} = 1 - \tilde{p} = \frac{u-(1+r)}{u-d}$. The no-arbitrage condition ensures $\tilde{p}$ and $\tilde{q}$ are correctly bounded by 0 and 1. If $1 + r > u$, investors could profit by shorting the stock and investing in the risk-free asset. Conversely, if $d > 1 + r$, they could earn guaranteed profits. Therefore, $d \le 1 + r \le u$ must hold, which leads to $0 \le \tilde{p}, \tilde{q} \le 1$. Furthermore, $\tilde{p} + \tilde{q} = 1$. Thus, our risk-neutral probabilities form a valid probability distribution over the two possible outcomes.