Master Thesis

Accelerated Interest Rate Option Pricing using Trinomial Trees

Authors:

Marek Hlava

Martin Metaksov

Supervisors:

Cosmin Oancea

Wojciech Michal Pawlak

University of Copenhagen

Copenhagen, Denmark

August 2018

Abstract

In this thesis we describe different parallelization strategies for the Hull-White Single-Factor Model that prices financial derivatives using the trinomial trees numerical method. The work is focused on finding the middle ground between the distinct levels of parallelism and trade-offs such as thread divergence vs. locality of reference, by applying various optimization techniques and transformations.

First, we will present a sequential solution, used to validate all parallel implementations later through the project.

Second, we will present a one-option per thread implementation, which will only deal with outer parallelism.

Third, we will present a multiple options per thread block implementation, aiming to exploit both levels of parallelism.

Fourth, we will present a fully-flattened implementation, which will put emphasis on the importance of finding the middle ground between parallelization trade-offs.

Finally, we will present a number of experiments conducted to help explore the performance impacts by each implementation and optimization previously presented. This empirical validation will be used to pinpoint the implementations with highest performance on each different dataset.

Keywords: Option pricing, Trinomial trees, Hull-White Single-Factor Model, Parallel programming, Flattening, GPGPU, CUDA, C/C++, Futhark

Acknowledgments

First and foremost, we would like to thank Cosmin E. Oancea and Wojciech M. Pawlak for their supervision and dedication to this project. For all the guidance on financial terminology, parallel transformations, CUDA programming and all related topics we have discussed. For the number of hours each of them have spent thoroughly reading this thesis and guided us in improving it.

We would also like to thank Troels Henriksen for his quick responses on GitHub and his help with Futhark.

Thank you.

Contents

1

Introduction

After over 50 years of miniaturization with the pace of Moore's law[25], transistor shrinking is reaching its limits[1], and the increase in CPU-clock speed has halted for a while now. Until quantum computers become a reality, or another hardware technology emerges, computation speed-ups will be achieved by scaling hardware parallelism—for example, current many-core architectures already support tens of thousands of hardware threads. In contrast to CPU-frequency scaling, which directly resulted in increased performance of the unmodified (sequential) program, unlocking the power of massively-parallel architectures presents significant challenges.

First, it requires the programmer to "think parallel", for example, to reason about the parallel nature of recurrences (loops) appearing at various nested levels, and then to make the parallelism semantics explicit in the program—ideally by means of parallel-language constructs rather than by unsafe directives such as OpenMP.

Second, while arguably, programs are (more) naturally expressed by combining parallel constructs at the same level and at various levels in a nest (a.k.a., nested parallelism), many-core architectures have little support for dynamic parallelism and morally exploit only flat parallelism. It follows that "someone" has to rewrite the nested-parallel program into a semantically-equivalent one that uses only flat-parallel constructs. Ideally, the compiler is the one that performs this translation (automatically), but unfortunately, current programming-model and compiler technology is far from effectively and reliably supporting real-world applications. It follows that in many cases, it is the programmer who needs to perform this rewriting by hand in addition to applying other (compiler) techniques aimed at optimizing locality of

[1] A prediction by the 2015 International Technology Roadmap for Semiconductors (ITRS)

reference, thread divergence, etc. In essence, efficiently porting by hand a complex application to modern hardware often breaks code modularity/maintainability and requires the programmer to have expert knowledge in compiler analysis. Furthermore, the mainstream parallel APIs are rather low-level, and thus very tedious to program with (e.g., OpenCL, CUDA are the "parallel assembly" of our time).

Third, to make matters worse, the optimization recipe is often sensitive to the dataset [1], i.e., optimal performance requires several semantically-equivalent code versions, each tailored to the particularities of a class of datasets. One potential driver in the process of code-version generation is the degree of parallelism that is actually mapped to the hardware. Common wisdom says that outer parallelism is more beneficial (than inner), so in principle, one should exploit as many levels of outer parallelism as there are needed to fully utilize the hardware, and should sequentialize the rest. This strategy also benefits locality of reference, because sequentializing inner parallelism may enable tiling opportunities. For example:

- The common implementation of dense matrix-matrix multiplication utilizes only the outer two levels of parallelism and it sequentializes and tiles the (innermost) dot product operation, which results in a compute-bound performance behaviour. However, if not enough parallelism is available in the outer levels, then it is necessary to also exploit the dot product parallelism, albeit resulting in memory-bound behavior.

- In the case of sparse-matrix vector multiplication, the computation can be carried out in multiple ways: by using row-wise decomposition, column-wise decomposition, or Checkboard decomposition. The performance of each depends on the distribution and the skewness of the dataset and can lead to other possible trade-offs, such as locality of reference vs. load balancing (thread divergence).

This thesis explores the space of optimization techniques aimed at efficiently porting to GPGPU hardware[2] of a difficult-to-parallelize, real-world application from the financial domain, namely option pricing by means of trinomial trees. This application is particularly challenging because:

- it uses irregular nested parallelism, which is notoriously difficult to map efficiently to hardware, and
- it exhibits thread-divergence on two (orthogonal) levels, which are difficult to optimize at the same time, but leaving either one unoptimized may significantly degrade performance.

An interesting finding of this thesis is that the aforementioned "common wisdom" is only partly correct. It still holds that sequentializing the inner-parallelism in excess (to the ones needed to saturate hardware) is the right thing to do when the divergence is randomly distributed — even though this method necessarily leaves one level of divergence unoptimized. However, when the distribution of divergence is skewed on both levels, it is better to exploit all parallelism because this allows to optimize all divergence.

Finally, although all discussed optimization techniques have been implemented manually using CUDA programming, we believe that this thesis may provide useful insights into how to integrate such code transformations in the repertoire of an optimizing compiler.

[2]General-purpose computing on graphics processing units

1.1 Problem Statement

To succeed on the market, financial organizations strive for higher performance in the tools they use on daily basis. Such examples are financial organizations managing investment portfolios with large number of assets, or financial software providers developing applications for pricing and risk management [3]. This has led to an increased interest in HPC [4] solutions, efficiently exploiting hardware parallelism. The Hull-White Single-Factor Model is one of the financial models widely used by financial organizations to simulate random changes in interest rates in order to price derivatives. This model can be implemented using trinomial trees [18, pg. 444], [21] - a generic numerical method known for its higher accuracy and stability compared to other popular models used for this purpose (e.g. binomial trees). While this method allows for the precise estimation of option prices, it is extremely expensive computationally, making it an interesting candidate for parallelization. This thesis studies several optimization recipes used to implement the Hull-White Single-Factor Model using trinomial trees, and it also studies how to combine the resulted code versions into one program that offers high performance across all classes of datasets. The main questions this thesis will try to answer are:

What are the performance trade-offs for parallelizing the Hull-White Single-Factor Model on modern massively parallel hardware and which optimization techniques can yield performance benefits?

Which techniques for parallelism optimization work best for the different data classes and how do we combine all parallel versions into one program that provides high performance on all data sets?

[3] SimCorp is one such organization trying to investigate and attempt to improve the core pricing functionalities in its product, Simcorp Dimension, by using various parallelization techniques to review the implementations of pricing models used by their clients.

[4] High performance computing

1.2 Birds-Eye View of our Approach

The thesis will begin with the creation of a proof-of-concept sequential implementation of the model. This will be done in accordance to description provided in Options, Futures, and Other Derivatives [18] and will be carried out in C++. Validation will be done only on single callable European zero-coupon bonds, which is the most basic data input to work with. The rationale for studying the financial background of the algorithm is that it would enable domain-specific optimizations in early design stages.

The algorithm is concerned with the creation of a trinomial tree, with the shape shown on Figure 1.1. The tree is built in two passes, referred to as forward and backward propagation. Forward propagation represents the construction of a term structure for the underlying asset by progressing one time step at a time. Backward propagation discounts the asset prices to estimate an option payoff at maturity by going back through the tree. The price of the derivative is estimated by the value obtained all the way back at the root.

As long as optimizations are concerned, the most important properties of the constructed trees are their widths and heights, which are clearly denoted on Figure 1.1. Both computation steps have a similar structure: the height dimension represents a time series, and is implemented as a sequential loop, because the computation of a certain timestep (level) in the tree depends on the values computed in the previous timestep. However, the computation along the width axis can be performed in parallel, i.e., for a certain timestep, all points at the same level in the tree can be computed with a map operation. It is straightforward to observe that the number of computations performed for a given option depends strongly on both the width and the height of the tree corresponding to that option. This thesis will be concerned with pricing a large number of derivatives (batches) in parallel, as efficiently as possible. Batch pricing exhibits two nested levels of parallelism: one across options, and one on the width dimension of each option tree. Furthermore, this nested parallelism is irregular, because the heights and widths may vary wildly across options.

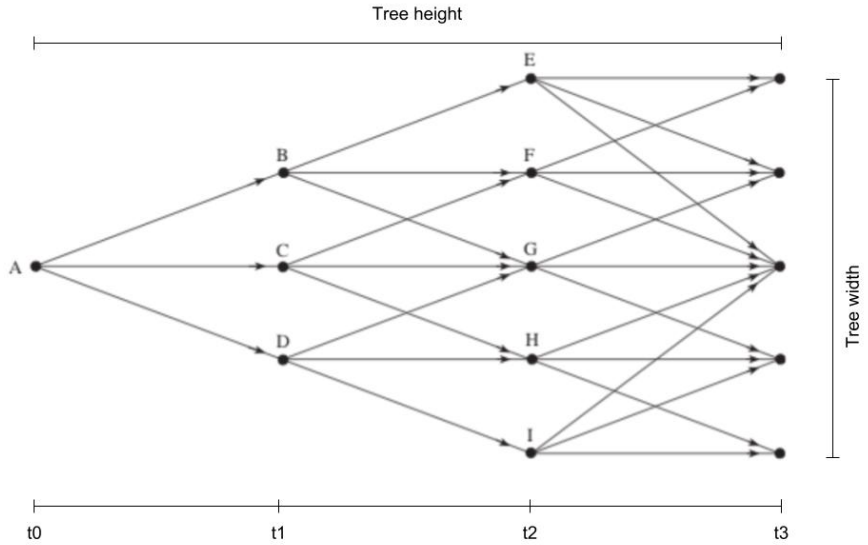


Figure 1.1: Example of a trinomial tree constructed for the Hull-White Single-Factor Model, illustrating its width and height.

Source: Modified by the authors, based on Options, Futures and Other Derivatives[18, pg. 699].

The divergence of widths and heights in a dataset presents several important challenges, but also many optimization opportunities. One of the challenges is to investigate which of the two nested-levels of application parallelism should be mapped to hardware - is it only the outer one or both?

Exploiting only the outer parallelism is most suited (i) when the batch is large enough to fully utilize hardware parallelism and (ii) when all widths and heights are equal. However, when the heights and widths are highly variant across options, this approach suffers from significant thread-divergence overhead (think load imbalance). It is possible to optimize one of the two divergence factors but not both, for example by sorting the options by their widths (or heights) in a descending order.

The alternative is to exploit both levels of parallelism. This has the advantage

that it naturally optimizes the width divergence, while the height divergence can be similarly optimized by sorting. The problem is that efficient execution requires that the flattening of the two levels of parallelism is performed on bins of options, whose summed widths fit the size of the CUDA block. This is because the flattening transformation introduces many parallel-prefix sums, map, and scatter operations, which would be prohibitively-expensive unless they are performed in fast (scratch-pad) memory, which is available only at CUDA-block level. On the other hand, this fast memory, is a very scarce resource. Increasing its footprint may result in decreasing the occupancy of the hardware. In essence this demonstrates an important performance trade-off: optimizing thread divergence may ultimately lead to a decline in hardware occupancy (or locality), i.e., "there is no free lunch".

The first parallel approach examined in this thesis refers to the version that exploits only outer parallelism, i.e., it computes a batch of options in parallel, but the pricing of an option is carried out sequentially inside one thread. The challenges for this version are caused by the irregularity of the widths and the heights; they specifically refer to (i) ensuring coalesced accesses to global memory, without wasting too much memory due to padding, and (ii) reducing the thread-divergence overhead by sorting the options based on their widths or heights. The implementations based on this approach have been described in detail in chapters 5, 7 and will be referred to as *CUDA-option* and *Futhark-basic* respectively.

The second parallel approach exploits both levels of parallelism, by packing multiple options and running them in parallel at CUDA-block level. The number of options that can be priced per block depends on their constructed tree widths, whose total sum must not exceed the maximal CUDA block size of 1024. This implies two things: First, this version is not suitable for pricing options with widths larger than 1024, as those cannot be packed into a CUDA block. Second, divergence across the width axis is implicitly optimized by bin-packing options at CUDA-block level. The main difficulty with this implementation is finding the right heuristic for flattening the nested parallelism. For example, the classical full-flattening approach [3], while general, is not applicable because it would generate too many arrays located in fast memory, which is a sparse resource. Similar to the previous approach, we can use

coalescing, padding and sorting to improve the performance. Exploiting both inner and outer parallelism should help discover performance benefits on some data sets, e.g. the ones which present skewed distributions for both widths and heights. This implementation is covered in detail in chapter 6 and will be referred to as *CUDA-multi*.

The last parallel approach explores the effects of the classical full flattening [3]. In contrast to the first two versions which have been implemented in CUDA, this version has been flattened out by hand in Futhark [17], a high-level data-parallel functional language. This approach is similar to the second one, except that all arrays are allocated in slow (global) memory rather than in fast (shared) memory. As such, this version is expected to yield significantly degraded performance in comparison to the remaining two. There is still a narrow niche for it: when the batch size is small and some widths are larger than the block size (then the other two approaches are inefficient or not applicable). However, the real purpose of this implementation is to further emphasize the trade-off between locality of reference and thread divergence: this version is by far the slowest in a large majority of cases, albeit its thread divergence is optimal. This implementation is described in more details in chapter 7 and will be referred to as *Futhark-flat*.

The thesis will also perform a systematic evaluation of the capabilities of all implementations, by generating and testing them on various datasets that exhibit different distributions of divergence. This will be done in an attempt to study the performance trade-offs on different data and to discover the impact of possible optimization techniques by empirical validation.

As an empirical evaluation, we will test all implementations on 7 distinct datasets (some of which are random, others skewed). We will show that *CUDA-option* dominates the benchmarks on most datasets (except on the skewed ones), reaching up to ~529× speed-up compared to the sequential implementation. The skewed dataset on the other hand benefits most from *CUDA-multi*, providing up to ~2× speed-up over *CUDA-option*. Furthermore, we will demonstrate that *CUDA-multi* outperforms *CUDA-option* by up to ~13× for floats and ~11× for doubles when the dataset is small enough.

1.3 Motivation and Relation to Related Work

This section briefly surveys related work with the goal of motivating the practical and scientific contributions of this thesis. A more detailed review can be found in Chapter 10. A rich body of scientific work has studied how to accelerate real-world, computing-starved applications by porting them to GPGPU hardware. Related solutions span several research directions: (i) building native GPGPU libraries, (ii) designing data-parallel languages and (iii) devising compiler transformations aimed at optimizing parallelism.

Many native GPGPU libraries have been already developed to support high-performance execution of commonly-used algorithms. Such examples are cuBLAS and cuDNN [27] that port linear-algebra and deep-learning algorithms, respectively, and a more recent effort of Mathworks and Nvidia that is aimed at accelerating Matlab libraries on Nvidia hardware. Similarly, in the financial domain, efforts have been aimed at accelerating risk modelling [1] and derivative pricing [29] using Monte-Carlo simulations. We argue that this thesis is of practical interest because, to our knowledge, there is no publicly-available GPGPU implementation of (batched) trinomial pricing.

The design of data-parallel hardware-independent languages have been a heavily scrutinized research topic. In this sense, many domain-specific languages (DSL) have been developed for accelerating image-processing pipelines [33], iterative stencils [41], data analytics [42], deep-learning and mesh computations [26, 39], or specific host-language constructs [23, 19, 40]. However, such DSLs do not typically support nested-composition of parallel constructs, which is the case of trinomial pricing. Finally, even though several data-parallel languages provide some support for nested parallelism [3, 17], they are not capable of expressing or deriving (at least) one of the two efficient implementations of trinomial pricing, which are the subject of this thesis.

Relation with compiler analysis. In this thesis we identify the important performance trade-offs for our application and solve them by hand, taking inspiration from related static and (more) dynamic analyses. As examples of applications of static analysis, we have optimized spacial locality (i.e., ensured memory coalescing) by working with arrays in transposed form [29] and we have performed loop fusion [32] to decrease the number of accesses to global memory. In the implementation of the code version that computes multiple options in one CUDA block, we have drawn inspiration from loop distribution [32] and Blelloch's flattening transformation [3]. This was necessary in order to rewrite the algorithmic specification, which exhibits two-level irregular parallelism, into a composition of flat-parallel constructs. As example of (more) dynamic analyses, we have used inspector-executor techniques [38, 31] to permute the order of the options in a way that minimizes the thread-divergence overhead, and we argue that a similar lightweight inspector can be used to derive a simple predicate that predicts the most suited version of the code for the current dataset. Finally, we argue that the manually-applied optimizations presented in this thesis (i) are likely applicable to other programs, and (ii) they provide useful insights into engineering the compiler infrastructure that would automate the optimization process.

1.4 Thesis Contributions

Scientific contributions: The main scientific contributions of this thesis are:

- Research on how the Hull-White Single-Factor Model works in practice and why it is useful
- Analysis of how the algorithm behind the Hull-White Single-Factor Model can be flattened to expose more parallelism
- Empirical validation that validates our claims and is supported by tables and plots
- Static and dynamic inspector/executor analysis of the model for performance optimization

Practical contributions: The main practical contributions of this thesis are:

- Multiple parallel implementations of the Hull-White Single-Factor Model in CUDA
- A one-option-per-thread parallel implementation of the Hull-White Single-Factor Model in Futhark
- A fully-flattened parallel implementation of the Hull-White Single-Factor Model in Futhark
- A well-structured project on GitHub containing all implementations with build instructions for each
- Data generator for testing edge cases and relevant data distributions

2

Background

This chapter will describe the essential terminology and techniques, laying down a foundation for the remaining chapters of this thesis. It is divided into four sections. First, we will cover fundamental financial concepts and methods essential for understanding the examined financial model and algorithm behind it. We will define all the parameters of the model that we implemented. We will proceed with a brief description of the CUDA parallel programming model — the main technology used in the project. Next, we will introduce the semantics and types of parallel operators we have used throughout the project and finally we will look into a definition of flattening parallel program transformations.

2.1 Financial Instruments

A *derivative* is a financial instrument that derives its value from and is dependent on the performance of some other basic underlying entity like asset, index, or interest rate. The most common underlying instruments include bonds, commodities, currencies, interest rates, market indexes and stocks. Over the last 40 years, various classes of derivatives like futures and options have grown in importance in finance being actively traded on daily basis in the markets all over the world. They significantly increase market efficiency and the transfer of risk in the economy. The derivatives market is much bigger (estimates go from $630 trillion to $1.2 quadrillion) than the market capitalization of the global stock markets ($73 trillion)[1]. Derivatives are mainly used for financial risk management as an insurance against rapid price

[1] http://www.visualcapitalist.com/worlds-money-markets-one-visualization-2017/

movements through hedging, increasing exposure to asset price movements through speculation or taking advantage of differences between two or more markets through arbitrage. [18, pg.1-18] This project is concerned with the valuation of financial derivatives.

2.1.1 Options

This thesis will focus on only one specific type of derivatives - *options*. These are contracts that give the holder the right to buy or sell an underlying asset at a certain point in time for a certain price, both specified when purchasing the option. This is in contrast with other derivatives - *forwards* and *futures*, where the holder is obligated to buy or sell the underlying asset. Another difference is where they are traded. Options and futures are standardized contracts traded on an exchange, while forwards are traded in the over-the-counter (OTC) market and can be customized for every transaction. [18, pg.22]

We identify two types of options. A *call option* gives the holder the right to buy the underlying asset by a certain date for a certain price. A *put option* gives the holder the right to sell the underlying asset by a certain date for a certain price. The price in the contract is called *strike price* and the expiration date is called *maturity*. Options that can be exercised at any time before maturity are known as *American options* and options that can be exercised only on the expiration date itself are known as *European options*. One contract usually allows to buy or sell 100 shares. [18, pg.7-8]

Option Example An investor spent 20.000 kr for an option to buy 100 Maersk shares for 9.600 kr each. The current market price for Maersk stock is 9.440 kr as of March 15, 2018. If the price does not rise above 9.600 kr by the maturity, the investor does not exercise the option and loses 20.000 kr. However, if Maersk stock is priced at 10.000 kr when the option can be exercised, the investor is able to buy 100 shares for the strike price of 9.600 kr and immediately sell them for 10.000 kr. This will generate a profit of $400 * 100 = 40.000$ kr minus the initial contract cost of 20.000 kr.

Table 2.1 illustrates an example of exercising this option at different dates. Even if the stock price rises above the strike price, the net profit might still be negative when the contract price is accounted for.

Table 2.1: Profit generated by a call option with strike price of 9.600 kr and contract price of $100 \times 200 = 20.000$ kr.

	March 2018	June 2018	Sept. 2018	Dec. 2018
Stock price (kr)	9.440	9.700	10.000	9.800
Share sale profit (kr)	not exercised	10.000	40.000	20.000
Net profit (kr)	−20.000	−10.000	20.000	0

2.1.2 Bonds and zero interest rates

Bonds are a form of debt that allow companies or governments borrow money from investors. Interest rate on an investment that starts today and lasts for n years are called n-year zero-coupon interest rates. All the interest and principal (known also as face value or par value) payments are realized at the bond maturity. For example, an investment of \$100 with a 5-year zero rate with *continuous* compounding at 5% p.a. grows to

$$100 \times e^{0.05 \times 5} = 128.40$$

In reality, most actively traded bonds pay in addition periodical coupons (interest) to the holder. [18, pg.80] However, we choose to deal with a most basic bond — a zero-coupon bond in this project, as coupon-bearing bonds introduce more complexity in valuation of their cash flows. We decide to do it for the reason of brevity, not to obstruct the main ideas behind the financial model that we implement. Moreover, in this project we deal with risk-free rates, interest rates that can be earned without assuming any risk, a common practice in derivative pricing. Another assumption we make is that we use continuous compounding frequency (as opposed to annual or monthly compounding) for measuring interest rates, an infinitesimally small compounding interval, that allows us to simplify interest rate calculations. [18, pg.76-79]

2.1.3 Yield Curve

Typically for most pricing models, the probability distributions of interest rates, bond prices, or other variables at a future point in time are lognormal. However, this does not provide any information of how interest rates evolve over time. [18, pg. 682]. This can be achieved by building a term structure model, also known as a yield curve. The term structure model describes the evolution of all zero-coupon interest rates as a function of maturity. [18, pg.83]

In practice, we do not usually have bonds with maturities equal to exactly 1.5 years, 2 years, 2.5 years, and so on. One approach is to use linear interpolation between the bond price data before it is used to calculate the zero curve. For example, if it is known that a 2.3-year bond with a coupon of 6% sells for $108 and a 2.7-year bond with a coupon of 6.5% sells for $109, it might be assumed that a 2.5-year bond with a coupon of 6.25% would sell for $108.5. In addition, it is also usually assumed that the yield curve is horizontal prior to the first point and horizontal beyond the last point. [18, pg.83]

For simplicity, in this thesis we have chosen to use a specific yield curve provided in Example 30.1 and Table 30.2 in [18, pg.706].

2.1.4 Black-Scholes model and formula

The Black-Scholes formula [8], is perhaps the world's most well-known options pricing model. The model has had a huge influence on the way that traders price and hedge derivatives [18, pg.299-324]. The main financial insight of the equation is that it allows to perfectly hedge an option by buying and selling the underlying asset and consequently eliminate the risk. This implies that there is only one right price for the option — computed by the formula. However using the Black-Scholes formula, assumes several simplifications:

- The option must be European and can only be exercised at expiration.
- No dividends are paid out during the life of the option.
- Markets are efficient (i.e., market movements cannot be predicted).

- There are no transaction costs in buying the option.
- The risk-free rate and volatility of the underlying are known and constant.
- The returns on the underlying are normally distributed.

They make this model too simplistic and unrealistic for more precise risk modelling.

2.1.5 Volatility

Volatility, denoted with Greek symbol σ, is a standard deviation of the logarithmic returns around the average of any random variable such as the price of a given security or a market index over a given time period. [18, pg. 303] The higher the volatility, the higher the risk of holding the security. It is a variable in option pricing formulas showing the extent to which the return of the underlying asset will fluctuate between current moment and the expiry of the option. A common practice in valuation of options is to use implied volatility, a theoretical value, that is derived directly from the market price of a derivative observed in a given moment in the market and input into an option pricing model such as Black-Scholes, that assumes that the volatility is constant. Volatility, however, as expressed as a percentage coefficient within option-pricing formulas, arises, among others, from daily trading activities and is itself observed to be random and unpredictable. Assets experience periods of high and low volatility rather than staying constant. Thus, apart from simplistic Black-Scholes model, there exist different stochastic models simulating the random changes volatility. In this project, the model, that we investigate, assumes in its standard form a constant time-independent volatility across time, and thus we are not concerned with modelling variability of volatility.

2.1.6 Mean Reversion

One-factor short-rate models are one example of models that are dependent on a single stochastic factor – the short rate [2]. In short-rate models, on the long run,

[2] A mathematical model that describes the future evolution of interest rates.

interest rates appear to be drawn back to their average level over time .[18, pg. 684] This phenomenon is known as *mean reversion*. When the short rate - typically denoted as r - is high, mean reversion tends to cause the interest rates to decrease; vice verse, when r is low, mean reversion tends to cause them to increase. This is explained visually on fig. 2.1 below.

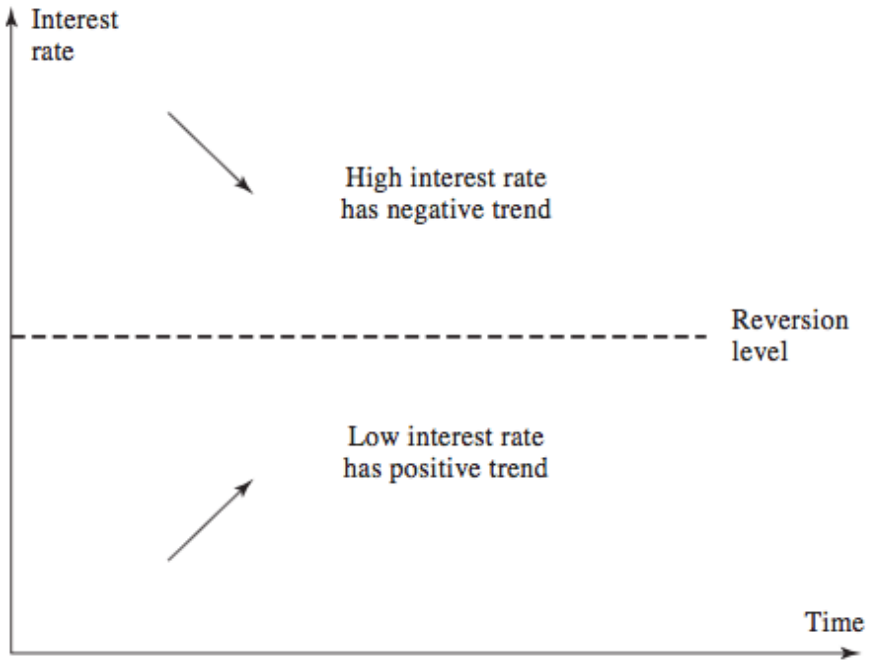


Figure 2.1: Illustration of mean reversion

Source: Based on Options, Futures and Other Derivatives [18, pg. 684].

2.1.7 Day-count Conventions

Day-count conventions is a method for computing the amount of accrued interest, i.e. accumulated or received payments or benefits over time. It is usually used to asses the present value when the next coupon payment is less than a full coupon period away. Typically, bond markets and financial instruments have their own day-count conventions, which vary on the type of instrument, the interest rate type, and the

issuance country. The standards were introduced to simplify accounting calculations and introduce constancy of time period, e.g. day, month, year. In the project, we chose to use Actual/365 Fixed method assuming the year to be 365 days and the actual differences to be calculated in days, i.e. the smallest interval is one day.

2.1.8 Numerical Methods for Option Pricing

Numerical methods are mathematical tools designed to give approximate but accurate solutions to various numerical problems. Most often used methods are iterative methods that converge to a satisfactory result with a certain assumed level of approximation by iterating in a finite number of steps from the initial conditions. In this project, we deal with interpolation and solutions to continuous-time stochastic differential equations describing financial phenomena like interest rate. As we can only represent problems with finite amount of data, we need to discretize these problems by finding values in a finite number of points in a problem domain. We use numerical methods to solve problems that do not have an analytical solution, e.g. some differential equations cannot be solved exactly. While using numerical methods, it is important to address the numerical stability of the used algorithm to estimate and control round-off errors arising from the use of floating point arithmetic.

Binomial Tree A most basic numerical method for pricing options involves the construction of trees or lattices. [18, pg. 253] A binomial tree (see fig. 2.2) is a numerical method, which allows to graphically represent the possible values that an option may take at different nodes or time periods. The value of the option depends on the underlying stock or bond, and computing values on the nodes is based on the probability that the price of the underlying asset will decrease or increase. The main advantage of binomial trees is that they are analytically tractable. This means that price valuation for a derivative can be performed on each node for every time step. This gives more flexibility and allows pricing of path-dependent derivatives, such as exotic options, having more complex cashflows. While the binomial model presented on fig. 2.2 is unrealistically simple, the life of an option may in practice span for 30

or more time steps, making it possible to implicitly consider about 2^{30}, or approx. 1 billion possible stock price paths. [18, pg. 268]

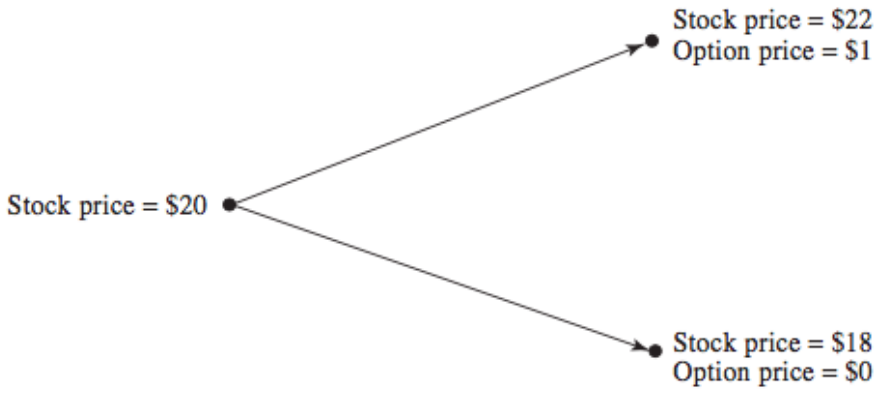


Figure 2.2: Illustration of a binomial tree

Source: Based on Options, Futures and Other Derivatives [18, pg. 254].

Trinomial Tree The trinomial option pricing model (an example is shown on fig. 2.3) is an alternative numerical method for constructing trees, with the difference that it consolidates another possible value per single time period. This makes the trinomial model even more relevant to real life situations, as it ensures the possibility that the value of the underlying asset may not change over a time period (taking the mid path on the tree). Calculations for a trinomial tree are analogous to those for a binomial tree.

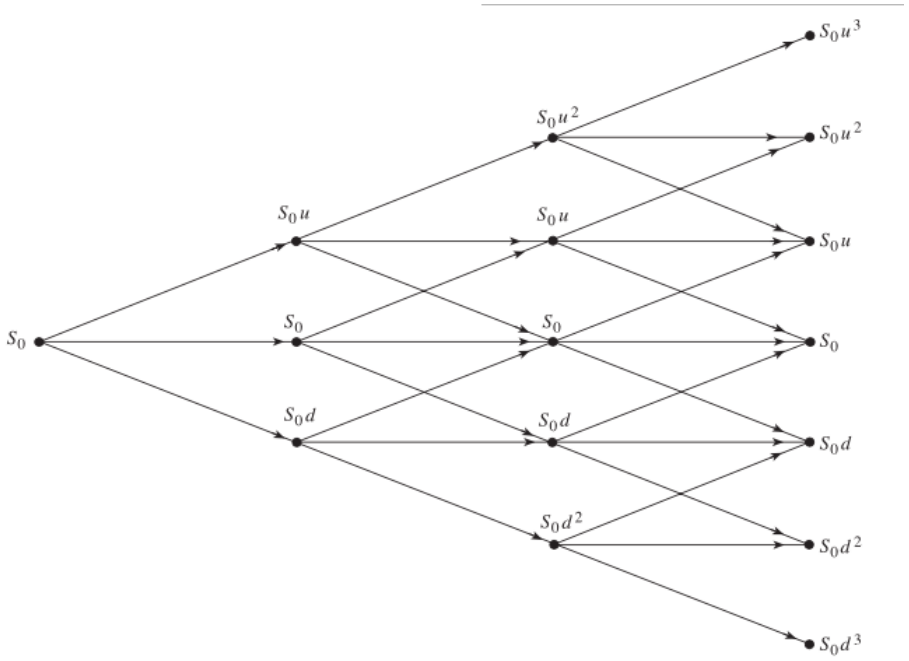


Figure 2.3: Illustration of a trinomial tree

Source: Based on Options, Futures and Other Derivatives [18, pg. 444].

In this project, we use trinomial tree method as a numerical method for option pricing. However, there exist other methods that might be more suitable and performing better under certain specific conditions, that might be mathemtically and computationally more complex.

Monte Carlo simulations samples different paths to obtain the expected payoff of the asset in a risk-neutral world and are then discounted at this risk-free rate. Monte Carlo Methods are particularly useful in the valuation of options with multiple sources of uncertainty (multiple dimensions) They are suitable for pricing instruments with complicated features, when the payoff depends on the path followed by the underlying variables, as this makes them difficult to value through a straightforward Black–Scholes analytical model or tree-based computation. Unfortunately, the method is computationally intensive and might be too slow to be competitive over an analytical solution or other numerical techniques like trees. On

the other hand, this constraint is less of concern in current computing environment with abundant and efficient compute capabilities. Another issue is that Monte Carlo procedures have to be adjusted to handle situations with early exercise opportunities, what increases their implementation complexity. [18, pg. 448]

Finite difference methods (FDM) value a derivative by solving the differential equation that the derivative satisfies. The differential equations are converted into sets of difference equations and are solved iteratively. This is similar to the trinomial trees method, since the computations also work back from the end of the derivative maturity to the beginning. In fact, tree based methods, if suitably parameterized, are a special case of the explicit finite difference method. [18, pg.455 - 466] These type of methods can solve derivative pricing problems that have, in general, the same level of complexity as those problems solved by tree approaches, but, given their relative complexity, are usually employed only when other approaches are inappropriate. Furthermore, like tree-based methods, they are limited in terms of the number of underlying variables, i.e. multiple dimensions, they can handle.

2.2 CUDA Background

CUDA[3] is a parallel computing platform and a programming model based on C/C++, developed by Nvidia with the purpose to simplify and make GPGPU programming more accessible. It allows developers to incorporate various CUDA-specific keywords (such as e.g. ___*device*___, ___*host*___, *cudaMemCpy*, *blockIdx* and more.) into their programs, in order to express the parallelism and indicate to the compiler the code that should be run on the GPU.

2.2.1 Process Flow

CUDA allows the compiler to distinguish between serial and parallel code by using the ___*host*___ and ___*device*___ function modifiers respectively. While the first will be run as any other normal C/C++ program on the CPU, the latter will be run on the GPU. Device functions can be called inside **kernel** functions that are defined with the ___*global*___ keyword and when called, are executed N times in parallel by N different CUDA *threads*. Since GPUs operate on their own memory, it is necessary that the input to the kernels is copied to the GPU memory beforehand. Furthermore, the results have to be copied back and both of these operations can be done with the use of the built-in *cudaMemCpy()* function. Note that copying data back and forth from the GPU takes a rather high amount of time, hence GPU computing is not always suitable for all applications. It is often the case that the input is too small and it makes more sense to run an algorithm sequentially, as that will result in a shorter runtime. Return or temporary arrays must also be pre-allocated on the GPU beforehand, which is done by the *cudaMalloc()* function.

Each parallel invocation of a kernel creates a CUDA ***block*** of multiple ***threads*** (currently up to 1024 threads). A *block* executes only on one multiprocessor, which allows (i) synchronization (by means of barriers) across all *threads* in the *block* and (ii) the use of *shared memory*, allowing *threads* within a *block* to communicate. On a larger scale, a set of *blocks* is called a ***grid***. The number of blocks and threads can

[3]Additional information about CUDA can be found on the Nvidia official documentation pages: `https://docs.nvidia.com/cuda/`

be specified by the programmer. A general overview of this structure is illustrated on fig. 2.4.

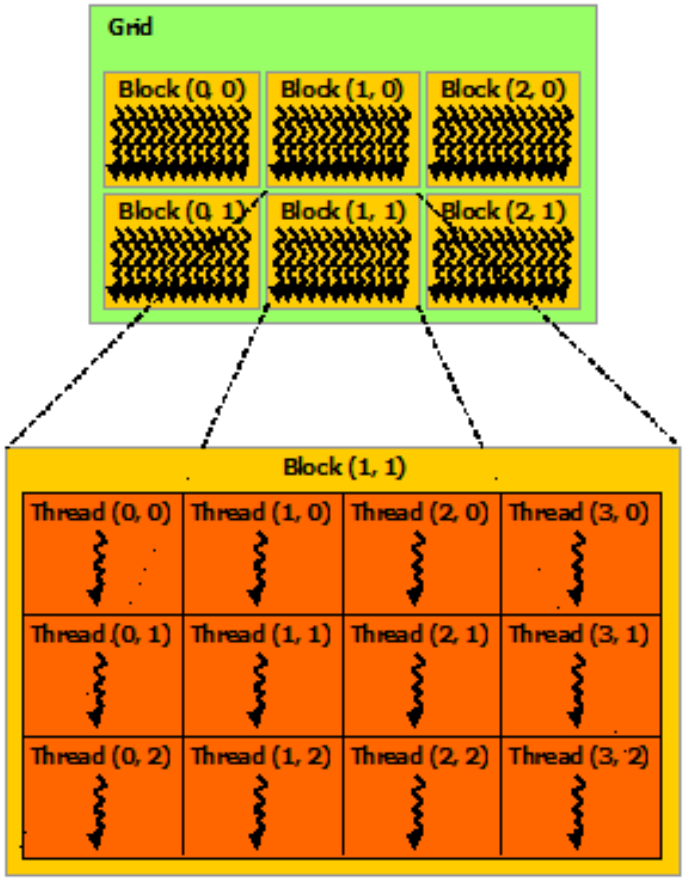


Figure 2.4: Grid of Thread Blocks

Source: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`

CUDA automatically allocates *thread blocks* and resources to the *Streaming Multiprocessors* (*SMs*). *SMs* are the part of the GPU which runs the provided CUDA *kernel* and they consist of several sub-components: *Memory caches* (e.g. *shared memory*, *constant memory*, and more); thousands of *registers*[4], which can be parti-

[4] *Register memory* is memory allocated inside a single *thread* and is only accessible by it through its lifespan. This is the fastest memory available in the CUDA memory model.

tioned among the threads for execution; a *warp scheduler*[5] and *execution cores* for both *integer* and *floating-point* operations. Fig. 2.5 illustrates the CUDA memory model.

Blocks and *threads* can be indexed inside the kernel in order to utilize the workload distribution among *threads*. CUDA also allows the use of ***shared memory*** between all threads within a block with the use of the *__shared__* keyword, used when declaring arrays. In contrast to ***global memory***, which is accessible from all threads, *shared memory* is much faster to operate with, due to its locality, as well as its different technology[6].

The parallel access to data can often lead to *data hazards*, such as *RAW* (read after write), *WAR* (write after read) or *WAW* (write after write). While good software design can often help with *data hazards* (i.e. as it will be seen in section 5.1), it is often insufficient. In those situations, the *__syncthreads()* function can be called, which awaits all *threads* within a *block* and prevents incorrect memory reads/writes. Note that this slows down the overall runtime, thus it is up to the programmer to ensure it is only used when necessary.

[5]When passed to the *SM*, *thread blocks* are split into *warps* (currently with a maximum size of 32 *threads*). All the *threads* in a *warp* execute concurrently on the resources of the *SM*.

[6]*Global memory* uses DRAM, while *Shared memory* uses SRAM, which tends to be overclocked, hence faster.

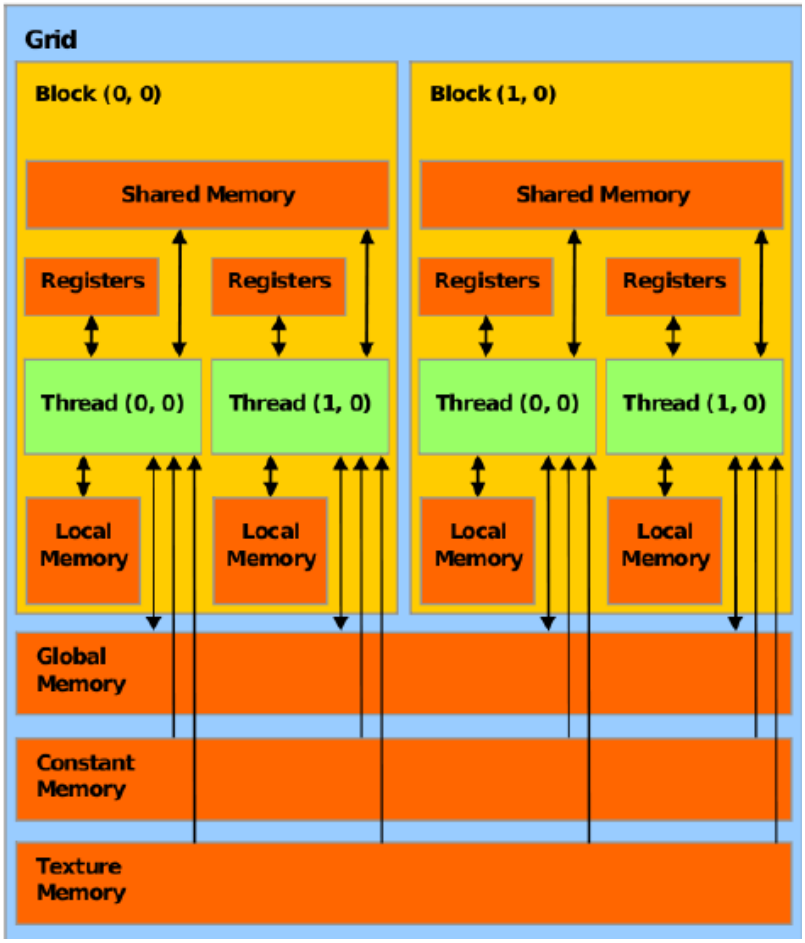


Figure 2.5: CUDA memory model

Source: Based on Nvidia CUDA Programming Guide 1.1, 2007

2.2.2 Memory Coalescing

One of the most significant hardware optimizations that CUDA provides is memory coalescing. It can be achieved when the threads in a warp access (when executing one read/write SIMD instruction) consecutive memory locations, making the access to be performed in one memory transaction. The consequences of not using memory coalescing can be as much as a warp different memory transaction to read/write this data. It is therefore important that the code is designed to take advantage of this

optimization. As it can be seen on fig. 2.6, an array can be restructured (transposed) allowing to take advantage of the order of elements. In the course of this project we have applied both coalesced and non-coalesced accesses and it has been interesting to observe the performance benefits it can lead to, as it will be shown in chapter 9.

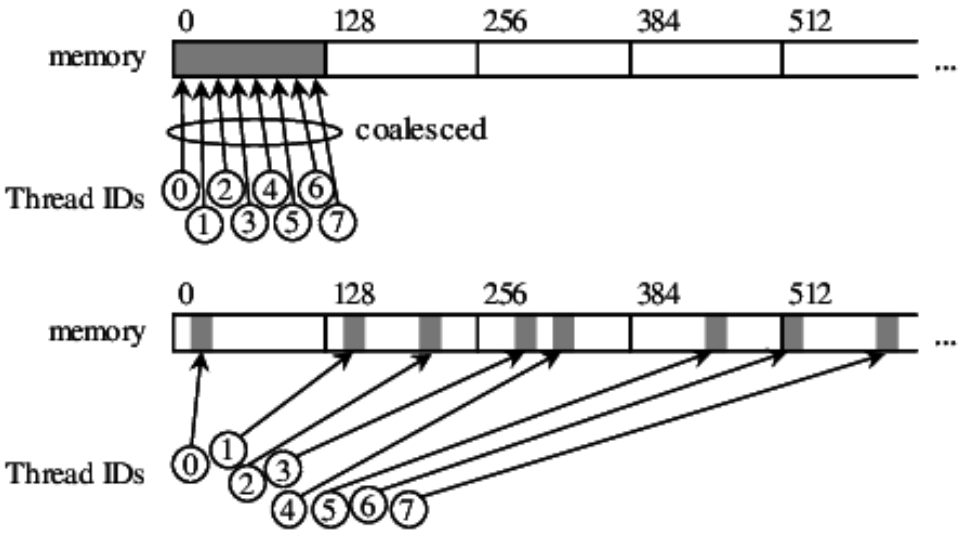


Figure 2.6: Memory coalescing

Source: Based on Real-Time FFT Computation Using GPGPU for OFDM-Based Systems [22]

2.3 Semantics and types of parallel operations

To understand the flattening transformations applied and derive implementations exploiting different levels of flattening, it is necessary to first understand the higher-order functions we have used. Note that the functions described in this section can be redundant in imperative languages, such as CUDA. Despite that, parts of the work of this thesis is done in Futhark, hence these functions will be met throughout this report. For simplicity, these functions have been extracted from the Futhark language. Therefore this section will also serve as a "Futhark background". Note that for all functions below, $[n]\alpha$ denotes an array of n elements of type α and e denotes the neutral element.

2.3.1 Zip and unzip

Zip and *unzip* are used when working with tuples of data, as the first creates a tuple of values, while the latter can be used to extract the values of it. These functions are redundant in CUDA, however we have used them in the Futhark implementations. The signature of *zip* is:

$$zip \ : \ [n]\alpha \to [n]\beta \to [n](\alpha, \beta)$$
$$zip \ [x_1, x_2, ...x_n] \ [y_1, y_2, ...y_n] = [(x_1, y_1), (x_2, y_2)..., (x_n, y_n)]$$

Conversely, the signature of *unzip* is:

$$unzip \ : \ [n](\alpha, \beta) \to ([n]\alpha, [n]\beta)$$
$$unzip \ [(x_1, y_1), (x_2, y_2)..., (x_n, y_n)] = ([x_1, x_2, ...x_n], [y_1, y_2, ...y_n])$$

2.3.2 Map

A *map* applies a given function to each element of a list. It therefore takes a function and an array as input. It has the following signature:

$$map \ : \ (\alpha \to \beta) \to [n]\alpha \to [n]\beta$$
$$map \ f \ [x_1, x_2, ..., x_n] = [f(x_1), f(x_2), ..., f(x_n)]$$

We have additionally used *map2*, *map3* and similar *map* variations in Futhark, which have different signatures than the ones above, where the map is applied to multiple arrays (*map2* for 2, *map3* for 3, etc.) of the same size. In these cases, the function f takes multiple parameters, one from each array. For simplicity, we have used only *map* in our pseudo-code, even though the maps may take multiple arrays as input. In these cases, the number of arguments in the lambda function indicate the number of input arrays if unclear.

2.3.3 Reduce

A *reduce* uses a given binary-associative operator, a neutral element, and an array, and it recursively accumulates the array elements with the combining operator, starting with the neutral element, building up the return value. *Reduce* is quite similar to a *scan* (discussed in the next subsection), with the only difference that it does not keep intermediate values. The function has the following signature:

$$reduce \ : \ (\alpha \to \alpha \to \alpha) \to \alpha \to [n]\alpha \to \alpha$$
$$reduce \ \odot \ e \ [x_1, x_2, ..., x_n] = e \odot x_1 \odot x_2 \odot ... \odot x_n$$

We have created an alias for *reduce*, named *reducePlus*, which corresponds to *reduce* $(+)$ 0 where the combining operator is + and 0 is the neutral element.

2.3.4 Scan

An *inclusive scan* (or just *scan*) uses a given binary-associative operator, a neutral element, and an array and recursively accumulates the array elements with the combining operator, starting with the neutral element, building up the return values. Note that (i) *scan* returns an array containing all intermediate values, differently from *reduce* and (ii) the neutral element is not returned together with the intermediate values. Another variation of *scan* is *exclusive scan/scanExc*, which shifts the return array of a typical *scan* to the left with one element, including the neutral element in the beginning and excluding the last element (hence the name exclusive).

Scan (inclusive) has the following signature:

$$scan \ : \ (\alpha \to \alpha \to \alpha) \to \alpha \to [n]\alpha \to [n]\alpha$$
$$scan \ \odot \ e \ [x_1, x_2, ..., x_n] = [e \odot x_1, ..., e \odot x_1 \odot ... \odot x_n]$$

while an exclusive scan has the signature:

$$scanExc \ : \ (\alpha \to \alpha \to \alpha) \to \alpha \to [n]\alpha \to [n]\alpha$$
$$scanExc \ \odot \ e \ [x_1, x_2, ..., x_n] = [e, e \odot x_1, ..., e \odot x_1 \odot ... \odot x_{n-1}]$$

Scan has been one of the most used functions in this project. To reduce redundancy in the pseudo-code of the following chapters, we have created an alias for it - *scanPlus*, which corresponds to *scan* (+) 0 where the combining operator is + and 0 is the neutral element.

2.3.5 Segmented scan

A two-dimensional irregular array — meaning the rows (segments) do not have the same lengths — can be represented by a flat array of values of primitive types together with a "flag" array, which denotes where each row (segment) starts—in essence a `true` or `1` flag value denotes the start of a new segment and a `false` or `0` value denotes that the current element is within a segment but not the first element of the segment. A *segmented scan/sgmScan* operator semantically performs in parallel an inclusive scan on each of the segments of the array. (Note that exclusive scan can also be segmented, but it will not be introduced, as it was not used in this thesis). *Segmented scan* has a type similar to *scan*, except that it also receives the flag array as parameter; moreover it is also straightforwardly implemented by means of *scan*.

Its type and implementation are presented below:

$$
\begin{aligned}
&\mathit{sgmScan} \ : \ (\alpha \to \alpha \to \alpha) \to \alpha \to [n]\texttt{bool} \to [n]\alpha \to [n]\alpha \\
&\mathit{sgmScan} \ \odot \ e \ \mathit{flags} \ \mathit{vals} = \\
&\quad \mathit{let} \ (\_, \mathit{vals}') \ = \ \mathit{scan} \ (\ (f_1, v_1) \ (f_2, v_2) \ \to \\
&\qquad\qquad\qquad\qquad \texttt{if} \ f_2 \ \texttt{then} \ (\texttt{true}, v_2) \ \texttt{else} \ (f_1 \ \texttt{||} \ f_2, \ v_1 \ \odot \ v_2) \\
&\qquad\qquad\qquad) \ (\texttt{false}, e) \ (\texttt{zip} \ \mathit{flags} \ \mathit{vals}) \\
&\quad \texttt{in} \ \mathit{vals}'
\end{aligned}
$$

Similar to *scanPlus* and *reducePlus*, we have also created an alias for *sgmScan* - *sgmScanPlus*, which corresponds to *sgmScan* (+) 0 where the combining operator is + and 0 is the neutral element.

2.3.6 Replicate

Replicate takes an integer n and a value m and creates an array of length n whose elements have value m. The function signature is:

$$
\begin{aligned}
&\mathit{replicate} \ : \ (n : \mathit{int}) \to \beta \to [n]\beta \\
&\mathit{replicate} \ n \ m = [m, ..., m]
\end{aligned}
$$

2.3.7 Scatter

Scatter is used for bulk in-place updates with multiple values and takes an array to write to, an array of indexes showing which elements of the first arrays should be updated, and an array of values, i.e. the updates. Note that CUDA supports in-place updates, hence this operation is not needed. The signature is shown below:

$$\mathit{scatter} \ : \ [n]\alpha \to [m]\beta \to [m]\alpha \to [n]\alpha$$

2.3.8 Iota

The *iota* function is Futhark-specific and is used to create an array of index values, used to map indexes over elements. It takes an integer as input. The signature is

shown below:

$$iota \ : \ (n : int) \rightarrow [n]int$$
$$iota \ n = [0, 1..., n-1]$$

2.3.9 Last

Last takes an array as input and returns the last element of it. The signature is:

$$last \ : \ [n]\alpha \rightarrow \alpha$$
$$last \ [x_1, x_2, ...x_n] = x_n$$

2.3.10 Length

Length takes an array as input and returns its length/size. The signature is:

$$length \ : \ [n]\alpha \rightarrow int$$
$$length \ [x_1, x_2, ...x_n] = n$$

2.4 Flattening Background

As briefly mentioned in the Introduction, complex applications are often composed of multiple nested levels of operations. This means that operations are often performed on nested arrays such as $a = [[1,2],[3,4,5],[6,7,8,9]]$. Suppose that we want to apply the function $f(x) = x + 1$ to each element in the array (note that iteration over arrays is done via a map). This cannot be done directly, as x is of type "array of integers". Instead, we can iterate over all nested arrays (with the use of another *map*) and apply the function to each sub-array as follows: $map\ (\backslash x \rightarrow map\ f(x))\ a$. In order to effectively utilize massively parallel hardware, it is necessary that nested higher-order functions (such as *map*, *scan*, *reduce*, etc.) are flattened, so that they can operate on flat arrays. Their transformation can be done through certain flattening techniques such as the ones described in this section. Flattening an array itself is straightforward, as the example above becomes $flat\_a = [1,2,3,4,5,6,7,8,9]$. Albeit, the flattened representation of an array does not contain any indication of how many nested arrays existed before, nor on their separation. Hence it is often necessary to introduce additional arrays such as *shape* and *flags*. In our example, $shape\_a = [2,3,4]$, contains the length of each sub-array of a. Denoting the length of $shape\_a$ by m and the length of $flat\_a$ by n, we can create the *flag* array by:

- using an exclusive scan to compute the indices where a `true` or `1` value should be placed in the flag array, i.e., $inds = scanExc\ (+)\ 0\ shape\_a = [0,2,5]$,
- updating an array of `false` values with `true` at the computed indices, i.e., $flags\_a = scatter\ (replicate\ n$ `false`$)\ inds\ (replicate\ m$ `true`$)$ which results in $[1,0,1,0,0,1,0,0,0]$.

The flag array can be later used to flatten nested operations, for example in the case of segmented scan. Note that only the flattening transformations that have been used in this project are described, even though many more transformations can be devised. Note also that these transformations are only guidelines and they can be simplified in practice, as some of the steps may be redundant in various combinations of operations.

2.4.1 Nested Map in a Map

A nested map inside a map is one of the simplest flattening transformations that can be performed. The procedure is as simple as applying the nested map on the flattened array, meaning that:

$$map\ (x \rightarrow map\ f(x))\ a\ \equiv\ map\ (x \rightarrow f(x))\ \mathit{flat\_a}$$

Applying the map to the flat array shown as an example in the beginning of this section, we get $map\ (x \rightarrow x + 1)\ [1, 2, 3, 4, 5, 6, 7, 8, 9] = [2, 3, 4, 5, 6, 7, 8, 9, 10]$

2.4.2 Nested Scan in a Map

Flattening *scan* is also done in a single step, as it is replaced with a *segmented scan*, taking an additional *flags* array. In general, the flattening transformation of a nested *scan* can be described as (where e is the neutral element):

$$map\ (x \rightarrow scan\ \odot\ e\ x)\ a\ \equiv\ sgmScan\ \odot\ e\ \mathit{flags\_a}\ \mathit{flat\_a}$$

Using the array a mentioned previously as an example, $(+)$ as a cumulative operator and 0 as a neutral element, we can apply
$sgmScan\ (+)\ 0\ [1, 0, 1, 0, 0, 1, 0, 0, 0]\ [1, 2, 3, 4, 5, 6, 7, 8, 9]\ = [1, 3, 3, 7, 12, 6, 13, 21, 30]$

2.4.3 Nested Replicate in a Map

As previously mentioned, replicate takes as inputs an integer and a value of some arbitrary type α. For simplicity (only in this section), we will name them n and x, where n is the number of times x should be replicated. The flattening of a *replicate* nested inside a *map* becomes a combination of scans and scatters. In the following we assume the flat array a is of type $[q](int, \alpha)$:

$$map\ (\backslash(n, x) \rightarrow replicate\ n\ x)\ a\ \equiv\ sgmScanInc\ (+)\ 0\ \mathit{flags}\ \mathit{vals}$$

where:
$(shape, xs)\ =\ unzip\ a$
$inds\ =\ scanExc\ (+)\ 0\ shape$
$\mathit{flatlen}\ =\ (last\ inds)\ +\ (last\ shape)$
$\mathit{flags}\ =\ scatter\ (replicate\ \mathit{flatlen}\ \texttt{false})\ inds\ (replicate\ n\ \texttt{true})$

$vals \;=\; scatter\ (replicate\ \mathit{flatlen}\ \texttt{false}))\ inds\ xs$

The above re-write rule says that a replicate operation nested in a map will generate a flat array (by means of the *sgmScan* operation), whose structure is described by the *shape* (and *flags*) arrays. Since the re-write rule is very dry, we work an example by hand to provide more insight.

Suppose that we have a $map\ ((n, x) \rightarrow replicate\ n\ x)\ [(1, 7), (3, 8), (2, 9)]$, that is, to replicate 7 one time, 8 three times, and 9 two times through the iterations of the map. We start by deriving
$(shape,\ xs) = unzip([(1, 7), (3, 8), (2, 9)]) = ([1, 3, 2], [7, 8, 9])$
$inds = scanExc\ (+)\ 0\ [1, 3, 2] = [0, 1, 4]$
$\mathit{flatlen} = (last\ [0, 1, 4]) + (last\ [1, 3, 2]) = 4 + 2 = 6$
$\mathit{flags} = scatter\ [0, 0, 0, 0, 0, 0]\ [0, 1, 4]\ [1, 1, 1] = [1, 1, 0, 0, 1, 0]$
$vals = scatter\ [0, 0, 0, 0, 0, 0]\ [0, 1, 4]\ [7, 8, 9] = [7, 8, 0, 0, 9, 0]$
and finally
$sgmReplicate = sgmScan\ (+)\ 0\ [1, 1, 0, 0, 1, 0]\ [7, 8, 0, 0, 9, 0] = [7, 8, 8, 8, 9, 9]$

2.4.4 Nested Reduce inside a Map

We cover now the case in which a reduce is nested inside a map:

$$map\ (\backslash x\ \rightarrow\ reduce\ \odot\ e\ x)\ a$$

where a is a two-dimensional irregular array, represented by (1) array *flat_a*, which holds the flattened values, and by (2) arrays *shape_a* and *flat_a* which encode the row structure of a, as explained before.

The result of flattening is a flat array containing as many elements as rows in a (i.e., length of *shape_a*), in which each element of the result is obtained by reducing the corresponding segment with the given operator and neutral element, a.k.a. segmented reduce. The re-write rule is given below (and it assumes no empty rows):

$$map\ (\backslash x \rightarrow reduce\ \odot\ e\ x)\ a\ \equiv\ map\ (\backslash i\ \rightarrow\ vals[i-1])\ shape\_scanned$$

where
$shape\_scanned\ =\ scan\ \odot\ e\ shape\_a$

$vals \;=\; sgmScan\ (\odot)\ e\ flags\_a\ flat\_a$

Suppose that we simplify the example in the beginning of this section, such that $\odot = +$, $e = 0$, $a = [[1,2],[3,4,5]]$; $flat\_a = [1,2,3,4,5]$; $shape\_a = [2,3]$, and $flags\_a = [1,0,1,0,0]$. Then the segmented reduce is obtained as follows:

$shape\_scanned \;=\; scan\ (+)\ 0\ shape\_a \;=\; [2,5]$
$vals \;=\; sgmScan\ (+)\ 0\ [1,0,1,0,0]\ [1,2,3,4,5] = [1,3,3,7,12]$
and finally
$sgmReduce \;=\; map\ (\backslash i \;\rightarrow\; vals[i-1])\ shape\_scanned \;=\; [3,12]$

Summary

This chapter has provided a brief introduction to the necessary concepts mentioned in the consequent chapters. It has covered financial topics such as options, volatility, numerical methods, and more, all needed in the implementations. Following was an introduction to the basics needed for understanding terminology and techniques in CUDA, which have been used throughout this thesis. Next, this chapter has covered the parallel operations used in the thesis, immediately followed by the flattening background needed for applying transformations in order to exploit thread divergence and experiment with different levels of nested parallelism.

3

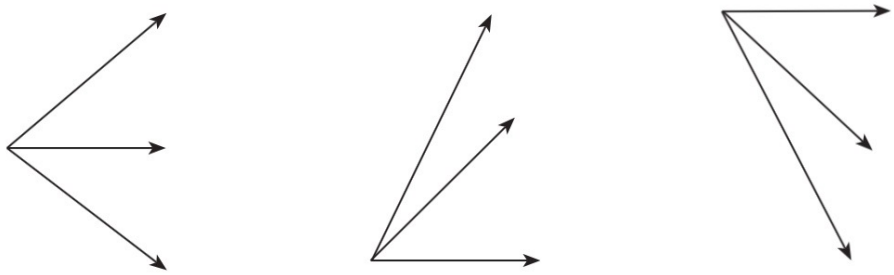
Hull-White Single-Factor Model

The following chapter will study the Hull-White Single-Factor Model to understand the algorithm behind it and prepare a strategy for a parallel implementation. It will first introduce a general overview of how a trinomial tree is being constructed, and later show how prices are being discounted from it.

3.1 Hull-White Trinomial Tree

In this project, we implement the trinomial tree numerical method to discretize the Hull-White model. In contrast to the standard trinomial tree, the tree used in the Hull-White model incorporates the mean-reversion of the interest rate, by using a width limit and modified branching methods for the tree. Standard branching (see fig. 3.1a) remains the same throughout the tree. At the bottom of the tree, where interest rates are very low, the "up one/straight along/down one" (see fig. 3.1b) branching is used. At the top of the tree, where interest rates are very high, the "straight along/down one/down two" branching is used (see fig. 3.1c).

We observe that this pruning characteristic allows us to asses the size of the tree upfront and use this knowledge to enable more specific implementation optimizations, in particular how to map the tree to the parallel device thread and memory architecture. Otherwise, the tree could grow infinitely in its width, making it impossible to map to limited parallel architecture memory resources and invalidating certain parallel implementations.



(a) Standard branching (b) Bottom branching (c) Top branching

Figure 3.1: Alternative branching methods for a trinomial tree.

Source: Based on Options, Futures and Other Derivatives[18, pg. 698].

3.2 Overview

Pricing a single option using Hull-White short-rate[1] single-factor trinomial tree model enables the term structure of interest rates at any given time to be obtained from the value of the short rate r at that time and the risk-neutral process for r. This shows that, once the process for r has been defined, everything about the initial zero curve and its evolution through time can be determined[18, pg. 683].

The model consists of two steps. The first (forward propagation along the tree) is the construction of the trinomial tree in order to obtain a list of alpha values for each time step. These alphas are later used in the second step (backward propagation along the tree) to fit the option/bond prices and obtain the option value back at the root node of the tree. The input fed to the algorithm consists of an option, which includes its strike price, maturity, time step length, mean-reversion rate[2] and volatility[3]. The output is the estimated price of the option/bond. The two steps can be generalized as follows:

[1]The short rate, r, at time t is the rate that applies to an infinitesimally short period of time at time t[18, pg. 682]

[2]denoted as a - determines the relative volatilities of long and short rates[20, pg.9]

[3]denoted as σ - determines the overall level of volatility [20, pg. 9]

1. **Forward propagation step:** Construct a term structure for the underlying asset by progressing one time step at a time. Determine neutral risk rate for a new time step using estimated yield curve data and estimated current asset values.

2. **Backward propagation step:** Discount the asset prices to estimate option payoff at maturity going from the leaves of the tree to its root.

Algorithm 1 shows a high-level overview of a function implementing this procedure for pricing one option. The input of the algorithm is an option and a yield curve (used for the computation of alphas) and the output is the estimated price of the option. The function consists primarily of two sequential (convergence) loops of count tree height, which contain inner parallel operators of count tree width, where tree height and width are specific to each option (and thus vary across options). The tree height is dependent on the number of time steps, i.e., maturity of the underlying bond and precision. The tree width is dependent on the number of terms and input parameters.

Different option/bond maturities (leading to different tree heights) and different level of pricing accuracy (number of simulated time steps leading to different tree dimensions) make the choice of an effective parallelization strategy difficult. It is necessary to have a deep understanding of the algorithm itself to achieve maximum parallelization efficiency. The book by John Hull[18] provides a solid background on the topic, describing the mechanics of interest rates, markets, as well as application of binomial trees and eventually trinomial trees to option pricing. Chapter 30 further narrows the topic of using trinomial trees as a numerical method and introduces a step by step walk-through of applying the algorithm on a basic example. While some of the calculation details are omitted in the book, the authors provide references to previous articles[20][21], where they provide a thorough explanation backed with more detailed examples.

It is important to mention that the construction of a trinomial tree is a discrete-time, lattice-based[4] numerical method, but the example in the book is simplified by cutting the tree at a certain height and using analytic formulas to produce a concrete result for a specific financial instrument - a zero-coupon bond maturing at time $(m + 1) * \triangle t$ [18, pg. 704]. These formulas have been found and proven to be effective by the authors of the book and the articles. While this simplification gives more precise results in the above mentioned specific case, constructing the entire tree and using all of the time steps provides a foundation for pricing other options with more sophisticated cashflows. All the implementations of this thesis will be focused on the described numerical approach.

The following sub-chapters are focused primarily on the intuition behind the algorithm, with the sole purpose to provide the reader with a general overview for it. For this reason, many of the details and formulas of calculating specific values are omitted, however they are thoroughly described in the book and the articles by Hull and White. As the model is best understood visually, we have included some of the supplementary images from the Hull and White book in order to support our algorithm explanation.

[4]A model that takes into account expected changes in various parameters e.g. interest rate over the duration of the option

Algorithm 1: High-level overview of pricing a single option using Hull-White Single-Factor Model

Input : Option, YieldCurve
Output: Price approximation

```
1
2  alphas[0] = Compute yield at initial interest rate
3  Qs[0][width/2] = 1                    /* Initialize the root node to 1$ */
4
5  /* Forward propagation (convergence) loop                              */
6  for i = 0 to height do
7  |   /* Compute Qs at the next time step                                */
8  |   for j = 0 to width do
9  |   |   Qs[i + 1][j] = Compute Q from Qs[i] and alphas[i]
10 |   end
11 |   Compute alphas[i + 1] from Qs[i]
12 end
13
14 /* Initialize prices at the last time step to 100$                     */
15 Prices[height − 1] = 100
16
17 /* Backward propagation (convergence) loop                             */
18 for i = height − 1 to 0 do
19 |   /* Compute prices at the previous time step                        */
20 |   for j = 0 to width do
21 |   |   Prices[i][j] = Compute price from Prices[i + 1] using alphas[i]
22 |   end
23 end
24
25 /* Return price at the root node                                       */
26 return Prices[0][width/2]
```

3.3 Forward Propagation

The forward propagation by itself consists of two stages. Each of them computes different values on the same tree. While the tree height can grow indefinitely, depending on the number of time steps, the width of the tree is limited by mean-reversion (as reasoned for in chapter 2), by determining its max. width (or as we refer to it for simplicity - width).

Indexing Since the tree is 2-dimensional, locating and computing values on individual nodes boils down to locating them by index first. Hence, we introduce the two indexes - i and j, used to indicate the tree height and its width respectively. To describe the meaning of i and j visually, we can use fig. 3.2, where nodes along the height - A, C, G - can be indexed as i = 0, 1, 2. Indexing across the width is different, as we intuitively denote the core of the tree - nodes A, C, G - with $j = 0$. Going down along the three decreases the value of j, while going up increases it. This means that nodes D,H are located on $j = -1$, nodes B,F are located on $j = 1$ and so on. We denote the highest node across the width as j_{max} and j_{min} as the lowest. Note that $j_{min} = -j_{max}$ due to the symmetry of the tree, making it possible to omit j_{min} occurrences in our implementations by replacing them with $-j_{max}$. On fig. 3.2 nodes E and I are at the top and at the bottom of the tree width, hence their j indexes are equal to j_{max} and j_{min} respectively. The index values in this example are $j_{max} = 2$ and $j_{min} = -2$. Note that our code cannot always be aligned with our intuition and indexes cannot be negative in our programs. Hence we can shift the indexing between j_{min} and j_{max} and count from 0 to $2 * j_{max} + 1$ instead, making the root node to be located at index j_{max}. Despite that, we have used j_{min} and j_{max} as width boundaries throughout this report, as they are easier to comprehend.

Stage 1 aims to construct a tree for a variable R^* that is initially 0 and follows the Ornstein–Uhlenbeck stochastic[5] process[6] $dR^* = -aR^*dt + \sigma dz$ which is

[5]With a random probability distribution or pattern that may be analyzed statistically but may not be predicted precisely.

[6]Tends to drift towards its long-term mean (also called mean-reverting).

symmetrical about $R^* = 0$[18, pg.698-699]. Together with R^*, p_u, p_m and p_d are calculated to match the respective up, mid and down probabilities that match the expected change and variance of the change in R over the next interval $\triangle t$. Since the width of the tree is limited between j_{min} and j_{max}, some of the branching is calculated differently, thus p_u, p_m and p_d depend on the node position. Naturally the tree construction happens iteratively, node by node, starting from the root. In the end of the stage, this first tree will always have a symmetrical shape[7] similar[8] to fig.3.2.

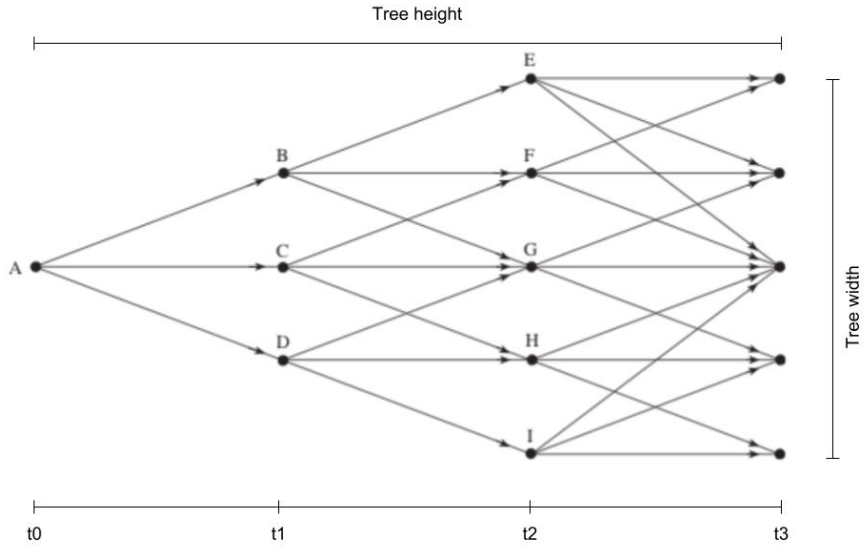


Figure 3.2: Example of the trinomial tree for R^*.

Source: Modified by the authors, based on Options, Futures and Other Derivatives[18, pg. 699].

[7]Trinomial trees are recombining, meaning that at any time, an up move followed by a down move has exactly the same effect on the price as a down move followed by an up move.

[8]Note that the width and height of the tree may differ based on the number of time steps and the maturity of the financial instrument

An important property of this tree include first of all that it is self-recombining, causing it to be symmetric. The probabilities on the lower part of the tree will be the negative of the probabilities on the upper part of the tree, e.g. probability that node A reaches node D is minus the probability of node A reaching node B. Furthermore, also due to symmetry, all unique probabilities can be stored in an array of size the width of the tree, because e.g. the probability of node A reaching node B is the same as the probability of node C reaching node F and so on. Probabilities are used both in stage 2 of the forward propagation, but also in the backward propagation, thus it is necessary to contain them to the end, if they are to be stored. Last but not least, the way probabilities are calculated is different on j_{min} or j_{max}, because of the difference in branching. This can be seen on fig. 3.2 where nodes E and I branch out differently in comparison to all other nodes.

Stage 2 In this stage, the rates at each node in the tree at each time step are shifted up by an amount - α, chosen so that the revised tree correctly prices discount bonds [21, pg. 6]. This is done by defining $Q_{i,j}$ as the present value of a security that pays off \$1 if node (i, j) is reached and 0 otherwise. The starting point is to set $Q_{0,0} = 0$ and α_0 to the interest rate at time $\triangle t$. Qs at the next time step are then calculated by using the generalized formula [18, pg.705]:

$$Q_{m+1,j} = \sum_k Q_{m,k} q(k,j) exp[-(\alpha_m + k\triangle r)\triangle t]$$

Assuming that we start at step m, to calculate the Qs on step $m+1$, we need to have the α on step m. Furthermore, once the Qs on step $m+1$ have been calculated, they are used to also find the α on $m+1$ later. This leads to conclude that αs and Qs are interrelated on each time step. αs are calculated using the generalized formula [18, pg.703]:

$$\alpha_m = \frac{\sum_{j=-n_m}^{n_m} Q_{m,j} e^{-j\triangle r\triangle t} - \ln P_{m+1}}{\triangle t}$$

At the end of this stage, the new tree will have changed visually. For example, the tree from fig. 3.2 can be re-shaped as shown on fig. 3.3.

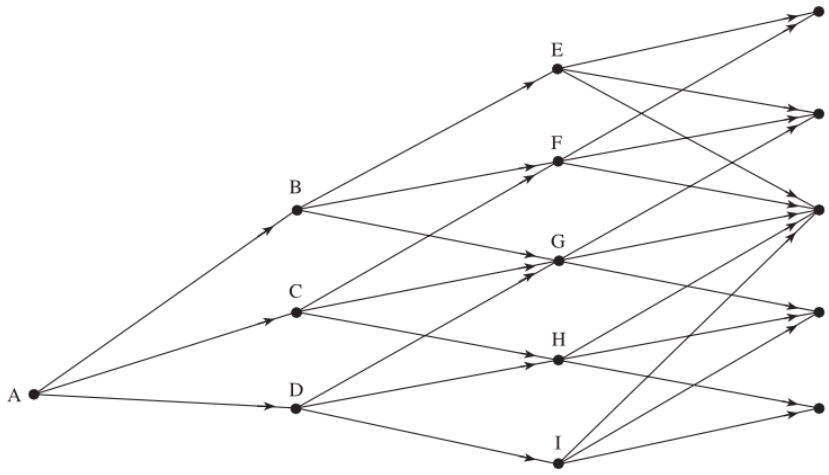


Figure 3.3: Example of the trinomial tree for R.

Source: Options, futures and other derivatives fig. 30.9 [18, pg. 702]

An important observation here is that the only outcome of this tree that is used in the backward propagation is the array of αs. Qs are in this case intermediary values, used to compute the α on each step and for this reason, the Q values do not need to be stored any longer once all αs have been computed.

3.4 Backward Propagation

The backward propagation starts with the previously constructed tree during the forward propagation step, in particular with array of αs. At each time-step the option payoff is computed as the discounted value of the expected value at the next step [21, pg. 6]. From this it follows that the nodes at time step i (e.g. the nodes without assigned letters in figures 3.2 and 3.3 above) are the starting point of the backward propagation. Their values are set to 100$ and are used to compute the previous set of nodes (at time step $i-1$). That is done by discounting bond price values up until the exercise of the option. At the option expiration time step, we decide if we exercise the option or let it expire worthless. To achieve that, we calculate the difference between bond price and the strike price. The positive values mean exercise, while

non-positive mean expiry worthless and are set to 0. We discount the option prices further down to the root of the tree to get the approximation of the option price on the valuation day. This is the output of the algorithm. We use the array of αs computed during the forward propagation through this procedure. It is important to note that determining the option price depends on the type of option (whether it is a put or a call option).

Summary

This chapter has provided a detailed overview of the Hull-White Single-Factor Model, in particular its two-stage procedure fo propagating along a trinomial tree. It will be used in the following chapter, which will introduce the challenges of implementing a sequential version of the algorithm in C++.

4

Sequential Implementation

In order to better understand the algorithm, we have started with a basic sequential implementation of it in C++. While this step could be done in any language, we have chosen to work with C++, as it would allow us to re-use pieces of code for the parallel CUDA implementations, described further in this report. Running this version with a large number of options will likely result in a significant amount of computation time. However, the purpose of this implementation is rather a proof of concept that the algorithm produces correct approximations, as well as to provide a set of results, which can be used to test against with the other implementations.

The algorithm described in the book is used to price one option at a time and the natural way to start a sequential implementation would be to create a single function that prices one option. Looping through all options in the data set and calling this function for each of them will then produce the end results. Pseudo-code in Algorithm 2 describes the approach we took based on the book and articles by Hull and White. Note that real is a data type that can be either single or double precision floating point number based on the required accuracy.

The implementation iterates through all given options, constructs a trinomial tree for each of them and propagates prices back through the tree, obtaining the price approximations for each option and returning them in the end. The algorithm follows the intuition provided in the previous chapter 3. The focus of this implementation is on correctness and simplicity.

4.1 Algorithm Description

Precomputation Pricing of one option starts with computing its constants such as tree width and rate step, tree height and time step, and other values needed to solve the formulas in Hull and White model. Afterwards, for each width step j, rate and probabilities (up, middle, down) are precomputed for use during both forward and backward propagations.

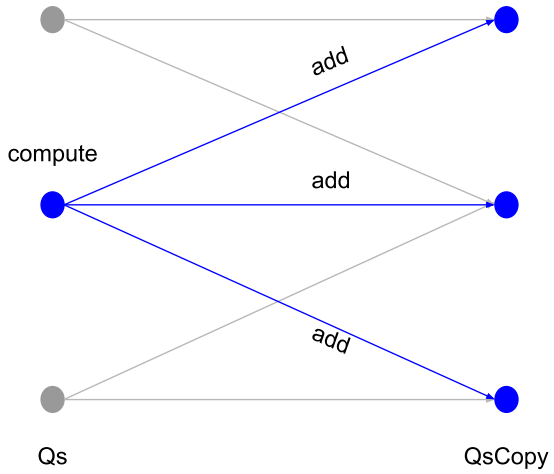
Algorithm 2: Sequential implementation

```
1  function ComputeOptionPrice
   Input  : option : { StrikePrice, Maturity, Length, TermUnit,
                      TermStepCount, ReversionRate, Volatility, Type }
            yields : { [Prices], [Timesteps] }
   Output: Price approximation for the option
2
3  Pre-compute probabilities for all width steps j
4  c : OptionConstants = Compute constants for the option
5  /* Option constants include:                                    */
6  /* c.t :  int - option length                                   */
7  /* c.X : real - option strike price                             */
8  /* c.dt :  real - time step (height)                            */
9  /* c.dr :  real - rate step (width)                             */
10 /* c.jmax :  int - max.  j of the tree                          */
11 /* c.width :  int - tree width (2 * c.jmax + 1)                 */
12 /* c.height :  int - tree height                                */
13 /* c.type :  CALL | PUT - option type                           */
14
15 /* Create an array of alphas and set the first alpha to the
      initial dt-period interest rate                              */
16 alphas : real[c.height + 1]
17 alphas[0] = Compute yield at c.dt
```

Forward propagation The purpose of forward propagation is to compute an array of alphas of size tree height + 1 that will be used during backward propagation. The first alpha is set to the interest rate at time of one time step. To capture the tree values at any given time step, only the current and previous tree levels of size tree width are needed, these two arrays are named *Qs* and *QsCopy*. The single starting value in the middle of the tree (the root of the tree) in *Qs* array is initalized to 1$.

After the arrays are initialized, the program iterates through time steps along the tree height. At each time step, it goes through the values Q computed in the previous step. Every value contributes to three values in the next time step (*QsCopy*) as illustrated in figure 4.1, according to the precomputed rates and probabilities. Note that this is an example of standard branching and there are also a bottom and a top branching, see fig. 3.1. After all *Qs* in the next step are computed, their values are aggregated to compute the next alpha. Lastly, arrays *Qs* and *QsCopy* are swapped and *QsCopy* is reset to zeros for the next iteration. Note that this approach combines stages 1 and 2 described in chapter 3.3 in a single iteration of the forward propagation loop.

Figure 4.1: Forward propagation - computing the next step



Source: Compiled by the authors

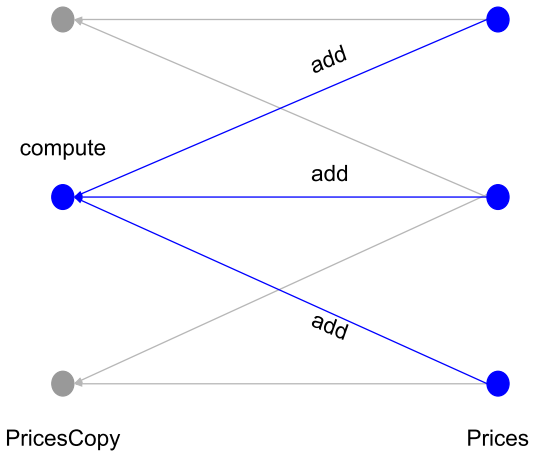
Algorithm 3: Sequential implementation - forward propagation

```
18 /* Forward propagation                                          */
19 Qs = real[c.width]
20 QsCopy = real[c.width]
21 Qs[c.jmax] = 1                              /* Set initial node to 1$ */
22
23 /* Iterate through nodes along tree height                      */
24 for i = 0 to c.height − 1 do
25     /* Compute the highest allowed j index on step i            */
26     jhigh : int = min(i, c.jmax)
27     alpha : real = alphas[i]
28
29     /* Iterate along width between j indexes on step i          */
30     for j = −jhigh to jhigh do
31         Compute and add to QsCopy on j + 1, j, j − 1
32     end
33
34     /* Iterate along width between j indexes on step i + 1      */
35     jhigh1 : int = min(i + 1, c.jmax)
36     alpha_p1 : real = 0
37     for j = −jhigh1 to jhigh1 do
38         Aggregate alpha_p1 based on QsCopy[j]
39     end
40
41     Compute alphas[i + 1] based on alpha_p1
42     Qs = QsCopy
43     Fill QsCopy with 0
44 end
```

Backward propagation After all alphas are computed, they are carried over to backward propagation along with two arrays of size tree width. These arrays called Prices and *PricesCopy* are used to store the current and previous tree levels similarly to forward propagation. Prices are initialized to 100$ which represents the payoff at bond maturity.

Afterwards, the program iterates through time steps along the tree height starting from the end of the tree. At each time step, the values at step $i - 1$ in *PricesCopy* are computed from three values in *Prices* at step i using alpha at i and the precomputed probabilities as illustrated in figure 4.2. If the current time step is the option maturity, every computed price is discounted by the option strike price, taking care of the option type being call or put as well. Lastly, arrays Prices and *PricesCopy* are swapped and *PricesCopy* is reset to zeros for the next iteration.

Figure 4.2: Backward propagation - computing the previous step



Algorithm 4: Sequential implementation - backward propagation

```
45 /* Backward propagation                                          */
46 Prices : real[c.width]
47 PricesCopy : real[c.width]
48 Fill Prices with 100                    /* Initialize prices to 100$ */
49
50 for i = c.height − 1 to 0 do
51     jhigh : int = min(i, c.jmax)
52     alpha : real = alphas[i]
53
54     for j = −jhigh to jhigh do
55         jind : int = j + c.jmax
56         Compute res based on Prices at j + 1, j, j − 1
57
58         if Step i is the option maturity then
59             if c.type is CALL then                    /* Call option */
60                 PricesCopy[jind] = max(res − c.X, 0)
61             else                                       /* Put option */
62                 PricesCopy[jind] = max(c.X − res, 0)
63             end
64         else
65             PricesCopy[jind] = res
66         end
67     end
68     Prices = PricesCopy
69     Fill PricesCopy with 0
70 end
71
72 /* Return the calculated current option price                   */
73 return Prices[c.jmax]
```

4.2 Validation

Results obtained by running this implementation will be used for validation of the parallel algorithms, so it is important that they are fully correct. We compared our intermediate array values of *alphas*, *Qs* and *Prices* along with the final results with values provided by our supervisor and made sure they are the same within a margin of error.

Table 4.1 compares the value of a three-year put option on a nine-year zero-coupon bond with a strike price of 63: mean-reversion rate $a = 0.1$ and volatility $\sigma = 0.01$, which is an example option in Hull & White [18, pg. 706]. The left table shows book results [18, pg. 707] and the right table shows our results for the same option with different time steps. Our approach is fully numerical, while their tree results are semi-analytic, since they do not build a tree for the whole nine-year bond, but only for the three-year option and then compute the rest using analytic formulas. Despite this fact, our result for daily time steps, i.e. 365×9 steps for the full tree, are within 0.02% difference of their analytic result.

Table 4.1: Sequential results compared on a book example

Source: Compiled by the authors, based on [18, pg. 707].

Steps	Tree	Analytic
10	1.8468	1.8093
30	1.8172	1.8093
50	1.8057	1.8093
100	1.8128	1.8093
200	1.8090	1.8093
500	1.8091	1.8093

Steps per year	Results
1	1.87996
5	1.83827
10	1.81851
25	1.81120
100	1.81053
365	1.80968

Summary

This chapter provided an overview of our sequential implementation with focus on explaining the computations in forward and backward propagations and how the final results are obtained. Finally, it described how the computations and results were validated with external sources. The following chapter will describe how this implementation was adapted for a parallel one option per thread version in CUDA.

5

One Option per Thread

This chapter describes the first parallel approach that exploits only outer parallelism, i.e. it computes a batch of options in parallel where one thread prices a single option. This algorithm is therefore similar to the sequential implementation with some caveats concerning GPGPU architectures that were tackled in an iterative process producing multiple versions of the code.

5.1 Sequential Implementation to CUDA

Global memory setup

It was necessary to identify arrays being used and consider how to store them in device memory. The input is a structure of arrays of size number of options, each array representing one parameter of the options. This structure is ideal for coalesced memory access because consecutive threads will load contiguous memory, optimizing the number of memory transactions and thus speed. The algorithm itself requires two arrays of tree width size for *Qs* and *QsCopy*, and one array of tree height size for *alphas* per option. We place the three arrays in GPU's global memory, where each thread uses a single part of each array which size depends on the option being computed.

Global memory access

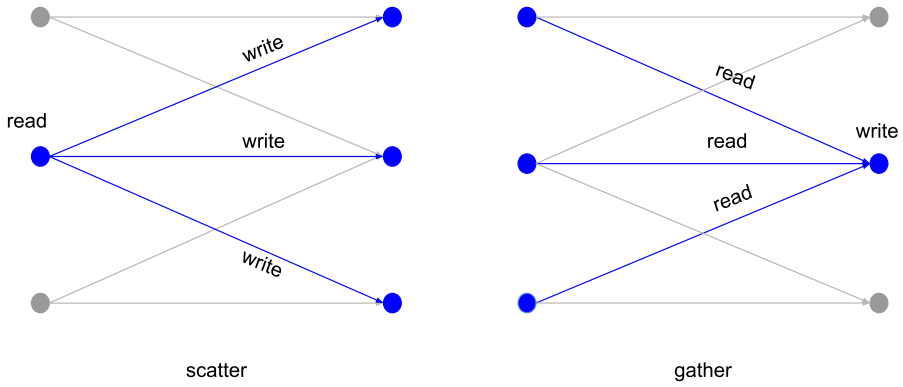
Array accesses had to be analyzed as well to avoid race conditions and optimize performance. The sequential implementation pre-computes rates and probabilities

for each node along the width and saves them. They are reused later both during the forward and backward propagations. In contrast, for this CUDA implementation we had to store the values in global memory, resulting in a slower access times. Instead, since they were accessed too few times, we opted to re-compute the values every time they were used in order to save query time and reduce the global memory consumption.

More importantly, in the forward propagation computing Qs at the next time step was done by a single computation on the current node and adding the value to three nodes in the next step as shown in figure 4.1. This performs 1 read for the computation and 3 reads followed by 3 writes for the addition. For all parallel implementations, this **scatter** pattern was replaced by a **gather** pattern illustrated in fig. 5.1, which performs 3 reads/computations and 1 write in this scenario.

This makes it necessary to pre-compute and save the values for Qs first to avoid computing the same value multiple times. The final result makes 1 read/computation/write in the pre-computation step followed by 3 reads and 1 write in computing the next time step. This approach eliminates 1 write and the need to have atomic additions, which would be necessary when exploiting inner parallelism described in chapter 6. However, since the tree is trinomial (fig. 3.2), a value can be computed from up to 5 values from the previous time step. Thus all possible types of branching have to be enumerated which makes the code more verbose and more difficult to maintain.

Figure 5.1: Comparison of scatter and gather operations



Source: Compiled by the authors

5.2 Implementations

We iteratively implemented 4 versions of this first parallel approach. They all share the same kernel but they store and access the three global arrays *Qs*, *QsCopy* and *alphas* in different ways.

The general steps for all these versions are as follows:

1. Load all options to GPU device memory.
2. Compute widths and heights for all options.
3. Allocate in global GPU memory two (expanded) arrays *Qs* and *QsCopy* which are large enough to hold option-width elements for all the options in the batch. What "large enough" means will be specified for each version.
4. Similarly, allocate in global GPU memory one array *alphas* which is large enough to hold option-height elements for all options in the batch.
5. Price all options using our CUDA kernel.
6. Copy results to host memory.

All pre-processing is performed on the GPU and is implemented using CUDA's Thrust library [1].

Version 1 - Naive

The first version stores arrays in a simple way where one thread gets contiguous parts of memory with sizes that match the computed option's width/height. Table 5.1 shows an example of storing *alphas* for 3 options of heights 2-4-3 computed by 3 threads in a single flat array. Each thread needs to know only the start index of its array chunk along with the option's width/height and then it can access array elements consecutively. The indices can be easily computed by running inclusive scans on widths and heights, obtaining also total sizes for the arrays in the process.

Table 5.1: Memory alignment in version 1

Source: Compiled by the authors

T1	T1	T2	T2	T2	T2	T3	T3	T3
α_0	α_1	α_0	α_1	α_2	α_3	α_0	α_1	α_2

This approach is very efficient in terms of storage space, however, it is very inefficient when it comes to performance. When we analyze how array elements are accessed in forward (alg. 3) and backward propagation (alg. 4), we find out that all threads access their α_0 at the same time, then move to α_1 and so on. This results in strided, un-coalesced access to global memory which ineffectively uses GPU hardware. The next 3 versions tackle this problem by padding and transposing the arrays on different levels, so as to ensure coalesced accesses to global memory whenever possible.

Version 2 - Global-level Padding

In order to make array access coalesced, the second version stores arrays padded to the maximum width/height across all options. Continuing with the example from

[1] `https://developer.nvidia.com/thrust`

above, the new alignment is illustrated in table 5.2. This obviously leads to some array elements not being used, unless the widths/heights are equal across options. However, when threads in a warp access the same array index at the same time, the access is now coalesced and can be performed in fewer memory transactions, greatly improving performance.

Table 5.2: Memory alignment in version 2

Source: Compiled by the authors

T1	T2	T3	T1	T2	T3	T1	T2	T3	T1	T2	T3
α_0	α_0	α_0	α_1	α_1	α_1		α_2	α_2		α_3	

To compute sizes of the arrays, it is necessary just to find out the maximum width/height and multiply them by the number of all options. Indexing to array elements can be simply computed as $index * optionsCount + optionIndex$.

The only downside is that the global padding might require very large memory chunks to be allocated but unused, especially if the dataset is skewed, i.e. with a small number of options that have very large widths/heights. According to our tests, discussed in detail in chapter 9, this version is up to ~10× faster than version 1 but the memory footprint is up to ~7× larger. The next two versions try to minimize the memory footprint by padding arrays on a smaller scale.

Version 3 - Block-level Padding

The third version is designed to save memory compared to the second version, while keeping memory access coalesced. Here we look at what options get computed in a single CUDA block (of up to 1024 threads). The maximum width/height of the options is computed per block and the total size of an array is then the sum of all block maxima multiplied by the block size.

The pre-processing of options is thus more complex. It is implemented using *reduce_by_key* and *transform_inclusive_scan* Thrust routines with custom operations. As a result, arrays *QsInds* and *alphasInds* storing indices to the respective

arrays are computed, created using two addition helper arrays, all of size $\lceil optionsCount/blockSize \rceil$. The computed indices represent the start of an array part specific to a block, elements can then be accessed as $blockSize * index + threadId$.

The downsides of this version are that if options in the dataset are not sorted by widths and heights, the amount of saved memory compared to version 2 may be very small while this version requires more pre-processing time and intermediate arrays. With correct sorting applied and the block size kept small, on our datasets this version uses only up to 4% more memory than version 1, while keeping up to ~10× performance lead like version 2. The last version further reduces the array padding even on bigger block sizes.

Version 4 - Warp-level Padding

The fourth version is similar to version 3 with a difference that we look at what options get computed by a single warp of 32 threads instead of the whole block. This is motivated by the fact that coalesced accesses to global memory are supported by hardware at the (half) warp level, i.e., the threads in a warp execute in lock step, and need to access consecutive memory locations in a SIMD instruction. In effect, array padding is performed at a lower granularity—that of a warp—while still preserving coalesced accesses. Depending on a dataset, the added warp-level padding might be smaller than block-level padding, in exchange for $blockSize/32$ more indices to be stored. When comparing version 4 with version 3, we achieved up to 70% decrease in memory for a CUDA block of size 1024.

Futhark implementation

This one option per thread parallel approach was also implemented in Futhark as a proof of concept and is equivalent to version 2 since Futhark uses global padding on arrays to ensure that memory accesses are coalesced. This version will be discussed further in chapter 7.

Optimizations

Choosing thread block size Since this implementation prices one option per thread, the thread block size represents the number of options being priced in parallel. Those options can have different widths and heights, leading to thread divergence on outer parallelism (heights) and inner parallelism (widths). It follows that bigger blocks might contain options that vary in widths/heights more than in small blocks, thus causing more threads to wait for completion of the block execution. On the other hand, smaller block sizes result in more thread blocks being scheduled, what adds extra overhead. Depending on the dataset, block sizes of either 128 or 256 are preferred.

Sorting input In order to (easily) reduce thread divergence, we can sort options before computation by either heights or widths. This would make one warp/block of threads compute options with similar number of execution steps and reduce the amount of time the threads have to wait for each other's execution. Our experiments show that sorting does have a large positive impact on performance, up to ~10× faster than no sorting, but the choice of sorting by height or width depends on a dataset with a difference of up to 20%.

5.3 Validation

To validate computed results from the CUDA implementations, we created a test case that uses the example mentioned in section 4.2. 100 instances of this option with gradually more time steps are first computed on the CPU using the sequential implementation from chapter 4 and then compared with results computed from all 4 versions. The test case was written using Catch2 [2] test framework for its simplicity.

The differences between floating point GPU results and CPU results must be interpreted carefully, since there are many reasons why the same sequence of operations may not be performed on the GPU and CPU, e.g. because of fused multiply-add

[2] https://github.com/catchorg/Catch2

on the GPU, rearranging operations for parallelization, higher than expected precision for computations on the CPU and rounding not required by common math operations by the IEEE 754 standard [44, pg. 16].

Despite this, we successfully validated our results using a small epsilon value of std::numeric_limits<**real**>::epsilon() ∗1000, where **real** is either single or double precision floating point number. This equals to 0.000119209 for single and 0.00000000000022204 for double precision, which makes double precision much more reliable.

Summary

This chapter provided an overview of our parallel one option per thread implementation using CUDA with focus on explaining the challenges of parallelizing the sequential implementation from chapter 4. It introduced 4 versions of this parallel implementation, each with its own advantages and disadvantages. Finally, it described how the GPU results were validated against the CPU results and the challenges of doing that. The following chapter will describe how this implementation was adapted to compute multiple options in a single CUDA thread block.

6

Multiple Options per Thread Block

This chapter describes our CUDA implementation that prices multiple options in one thread block. This approach exploits inner parallelism, leveraging fast shared memory for computations of *Qs* and *Prices*. The limitation of this version is that it cannot price options with widths bigger than the size of a thread block (1024), since every thread computes one value along the tree width. The flattening transformations applied to derive a multiple options per thread block implementations and the ones used to derive a fully flattened implementation are semantically the same, as it will be shown later in section 7.4. For this reason, we have explained them in thorough details in the context of a functional language in chapter 7, making them easier to understand. Instead, the descriptions in this chapter are more focused on CUDA-specific challenges.

6.1 CUDA-option to CUDA-multi

To implement this parallel approach, we used CUDA-option as a basis, reusing code that is not specific to each version, such as parts of pre-processing and some computations.

6.1.1 Memory setup

In this implementation one thread does not compute the whole tree for an option, rather it computes a single value on the tree width. This allows us to move the array of *Qs* to shared memory and even remove the array of *QsCopy*, as it was used to hold *Qs* temporarily between computations, and now a single Q value can be temporarily

stored in a thread register. Shared memory thus comprises of one real array of *Qs* and one *uint*16_*t* array of *flags* (used by segmented scans), both of thread block size. Global memory then stores all the options along with their pre-computed widths and heights, and the array of all *alphas*.

6.1.2 Pre-processing

After the options are loaded to GPU memory and their widths and heights are computed, they have to be split into chunks. A chunk represents one or more options whose combined widths can fit into the chosen thread block size. This process is also known as bin packing, which is a combinatorial NP-hard problem, so we decided to implement it in a simple way on the CPU, since it is not a focus of this thesis. The options are packed by a for-loop, in which an option is put into the current chunk if the sum of width sizes for options stored already in the chunk does not exceed the thread block size, otherwise a new chunk is created and the current option index is added to an array of indices. This produces an array of option indices, e.g. $[1, 3, 5]$, which describes that there are 5 options computed by 3 thread blocks, the first block will compute option 0, the second one options 1 and 2, and the third one options 3 and 4.

It is obvious that this simple implementation packs options into a smaller number of chunks/blocks if the options are sorted by width. However, it should also be desirable to sort the options by height to optimize thread divergence, since computations on the tree width are parallelized. Therefore, a better bin packing implementation might improve performance by packing more options into chunks by their widths, while optimizing thread divergence on heights, probably as a trade-off for more pre-processing time.

6.1.3 Flattening

Algorithm 5-9 outlines the kernel written for this implementation, which is much more complex than the kernel that computes one option per thread described in chapter 5. Since computed values are dependent on results from multiple threads, it

is important to synchronize threads in the block to prevent race conditions. However, special care has to be taken for this not to result in deadlocks because of thread divergence, caused by options with different widths and heights inside a block.

Initialization First, option indices *inds* for a block have to be distributed to threads based on option widths, such that each thread can compute one value in *Qs*. For example, threads 0-200 compute option 0 with width 201 and threads 201-1000 compute option 1 with width 799, leaving threads 1001-1023 unoccupied. This is done by a series of segmented scans on arrays with indices and widths, resulting in every thread in block getting an index of the option to compute (*optionIdx*) and *scannedWidthIdx* representing the start index of *Qs* (Alg. 5, 6, 7 lines 10-65). Afterwards, one thread per option initalizes the first Q and *alpha* value (Alg. 7 lines 69-71). To access the global array of *alphas*, a thread uses helper functions *getAlphaAt(index, optionIndex)* and *setAlphaAt(index, value, optionIndex)* that compute global array indices based on the specific version described later in section 6.2. It can be noted that the actions performed in the initialization have close resemblance to some of the transformations described in chapter 2.4. However, it is difficult to relate them to a single specific flattening transformation (e.g. *map*, *reduce*, *replicate*), as CUDA is not using any of those functions directly.

Forward propagation In the loop, it is important that all threads use the maximum height of options in the block, not the height of their option which might be smaller. This way block-level synchronization can be used inside the loop without the possibility of deadlocks. However, at each time step a thread has to check if it should even compute new *Qs* based on its option's height. First, a thread precomputes the next Q value, then all threads can quickly compute the next Q value from multiple *Qs* in the previous step and save it in a local variable (Alg. 8 lines 75-88). Afterwards, *Qs* are multiplied in order to be summed up using parallel segmented scan (Alg. 8 lines 90-92). Next, a new alpha value is computed from the last scanned *Qs* per option (Alg. 8 lines 95-98). Lastly, all threads set *Qs* to the new Q values (Alg. 8 line 99).

Backward propagation Before the loop, each threads sets *Qs* to 100 (called *Prices* in the sequential implementation, reusing the array in this version) (Alg. 9 lines 103-104). Then in the loop, threads have to use max height again in order to use thread synchronization. Each thread computes one price in the previous time step (if valid for current option) and after all of them are done, they store the values and move to another step (Alg. 9 lines 106-119). After the whole tree is traversed, one thread per option sets the price result in the global results array (Alg. 9 lines 122-124).

Algorithm 5: Multiple options per thread block kernel

```
1  function kernelMultipleOptionsPerBlock
   Input  : options : { [StrikePrices], [Maturities], [Lengths], [TermUnits],
                       [TermStepCounts], [ReversionRates], [Volatilities], [Types], [Widths],
                       [Heights] },
            yields : { [Prices], [Timesteps] }
            [inds], [alphas], [results]
   Output: Price approximations for the options in block
2
3  /* Initialize shared memory references                                    */
4  volatile extern ___shared___ char sh_mem[] /* Memory array for block */
5  Qs = (real *)&sh_mem                       /* Qs are the first part */
6  values = (int32_t *)&sh_mem   /* Helper int array, overwrites Qs */
7  flags = (uint16_t *)&sh_mem[blockDim.x * sizeof(real)]     /* After Qs */
8
9  /* Compute option indices and scanned widths                              */
10 idxBlock = blockIdx.x == 0 ? 0 : inds[blockIdx.x − 1]
11 idxBlockNext = inds[blockIdx.x]
12 idx = idxBlock + threadIdx.x
13 width = 0
14 if idx < idxBlockNext then  /* Don't fetch options from next block */
15 |   width = options.Widths[idx]
16 |   values[threadIdx.x] = width
17 else
18 |   values[threadIdx.x] = 0
19 end
20 ___syncthreads
```

Algorithm 6: Multiple options per thread block kernel - cont. 2

```
21 /* Scan widths inplace to obtain indices to Qs for each option */
22 scanPlus values
23 scannedWidthIdx = −1
24 if idx <= idxBlockNext then
25 |   /* Get the scanned width as in exclusive scan               */
26 |   scannedWidthIdx = threadIdx.x == 0 ? 0 : values[threadIdx.x − 1]
27 end
28 ___syncthreads
29
30 /* Send option indices to all threads                           */
31 values[threadIdx.x] = 0                    /* Clear values and flags */
32 flags[threadIdx.x] = 0
33 ___syncthreads
34
35 /* Set values to option indices and flags to option widths      */
36 if idx < idxBlockNext then
37 |   values[scannedWidthIdx] = threadIdx.x
38 |   flags[scannedWidthIdx] = width
39 else if idx == idxBlockNext and scannedWidthIdx < blockDim.x then
40 |   /* Fill the remaining part of the block (if any)            */
41 |   values[scannedWidthIdx] = threadIdx.x
42 |   flags[scannedWidthIdx] = blockDim.x − scannedWidthIdx
43 end
44 ___syncthreads
45
46 /* Scan option indices with widths as flags to distribute them */
47 sgmScanPlus values flags
48 optionIdxBlock = values[threadIdx.x]    /* Option index within block */
```

Algorithm 7: Multiple options per thread block kernel - cont. 3

```
49 /* Let all threads know about their scannedWidthIdx (Q start)  */
50 if idx <= idxBlockNext then
51 |   flags[threadIdx.x] = scannedWidthIdx
52 end
53 ___syncthreads
54 scannedWidthIdx = flags[optionIdxBlock]
55
56 /* Get the option for thread and compute its constants          */
57 OptionConstants c
58 optionIdx = idxBlock + optionIdxBlock
59 if optionIdx < idxBlockNext then
60 |   c = Compute constants for options[optionIdx]
61 else                                  /* Fake option to fill block */
62 |   c.n = 0
63 |   c.width = blockDim.x − scannedWidthIdx
64 end
65 ___syncthreads
66
67 /* Initialize Qs and alphas in one thread per option            */
68 if threadIdx.x == scannedWidthIdx and optionIdx < idxBlockNext then
69 |   alpha = compute yield at dt                 /* Initial alpha value */
70 |   setAlphaAt(0, alpha, optionIdx)
71 |   Qs[scannedWidthIdx + jmax] = 1                /* Initial Q value */
72 end
```

Algorithm 8: Multiple options per thread block kernel - cont. 4

```
73  /* Forward propagation                                                    */
74  for i = 1 to maxHeight do
75      jhigh = min(i, c.jmax)
76      j = threadIdx.x − c.jmax − scannedWidthIdx
77      /* If both height and width steps are valid for this option          */
78      if i <= c.height and j >= −jhigh and j <= jhigh then
79          alpha = getAlphaAt(i − 1, threadIdx.x)
80          Qs[threadIdx.x] *= exp(...)        /* Pre-compute Qs using alpha */
81      end
82      ___syncthreads
83
84      Q = 0
85      if i <= c.height and j >= −jhigh and j <= jhigh then
86          Q = Compute next step from Qs
87      end
88      ___syncthreads
89
90      Qs[threadIdx.x] = Q ∗ exp(...)              /* Set Qs for summation */
91      ___syncthreads
92      sgmScanPlus Qs flags                                  /* Sum up Qs */
93
94      /* Get last values of segmented scans (reduced results)              */
95      if i <= c.height and threadIdx.x == scannedWidthIdx + c.width − 1 then
96          alpha = Compute alpha from Qs[threadIdx.x]
97          setAlphaAt(i, alpha, optionIdx)
98      end
99      Qs[threadIdx.x] = Q                         /* Set Qs to new values */
100     ___syncthreads
101 end
```

Algorithm 9: Multiple options per thread block kernel - cont. 5

```
102 /* Backward propagation                                          */
103 Qs[threadIdx.x] = 100                          /* Init prices to 100$ */
104 ___syncthreads
105
106 for i = maxHeight − 1 to 0 do
107     jhigh = min(i, c.jmax) j = threadIdx.x − c.jmax − scannedWidthIdx
108     price = Qs[threadIdx.x]
109     ___syncthreads
110
111     if i <= c.height and j >= −jhigh and j <= jhigh then
112         alpha = getAlphaAt(i − 1, optionIdx)
113         price = Compute new price using alpha
114     end
115     ___syncthreads
116
117     Qs[threadIdx.x] = price                /* Set prices to new values */
118     ___syncthreads
119 end
120
121 /* Set results to prices on the first nodes                       */
122 if optionIdx < idxBlockNext and threadIdx.x == scannedWidthIdx then
123     results[optionIdx] = Qs[scannedWidthIdx + c.jmax]
124 end
```

6.2 Implementations & Validation

We derived 3 implementations that differ only in the way how the array of *alphas* in global memory is stored and accessed. It is similar to the 4 CUDA-option versions, except that the array of *Qs* is not of concern here because it is in shared memory.

Version 1 - Naive

The first simple version was created to be a starting point for the other versions and for comparison. The *alphas* are padded on a global level, thus the size equals the maximum height of all options times the number of options. Values are accessed in a straightforward way as $maxHeight * optionIndex + index$. However, this access is not coalesced and the next two versions solve that to improve performance.

Version 2 - Global-level Padding with Coalescing

The second version is similar, since the *alphas* are padded on a global level. However, the values are being accessed in transposed form as $optionsCount * index + optionIndex$. This simple change in indexing should result in performance speed-ups for no additional cost, either in terms of storage requirements or pre-processing time. Interestingly, in our experiments, this version did not lead to noticeable performance gains, probably because *alphas* are not accessed as often as *Qs* which are already in shared memory, and version 1 does have global-level padding.

Version 3 - Block-level Padding

The third version tries to improve on storage requirements as it uses padding for *alphas* on block level. It does so by computing an array of indices to *alphas*, each value representing the beginning of a segment allocated for a single block. One segment is of size maximum height of options in the block times the number of options in the block. The values are then accessed in a slightly more complicated way as $alphaIndexForBlock + optionIndexInBlock + optionsCountBlock * index$. This

should result in less global memory being allocated in trade-off for an array of indices created, and then accessed in the kernel. It might also lead to speed-ups due to improved locality of reference. For our datasets, this version performs up to ~3× faster while using up to 6x less memory than versions 1 and 2. Note that warp-level padding is not possible to achieve in this approach as one option can be computed by multiple warps.

Optimizations

Choosing thread block size Due to the nature of this approach, it is best to choose the biggest thread block size available to be able to manually pack as many options into a block as possible. All versions when compiled use more than 64 registers which effectively limits the thread block size to 512. However, we can also limit the amount of registers[1] in order to be able to use the maximum block size of 1024. During our experiments, we observed that limiting the number of registers to 32 and setting block size to 1024 gives the best performance (up to ~2.3× faster) as it lead to full occupancy of the device SMs.

Sorting input This approach eliminates thread divergence caused by different option widths by applying flattening to the inner parallelism. In order to reduce thread divergence on heights, it should be beneficial to sort the options by height before computation. However, sorting options by width might lead to more efficient packing of options into chunks. Experiments show that sorting by height gives better performance for all tested datasets, up to ~2.3× faster than no sorting.

Validation

Computed results from all 3 versions are validated by expanding the test case described in section 5.3. Furthermore, chapter 9 will describe tests using bigger datasets.

[1]We can limit the number of registers by setting the *nvcc* compiler flag *–maxrregcount=32*, as mentioned in https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.htmloptions-for-steering-gpu-code-generation

Summary

This chapter provided an overview of our parallel multiple options per thread block implementation using CUDA with focus on explaining the challenges of exploiting the inner parallelism, compared to the one option per thread implementation from chapter 5. It introduced 3 versions of this parallel implementation, the final third version being the best performing one. Finally, it described how the GPU results were validated against the CPU results using our common CUDA test case. The following chapter will introduce the third parallel approach — full flattening.

7 Full Flattening

This chapter will introduce the classic, fully-flattened implementation of the model. It has been written in Futhark, a functional, high-level data-parallel language, with the main reason for this being simplicity.

The purpose of this implementation is to further underline the importance of the locality of reference vs. thread divergence trade-off, particularly when working with large data sets. In this case, thread divergence optimization is at its peak, while locality of reference is at its worst. The implementation exploits both inner and outer parallelism, as it processes all input options at the same time.

7.1 Sequential Version to Futhark-basic

As full-flattening is difficult to imagine and compose from scratch, we have started by creating a basic, one option per thread Futhark implementation, which we refer to as *Futhark-basic*. Due to the functional language semantics, some of the optimizations done in *CUDA-option* were not possible here. This involves particularly the memory allocations, which are handled automatically by the Futhark compiler, allowing only global memory padding. Nevertheless, sorting on either width or height can still turn to be useful in order to reduce thread-divergence overhead.

As it will be shown in chapter 9, albeit its drawbacks, the performance of *Futhark-basic* has the full potential of competing with both CUDA implementations. Despite that, its main purpose is to serve as a template for deriving a fully-flattened version, which we refer to as *Futhark-flat*.

Using a functional language to represent a data-processing algorithm is rather

straight-forward. Once the options are read in the sequential version, they are iterated over by using a *for-loop*. In Futhark, this is done by a *map* operation. Since this is the first *map* encountered in the Futhark program, and it is not nested, the function being mapped will be executed in parallel over all options. Temporary arrays such as *Qs*, *QsCopy* and *alphas* are declared with Futhark's *let*-binding and every variable gets allocated in global memory. While C programs usually first allocate an array, then enter a loop to provide its initial values, in Futhark this will be done by a composition of *replicate*, *iota* or *map*.

The rest of the code translation followed the sequential code inside the loop, by replacing C++ code with its functional Futhark equivalent. Two optimizations have been done in order to simplify and reuse code, which included (i) moving the backward and forward helpers into reusable functions, together with the computation of *jvalues* and (ii) replacing the scatter in the forward helper by a gather operation in order to optimize writes. Other more complex transformations include summing up elements in an array, done by a loop in C++ and by a reduce operation in Futhark. However, none of these transformation can be described as non-trivial, as they are typical to functional languages. The complete code base for Futhark-basic can be found in file *futhark/futhark-basic.fut* and is outlined in algorithm 10.

Algorithm 10: Futhark-basic

```
1  function trinomialBasic
   Input  : options : [ { StrikePrice, Maturity, Length, TermUnit,
                         TermStepCount, ReversionRate, Volatility, Type } ],
            yields : [ { Price, Timestep } ]
   Output: Price approximations for all options
2  results = map options with function:
3      constants = compute constants from option
4      Qs = replicate width 0                                  /* Init Qs to 0 */
5      Qs[jmax] = 1                                          /* Initial Q value */
6      alphas = replicate height 0                         /* Init alphas to 0 */
7      alphas[0] = compute yield at constants.dt    /* Initial alpha value */
8      /* Forward propagation                                                  */
9      loop (Qs,alphas) for i < height do
10         /* f1, f2, f3 check for out-of-bounds on j-index and set 0s  */
11         Qs = map f1 Qs                                    /* Pre-compute Qs */
12         Qs = map f2 (iota width)   /* Get next step with fwdHelper */
13         tmpQs = map f3 Qs                /* Compute tmpQs for reduce */
14         alpha = reducePlus tmpQs                /* Compute next alpha */
15         alphas[i + 1] = alpha                        /* Set next alpha */
16     end
17     /* Backward propagation                                             */
18     Prices = replicate width 100                 /* Init Prices to 100$ */
19     loop (Prices) for i=(max_height-1) ≥ 0 do
20         /* f4 check for out-of-bounds on j-index and sets 0s      */
21         Prices = map f4 (iota width)   /* Prev step with bkwdHelper */
22     end
23     return@map Prices[jmax]                 /* Result for one option */
24 end
25 return results
```

7.2 Futhark-basic to Futhark-flat

Once Futhark-basic was validated against the C++ sequential implementation and the book example, *Futhark-flat* could be derived. In Futhark, all arrays are stored in (slow) global memory, which increases the time each thread needs for reading the data. The two implementations work with different levels of parallelism. *Futhark-basic* uses nested parallelism, where the pricing of one option happens sequentially, but multiple options are priced simultaneously with the use of a parallel map. In *Futhark-flat* many nested computations are performed in parallel, operating on enormous arrays, each containing data for all options. Helper *index* and *flag* arrays are used to indicate the start and end memory addresses of each option in such arrays, in order to allow multiple threads to work on the same array in parallel. The way threads are allocated to work on each array happens behind the scenes, automated by the *Futhark* compiler. The transformations performed in *CUDA-multi* (see chapter 6) and *Futhark-flat* should in theory be the same, as the difference between the two only lies in the way memory is allocated, the use of in-place updates in CUDA, and the amount of options processed in parallel.

The program expects to read the dataset from input as a structure of arrays. The *trinomialFlat* function is then invoked with an array of all options as input, where multiple arrays are computed from it, representing a series of *OptionConstants* mentioned in algorithm 2 in chapter 4. It can be seen that memory usage increases proportionally with the number of options. Furthermore, every array created inside *trinomialFlat*, such as *Qs* and *alphas* is also stored in global memory. In this implementation, the size of the width-dependent arrays (e.g. *Qs* and *QsCopy*) is the sum of all widths times the data type size (i.e. whether a float or double precision is used). Height-dependent arrays (*alphas*) on the other hand are computed as the number of options times the maximum height times the data type size (Note that an optimization can be made here, to avoid the padding of height-dependent arrays). While this implementation is expected to perform fast enough on small data sets, it is also expected that the performance will significantly degrade with the increased number of options, because of the excessive amount of memory it requires.

Since *trinomialFlat* works with arrays of individual properties for all options, differently from the function in *Futhark-basic*, which works with individual properties for one option at a time, several flattening principles were used in order to apply the same permutations to all elements in the arrays at the same time. After inspecting *Futhark-basic* code, we have extracted functions that have to be flattened in order to apply them on flat arrays.

Flattening transformations

It can be seen immediately on Algorithm 10 that there is a repetitive use of operations, i.e., there are two *replicates* on widths and 4 *maps* on widths. This allows efficient reuse of many of the temporary arrays, used in the process of flattening, such as *inds* and *flags*. Furthermore, it is possible to reuse arrays between the different transformations, as e.g. *replicate* and *map* are both done on the widths array. The complete pseudo-code for *Futhark-flat* can be found under algorithms 11-12 further in this section. Before the algorithm itself, we introduce the specific flattening operation we have used in order to implement it.

The code transformation starts from a *replicate* on the widths (Alg. 10 line 4). This is used to initialize the array of *Qs*. As shown in the example in section 2.4.4, first step is obtaining *ns* and *ms*. The *ns* in this case are the widths, which we can get from the option constants. We can then obtain the *inds* and the *size*. The *inds* are computed by performing an exclusive scan. Since we needed the *scanned_lens* (an inclusive scan on widths) array further in the code, we omit the exclusive scan and instead perform a map, adding the neutral element 0 in the beginning and excluding the last element of *scanned_lens* (Alg. 11 line 7):

$$len\_inds = map\ (i\ \rightarrow\ if\ (i == 0)\ then\ 0\ else\ scanned\_lens[i-1])\ (iota\ numAllOptions)$$

Furthermore, we can obtain the *size* from *last scanned_lens*. The next step is obtaining a *flags* array as *flags* = *scatter* (*replicate w* 0) *len_inds widths* (Alg. 11 line 8). The array of *ms* on the other hand is obtained by performing a *replicate w* 0.

Computing the *vals* array is the next step of the transformation. It can be seen in *Futhark-basic*, however, that the *Qs* array is not only initialized, but also the $jmax^{th}$ element of it is set to 1 (Alg. 10 lines 4-5). We have decided to simplify the process by combining these two operations. We start by creating a *sgm_inds* array by (Alg. 11 line 9-10):

$$scatter\ (replicate\ w\ 0)\ len\_inds\ (iota\ numAllOptions)$$

where *numAllOptions* is obtained through *length options*. We then perform *sgmScanPlus flags sgm_inds* which results in an array containing indexes of all options, spread across the *size* of *Qs*. In the next step, we create the array (Alg. 11 line 13)

$$q\_lens = map(x \rightarrow x - 1)\ (sgmScanPlus\ flags\ (replicate\ w\ 1))$$

which contains segmented enumerators for each of the option widths. This array is going to be useful for getting $j - indexes$ from the *Qs* arrays later on. Finally, we apply (Alg. 11 line 14)

$$Qs = map\ ((i,k) \rightarrow if\ (i == jmaxs[sgm\_inds[k]])\ then\ 1\ else\ 0)\ q\_lens\ (iota\ w)$$

This concludes the first replicate on widths.

The next step is a *replicate* on *alphas* (Alg. 10 line 6) along the height of all trees. We have approached this transformation by determining the max height from all options, which could then be used to create an enormous array of size $total\_len = numAllOptions * max\_height$ (Alg. 11 line 16), where *max_height* is obtained with a custom *reduce* on the heights, which finds the highest element. Once again we want to combine the initialization of *alphas* to 0 and the computation of *alphas*[0] with the use of the *yield curve*, however, for all options. This means that every 0^{th} element of each segment of *alphas* has to be assigned with a value from the yield curve. We can do this in 1 step, but for simplicity, the pseudo-code of *Futhark-flat* divides these first computations of alphas in two steps. We first *replicate total_len* 0 to initialize the array. Finally, we obtain (Alg. 11 line 21)
$alphas = scatter\ alphas\ (map\ (i\ \rightarrow\ i * seq\_len)\ (iota\ numAllOptions))\ yields$.

The next three operations are two maps on *Qs* and one on *iota width* (Alg. 10 line 11-13). As mentioned in 2.4.1, a nested *map* is simply the same function applied to the flat array. Since we already have the flat *Qs*, we can safely apply the maps (Alg. 12 lines 25-27). Similarly, we have *q_lens*, which contains segmented enumerators of all *widths*. Note that, as before, the mapping functions consist of a safe mechanism, checking if the j-value is going out of bounds. This can happen in the beginning of the tree construction, as j_{min} and j_{max} have not been reached yet, hence some nodes are missing, e.g. there are no nodes above node B, or below node D on fig. 3.2.

A reduce on *tmpQs* is done next (Alg. 10 line 14), which needs to be flattened in order to obtain the new alphas for the next steps of all options. Luckily, *tmpQs* are already flat, as it is a result of a *map* operation performed on *Qs*. As mentioned in chapter 2.4.4, we can obtain the *vals* in a *sgmReduce* by using a *segmented scan* (Alg. 12 line 28):

$$alphaVals = sgmscanPlus\ flags\ tmpQs$$

We remove the redundant step of computing the actual reduce result and instead write the next step of *alphas* directly, with the use of the *alpha_indsp1* helper array (Alg. 12 lines 31-34).

$$\begin{aligned} alpha\_indsp1 &= map\ f_6\ (iota\ numAllOptions) \\ alphas &= scatter\ alphas\ alpha\_indsp1\ alpha\_vals \end{aligned}$$

The backward propagation begins with the initialization of the *Prices* array, which is a *map* over *iota w*, which is already flat, hence the map function is simply applied to all flat elements (Alg. 12 line 38). Furthermore, we observe that Prices inside the backward propagation step are also computed with the use of a *map* over *iota w*, hence the same rule applies (Alg. 12 line 41).
Finally, *Prices* at all *jmax* must be returned. For this step, we can easily obtain *inds* of all root elements in *Prices* and their respective *vals*. The algorithm is finalized by returning *res* = *scatter* (*replicate numAllOptions* 0) *inds vals* (Alg. 12 line 45-49), which consists of the final price estimates for all options.

Algorithm 11: Futhark-flat

```
1  function trinomialFlat
   Input  : options : [ { StrikePrice, Maturity, Length, TermUnit,
                         TermStepCount, ReversionRate, Volatility, Type } ],
            yields : [ { Price, Timestep } ]
   Output: Price approximations for all options
2                                                  /* Get option constants */
3  (widths, heights, constants) = unzip (map f_1 options)
4  numAllOptions = length options                  /* Get length of options */
5  scanned_lens = scanPlus widths                        /* Scan widths */
6  w = last scanned_lens                /* Total size of all width arrays */
7  len_inds = map (i -> if (i == 0) then 0 else scanned_lens[i - 1]) (iota
     numAllOptions)                                 /* Get width indexes */
8  flags = scatter (replicate w 0) len_inds widths    /* Get width flags */
9  sgm_inds = scatter (replicate w 0) len_inds (iota numAllOptions)
10 sgm_inds = sgmscanPlus flags sgm_inds   /* Get segm.  width inds */
11
12 /* Get j_min to j_max range values for all options, represented in
      the range from 0 to 2 * j_max + 1                                  */
13 q_lens = map (textx -> x - 1) (sgmscanPlus flags (replicate w 1))
14 Qs = map ((i,k) -> if (i == jmaxs[sgm_inds[k]]) then one else zero) q_lens
     (iota w)                                          /* Initialize Qs */
15 /* Get max height                                                     */
16 max_height = reduce ((x,y) -> if (x > y) then x else y)) 0 heights
17 seq_len = max_height + 1     /* Compute length of max tree height */
18 total_len = numAllOptions * seq_len        /* Copmute alphas length */
19 alphas = replicate total_len 0           /* Init alphas array with 0s */
20 /* Init alphas array with initial alpha values on starting
      indexes for all options                                            */
21 alphas = scatter alphas (map (i -> i * seq_len) (iota numAllOptions)) yields
```

Algorithm 12: Futhark-flat - cont. 2

```
22 /* Forward propagation                                              */
23 for (Qs, alphas) for i < max_height do
24     /* f2, f3, f4 check for out-of-bounds on j-index and set 0s     */
25     Qs = map f2 (Qs) (q_lens) (iota w) /* Precompute Qexp on Qs array
        (uses sgm_inds, seq_len, alphas, other constants) */
26     Qs = map f3 q_lens (iota w)    /* Compute Qs in the next step */
27     tmpQs = map f4 q_lens (iota w)    /* Compute tmpQs for reduce */
28     alpha_vals = sgmscanPlus flags tmpQs
29
30     /* f5, f6 check for out-of-bounds on i-index and set 0s       */
31     alpha_vals = map f5 (iota numAllOptions)      /* Compute alphas */
32     alpha_indsp1 = map f6 (iota numAllOptions)
33     /* Update alphas at next step                                  */
34     alphas = scatter alphas alpha_indsp1 alpha_vals
35 end
36
37 /* Backward propagation                                           */
38 Prices = map f7 (iota w)                    /* Init Prices to 100$ */
39 for (Prices) for i = (max_height − 1) ≥ 0 do
40     /* f8 check for out-of-bounds on j-index and sets 0s          */
41     Prices = map f8 q_lens (iota w)  /* Compute Price at prev step */
42 end
43
44 /* Get root inds and Prices                                       */
45 (inds, vals) = unzip (map f9 (iota numAllOptions))
46 /* Scatter prices for all options                                 */
47 Prices = scatter (replicate numAllOptions 0) inds vals
48
49 return Prices                      /* Return results for all options */
```

7.3 Validation

To validate the correctness of *Futhark-flat*, we have used *futhark-bench*, a built-in tool, which is the recommended way to benchmark Futhark programs. The code is compiled using the specified compiler and ran a specified number of times for each test case. The output is validated against the output files in the *out* folder, previously created by running the Sequential C++ implementation, described in chapter 4. The average runtime is also printed to the standard output. *Futhark-flat* has been successfully validated on all input data sets.

7.4 Comparison with CUDA-multi

The core differences between *CUDA-multi* and *Futhark-flat* are (i) the number of options that can be priced in parallel and (ii) the arrangement of memory. While CUDA provides the concept of *thread blocks*, where all threads in a single block are run on the same multiprocessor (hence allowing the use of *shared thread memory* and *register memory* for faster data access), Futhark operates on a larger granularity, thus is only able to operate with the much slower global memory. This difference makes it possible to derive a multiple options per thread block in CUDA, where a chunk of options can be priced in parallel, but not in Futhark. Despite that, both implementations operate on a flat list of options. Whether this list comprises of the number of options whose widths can fit in a CUDA block (1024), or of all options that were inputted, the flattening transformations of both versions remain semantically the same.

When comparing the kernel function from *CUDA-multi* with the *trinomialFlat* function in *Futhark-flat*, the first noticeable difference is the computation of option constants. While Futhark-flat computes constants for all options in one map operation and stores them in separate arrays in global memory, *CUDA-multi* computes constants only for options in the current thread block (as intended) and stores them in fast thread registers.

In CUDA's case, option constants could also be stored in shared memory to ease

the register pressure in order to fit more blocks on SMs, but that would mean each memory access is always slower. On the other hand, we can also enforce a limit on the number of registers, but it might result in register spilling to slow global memory. However, spilled registers can still get cached in L1 cache which has the same speed as shared memory. We tried to experiment with putting the constants into shared memory and it indeed eased register pressure, but it also noticeably hurt the performance. Possible optimization in *CUDA-multi* could be some combination of shared memory for rarely accessed constants and registers for frequently accessed ones.

Another similar difference is in forward propagation, where *CUDA-multi* stores temporary values *tmpQs* in thread registers which results in much faster access.

Summary

This chapter has provided an overview of our fully flattened parallel implementation using Futhark. It has started by introducing *Futhark-basic*, a one-option per thread implementation in Futhark, created and used as a template to derive *Futhark-flat*, together with the flattening transformations applied and the method we have used to validate its correctness. At the end, it compared flattening implementation in *Futhark-flat* with the one in *CUDA-multi*. This concludes the last two algorithm implementations and leads to the methodology and experiments performed in order to determine the pros and cons of each version and more importantly their performance.

8

Experimental Methodology

This chapter will introduce the reader to the methodology used to experiment our work. This includes the data sets we have generated and their difference in distributions, which have helped us find key differences in the performances of our implementations. Later in this chapter we present the experimental environment, which includes the hardware and software we have used and finally we conclude with an evaluation of the experiments we have created and the reasons why.

8.1 Data generation

Until this point, all implementations have been tested and validated on examples from the book (*book.in*). Doing this helps determine the correctness of the implementations and gives some hints about the running times of each version, however, book.in is too small to draw any meaningful conclusions from it. To challenge the implementations we have created a simple dataset generator, which works with several different distributions. This chapter will introduce the reader to the dataset generator and the sets that were generated to put the implementations to test and help discover performance differences.

8.1.1 Generator Overview

The generator is implemented in C++ and takes 3 arguments as input - total number of options in the set, a skewness parameter and the data distribution type to be generated. The inputted number of options is used as a max limit when generating options. We have generated six essential datasets with $2^{16} = 65536$ options, which

is double the total number of available threads on the GPU that was used for this thesis[1] (32768). Additionally, we have created variations for each essential data distribution with 1000, 10000 and 30000 options in each (that is $6 * 3 = 18$ more files), which could be useful to test performance differences when files have different number of options.

The skewness parameter represents the amount (in percent) of options that will be skewed (have significantly different height and/or width than the rest of the options). This parameter is applied only for the skewed distributions, which will be described later in this chapter. Lastly, the data distribution type is used to specify the dataset which will be generated. The generator currently works with 6 different types, which will be described next in this section. Plots and statistics to show the data distributions are available in Appendix 11. Note that all data distribution plots consist of a scatter plot where each dot represents an option, and histogram plots next to their corresponding axis, indicating on the data distribution. We have additionally included statistics for both the widths and the heights of different options in the file.

8.1.2 Uniform

The uniform data distribution consists of the same option replicated multiple times. Each entry in this set has the same height and the same width as the others. All widths in the newly generated set are equal to 47 and all heights to 109. As this set is uniformly distributed, all other statistics such as variance, std, skewness are 0 for both widths and heights. The data distribution and statistics about it are shown on fig. 8.1, where it can be seen that a dot is formed in the center of the plot. While pricing the same option this many times is not practically/financially useful, there is a possibility that many real-life inputs will have a uniform distribution, where both their widths and heights will have close values. In such a case, the dots on the plot will be separated, but will still remain close to the center. This suggests that in these distributions, pricing individual options will also take similar times. In our generated set, each option should be priced in exactly the same amount of time. Furthermore,

[1] Hardware will be described later in section 8.2

since there is no difference in the heights and the widths, it is not expected that pricing this data distribution in parallel will benefit from any sorting or padding, which should be an interesting experiment.

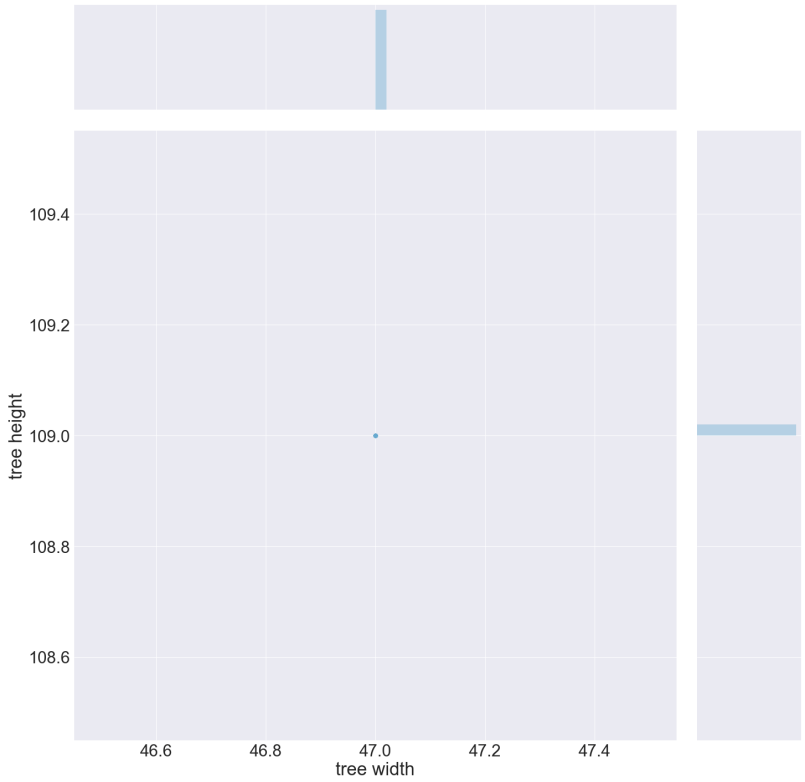


Figure 8.1: Illustration of the uniformly distributed generated file

8.1.3 Random

The random data distribution (see fig. 8.2) consists of options with both uniformly distributed random widths and uniformly distributed random heights. This dataset is interesting, as it presents a wide variety of option sizes. Both padding and sorting can benefit the processing of such a data distribution, hence it can help answer questions concerned with the various optimization techniques that can possibly improve the performance of the algorithm.

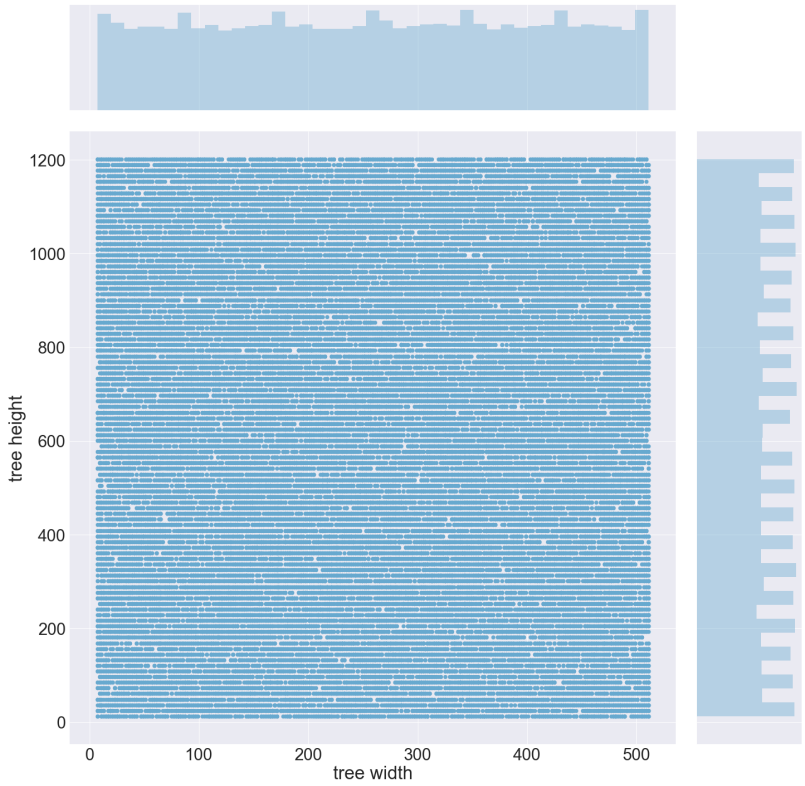


Figure 8.2: Illustration of the randomly distributed generated file

8.1.4 Random with Constant Height/Width

The following two data distributions (see Appendix 11 for plots and details) have similar structures as the random one described above, however, one of their parameters is being held in place (kept constant). In the case of constant height, the width is uniformly randomly distributed, while the height remains the same throughout all options. The other case is vice verse, where the width remains the same, while the height is randomly distributed. It will be interesting to experiment whether having a constant width or height benefits the performance of any implementation. It should also be interesting to see if sorting and padding can benefit the performance.

8.1.5 Skewed

This data distribution introduces data skewness, where a small percent of all options is significantly different than the rest. As it can be seen on fig. 8.3, the majority of the data has widths up to approx. 100 and heights up to approx. 400. Several options with much larger heights and widths stand out with a larger range for both widths and heights. This data distribution can also often occur in real life situations, where several data entries significantly deviate from the rest. This introduces problems with memory padding in some of the implementations, but can benefit from sorting both along the height and along the width. It will be interesting to experiment with the behaviour of *CUDA-multi* on similar datasets where the majority of options have small widths, hence allowing to pack and process more options in parallel.

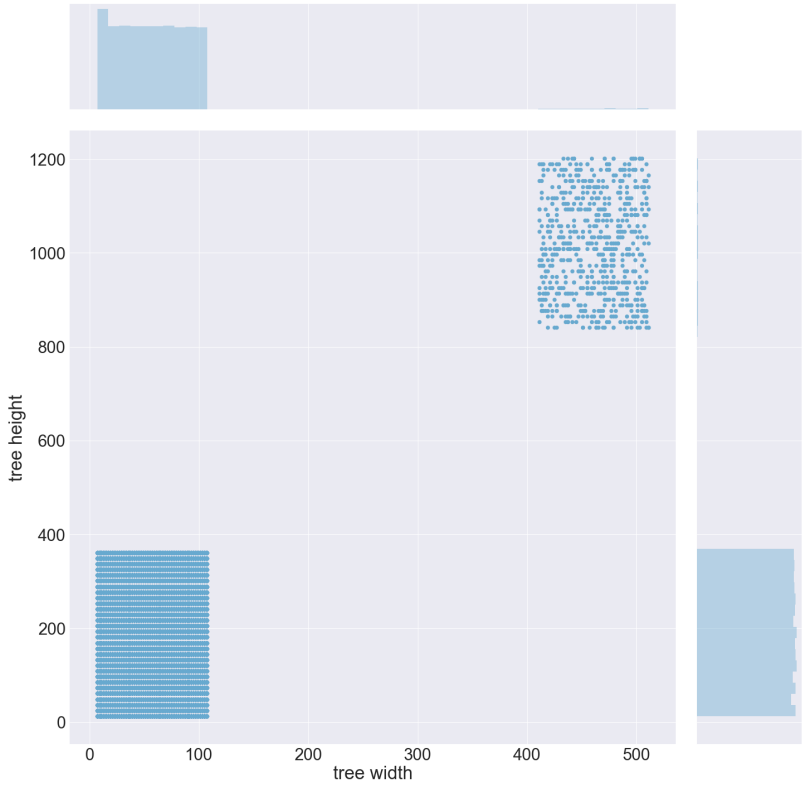


Figure 8.3: Illustration of the skewed generated file

8.1.6 Skewed with Constant Height/Width

The last two data distributions (see Appendix 11 for plots and details) introduce similar concepts as the skewed dataset. Similarly, the majority of the data has a relatively small uniform random distribution on both axes. The difference comes in the skewed part, where either the height or the width are constant, with a much larger values than the rest of the set. he free parameter (the one that is not held constant) in these cases has the same distribution as the rest of the options. These two sets can show interesting information about the dominance of either widths or heights in a dataset, and help determine whether any of the implementations perform better on widths or on heights. It should also be interesting to observe the runtime when sorting and padding are used on each of the different axes.

8.2 Experimental Environment

Hardware In order to test our code, we have used the GPU cluster at DIKU[2]. It is composed of 5 servers with multicore CPUs and 2 GPUs per machine. We have only used four of the machines (GPU01-04) each with 2 x *nVidia GeForce GTX 780 GPUs*. The complete hardware specification for each of the 4 GPUs is described below:

- **Case**: 1x Supermicro SYS-7047GR-TPRF, 4U/Tower barebone LGA2011, 2x1620W PSU, 8x3.5" htswp trays
- **CPU**: 2x Intel Xeon E5-2650v2, 8-core CPU, 2.6GHz, 20MB cache, 8GT/s QPI
- **RAM**: 8x Samsung 16GB DDR3(128GB total) 1866MHz Reg. ECC server module
- **GPU**: 2x nVidia GeForce GTX 780 Ti, 3072MB, 384 bit GDDR5, PCI-E 3.0 16x, 15 streaming multiprocessors with 2880 CUDA cores (single precision)

[2]Find more information on `https://di.ku.dk/it/documentation/gpu/`

and 960 CUDA cores (double precision) and compute capability 3.5

- **SSD**: 1xIntel S3500 serie 240GB SATA
- **HDD**: 1x Seagate Constellation ES.3 4TB 7200RPM SATA 6Gb/s 128MB cache 3,5"

The GTX 700 series were first released in 2013 and GPU technology has noticeably improved since. Despite that, both CUDA and Futhark provide portability, allowing to easily switch hardware, or scale the solution, allowing to run the code on even more modern hardware, without the need to re-write it.

Software We also turn to software as another aspect of portability. Even though CUDA and Futhark allow code to be run on many different architectures and operating systems, it is often a good idea to align compiler versions on different systems. All experiments described in chapter 9 have been run on **Red Hat Enterprise Linux Server 7.5 (Maipo)** with a **Linux 3.10.0-862.6.3.el7.x86_64** kernel. All CUDA programs have been compiled by **Cuda compilation tools, release 9.2, V9.2.148** using **C++11** and all Futhark programs by the **Futhark 0.6.0** compiler. Note that the use of older versions of both may result in compile errors, as we have used modern language features introduced in the newer releases. The same applies for newer compiler versions, since we cannot guarantee that neither Nvidia products, nor the Futhark language are going to be backward compatible.

Experiments With the large number of combinations between datasets, computation precision, implementations, optimizations and more, we have created a testing framework for our CUDA implementations. It runs one combination at a time and writes the measurements to a file. For each run we obtain the name of the file; the precision; number of registers; the implementation version; the block size; the sort option; kernel time in microseconds; total time in microseconds and the total allocated memory in bytes. Futhark implementations are tested using the built-in *futhark-bench* tool.

Evaluation Throughout all experiments, we have tested for multiple performance factors:

- Using both **float** and **double** precision. Although we do not expect that computing doubles will perform better than floats, it has been interesting to observe how the algorithms perform on different levels of precision.
- To determine the highest performing implementation, we evaluate the **runtimes** of all approaches. For simplicity, we align our CUDA findings with *futhark-bench*'s measurement strategy - excluding input reading, device context initialization, copying of input and output to/from the device and writing the output. Pre-computations of the data, such as sorting and index computations/padding are still included in our results. It has been interesting to observe the speed-ups achieved when different variations of sorting were applied, when different types of padding were applied and when different block sizes were used[3]. All runtimes are in seconds, as we have found that measure to be best visually representative on the plots.
- Another important measurement we have considered was **memory**. We have created multiple versions both for *CUDA-option* and *CUDA-multi*, where memory was optimized, thus it has been interesting to observe the impacts of each version. Memory has been measured in megabytes (MB).

Summary

This chapter provided an overview of the methodology used in order to set up the experiment environment. This has included the datasets we have generated in order to put the implementations to a test, the hardware and software used to run them and a brief description of what exactly we have put to the test in order to measure performance. The following chapter will introduce the actual experiments and elab-

[3]Note that *CUDA-multi* is always expected to perform better with the largest block size available - namely 1024, hence we have tested different block-sizes only for *CUDA-option*.

orate on the results we have obtained, in order to determine different performance characteristics and obtain an empirical validation for answering the thesis questions.

9

Experimental Results

This chapter will introduce the actual experiments we have conducted in this project. It will first discuss the performance benefits of the different optimization techniques we have used and generalize with the speed-ups achieved over the sequential set-up we have discussed in chapter 4. The chapter will conclude with a summary of our findings aiming to answer the thesis questions stated in the beginning of the report.

As mentioned in the previous chapter, we ran our 5 programs on 7 datasets with many combinations of parameters such as version, sorting and block size, which gave us a large number of results (3542 in total). Therefore, it is important to show only the results that matter the most, and we decided to make plots that show the best runtimes and average memory we measured for the combination shown on the plot. Even though we have created a variety of plots to support our findings, in this chapter we will only show the ones which can clearly show the differences we discuss. Since we have distributions of different nature, we expect to see opposing differences for some optimizations. Therefore, to underline the trade-offs and the importance of each optimization, we have tried to show on plots the two datasets that have shown the most contrast in each experiment. Nevertheless, all results can be found in tables under the two chosen plots and all plots can be found in Appendices B, C and D.

9.1 CUDA-option Performance

9.1.1 Coalescing

As it can be seen on fig. 9.2, coalescing (versions 2, 3 and 4 all provide memory coalescing) has proven to be a successful technique for *CUDA-option*. The highest performance was achieved on the random dataset with constant height, where we have achieved as much speed-up as ~10× on floats and ~2× on doubles. The least impact by coalescing has shown to be on the skewed datasets, even though still ~2× faster.

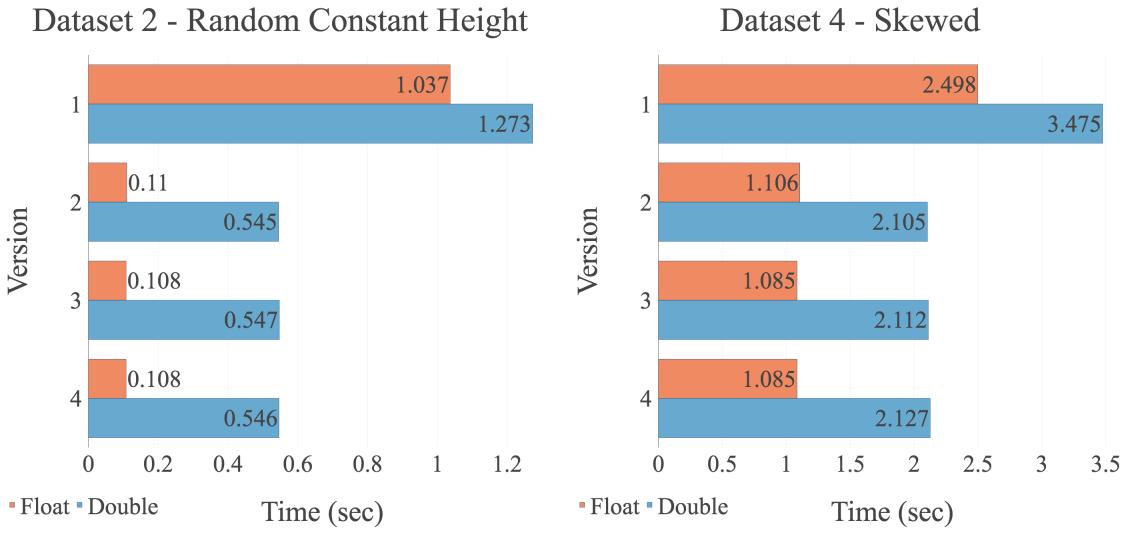


Table 9.1: Float runtime for all 7 datasets (in seconds)

version	DAT 0	DAT 1	DAT 2	DAT 3	DAT 4	DAT 5	DAT 6
1	0.347	12.64	1.037	2.05	2.498	0.973	0.879
2	0.055	2.125	0.11	0.299	1.106	0.248	0.234
3	0.054	2.043	0.108	0.294	1.085	0.243	0.227
4	0.055	1.993	0.108	0.293	1.085	0.243	0.227

Table 9.2: Double runtime for all 7 datasets (in seconds)

version	DAT 0	DAT 1	DAT 2	DAT 3	DAT 4	DAT 5	DAT 6
1	0.493	15.329	1.273	3.026	3.475	1.079	1.101
2	0.248	8.355	0.545	1.497	2.105	0.66	0.632
3	0.249	8.427	0.547	1.502	2.112	0.661	0.636
4	0.252	8.409	0.546	1.504	2.127	0.661	0.642

Figure 9.2: Performance impact of coalescing in *CUDA-option* (best runtimes)

9.1.2 Global-level Padding

While the runtime improvements were significant after the coalescing, memory consumption had also increased with the usage of global-level padding in version 2 (up to ~7× more for both floats and doubles), in comparison to version 1. This can be seen on fig. 9.4 showing the memory consumption difference between all versions for datasets 1 - Random and 4 - Skewed. This was obviously unwanted and it was therefore necessary to try and reduce the memory usage.

9.1.3 Block-level Padding

As described in chapter 5, block-level padding (version 3) is a memory optimization technique, which attempts to reduce the large memory usage, while preserving coalesced memory access. This version has proven to significantly reduce memory size, as it can be seen on fig. 9.4, while the performance remained similar. On average, requiring ~1.4× less memory than version 2 for floats and ~3× for doubles, but still ~2× more than version 1.

9.1.4 Warp-level Padding

Warp-level padding (version 4) was also described in chapter 5, where arrays were padded per 32 threads (warp) instead of a block, compared to version 3. On average, it used ~1.4× less memory than block-level padding, bringing it down to ~1.4× more memory than no padding (fig. 9.4), while performance remained similar again (fig. 9.2).

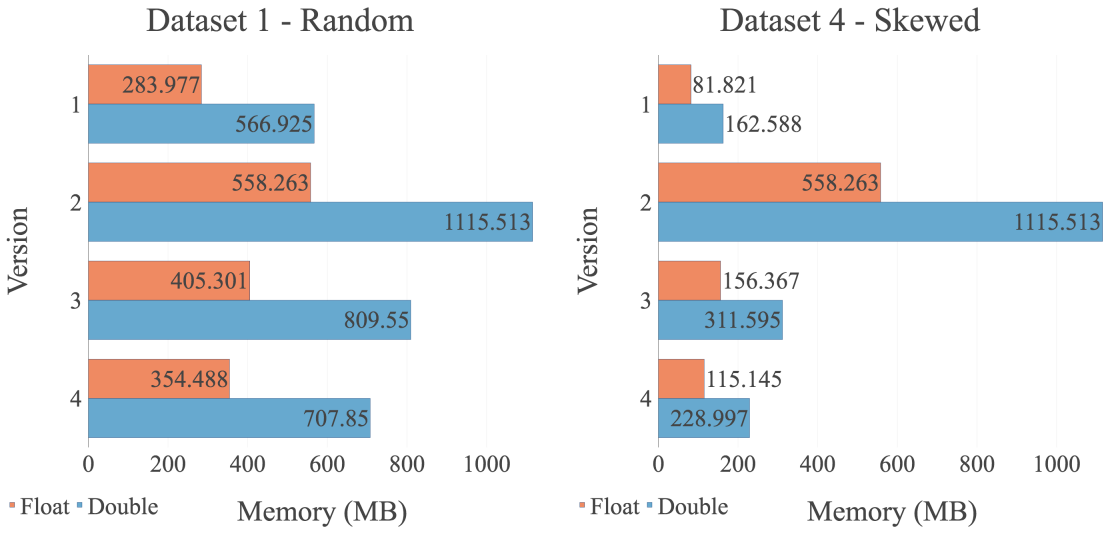


Table 9.3: Float memory size for all 7 datasets (in MB)

version	DAT 0	DAT 1	DAT 2	DAT 3	DAT 4	DAT 5	DAT 6
1	53.263	283.977	159.031	177.772	81.821	80.399	80.043
2	53.263	558.263	285.263	326.263	558.263	356.263	347.263
3	53.521	405.301	184.502	207.782	156.367	149.116	139.867
4	53.536	354.488	183.127	206.254	115.145	108.296	106.469

Table 9.4: Double memory size for all 7 datasets (in MB)

version	DAT 0	DAT 1	DAT 2	DAT 3	DAT 4	DAT 5	DAT 6
1	105.513	566.925	317.034	354.532	162.588	159.747	159.035
2	105.513	1115.513	569.513	651.513	1115.513	711.513	693.513
3	105.771	809.55	367.724	414.292	311.595	297.848	278.65
4	105.786	707.85	364.956	411.222	228.997	215.484	211.654

Figure 9.4: Memory impact of padding in *CUDA-option* (average global memory size)

9.1.5 Sorting

An important optimization technique which affects thread divergence is sorting. We tested 4 types of sorting:

1. ascending height first, width second
2. descending height first, width second
3. ascending width first, height second
4. descending width first, height second

As shown on fig. 9.6, dataset 0 - Uniform data distribution, is negatively affected by it, as we also have to spend a few milliseconds to sort the data and gain no actual benefit for doing that. This is expected, since all options in the file have the exact same width and height. We expect, however, that sorting similar data distributions, where the distribution of data is centered around the middle (similar to fig. 8.1 but not just one point), can lead to a small, insignificant speed-up. Hence the only situation where we found sorting useless was when pricing a large number of duplicated options, which is impractical.

In contrast, sorting has proven to be quite successful on other datasets. However, as mentioned in chapter 5, and shown on fig. 9.6, choosing a good sorting strategy can be data-sensitive. We can see on the plot for dataset 5 - Skewed constant height that any sorting is better than no sorting at all. Despite that, different sorting options can produce different speed-ups with up to 20% of margin.

Due to the thread divergence, we risk running large options (e.g. options with large heights) in the end, which likely results in CUDA waiting on a few threads to process them, while a lot of other threads are idling. Intuitively, reducing the idle time for threads can produce significant speed-ups, hence we expect that sorting by descending should be generally better in most cases. Looking at the additional plots in Appendix B.3 or tables in fig. 9.6, we can confirm our observation, as sorting by both width and height in descending order tends to be faster.

Additionally, we show that pre-processing (primarily affected by sorting) takes a negligible amount of time in all cases, as seen in fig. 9.7, where the highest average pre-processing time for doubles takes 12.3 milliseconds when sorting by ascending width.

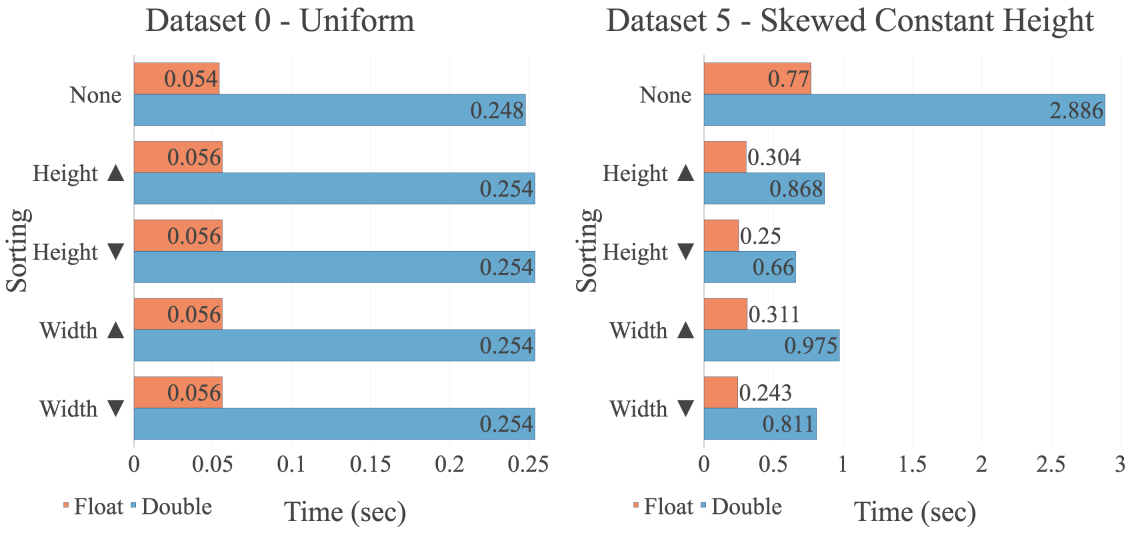


Table 9.5: Float runtime for all 7 datasets (in seconds)

sort	DAT 0	DAT 1	DAT 2	DAT 3	DAT 4	DAT 5	DAT 6
None	0.054	8.422	1.037	0.644	2.498	0.77	0.706
Height ▲	0.056	2.554	0.115	0.337	1.181	0.304	0.299
Height ▼	0.056	1.993	0.108	0.294	1.129	0.25	0.243
Width ▲	0.056	2.585	0.116	0.337	1.138	0.311	0.285
Width ▼	0.056	2.069	0.108	0.293	1.085	0.243	0.227

Table 9.6: Double runtime for all 7 datasets (in seconds)

sort	DAT 0	DAT 1	DAT 2	DAT 3	DAT 4	DAT 5	DAT 6
None	0.248	26.203	1.273	2.973	9.776	2.886	2.575
Height ▲	0.254	9.18	0.551	1.551	2.322	0.868	0.862
Height ▼	0.254	8.355	0.545	1.497	2.162	0.66	0.696
Width ▲	0.254	10.752	0.55	1.551	2.279	0.975	0.848
Width ▼	0.254	9.96	0.545	1.497	2.105	0.811	0.632

Figure 9.6: Performance impact of sorting in *CUDA-option* (best runtimes)

Table 9.7: Float pre-processing time for all 7 datasets (in **milliseconds**)

sort	DAT 0	DAT 1	DAT 2	DAT 3	DAT 4	DAT 5	DAT 6
None	1.325	2.947	1.597	1.619	1.69	1.644	1.657
Height ▲	3.022	5.458	3.376	3.396	3.46	3.436	3.477
Height ▼	3.039	5.407	3.388	3.403	3.439	3.424	3.483
Width ▲	3.03	6.131	3.398	3.425	3.458	3.459	3.494
Width ▼	3.033	5.297	3.375	3.418	3.453	3.441	3.479

Table 9.8: Double pre-processing time for all 7 datasets (in **milliseconds**)

sort	DAT 0	DAT 1	DAT 2	DAT 3	DAT 4	DAT 5	DAT 6
None	1.283	2.678	1.788	1.87	2.036	1.807	1.795
Height ▲	7.098	10.664	7.586	7.592	7.629	7.524	7.516
Height ▼	7.104	9.206	7.601	7.593	7.636	7.516	7.521
Width ▲	7.21	12.304	7.561	7.703	7.639	7.551	7.524
Width ▼	7.107	11.987	7.547	7.622	7.632	7.567	7.511

Figure 9.7: Pre-processing cost in *CUDA-option* (average pre-processing times)

9.1.6 Block Sizes

Since *CUDA-option* is concerned with running a single option per thread, the block size determines the number of options that can be processed in parallel. This was also described at the end of chapter 5, together with an explanation of the trade-offs for choosing different block sizes. The two experiments we have chosen to display in this section (see fig. 9.9) show that the optimal block-size is also dependent on the data. Even though the impact of block sizes is not significant (i.e. ~1.5× for floats and ~1.2× for doubles), dataset 1 - Random performs best with a block size of 128. In contrast, dataset 5 - Skewed Constant Height operates best with block size of 512. Despite that, smaller block sizes prevail to show optimal performance on all other experiments (as seen in Appendix C.2 or tables in fig. 9.9).

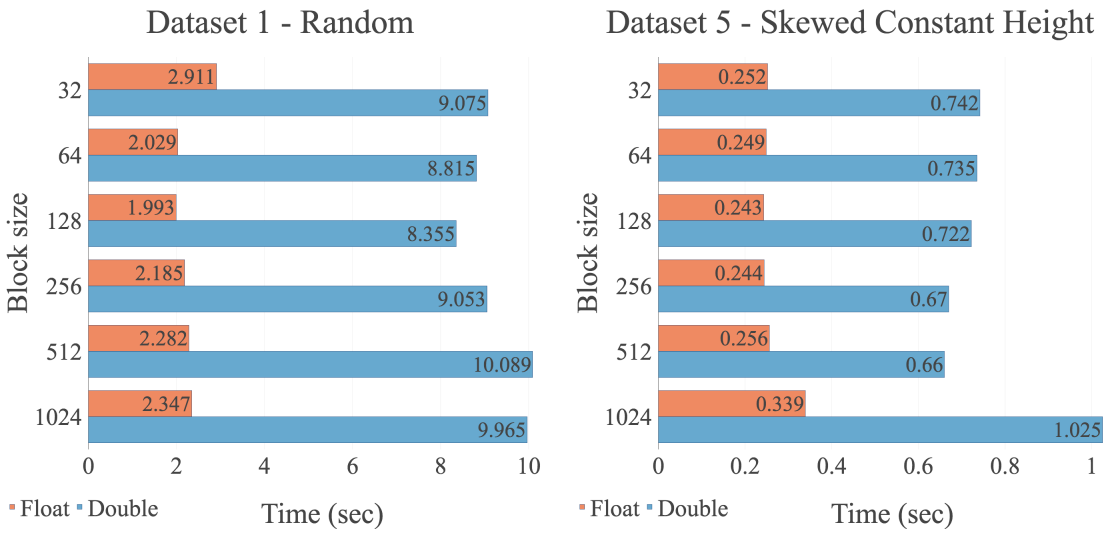


Table 9.9: Float runtime for all 7 datasets (in seconds)

block	DAT 0	DAT 1	DAT 2	DAT 3	DAT 4	DAT 5	DAT 6
32	0.083	2.911	0.176	0.491	1.085	0.252	0.233
64	0.054	2.029	0.108	0.294	1.093	0.249	0.23
128	0.054	1.993	0.108	0.293	1.092	0.243	0.227
256	0.055	2.185	0.11	0.294	1.115	0.244	0.229
512	0.055	2.282	0.112	0.294	1.174	0.256	0.23
1024	0.059	2.347	0.113	0.302	1.56	0.339	0.306

Table 9.10: Double runtime for all 7 datasets (in seconds)

block	DAT 0	DAT 1	DAT 2	DAT 3	DAT 4	DAT 5	DAT 6
32	0.257	9.075	0.548	1.517	2.153	0.742	0.696
64	0.248	8.815	0.545	1.5	2.156	0.735	0.702
128	0.248	8.355	0.545	1.497	2.105	0.722	0.682
256	0.259	9.053	0.55	1.514	2.327	0.67	0.632
512	0.267	10.089	0.562	1.544	3.312	0.66	0.722
1024	0.316	9.965	0.613	1.638	5.431	1.025	1.048

Figure 9.9: Performance impact of different block sizes in *CUDA-option* (best runtimes)

9.2 CUDA-multi performance

9.2.1 Coalescing

Both version 1 and version 2 of *CUDA-multi* use global-level padding to allocate height-dependent arrays (alphas). Their difference comes in the memory coalescing. Interestingly enough, all experiments we have created to test the performance benefits of coalescing on *CUDA-multi* have shown no significant improvement. This can be seen on fig. 9.11. The only reasonable explanation for this is that *CUDA-multi* does not access that many alphas at the same time. For example, in *CUDA-option* on a uniform dataset, using a block size of 1024 will make all 1024 threads access alphas at the same time, which makes it easy to optimize data transfers by coalescing. On the other hand, in *CUDA-multi* the number of alphas accessed at the same time depends primarily on the number of options in the block and their widths. Nevertheless, this number is going to be much smaller, making optimizations much less significant.

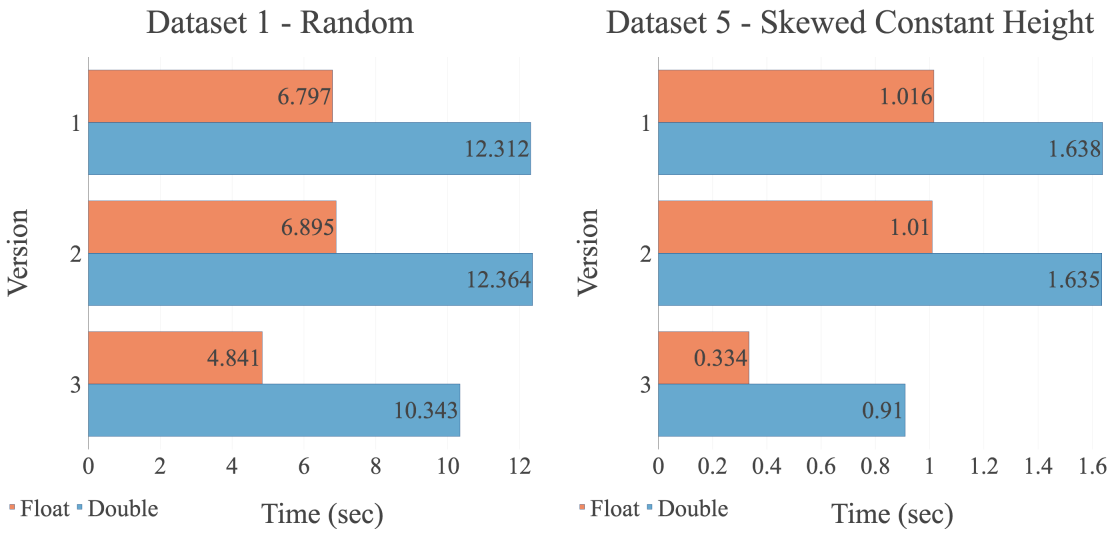


Table 9.11: Float runtime for all 7 datasets (in seconds)

version	DAT 0	DAT 1	DAT 2	DAT 3	DAT 4	DAT 5	DAT 6
1	0.146	6.797	0.743	1.192	1.138	1.016	0.456
2	0.148	6.895	0.758	1.204	1.135	1.01	0.462
3	0.15	4.841	0.75	0.888	0.45	0.334	0.336

Table 9.12: Double runtime for all 7 datasets (in seconds)

version	DAT 0	DAT 1	DAT 2	DAT 3	DAT 4	DAT 5	DAT 6
1	0.439	12.312	1.367	2.794	1.877	1.638	1.014
2	0.439	12.364	1.374	2.798	1.877	1.635	1.016
3	0.444	10.343	1.388	2.465	1.147	0.91	0.896

Figure 9.11: Performance impact of memory coalescing on *CUDA-multi* (best runtimes)

9.2.2 Global-level Padding

As noted in the previous section, both version 1 and 2 use global-level padding and no memory-optimizing technique has been applied. Hence we do not expect any memory differences between the two. Fig. 9.13 further confirms our expectations.

9.2.3 Block-level Padding

Similarly to *CUDA-option*, we can apply block-level padding in an attempt to optimize the memory usage of the implementation. We can see on fig. 9.13 that on the random data distribution we can achieve up to ~2× memory improvement for both floats and doubles. Furthermore, on the skewed dataset, we can see memory improvements up to ~5× for both floats and doubles. Due to the improved locality of reference, as mentioned in chapter 6.2, we also expect to see runtime improvements after this optimization. Indeed, if we look at fig. 9.11 we can see that version 3 performs up to ~3× faster on floats and ~1.8× on doubles. Furthermore, we can observe that all skewed datasets are significantly and positively impacted by version 3.

Intuitively, the next step would be to experiment with warp-level padding and attempt to improve the memory optimization even further. However, since one option can be computed by multiple warps, this optimization is not available in *CUDA-multi*.

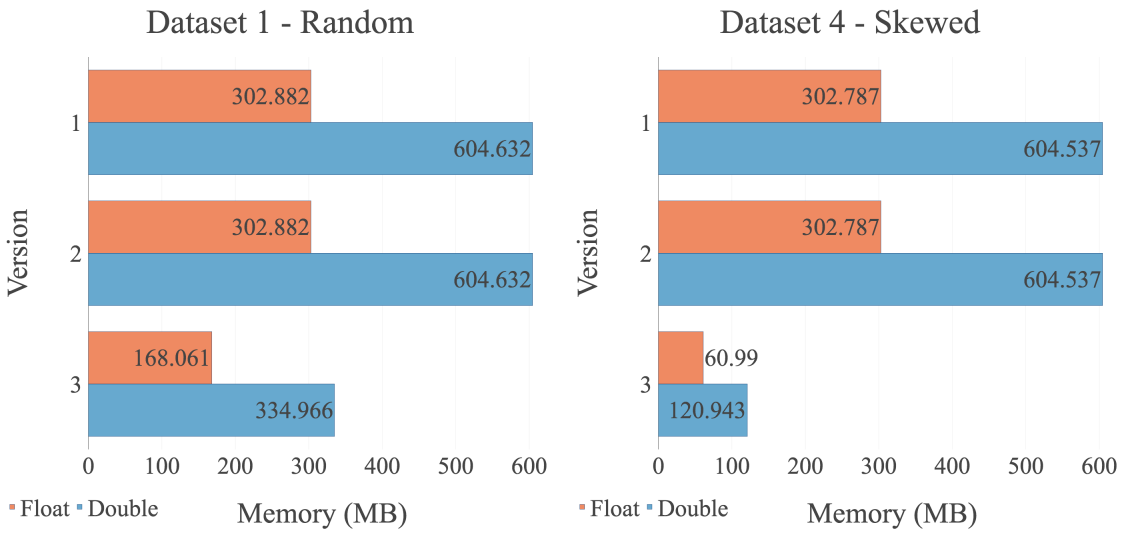


Table 9.13: Float memory size for all 7 datasets (in MB)

version	DAT 0	DAT 1	DAT 2	DAT 3	DAT 4	DAT 5	DAT 6
1	29.781	302.882	29.881	302.781	302.787	302.785	92.787
2	29.781	302.882	29.881	302.781	302.787	302.785	92.787
3	29.8	168.061	30	180.213	60.99	66.086	57.208

Table 9.14: Double memory size for all 7 datasets (in MB)

version	DAT 0	DAT 1	DAT 2	DAT 3	DAT 4	DAT 5	DAT 6
1	58.531	604.632	58.632	604.531	604.537	604.535	184.537
2	58.531	604.632	58.632	604.531	604.537	604.535	184.537
3	58.55	334.966	58.75	359.377	120.943	131.189	113.374

Figure 9.13: Memory impact of global-level padding *CUDA-multi* (average global memory size)

9.2.4 Sorting

As mentioned at the end of chapter 6, reducing the thread divergence on heights can be done by sorting by height, while sorting by widths can improve the options packing. We can see on fig. 9.15, that any sorting gives up to ~1.7× speed-up for floats and up to ~1.4× for doubles. The differences between sorting options is insignificant, however we can see that sorting by height often results in a slightly better speed-up. Supplementary plots in Appendix. C.3 and tables in fig. 9.15 further underline this.

As also seen in section 9.1.5, uniform datasets obviously do not benefit from sorting, but it has been shown that the process of sorting only slightly degrades performance.

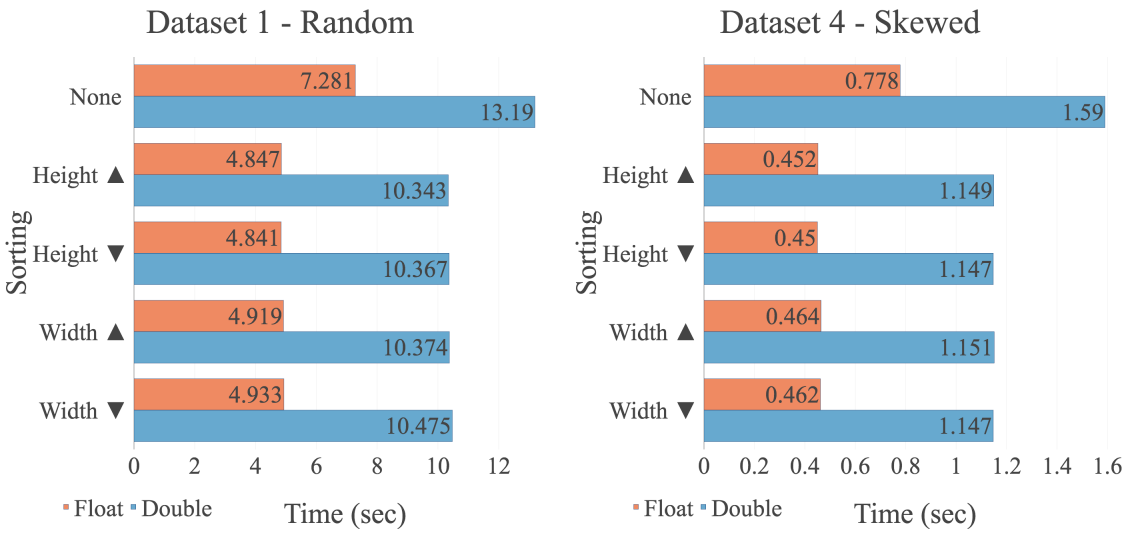


Table 9.15: Float runtime for all 7 datasets (in seconds)

sort	DAT 0	DAT 1	DAT 2	DAT 3	DAT 4	DAT 5	DAT 6
None	0.146	7.281	0.778	1.551	0.778	0.769	0.6
Height ▲	0.148	4.847	0.743	0.894	0.452	0.342	0.337
Height ▼	0.148	4.841	0.743	0.89	0.45	0.334	0.336
Width ▲	0.148	4.919	0.743	0.893	0.464	0.366	0.344
Width ▼	0.148	4.933	0.743	0.888	0.462	0.365	0.345

Table 9.16: Double runtime for all 7 datasets (in seconds)

sort	DAT 0	DAT 1	DAT 2	DAT 3	DAT 4	DAT 5	DAT 6
None	0.439	13.19	1.428	3.442	1.59	1.484	1.265
Height ▲	0.444	10.343	1.367	2.474	1.149	0.921	0.897
Height ▼	0.445	10.367	1.369	2.465	1.147	0.91	0.896
Width ▲	0.445	10.374	1.367	2.476	1.151	0.94	0.896
Width ▼	0.444	10.475	1.37	2.466	1.147	0.939	0.899

Figure 9.15: Performance impact of sorting padding *CUDA-multi* (best runtimes)

9.2.5 Block Sizes

The experiments on block sizes (see fig. 9.17) have proven that using block size of 1024 and limiting registers to 32 is always preferable, compared to using block size of 512 without limiting registers. We can see a speed-up of up to ~1.4× on floats and ~1.5× on doubles. Bigger block size allows packing more options in one block, while limiting the number of registers helps to achieve full occupancy on SMs, proving our performance expectations from chapter 6.2.

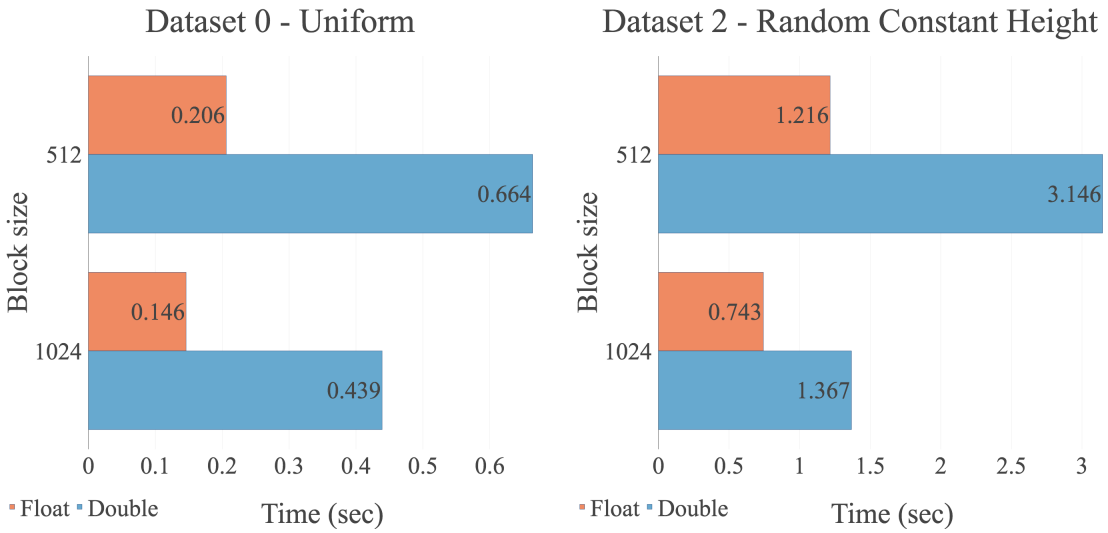


Table 9.17: Float runtime for all 7 datasets (in seconds)

block	DAT 0	DAT 1	DAT 2	DAT 3	DAT 4	DAT 5	DAT 6
512	0.206	7.603	1.216	1.247	0.631	0.467	0.471
1024	0.146	4.841	0.743	0.888	0.45	0.334	0.336

Table 9.18: Double runtime for all 7 datasets (in seconds)

block	DAT 0	DAT 1	DAT 2	DAT 3	DAT 4	DAT 5	DAT 6
512	0.664	20.91	3.146	3.869	1.894	1.444	1.44
1024	0.439	10.343	1.367	2.465	1.147	0.91	0.896

Figure 9.17: Performance impact of block sizes on *CUDA-multi* (best runtimes)

9.2.6 Bin Packing

As mentioned in chapter 6, in *CUDA-multi*, all options are packed sequentially before invoking the kernel. Ideally, we can optimize this by implementing it in parallel. Despite that, we show on tables 9.19 and 9.20 that pre-processing of the data has had insignificant impact on the performance, delaying it up to ~7.3 milliseconds on average. We can also note here that (i) the times shown in the two tables show the average pre-processing time for each version (hence there may be larger runtimes than 7.3, but still insignificant) and (ii) the runtimes include sorting too, as sorting can also have an impact on bin packing.

Table 9.19: Float pre-processing time for all 7 datasets (in **milliseconds**)

version	DAT 0	DAT 1	DAT 2	DAT 3	DAT 4	DAT 5	DAT 6
1	2.734	3.688	3.211	3.251	3.314	3.29	3
2	2.701	3.657	3.268	3.264	3.283	3.255	3.089
3	2.667	3.517	3.381	3.001	3.015	2.885	3.045

Table 9.20: Double pre-processing time for all 7 datasets (in **milliseconds**)

version	DAT 0	DAT 1	DAT 2	DAT 3	DAT 4	DAT 5	DAT 6
1	5.93	7.257	6.627	6.706	6.762	6.675	6.198
2	5.985	7.3	6.603	6.676	6.658	6.702	6.161
3	6.174	6.967	6.659	6.56	6.251	6.363	6.293

Figure 9.18: Pre-processing cost in *CUDA-multi* (average pre-processing times)

9.3 Parallel Speed-up

First and foremost, we underline the performance benefits that porting code to GPGPU hardware can provide. Figures 9.19 and 9.20 show speed-up achieved by all parallel implementations compared to the sequential implementation, for both single (floats) and double precision respectively. Even though the sequential code could be further optimized, it does already provide a useful metric for comparing datasets with different workloads. As it can be seen, *CUDA-option* is the principal winner of this comparison, with speed-ups up to ~529× faster than the sequential implementation on floats and up to ~87× faster on doubles. *CUDA-multi* prevails on dataset 4 - Skewed with a speed-up of ~161× on floats and ~54× on doubles.

Here we find out that if the data distribution is skewed on both heights and widths, it is better to exploit both levels of parallelism because this allows to further optimize thread divergence. This feature of the dataset can easily be computed using the **skewness** statistical measure. Dataset 4 - Skewed has 2.40 skewness on heights and 5.19 on widths (as seen in Appendix A.5), in contrast with other datasets that have skewness close to zero on heights or widths or both.

Additionally, the experiments we have performed on smaller datasets[1] (see fig. 9.21 and 9.22) have exposed that *CUDA-multi* outperforms *CUDA-option* up to ~13× in runtime when the datasets are small enough. Furthermore we can see that the performance of *CUDA-option* and *Futhark-basic* increases with the number of options, while *CUDA-multi* is not affected by it. This leads us to believe that there exists a certain threshold for each data distribution, which can be used to dynamically switch between *CUDA-multi* and *CUDA-option* for optimal performance.

[1]Even more experiments can be seen in Appendix C.4

Figure 9.19: Comparisons of parallel speed-ups over sequential implementation using single precision. CUDA-option is faster on all datasets except 4 - Skewed, where CUDA-multi is faster.

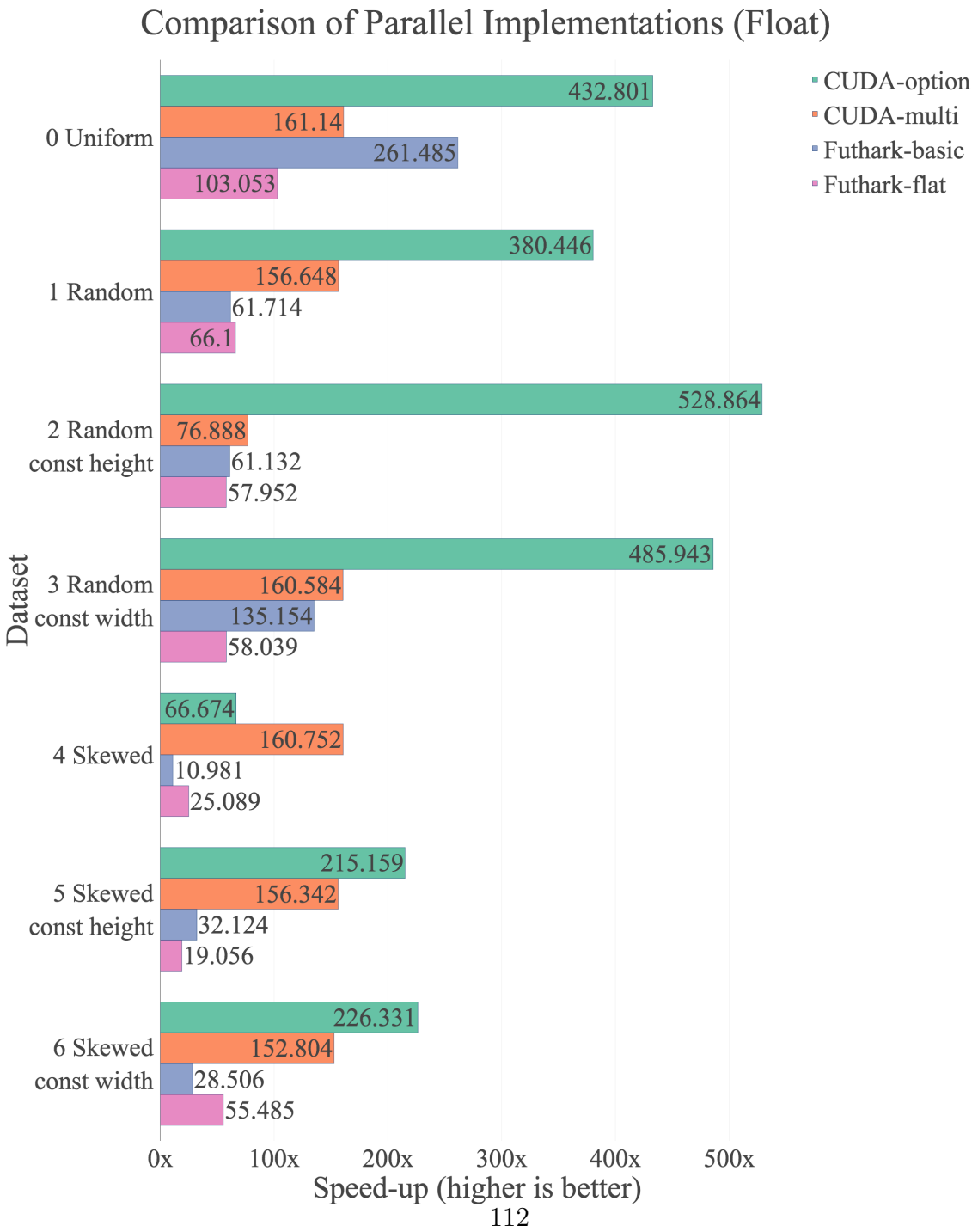


Figure 9.20: Comparisons of parallel speed-ups over sequential implementation using double precision. CUDA-option is faster on all datasets except 4 Skewed, where CUDA-multi is faster.

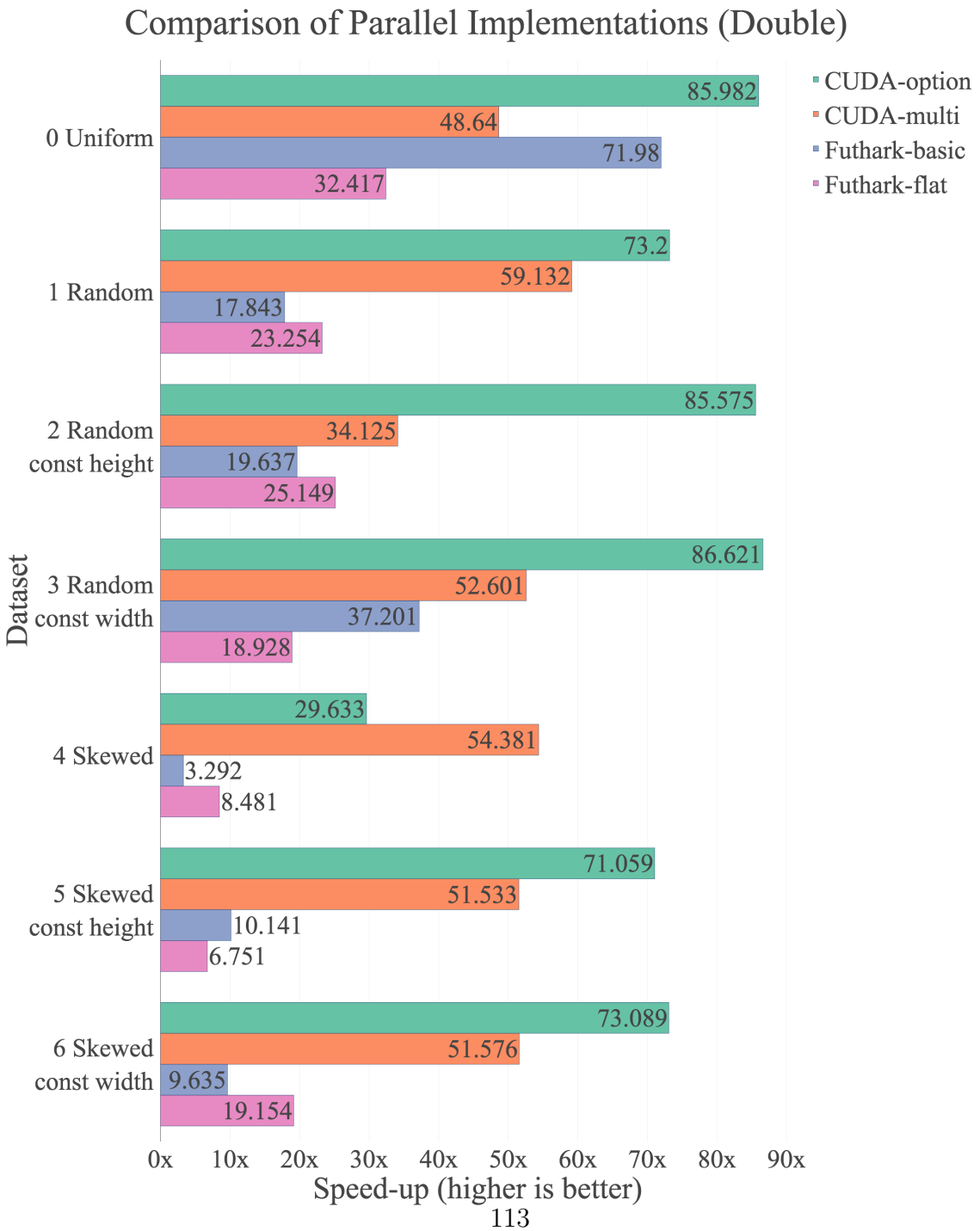


Figure 9.21: Comparisons of parallel speed-ups on smaller datasets over sequential implementation using single precision.

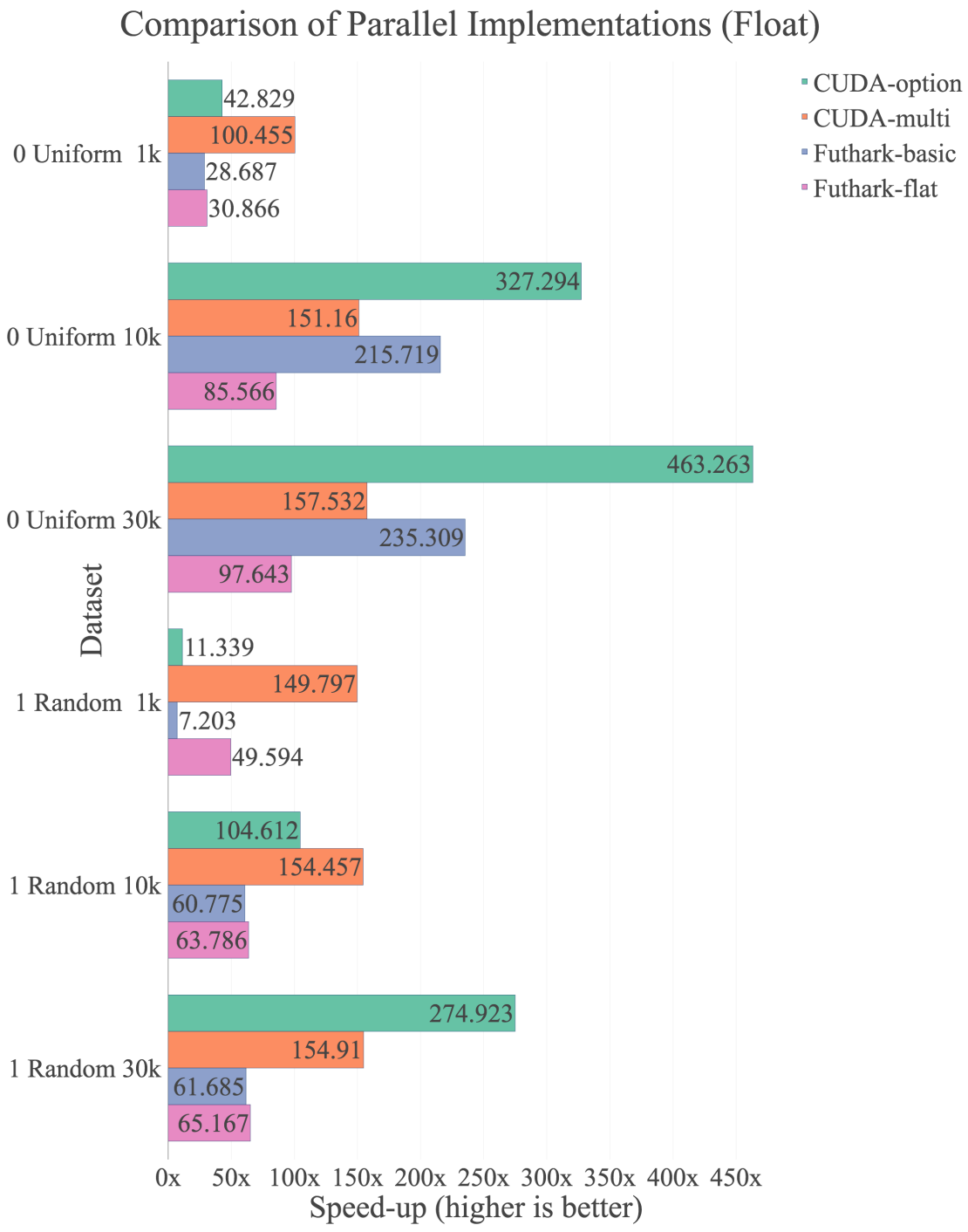
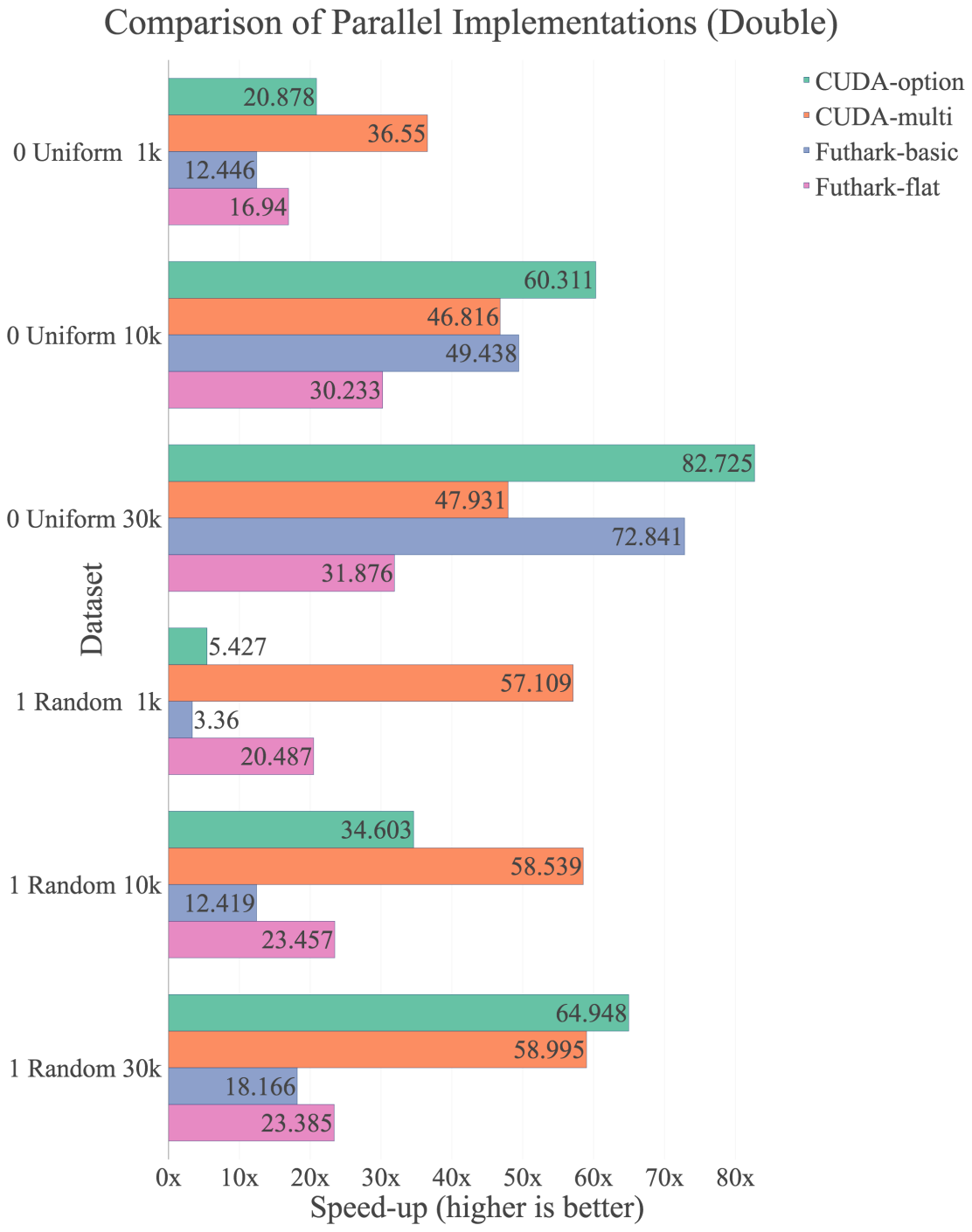


Figure 9.22: Comparisons of parallel speed-ups on smaller datasets over sequential implementation using double precision.



9.4 Key findings

The experiments have shown that:

- Memory coalescing has positive impact on the runtime performance of *CUDA-option*, at the cost of higher memory consumption. It is therefore important to optimize the global arrays and reduce their padding as much as possible. Coalescing on *CUDA-multi* on the other hand did not appear to be as effective but more granular padding has shown performance gains, most probably because of improved locality of reference (caching).

- Sorting has shown positive performance impact on both *CUDA-option* and *CUDA-multi* runtimes at neglectable runtime costs. An exception can be found when the data distribution is uniform, and consists of the same option replicated many times. However, this seems highly impractical in real life situations and can be easily detected. In terms of choosing the order, descending sorting by both width and height has shown the best results, but we have also shown that the correct choice is mostly dependent on the data.

- Choosing the correct block sizes has also shown to produce significant runtime impact on *CUDA-option*. However, this is also dependent on the data distribution, as skewed datasets tend to benefit more from larger block sizes, such as 256 and 512, while uniform and random datasets tend to have highest performance with block size of 128. For *CUDA-multi*, it is best to chose block size 1024 to pack in as many options as possible, while limiting the number of registers to 32 to achieve full occupancy of SMs.

- In terms of runtime, *CUDA-option* has been dominant on all datasets except 4 - Skewed, where *CUDA-multi* wins by almost a double margin. This discovery underlines the statement that parallel optimization is often sensitive to the dataset, hence it is important to explore the impacts of different degrees of parallelism. Furthermore, this allows the use of dynamic analysis in order to determine the optimal strategy for each dataset.

- There exists a certain threshold for the number of options in each data distribution, which can be used to dynamically choose *CUDA-multi* over *CUDA-option* if the dataset has small enough number of options.

Another interesting finding of less relevance to the thesis is that the sequential implementation is on average 15% faster with doubles than with floats (can be seen on the figures in Appendix C.4). In all our parallel implementations, float achieves 2 to 6× higher speed-ups than double. This difference can be explained with the fact that double precision (taking twice the memory) is much heavier on registers and cache which are more complex on the CPU cores than on the GPU's CUDA cores.

Summary

This chapter has shown the experiments we have performed to put our implementations to the test. It has introduced the impacts of the optimizations we have done, both in terms of runtime and memory consumption. Later in this chapter we have shown a comparison of all implementations, which has also led to the most important discovery - that *CUDA-multi* can show speed-up over *CUDA-option* on some datasets. Additionally, we describe all key findings of our experiments in the end of the chapter, with the idea of drawing meaningful conclusions later in the thesis. The next chapter will introduce related work on the topic, leading up to our conclusions, which will follow right after.

10 Related Work

In comparison with CPU-oriented architectures, modern many-core hardware offers much higher theoretical peak performance, but it also require significantly more effort and expertise in order to unlock this power. It follows that a rich body of work was aimed at porting and optimizing real-world applications for the many-core architectures. This chapter reviews several related research directions and is structured as follows: Section 10.1 reviews work aimed at accelerating various important algorithms by means of libraries, and concludes that an accelerated implementation of trinomial pricing is, to our knowledge, not publicly available. Section 10.2 surveys several data-parallel languages and argues that none of them can derive the two efficient code versions presented in this thesis. Finally, Sections 10.3 recounts several static, dynamic and hybrid analysis techniques, and points out the ones that have inspired our implementations.

10.1 Library Solutions

Library solutions have been aimed at providing efficient implementations for common-used algorithms. For example, Nvidia provides CUDNN for accelerating deep learning algorithms [27], and more recently, Mathworks and Nvidia are working together to accelerate Matlab libraries, resulting in the Matlab Parallel Computing Toolbox[1]. The toolbox provides high-level constructs, parallel for-loops, special array types and parallelized numerical algorithms.

[1]More information can be found on https://www.mathworks.com/products/parallel-computing/features.html

Furthermore, a more related body of work has been carried out on investigating the acceleration (in OpenCL) of risk modelling and derivative pricing using Monte-Carlo simulations [1, 29]. The findings of this studies have led to extending the Futhark language with new constructs and compiler analysis, which are useful and applicable in general (i.e., beyond the scope of the studied applications).

This thesis takes a similar approach, by providing a CUDA-library solution for accelerating trinomial pricing of options, which we are not aware of being publicly available before. Also we hope that this work will inspire new language/compiler technology, because, as the next section will show, none of the data-parallel languages can generate both efficient code versions that have been studied in this thesis.

10.2 Data-Parallel Language Design

The design of data-parallel languages is another research direction that has received a lot of attention in the past decade. The main idea here is (i) for the language to support a (smallish) set of operators, which have inherently parallel semantics (map, reduce, scan, etc.), and (ii) to build programs like puzzles by combining such parallel operators either horizontally (i.e., composition) or also in a nested fashion. This allows the language to statically provide not only safety guarantees—e.g., the absence of data races, but also cost-model guarantees—e.g., modeling the operational (asymptotic) complexity of the program. Unfortunately, as of today, the parallel-language technology is not at the stage where it can efficiently and reliably execute complex real-world applications, such as the one discussed in this thesis.

For example SAC [9, 10, 11] is a data-parallel language that uses an imperative-C notation aimed at achieving widespread use, but is purely functional underneath. It supports (i) shape polymorphism—i.e., one can write functions that operate uniformly on arrays of arbitrary rank, and (ii) a parallel construct named "with-loop", which can be seen as a sophisticated composition of map-reduce operators. Unfortunately, the optimizing compiler is mainly aimed at multi-cores, rather than many-cores, architectures.

Accelerate [23] is an array language, embedded in Haskell, aimed at GPGPU exe-

cution. It initially supported neither nested parallelism nor expression of sequential recurrences (loops) inside parallel code, but recent extensions [5] support a limited form of nested parallelism that is amenable to streaming. It follows that neither the one-option-per-thread nor the multiple-options-per-block code versions of trinomial pricing could be expressed in or derived by Accelerate. (We expect that the fully flat version can be written by hand in Accelerate, much like it has been done in Futhark.)

The seminal work on Nesl shows that it is possible to support the expression of arbitrary-nested and irregular parallelism, by a transformation named "flattening", which systematically rewrites the nested parallelism into only flat-parallel operators [3], which can be further mapped to GPGPU hardware [2]. More importantly, this transformation guarantees that the resulted (flat-parallel) program respects the work-depth (operational) asymptotic of the original (nested-parallel) program. Unfortunately, while this transformation is theoretically appealing, it is sometimes infeasible in practice because full utilization of application parallelism prevents locality-of-reference optimizations and may require asymptotically-larger memory footprint. It follows that Nesl would allow to elegantly express the trinomial pricing in its nested-parallel fashion, and it would automatically derive the fully-flat implementation. However, it would not be able to derive the two (more efficient) code versions, which are the subject of this thesis.

Finally, Futhark [16] is another data-parallel purely-functional language, whose design seeks a common-ground with imperative solutions. For example, it supports sequential loops with in-place modifications of array elements inside parallel regions, and an aggressive fusion mechanism [15]. It also supports a "regular" flavor of nested parallelism—i.e., the sizes of the array dimensions produced at any program point should be invariant to the parallel-nest in which the array is defined. In this context, the flattening transformation (i) is implemented by a combination of imperative transformations [17], such as map interchange and map fission (distribution), and (ii) it features the nice property that it can stop at an arbitrary level in the parallel nest and sequentialize from there all inner parallelism. In essence, the latter property allows further optimization of locality of reference. Finally, Futhark allows easy integration [14] of computational kernels written in it, with real-world applica-

tions written in mainstream-productivity languages such as Python and C. In what trinomial-option pricing is concerned, Futhark supports reasonably well the code version that executes one option per thread, and it also allows manual expression of the fully-flattened program. However, Futhark cannot derive the other efficient version that processes several options in one CUDA block. We believe that this work will provide useful insights into how to engineer a restricted flavor of flattening irregular parallelism, which is carried out in fast memory and is thus efficient.

10.3 Compiler Analysis

In this thesis we have studied how to accelerate trinomial pricing: we have identified several important performance trade-offs and have addressed each one by hand, by taking inspiration from related compiler analysis literature. This section surveys such analyses and discusses the ones that we have used.

10.3.1 Static Analysis

Static analysis refers to analysis performed at compile time, which by definition does not take into account the particularities of the input datasets. Such analysis carries no runtime overhead, but it may be inaccurate (i.e., yields conservative results). For example, analysis based on the polyhedral model [32, 4, 43] can reliably and aggressively optimize locality of reference and identify the parallelism of sequentially-written loops as long as the program is affine[2]. This is achieved by composing a sequence of non-trivial transformations such as loop interchange, loop distribution, various kinds of loop tiling [12]. However, if the program is not affine (e.g., due to subscripted subscripts) then analysis is likely to fail. Analysis failing does not necessarily mean that a certain loop is not parallel, it simply means that the static analysis was not able to prove some required property. Even worse, if that property refers to a subscripted subscript, and the indirect array is part of the dataset, then

[2]In affine code, all array subscripts and branch conditions are expressed as affine formulas in terms of the loop-nest indices

it follows that no static analysis can possibly prove that property (because its proof requires information about the elements of the indirect array, which is unknown until runtime).

Another example of static analysis is the flattening transformation [3], which allows for increasing the amount of parallelism that can be statically mapped to hardware and which has been mentioned in the previous section. It is worth noticing that flattening is more general than the combination of loop interchange and distribution supported by Futhark—for example, Blelloch's transformation applies to parallelism which is even nested inside divide-an-conquer function calls (e.g., it is possible to flatten quicksort starting from its clearest and simplest divide-and-conquer formulation). However, even if flattening guarantees to preserve the work-depth asymptotics of the original nested-parallel program, the analysis is still inaccurate because it fails to take into consideration communication costs or locality of reference. For example the resulted parallel (flat) operators will not contain any (sequential) recurrences/loops. While this can be good for the purpose of vectorization, it does not suit well many-core architectures and makes it impossible to optimize locality of reference.

Our implementations took inspiration from several static analyses. For example, in the one-option-per-thread implementation, we have performed loop fusion by hand and we have optimized spatial locality by working on the arrays in their transposed form [29] as a way to create array accesses that are coalesced in memory—i.e., consecutive threads access consecutive memory addresses. This memory access pattern is efficiently supported by GPGPU hardware: it leads to optimal utilization of global-memory bandwidth and thus it reduces the query time of data, resulting in significant speedups. Similarly, in the multiple-options-per-block implementation, we have taken inspiration from loop distribution (map fission) and more generally from the flattening transformation, to preform the flattening of the parallelism of several options that fit in a CUDA block.

10.3.2 Dynamic Analysis

Dynamic analysis refers to optimizations that aggressively rely on inspecting the data at runtime. The result of this inspection is used to reorganize the data layout or the scheduling of instructions (commonly entire loop iterations) in a way that optimizes the data locality or the degree of parallelism, or the code on the hot execution path. The runtime of the code that is performing the inspection and the reorganization constitutes pure overhead; it follows that the performed optimization need to be highly profitable in order to outset the inspection overhead.

An illustrative example of dynamic analysis is thread-level speculation [6, 34] (TLS). TLS speculatively executes loop iterations concurrently without any static guarantees that the loop is actually parallel. The inter-iterations dependencies are tracked and "fixed" at runtime. When such dependences are detected, the state is rolled back to a previously-known correct state, and speculation is resumed from there. This analysis is implemented by instrumenting each read and write access to memory with code that resembles an enhanced cache-coherency protocol—in fact TLS has been successfully implemented in hardware. On the plus side, TLS can be safely applied to automatically parallelize any application, it can exploit partial parallelism, and can be also used to optimize communication overhead in a distributed setting [28]. The shortcomings are that the overheads (both runtime and memory) are high, which makes it suboptimal in all cases and prohibitively expensive in many cases. This is because TLS has no "smarts", in that it does not use any static information related to the code it attempts to parallelize. While various optimizations have been proposed, such as aimed at optimizing the footprint or the data layout of speculative memory [30], TLS' overheads remain significant, which has caused TLS to loose ground in favor of techniques that combine static and dynamic analysis.

While in the case of TLS the inspection and execution of the data and code are intermingled, most of the other instances of dynamic analysis are separating the two stages: an inspector is extracted by slicing the code of interest, and it reorganizes the data layout or the schedules of instructions, in a way that improves the execution of the original code. For example, such analysis has been used to separate at runtime the

execution of (originally) dependent loops, into waves that can be safely executed in parallel [35]. Similarly, it has been used to optimize communication in a distributed setting [36]—here the inspctor solves a graph-partitioning problem at runtime in order to determine the optimal scheduling of loop iterations to nodes.

10.3.3 Hybrid Analysis (Static + Dynamic)

Hybrid analysis attempts to find a common ground between static and dynamic analysis that combines the advantages of each, i.e., accurate analysis carrying small runtime overhead. For example, seminal work has shown how to optimize locality of reference of highly irregular computations [7, 38], such as from the molecular-dynamics field, which make heavy use of statically-unanalyzable subscripted subscripts. This is achieved by an inspector that permutes both the array layout and the order in which loop iterations are scheduled. The technique however is enhanced to make use of static analysis: for example it fires the transformation only when the inspector overhead can be amortized (e.g., by hoisting it outside a loop), and it rewrites the code in a manner that minimizes the number of subscripted subscripts (which requires two memory operations and puts pressure on the cache system).

Other examples of hybrid analysis are related to aggressive automatic-parallelization technique aimed at statically-unanalyzable Fortran loops. There, memory accesses are summarized at instruction, iteration and loop level under a symbolic-set representation and loop parallelism is then modeled as an equation on these sets [13, 37]. These equations can then be solved at runtime, but the corresponding inspector has a sequential nature and it would lead to significant overhead in many cases. Instead, the equation is mapped to a language of predicates, by a logical inference analysis that extracts sufficient conditions under which the equation holds [24, 31]. The resulting predicates have a parallel semantics, and incur negligible runtime overhead in most cases, and, in the worst case, an overhead that scales down with the degree of parallelism.

Our implementation takes inspiration from such hybrid analyses. For example, we have used inspectors that sort the options in descending order of the height/width of

their trinomial tree, and we have shown that (i) this data re-ordering has a positively-high impact on performance, and that (ii) the overhead introduced by sorting is small in practice, because the sorting time is asymptotically smaller than the work necessary to price the options. Similarly, we have argued that a measure of the divergence overhead can be similarly derived by a lightweight inspector, and that a simple predicate can be autotuned to predict the most-suitable version of the code for the current dataset.

11
Conclusion

This thesis has presented multiple different parallelization strategies for the Hull-White Single-Factor Model with the use of the trinomial trees numerical method. The main purpose of these implementations is to improve the model performance, by pricing as many options in as little time as possible.

We have presented a sequential implementation, serving as a proof-of-concept and used it for validating the correctness of our parallel approaches. We have shown two parallel one-option per thread implementations — one in CUDA and another one in Futhark, exploiting only outer parallelism. We have demonstrated the steps applied to transform the one-option per thread version into a multiple options per thread block parallel version in CUDA, in an attempt to harness the possibilities of inner parallelism. We have also derived and presented a fully-flattened parallel approach written in Futhark, showing an important trade-off, that fully optimizing thread divergence comes with the price of degrading locality of reference.

We have created 7 distinct datasets, and ran numerous experiments to highlight the impact of our implementations and we have achieved as much as ~529× performance increase of *CUDA-option* over the sequential implementation. These experiments have also validated our claims that there is no single implementation which works best for all data distributions, by demonstrating that *CUDA-multi* can result in a speedup of ~2× on skewed datasets and up to ~13× on smaller datasets over *CUDA-option*. We have demonstrated that optimizations such as memory coalescing, sorting and block size choices can have a high positive impact on the performance of the parallel implementations (both in terms of runtime and memory consumption). Despite that, we have also established that each of these optimizations comes with

its trade-offs (e.g. coalescing improves the runtime, but also increases the memory consumption, smaller block sizes may result in block execution overhead, while larger block sizes may degrade thread divergence), which need to be considered. This has led up to the usefulness of dynamic analysis, which can be used to determine the optimal implementation based on the dataset.

All code and datasets used in this project can be found on our public GitHub repository:

https://github.com/MartinMetaksov/diku.OptionsPricing

Limitations and Future Work Despite answering the research questions, further optimizations could be done to decrease the runtimes of our implementations even more. The primary example of this is *CUDA-multi*, where option packing is not optimal and is done sequentially. As mentioned in chapter 6.1.2, this is a combinatorial NP-hard problem, which can be researched separately. There is a high possibility that an optimized bin packing solution may have some impact on the overall performance of *CUDA-multi*, making it a potential candidate for future improvements.

Furthermore, we could derive an inspector/executor technique that chooses between the two of our fast parallel implementations — *CUDA-option* and *CUDA-multi* based on the size, skewness and option widths of an input dataset.

Finally, another downside was the lack of optimizations on *Futhark-flat*. As mentioned in chapter 7, height-size arrays are padded on global-level, potentially degrading the performance. While we still do not expect it to outperform *CUDA-option*, we believe that fully optimizing this version can potentially lead to new discoveries.

Bibliography

[1] Christian Andreetta, Vivien Bégot, Jost Berthold, Martin Elsman, Fritz Henglein, Troels Henriksen, Maj-Britt Nordfang, and Cosmin E. Oancea. Finpar: A parallel financial benchmark. *ACM Trans. Archit. Code Optim.*, 13(2):18:1–18:27, June 2016.

[2] Lars Bergstrom and John Reppy. Nested data-parallelism on the GPU. *SIGPLAN Not.*, 47(9):247–258, September 2012.

[3] Guy E Blelloch, Jonathan C Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. Implementation of a Portable Nested Data-Parallel Language. *Journal of parallel and distributed computing*, 21(1):4–14, 1994.

[4] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[5] Robert Clifton-Everest, Trevor L. McDonell, Manuel M. T. Chakravarty, and Gabriele Keller. Streaming irregular arrays. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, Haskell 2017, pages 174–185, New York, NY, USA, 2017. ACM.

[6] Francis Dang, Hao Yu, and Lawrence Rauchwerger. The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops. In *Int. Par. and Distr. Processing Symp. (PDPS)*, pages 20–29, 2002.

[7] Chen Ding and Ken Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 229–241, New York, NY, USA, 1999. ACM.

[8] Myron Scholes Fischer Black. The pricing of options and corporate liabilities. 1973.

[9] Clemens Grelck. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming (JFP)*, 15(3):353–401, 2005.

[10] Clemens Grelck and Sven-Bodo Scholz. SAC: A functional array language for efficient multithreaded execution. *Int. Journal of Parallel Programming*, 34(4):383–427, 2006.

[11] Clemens Grelck and Fangyong Tang. Towards Hybrid Array Types in SAC. In V. Zsók, Z. Horváth, and R. Plasmeijer, editors, *7th Workshop on Prg. Lang., (Soft. Eng. Conf.)*, pages 129–145, 2014.

[12] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. Hybrid hexagonal/classical tiling for gpus. In *Procs. Int. Symposium on Code Generation and Optimization*, CGO '14, pages 66:66–66:75. ACM, 2014.

[13] Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. Interprocedural Parallelization Analysis in SUIF. *Trans. on Prog. Lang. and Sys. (TOPLAS)*, 27(4):662–731, 2005.

[14] Troels Henriksen, Martin Dybdal, Henrik Urms, Anna Sofie Kiehn, Daniel Gavin, Hjalte Abelskov, Martin Elsman, and Cosmin Oancea. APL on GPUs: A TAIL from the Past, Scribbled in Futhark. In *Procs. of the 5th Int. Workshop on Functional High-Performance Computing*, FHPC'16, pages 38–43, New York, NY, USA, 2016. ACM.

[15] Troels Henriksen, Ken Friis Larsen, and Cosmin E. Oancea. Design and GPGPU performance of futhark's redomap construct. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2016, pages 17–24, New York, NY, USA, 2016. ACM.

[16] Troels Henriksen and Cosmin E. Oancea. Bounds checking: An instance of hybrid analysis. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, pages 88:88–88:94, New York, NY, USA, 2014. ACM.

[17] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 556–571, New York, NY, USA, 2017. ACM.

[18] John C. Hull. *Options, Futures, And Other Derivatives*, volume 8. Pearson College Div, 2011.

[19] Kazuaki Ishizaki, Akihiro Hayashi, Gita Koblents, and Vivek Sarkar. Compiling and optimizing java 8 programs for gpu execution. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, pages 419–431, Washington, DC, USA, 2015. IEEE Computer Society.

[20] Alan White John Hull. Numerical procedures for implementing term structure models I: Single-factor models. *The Journal of Derivatives*, 1994.

[21] Alan White John Hull. Using Hull-White interest-rate trees. *The Journal of Derivatives*, 1996.

[22] Selcuk Keskin, Ömer Çetin, and Taşkın Koçak. Real-time fft computation using gpgpu for ofdm-based systems. 35, 06 2015.

[23] Trevor L. McDonell, Manuel M.T. Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising purely functional gpu programs. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 49–60, New York, NY, USA, 2013. ACM.

[24] Sungdo Moon and Mary W. Hall. Evaluation of Predicated Array Data-Flow Analysis for Automatic Parallelization. In *Int. Symp. Princ. and Practice of Par. Prog. (PPoPP)*, pages 84–95, 1999.

[25] Gordon E. Moore. Cramming more components onto integrated circuits. 1965.

[26] G.R. Mudalige, M.B. Giles, J. Thiyagalingam, I.Z. Reguly, C. Bertolli, P.H.J. Kelly, and A.E. Trefethen. Design and initial performance of a high-level unstructured mesh framework on heterogeneous parallel systems. *Parallel Comput.*, 39(11):669–692, November 2013.

[27] NVIDIA. CUDNN library user guide du-06702-001 v07. Technical report, NVIDIA Corporation, 2017.

[28] C. E. Oancea, J. W. A. Selby, M. Giesbrecht, and S. M. Watt. Distributed Models of Thread-Level Speculation. In *Proceedings of the PDPTA'05*, pages 920–927, 2005.

[29] Cosmin E. Oancea, Christian Andreetta, Jost Berthold, Alain Frisch, and Fritz Henglein. Financial Software on GPUs: Between Haskell and Fortran. In *Proceedings of the 1st ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '12, pages 61–72, New York, NY, USA, 2012. ACM.

[30] Cosmin E. Oancea and Alan Mycroft. Set-congruence dynamic analysis for thread-level speculation (TLS). In José Nelson Amaral, editor, *Languages and Compilers for Parallel Computing*, pages 156–171, Berlin, Heidelberg, 2008. Springer-Verlag.

[31] Cosmin E. Oancea and Lawrence Rauchwerger. A Hybrid Approach to Proving Memory Reference Monotonicity. In *Int. Languages and Compilers for Parallel Computing (LCPC'11)*, volume 7146 of *LNCS*, pages 61–75, 2013.

[32] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. Loop transformations: Convexity, pruning and optimization. In *Proceedings of the 38th Annual*

ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '11, pages 549–562, New York, NY, USA, 2011. ACM.

[33] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[34] L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *IEEE Trans. Parallel Distrib. System*, 10(2):160–199, 1999.

[35] Lawrence Rauchwerger, Nancy Amato, and David Padua. A Scalable Method for Run Time Loop Parallelization. *Int. Journal of Par. Prog*, 26:26–6, 1995.

[36] Mahesh Ravishankar, John Eisenlohr, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic parallelization of a class of irregular loops for distributed memory systems. *ACM Trans. Parallel Comput.*, 1(1):7:1–7:37, October 2014.

[37] Silvius Rus, Jay Hoeflinger, and Lawrence Rauchwerger. Hybrid Analysis: Static & Dynamic Memory Reference Analysis. *Int. Journal of Par. Prog*, 31(3):251–283, 2003.

[38] Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. Compile-time composition of run-time data and iteration reorderings. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 91–102, New York, NY, USA, 2003. ACM.

[39] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embed. Comput. Syst.*, 13(4s):134:1–134:25, April 2014.

[40] Joel Svensson. *Obsidian: GPU Kernel Programming in Haskell*. PhD thesis, Chalmers University of Technology, 2011.

[41] Yuan Tang, Rezaul Alam Chowdhury, Bradley C Kuszmaul, Chi-Keung Luk, and Charles E Leiserson. The pochoir stencil compiler. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 117–128. ACM, 2011.

[42] Ehsan Totoni, Todd A. Anderson, and Tatiana Shpeisman. HPAT: High Performance Analytics with Scripting Ease-of-use. In *Proceedings of the International Conference on Supercomputing*, ICS '17, pages 9:1–9:10, New York, NY, USA, 2017. ACM.

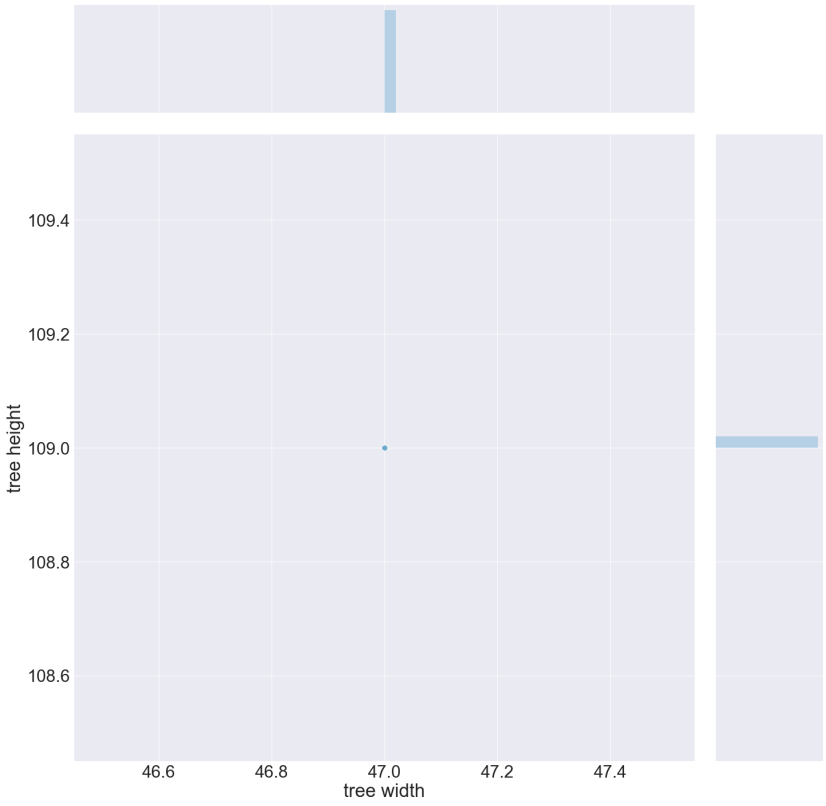
[43] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, January 2013.

[44] N. Whitehead and A. Fit-Florea. Precision and Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs. Technical report, NVIDIA Corporation, 2018.

Appendices

Appendix A. Generated Data

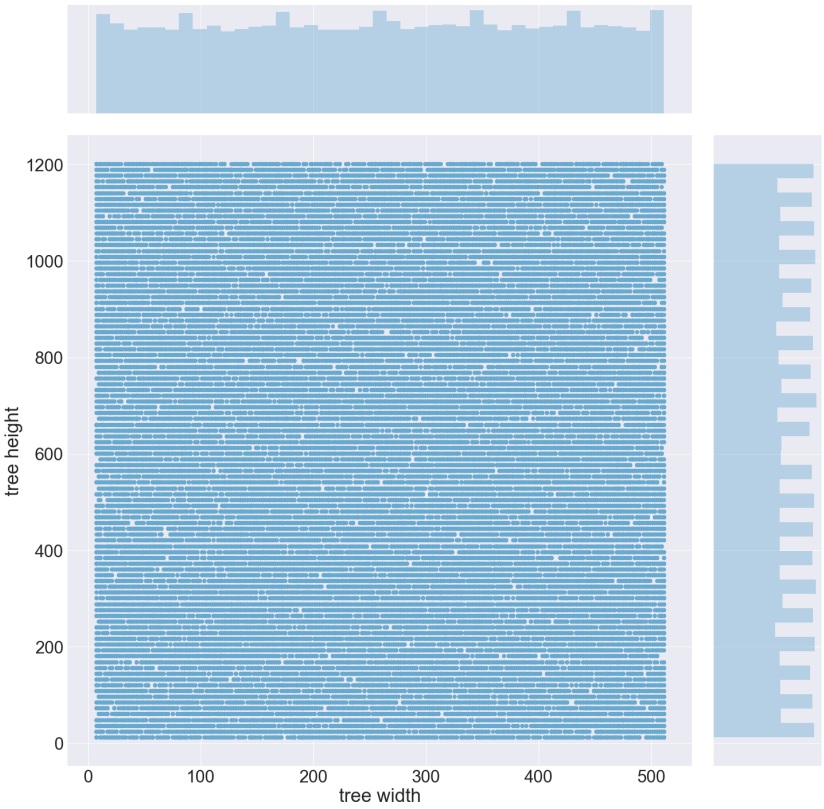
A.1 Uniform



Widths:
min: 47
max: 47
mean: 47.00
variance: 0.00
std: 0.00
skewness: 0.00
kurtosis: −3.00

Heights:
min: 109
max: 109
mean: 109.00
variance: 0.00
std: 0.00
skewness: 0.00
kurtosis: −3.00

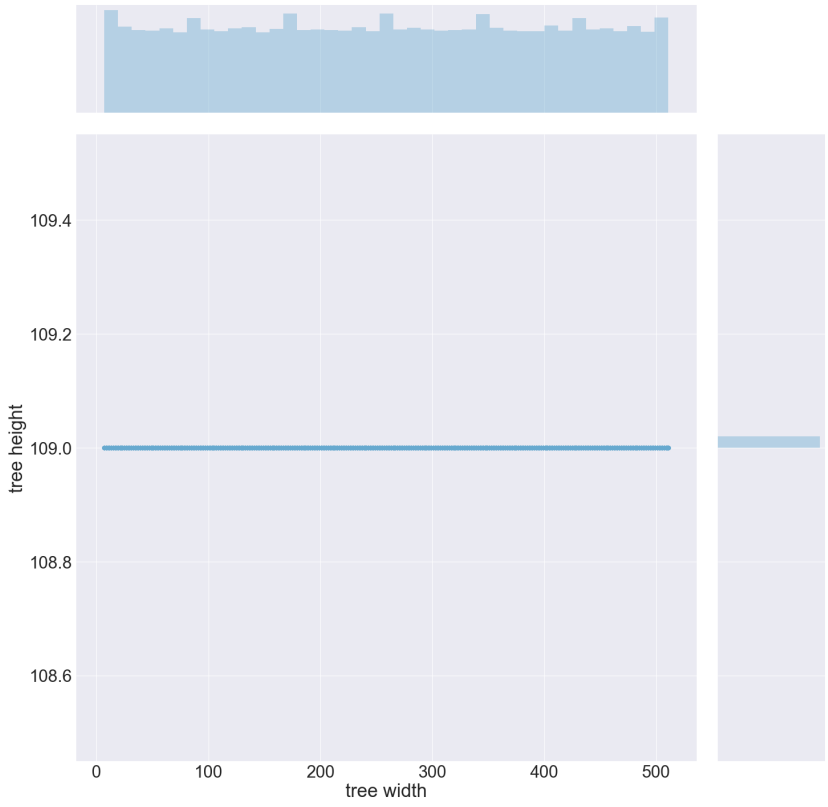
A.2 Random



Widths:
min: 7
max: 511
mean: 259.78
variance:
21293.52
std: 145.92
skewness: −0.01
kurtosis: −1.20

Heights:
min: 13
max: 1201
mean: 606.26
variance:
120065.68
std: 346.50
skewness: 0.00
kurtosis: −1.20

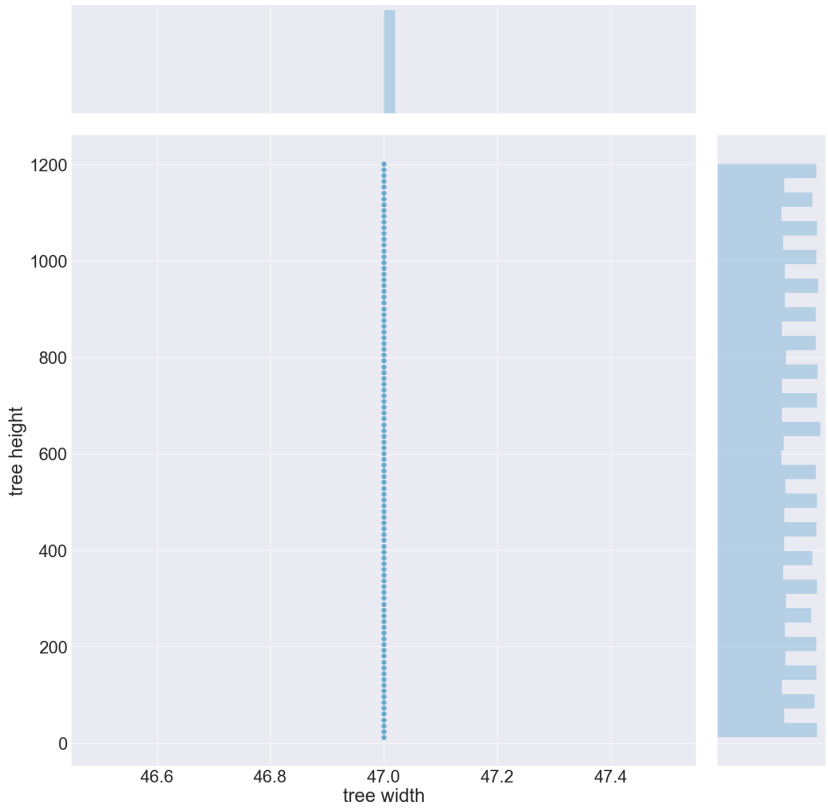
A.3 Random with Constant Height



Widths:
min: 7
max: 511
mean: 258.52
variance: 21358.88
std: 146.15
skewness: −0.00
kurtosis: −1.20

Heights:
min: 109
max: 109
mean: 109.00
variance: 0.00
std: 0.00
skewness: 0.00
kurtosis: −3.00

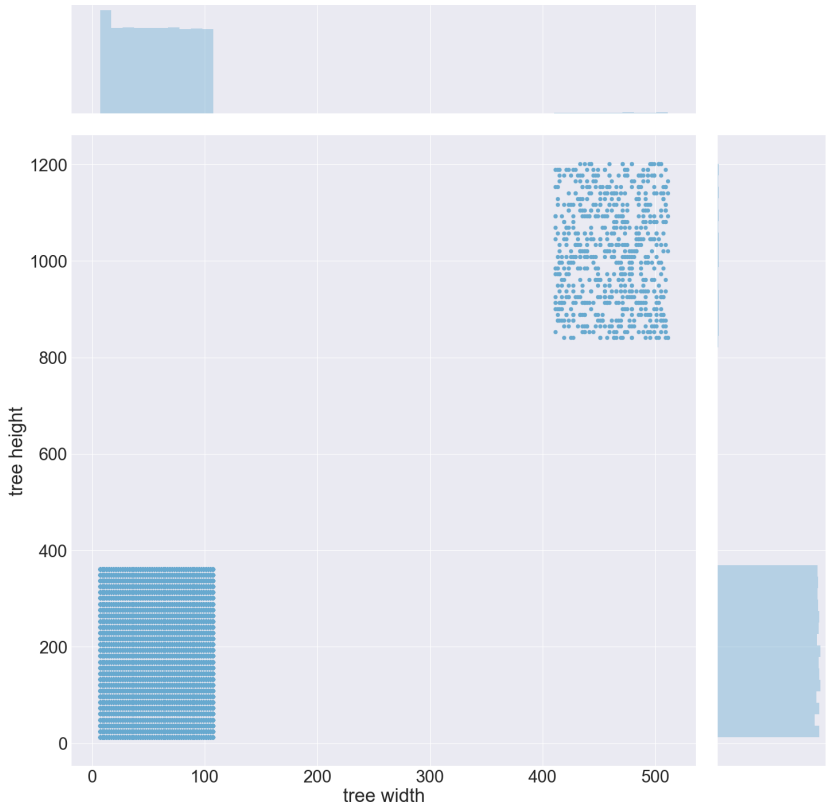
A.4 Random with Constant Width



Widths:
min: 47
max: 47
mean: 47.00
variance: 0.00
std: 0.00
skewness: 0.00
kurtosis: −3.00

Heights:
min: 13
max: 1201
mean: 607.04
variance: 119714.09
std: 346.00
skewness: −0.00
kurtosis: −1.20

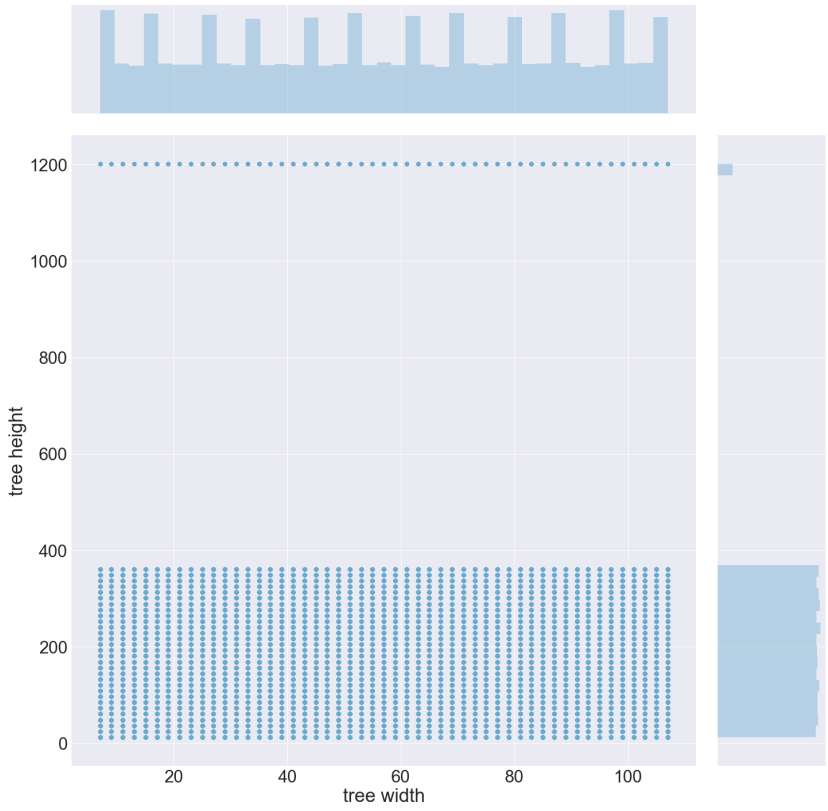
A.5 Skewed



Widths:
min: 7
max: 511
mean: 60.90
variance: 2490.82
std: 49.91
skewness: 5.19
kurtosis: 40.29

Heights:
min: 13
max: 1201
mean: 195.35
variance: 17598.95
std: 132.66
skewness: 2.40
kurtosis: 14.07

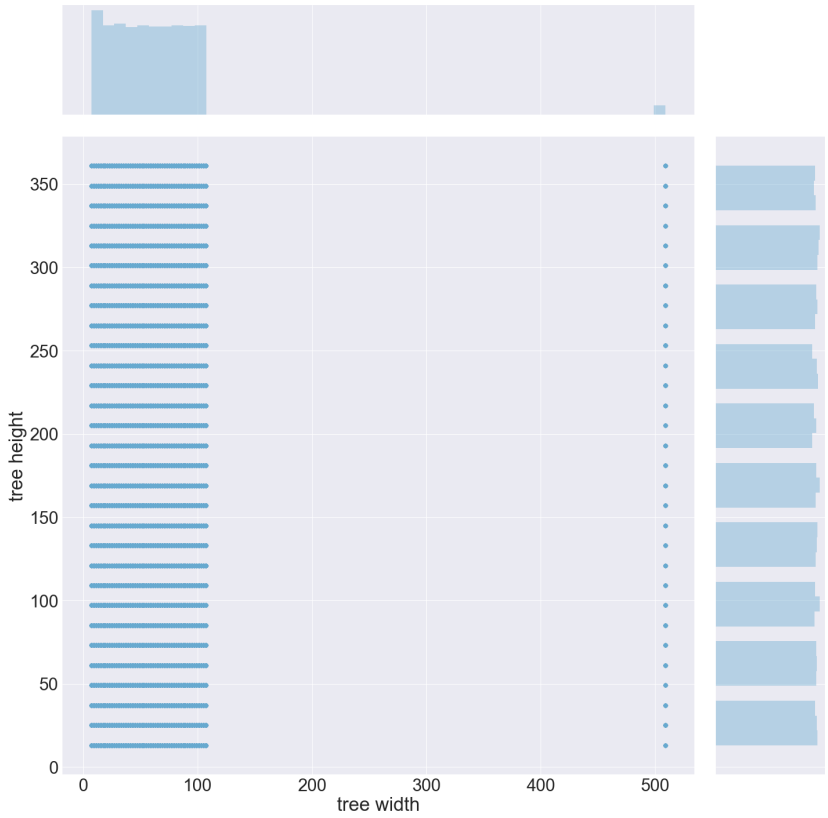
A.6 Skewed with Constant Height



Widths:
min: 7
max: 511
mean: 61.06
variance: 2493.88
std: 49.94
skewness: 5.15
kurtosis: 39.87

Heights:
min: 13
max: 1201
mean: 602.58
variance: 121471.29
std: 348.53
skewness: 0.01
kurtosis: -1.22

A.7 Skewed with Constant Width

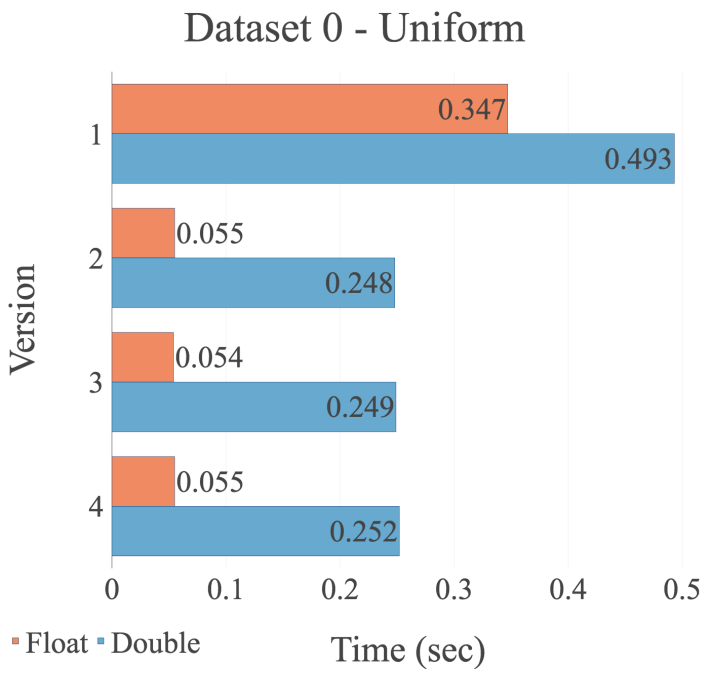


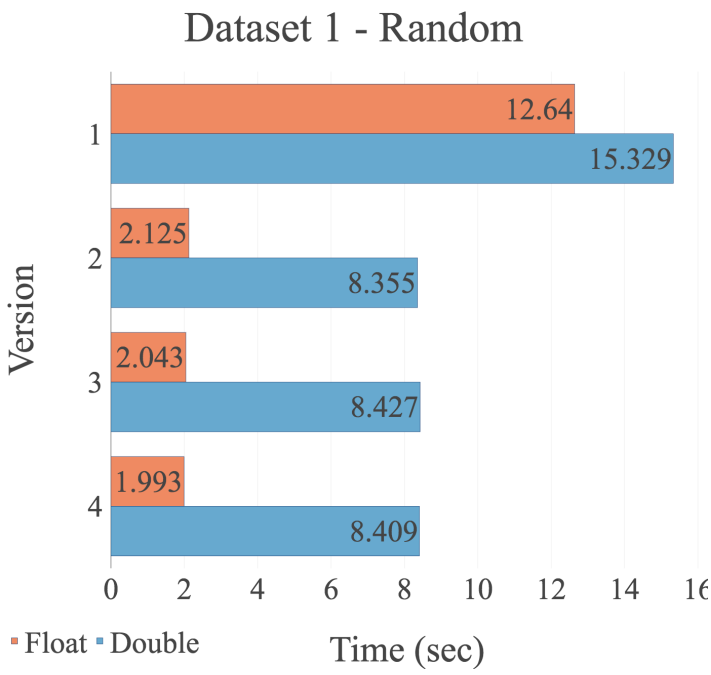
Widths:
min: 7
max: 507
mean: 254.24
variance: 21164.27
std: 145.48
skewness: 0.02
kurtosis: -1.21

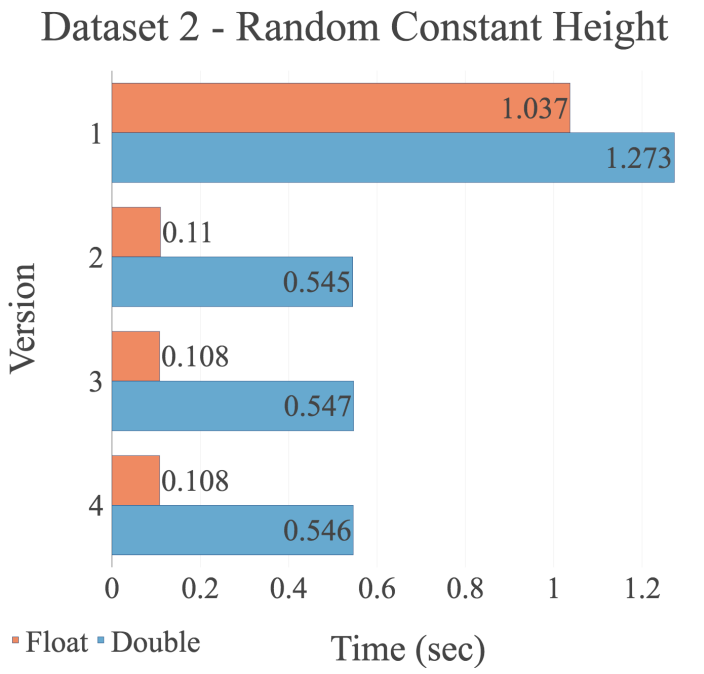
Heights:
min: 13
max: 1201
mean: 194.23
variance: 17615.88
std: 132.72
skewness: 2.41
kurtosis: 14.05

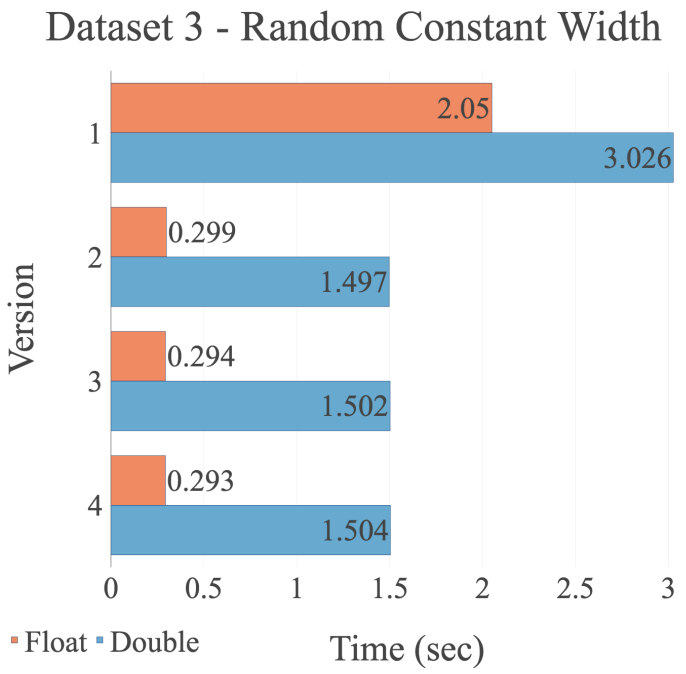
Appendix B. CUDA-option Experiments

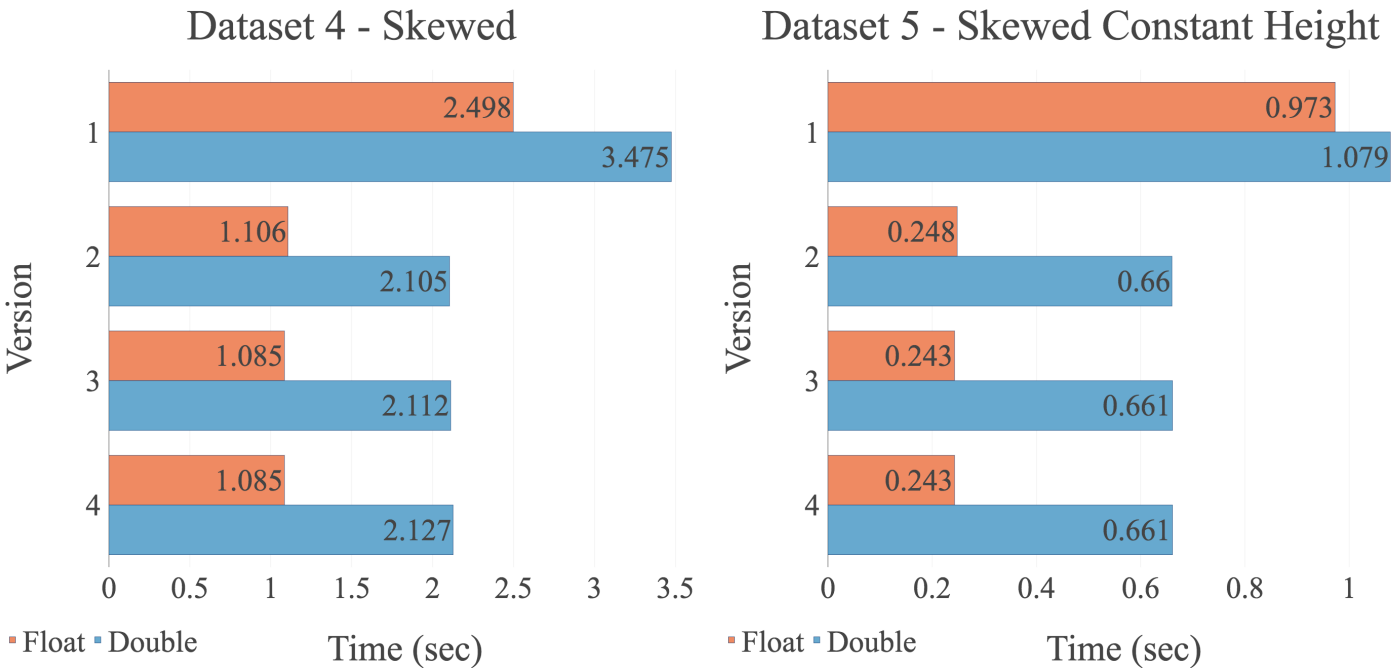
B.1 Version Experiments (Best Runtimes)

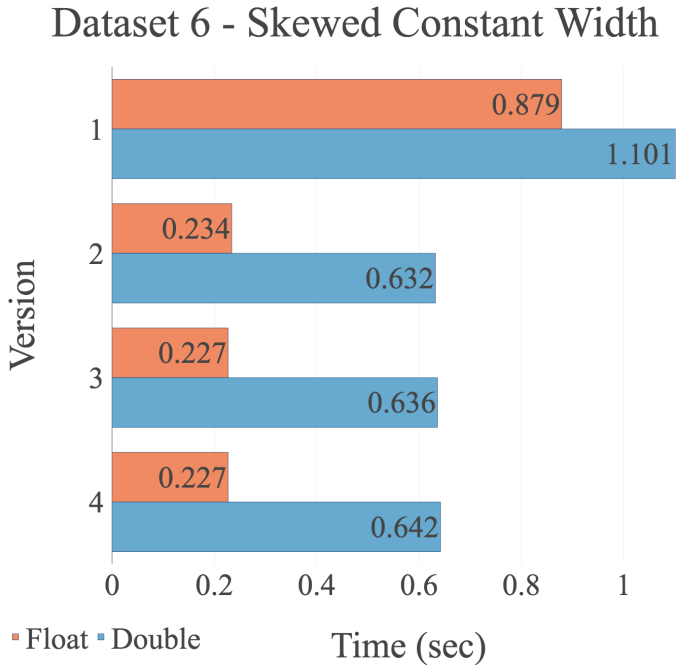




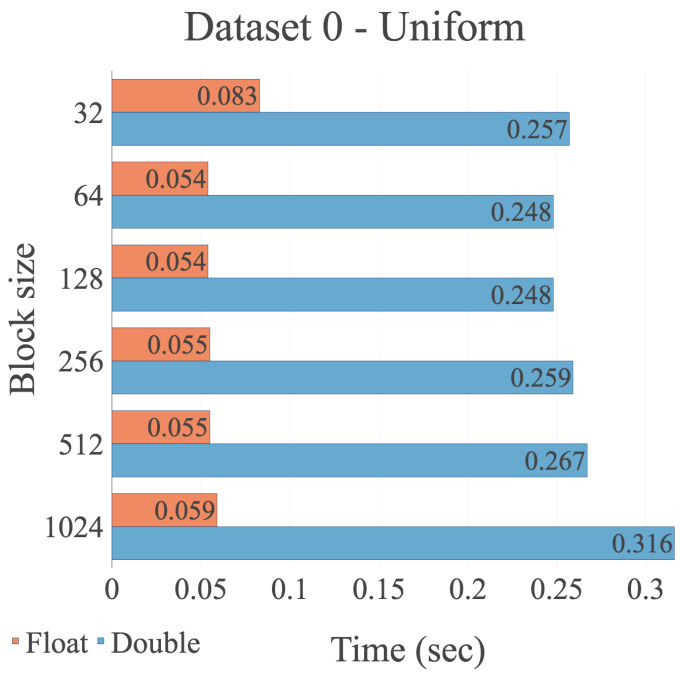


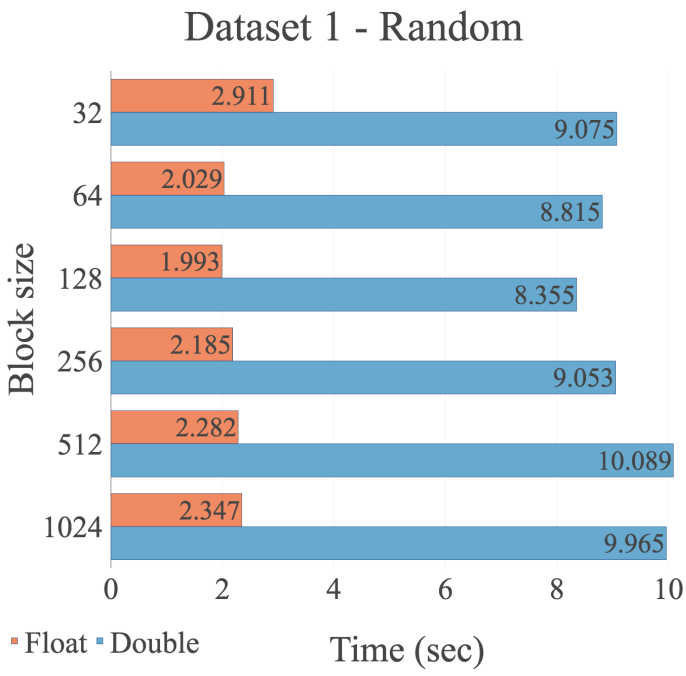


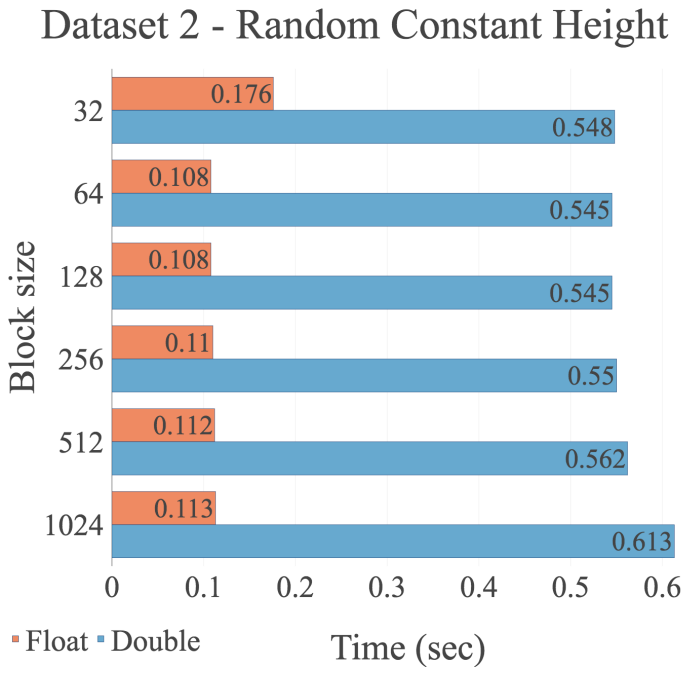

Dataset 4 - Skewed
Version
1
2
3
4
2.498
3.475
1.106
2.105
1.085
2.112
1.085
2.127
0
0.5
1
1.5
2
2.5
3
3.5
Float
Double
Time (sec)
Dataset 5 - Skewed Constant Height
Version
1
2
3
4
0.973
1.079
0.248
0.66
0.243
0.661
0.243
0.661
0
0.2
0.4
0.6
0.8
1
Float
Double
Time (sec)

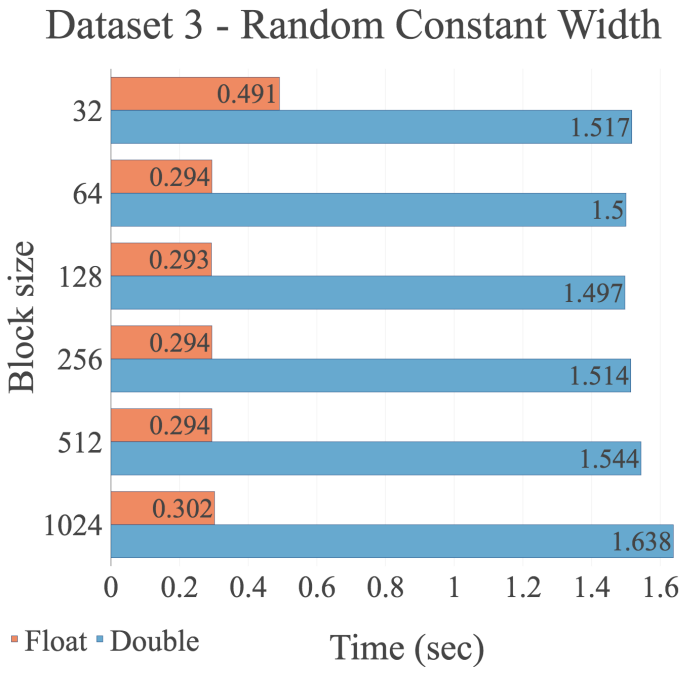

Dataset 6 - Skewed Constant Width
Version
1
2
3
4
0.879
1.101
0.234
0.632
0.227
0.636
0.227
0.642
0
0.2
0.4
0.6
0.8
1
Float
Double
Time (sec)

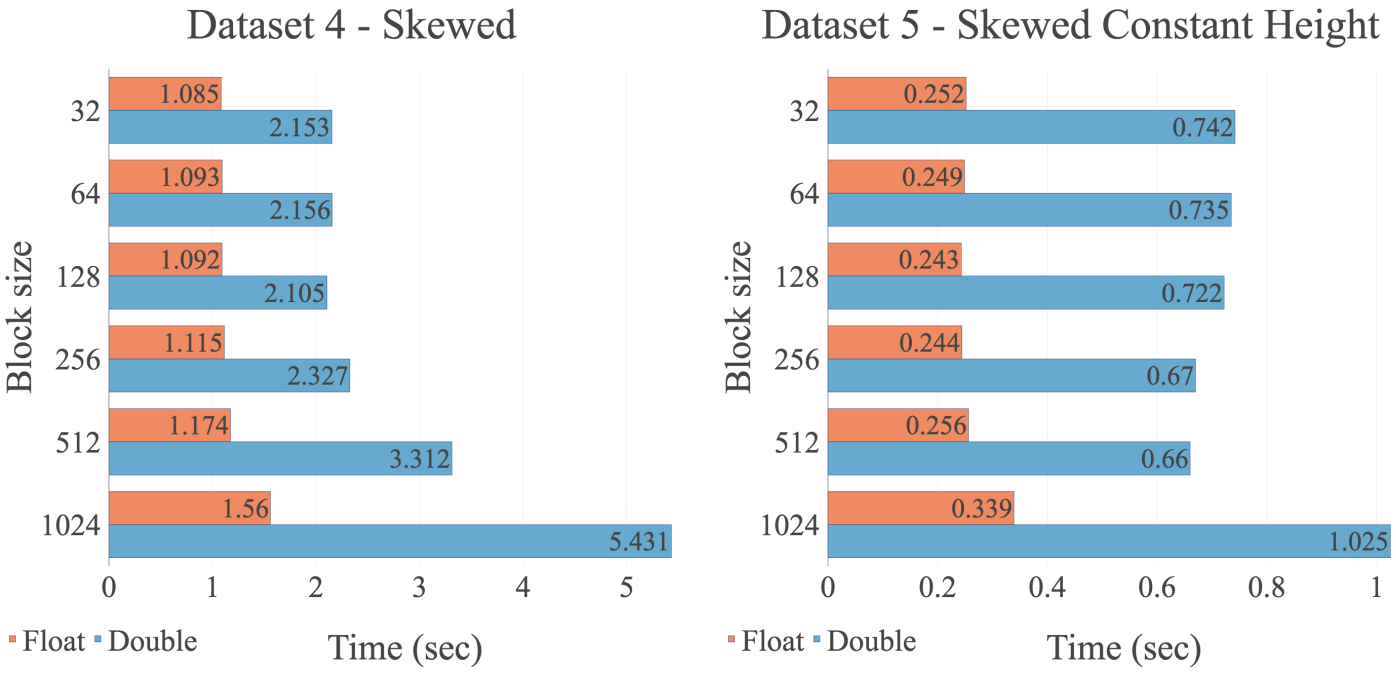
B.2 Block Size Experiments (Best Runtimes)

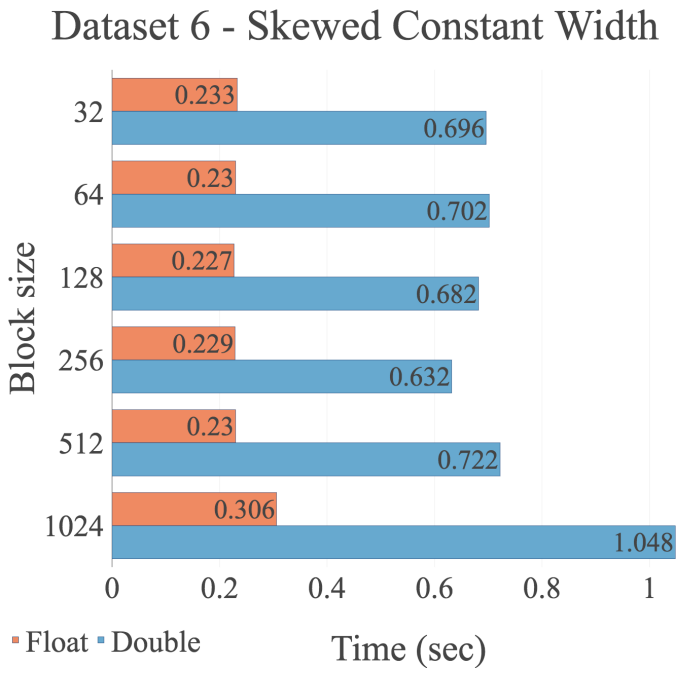




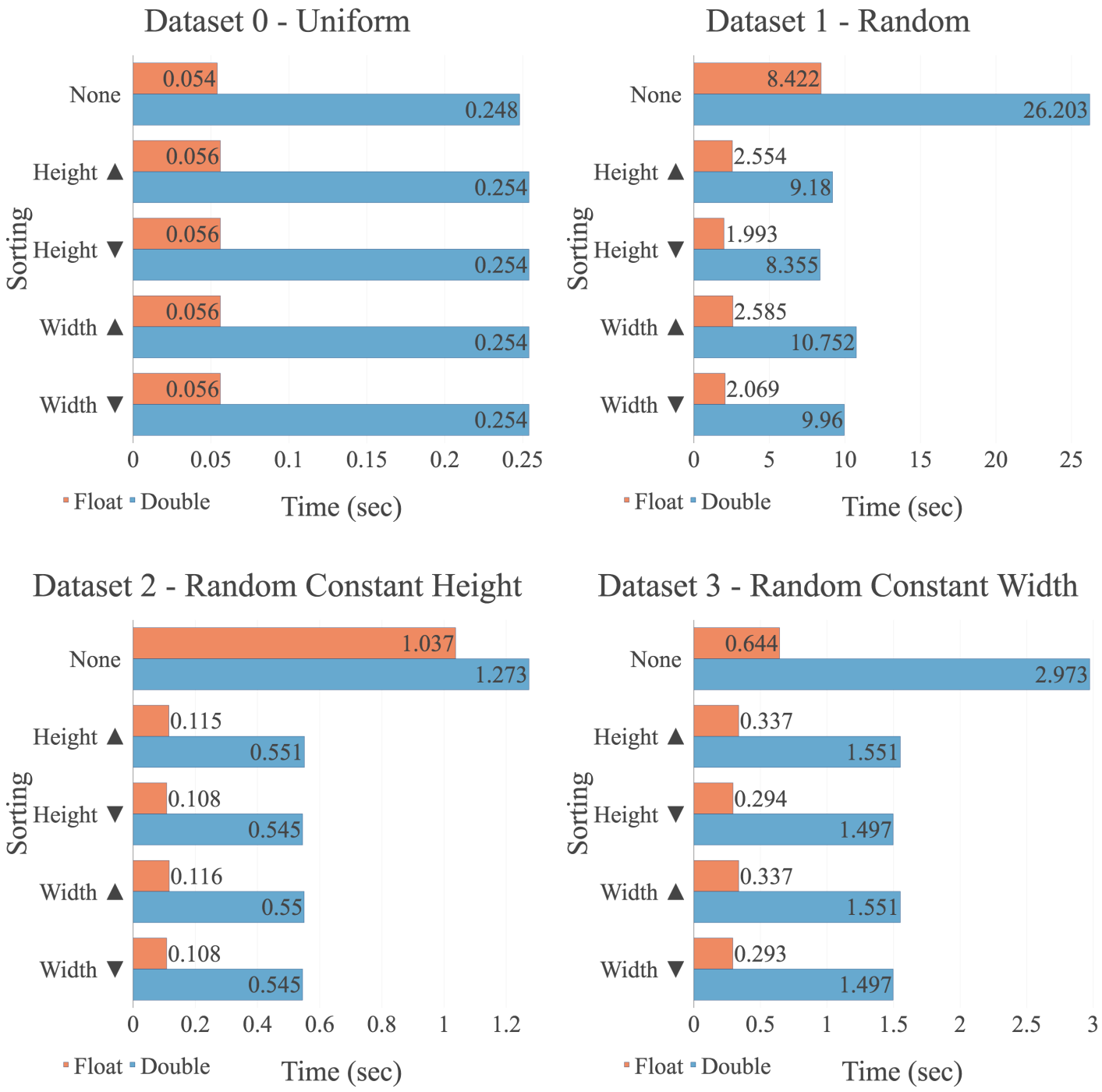


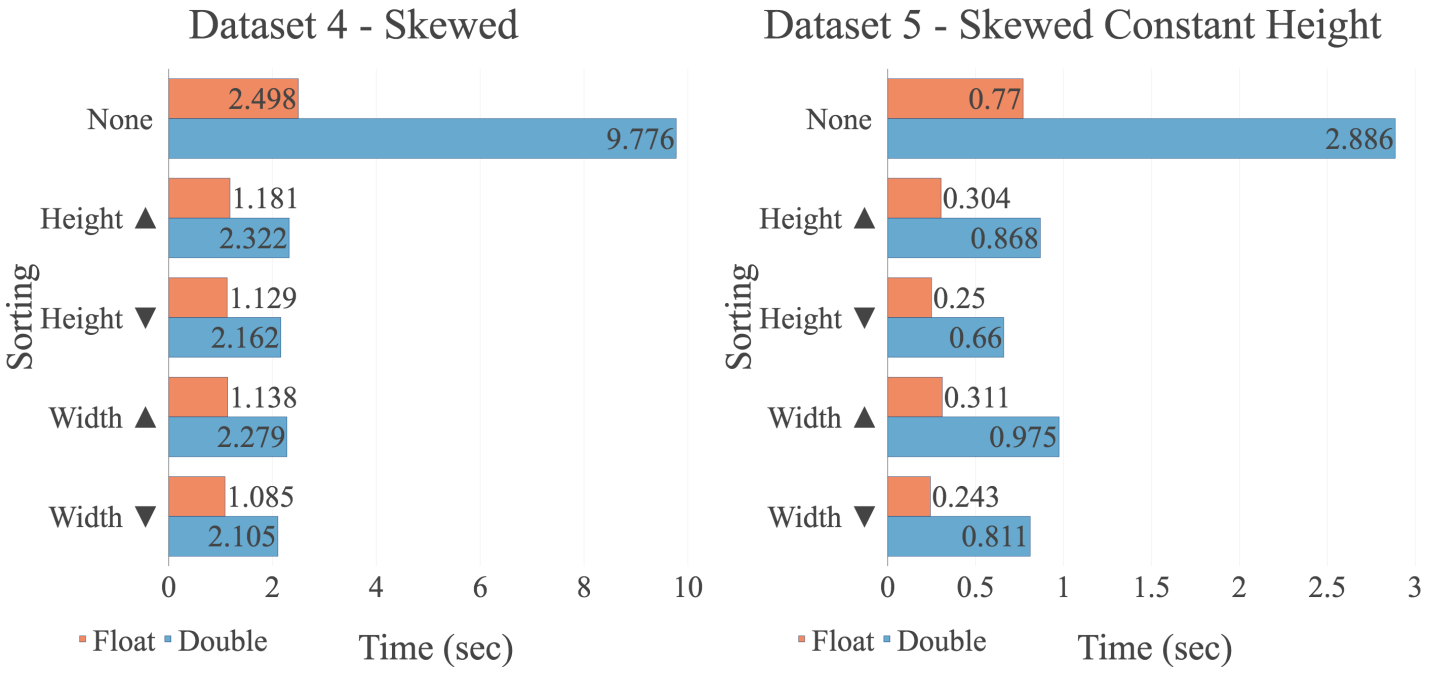


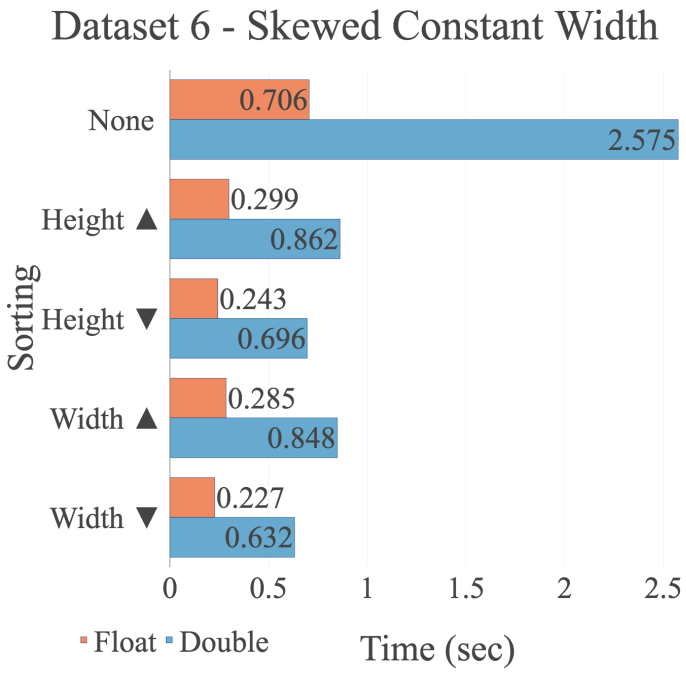

Dataset 4 - Skewed
Block size
32
64
128
256
512
1024
1.085
2.153
1.093
2.156
1.092
2.105
1.115
2.327
1.174
3.312
1.56
5.431
0
1
2
3
4
5
Float
Double
Time (sec)
Dataset 5 - Skewed Constant Height
Block size
32
64
128
256
512
1024
0.252
0.742
0.249
0.735
0.243
0.722
0.244
0.67
0.256
0.66
0.339
1.025
0
0.2
0.4
0.6
0.8
1
Float
Double
Time (sec)


Dataset 6 - Skewed Constant Width
Block size
32
64
128
256
512
1024
0.233
0.696
0.23
0.702
0.227
0.682
0.229
0.632
0.23
0.722
0.306
1.048
0
0.2
0.4
0.6
0.8
1
Float
Double
Time (sec)

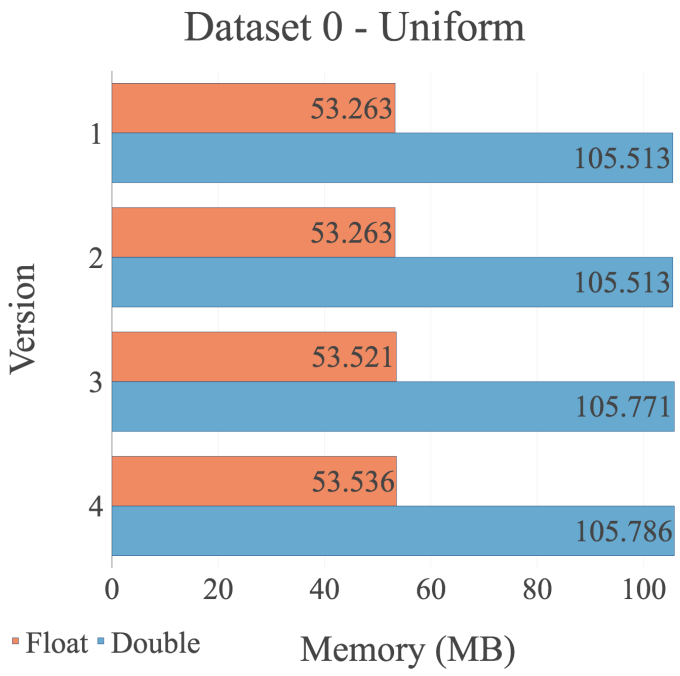
B.3 Sorting Experiments (Best Runtimes)

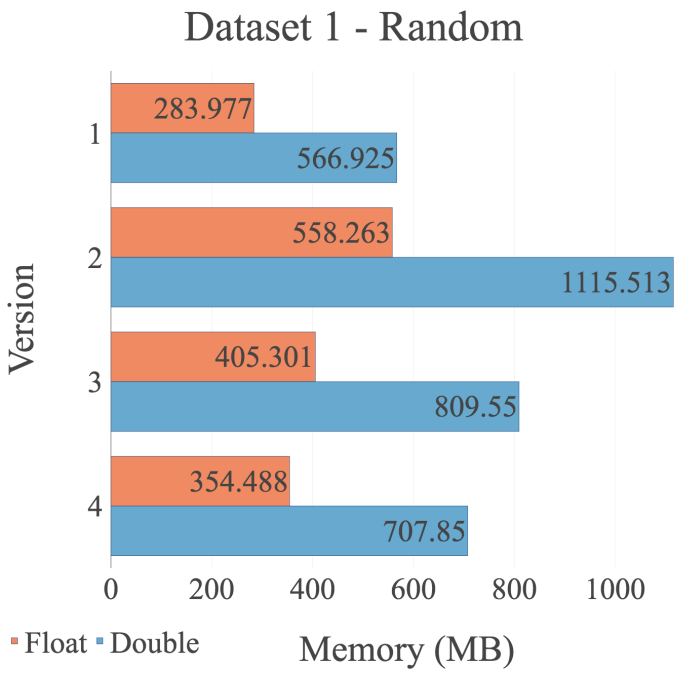


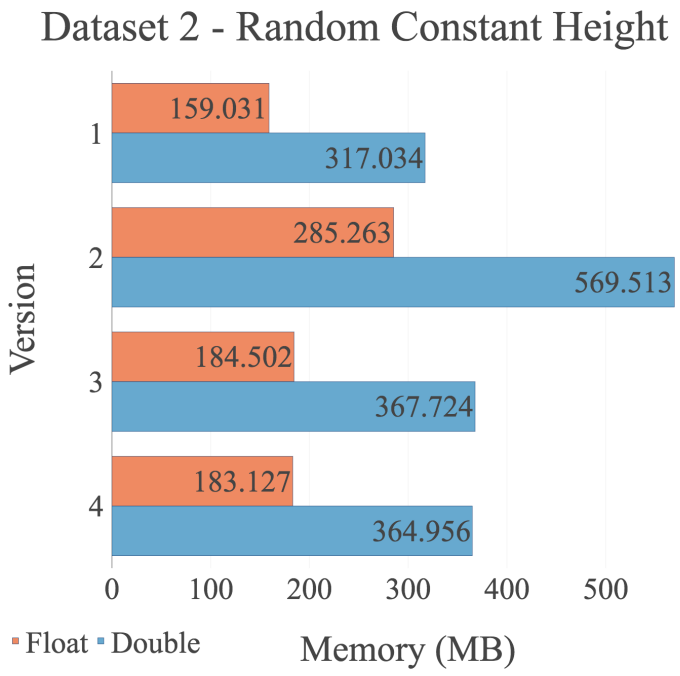

Dataset 4 - Skewed
None
Height ▲
Height ▼
Width ▲
Width ▼
Sorting
2.498
9.776
1.181
2.322
1.129
2.162
1.138
2.279
1.085
2.105
0
2
4
6
8
10
Float
Double
Time (sec)
Dataset 5 - Skewed Constant Height
None
Height ▲
Height ▼
Width ▲
Width ▼
Sorting
0.77
2.886
0.304
0.868
0.25
0.66
0.311
0.975
0.243
0.811
0
0.5
1
1.5
2
2.5
3
Float
Double
Time (sec)

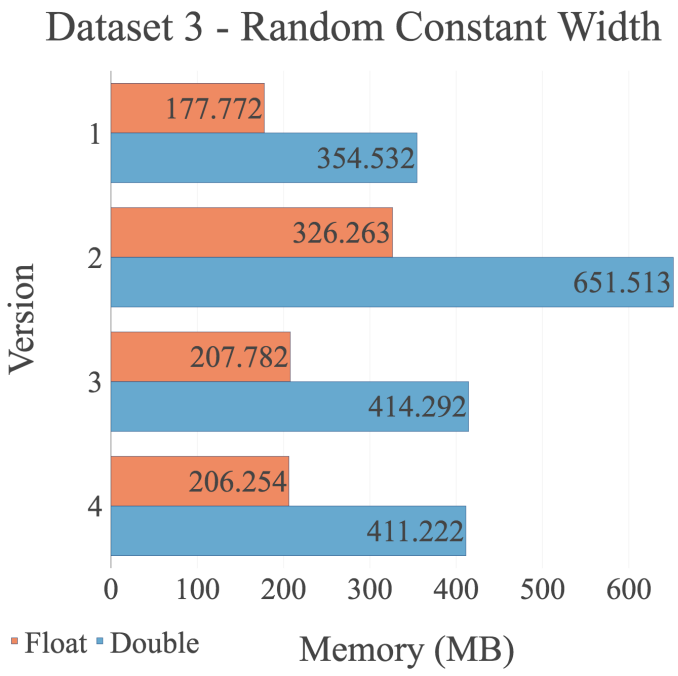

Dataset 6 - Skewed Constant Width
None
Height ▲
Height ▼
Width ▲
Width ▼
Sorting
0.706
2.575
0.299
0.862
0.243
0.696
0.285
0.848
0.227
0.632
0
0.5
1
1.5
2
2.5
Float
Double
Time (sec)

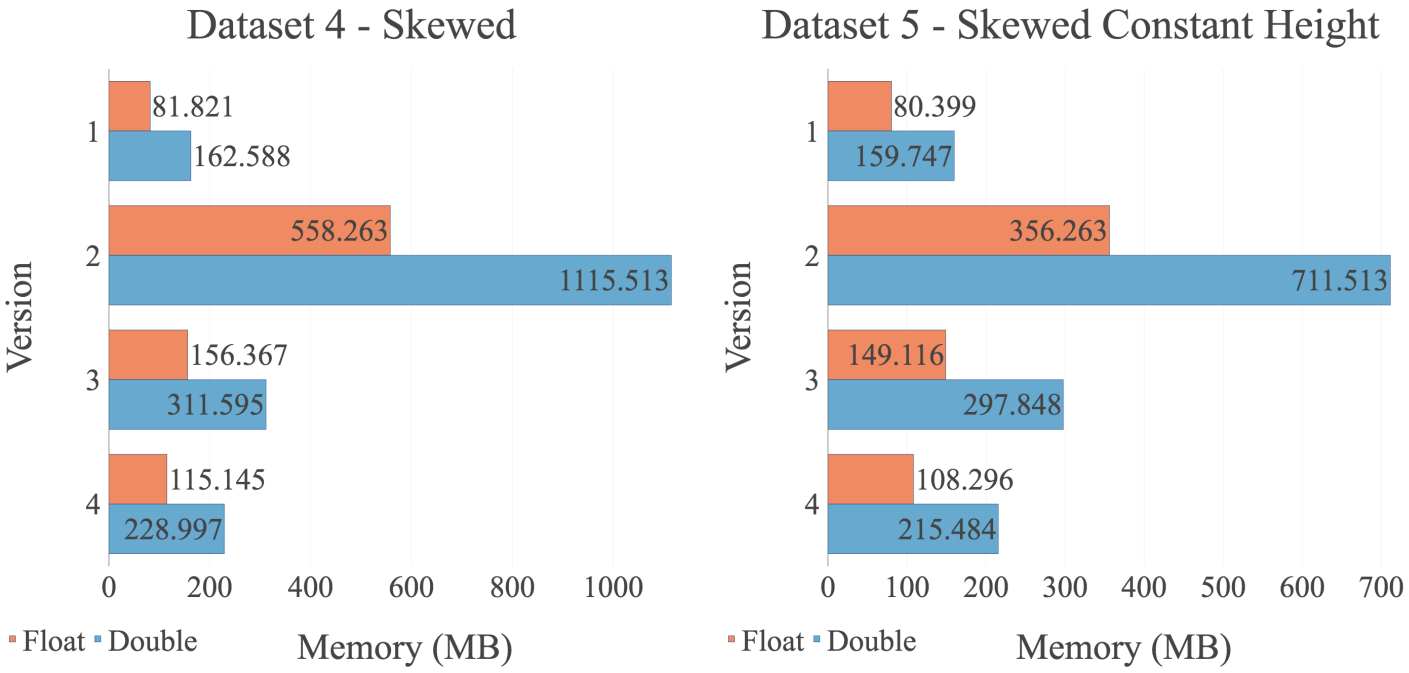
B.4 Version Experiments (Average Memory)

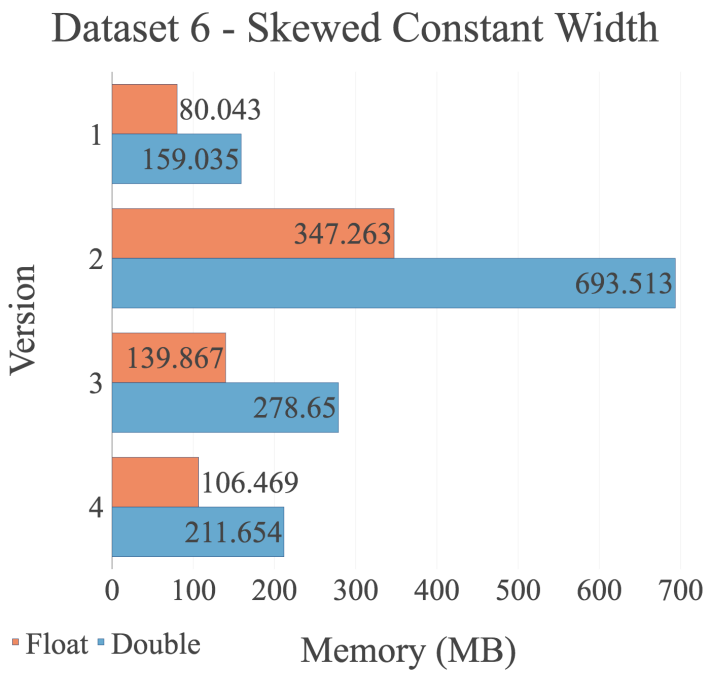






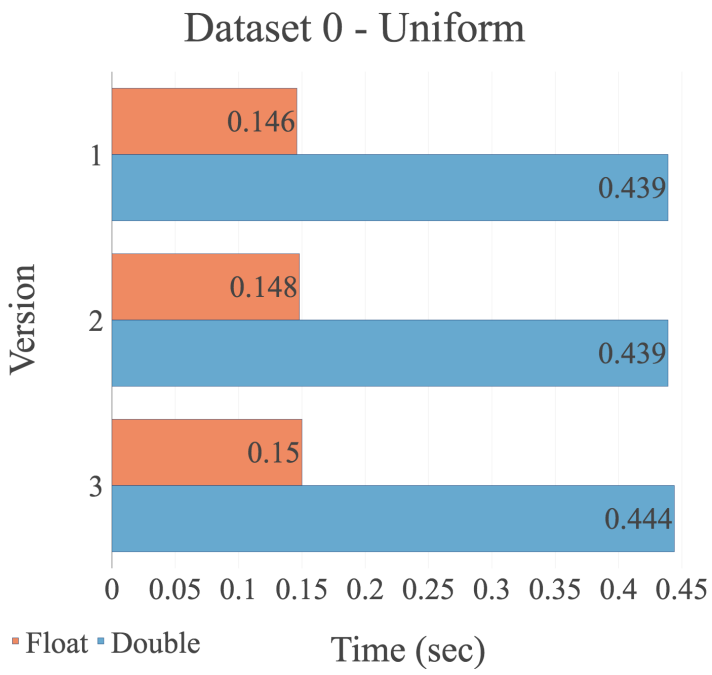


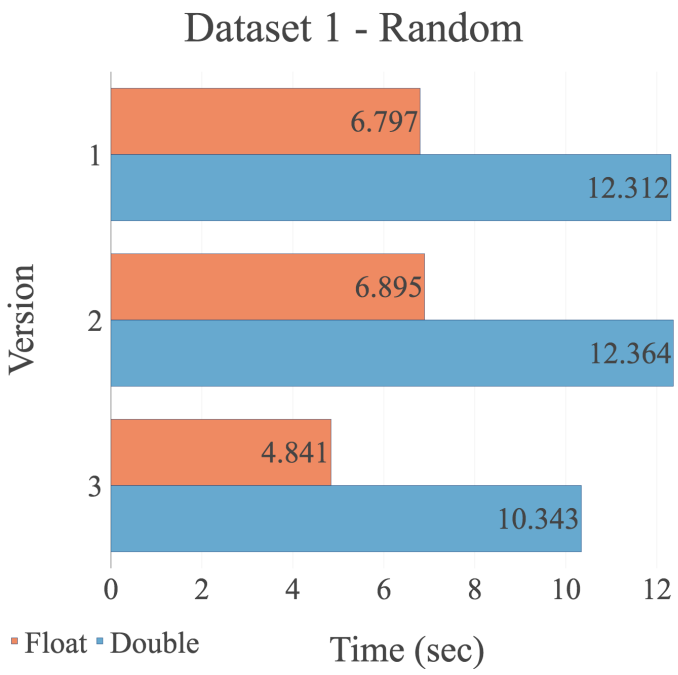

Dataset 4 - Skewed
Version
1
2
3
4
81.821
162.588
558.263
1115.513
156.367
311.595
115.145
228.997
0
200
400
600
800
1000
Float
Double
Memory (MB)
Dataset 5 - Skewed Constant Height
Version
1
2
3
4
80.399
159.747
356.263
711.513
149.116
297.848
108.296
215.484
0
100
200
300
400
500
600
700
Float
Double
Memory (MB)

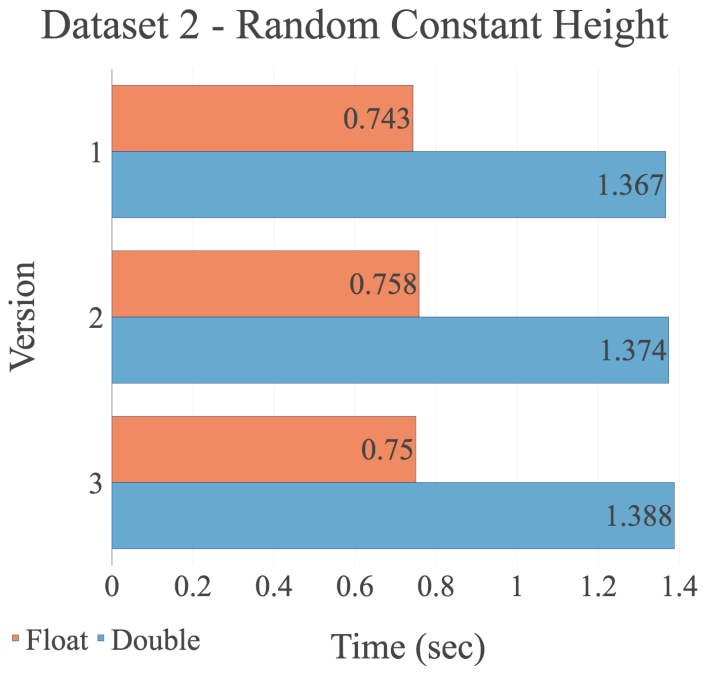

Dataset 6 - Skewed Constant Width
Version
1
2
3
4
80.043
159.035
347.263
693.513
139.867
278.65
106.469
211.654
0
100
200
300
400
500
600
700
Float
Double
Memory (MB)

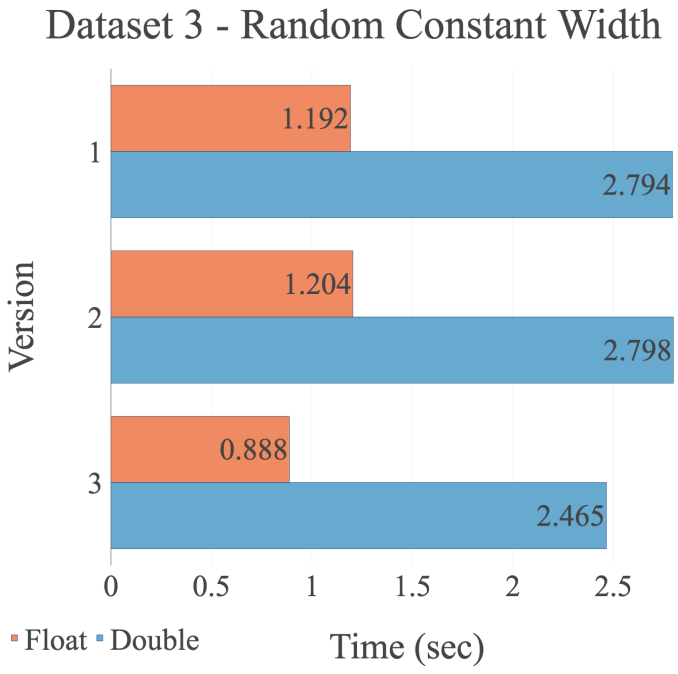
Appendix C. CUDA-multi Experiments

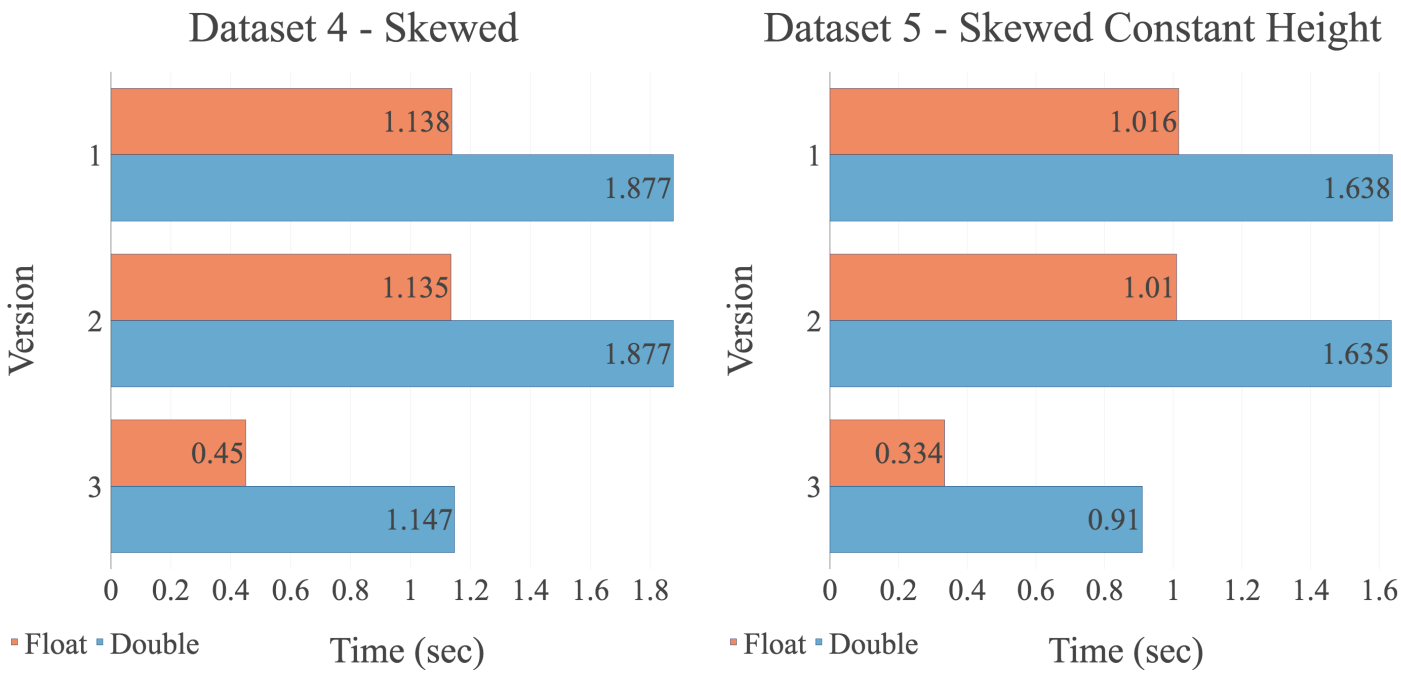
C.1 Version Experiments (Best Runtimes)

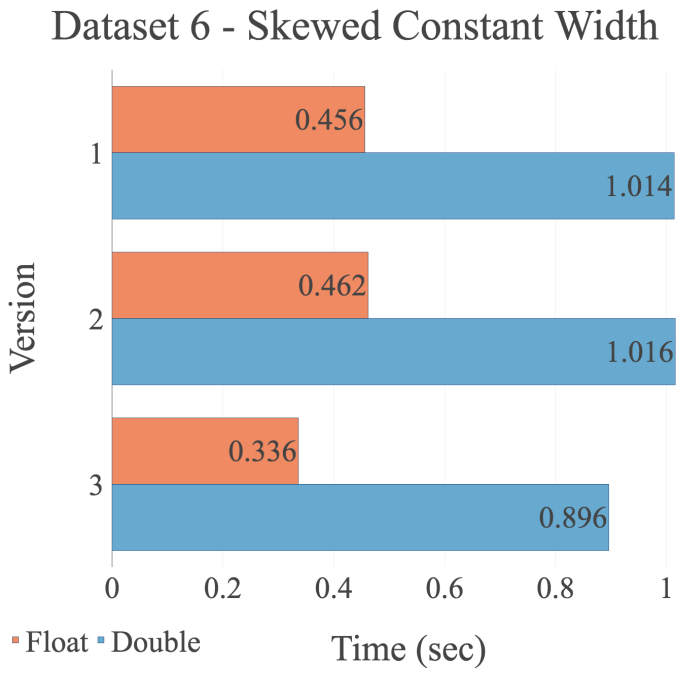




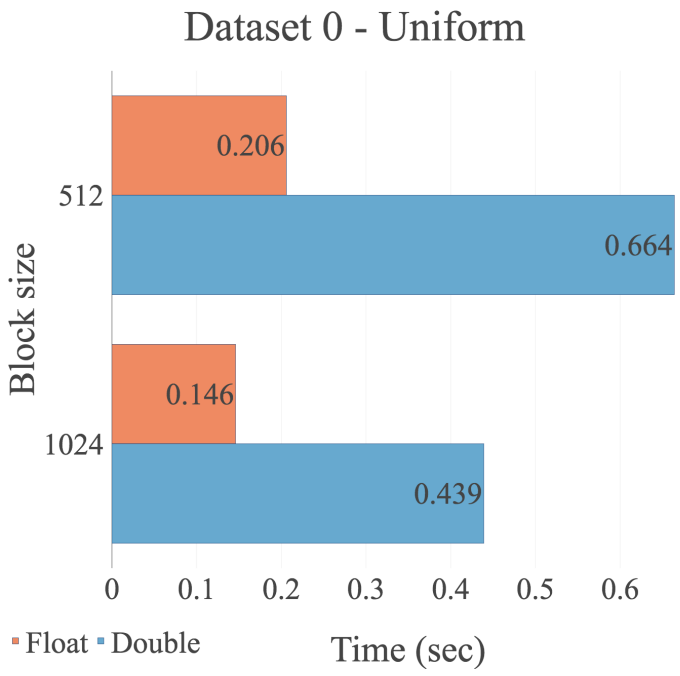


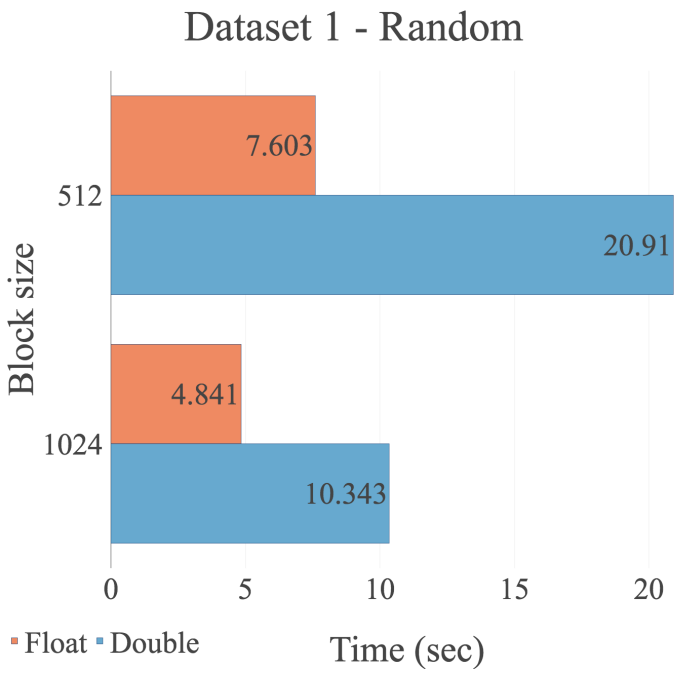


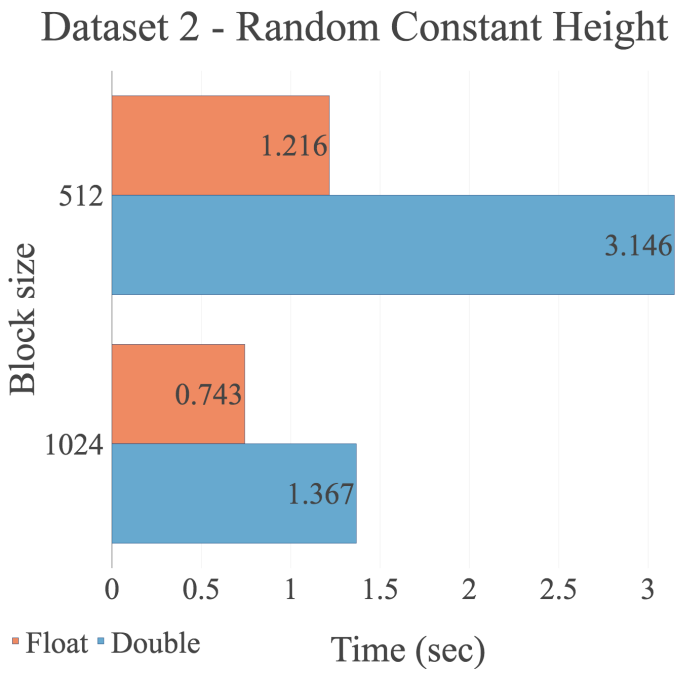

Dataset 4 - Skewed
Version
1
2
3
1.138
1.877
1.135
1.877
0.45
1.147
0 0.2 0.4 0.6 0.8 1 1.2 1.4 1.6 1.8
Float Double
Time (sec)
Dataset 5 - Skewed Constant Height
Version
1
2
3
1.016
1.638
1.01
1.635
0.334
0.91
0 0.2 0.4 0.6 0.8 1 1.2 1.4 1.6
Float Double
Time (sec)

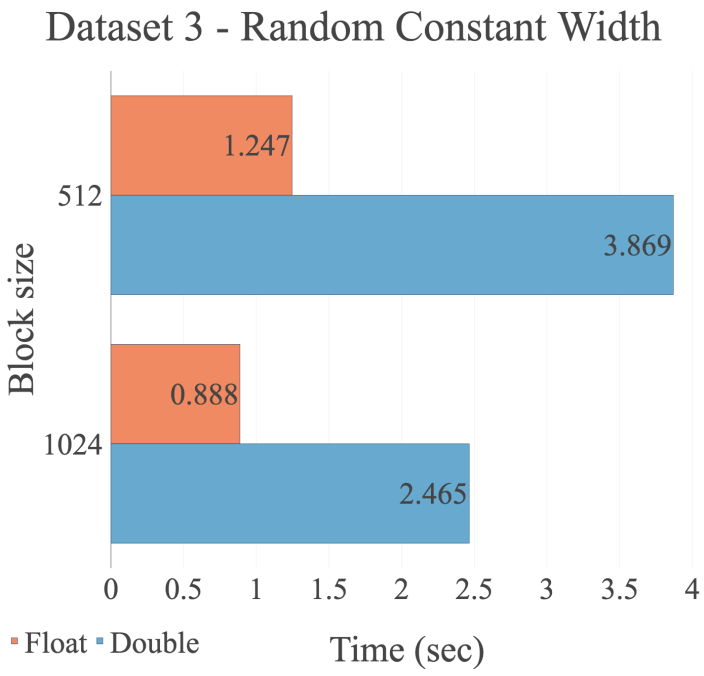

Dataset 6 - Skewed Constant Width
Version
1
2
3
0.456
1.014
0.462
1.016
0.336
0.896
0 0.2 0.4 0.6 0.8 1
Float Double
Time (sec)

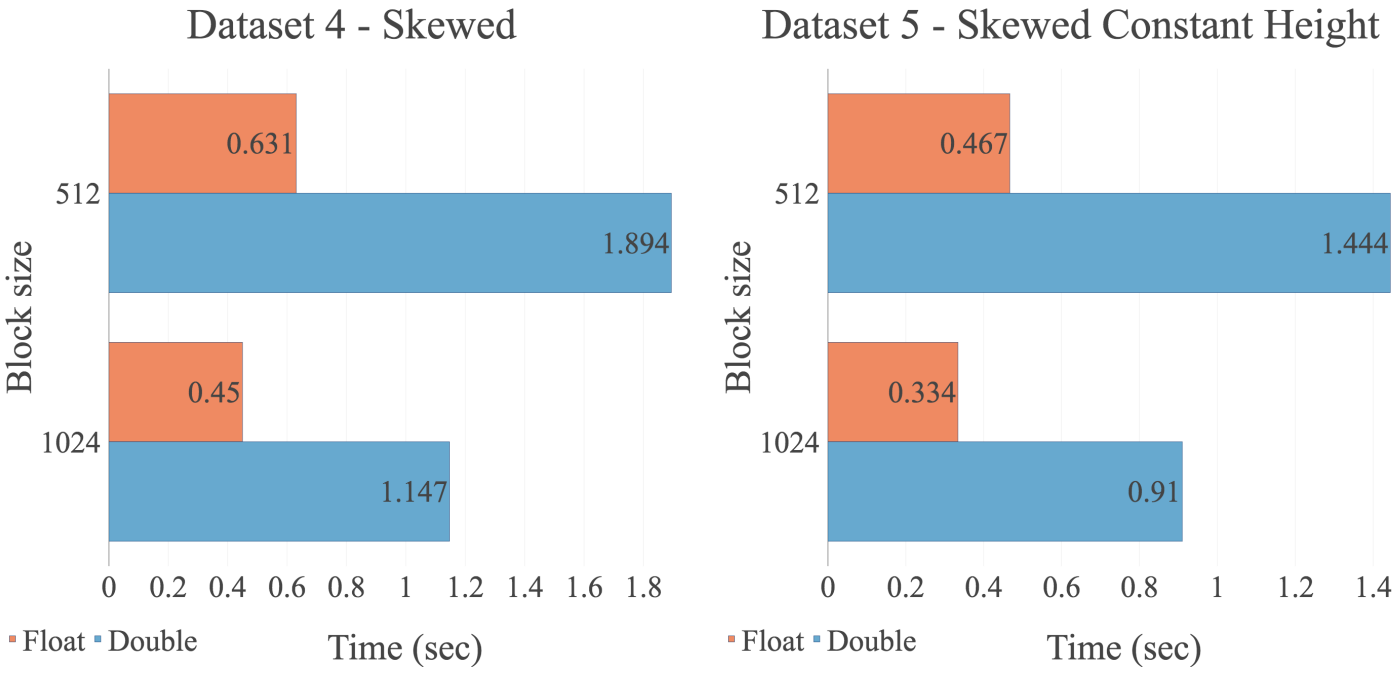
C.2 Block Size Experiments (Best Runtimes)

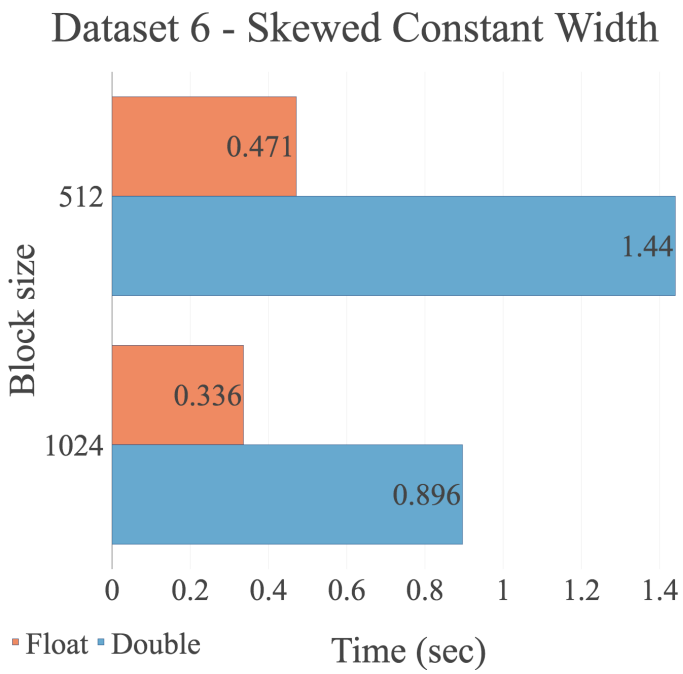




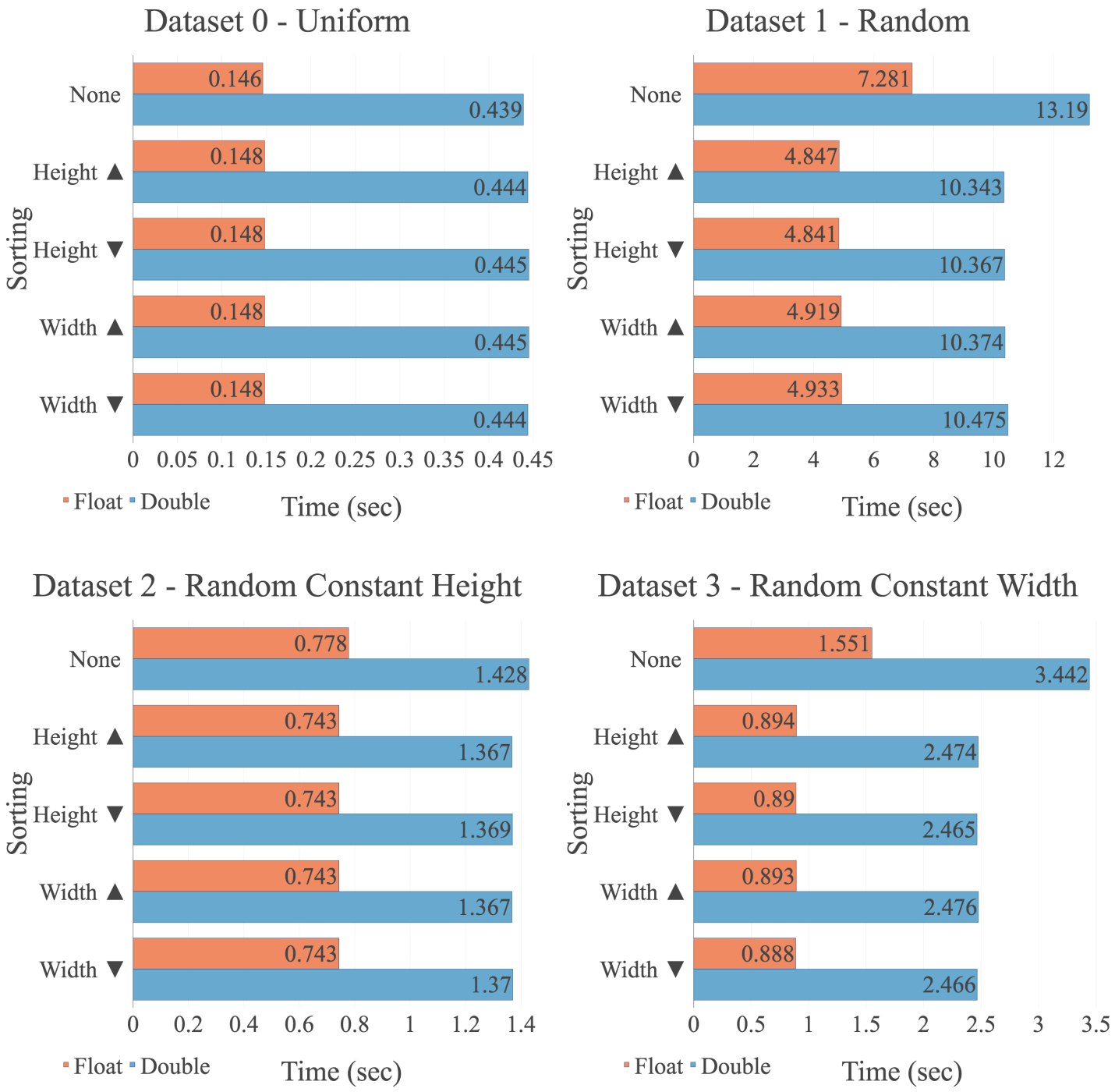


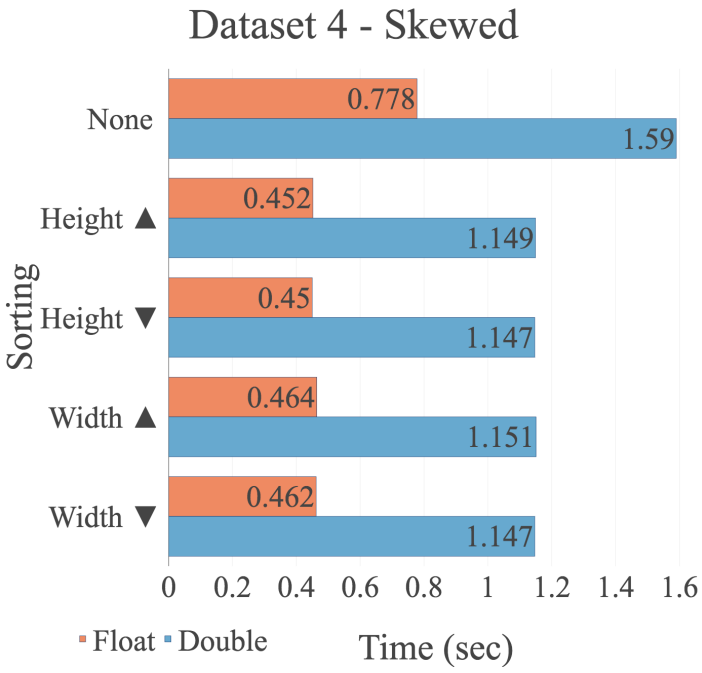


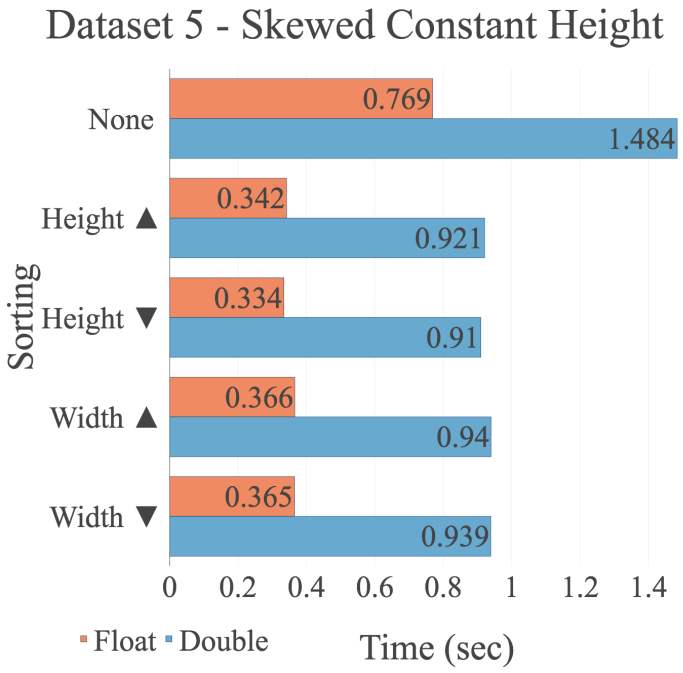

Dataset 4 - Skewed
Block size
512
1024
0.631
1.894
0.45
1.147
0 0.2 0.4 0.6 0.8 1 1.2 1.4 1.6 1.8
Float Double
Time (sec)
Dataset 5 - Skewed Constant Height
Block size
512
1024
0.467
1.444
0.334
0.91
0 0.2 0.4 0.6 0.8 1 1.2 1.4
Float Double
Time (sec)

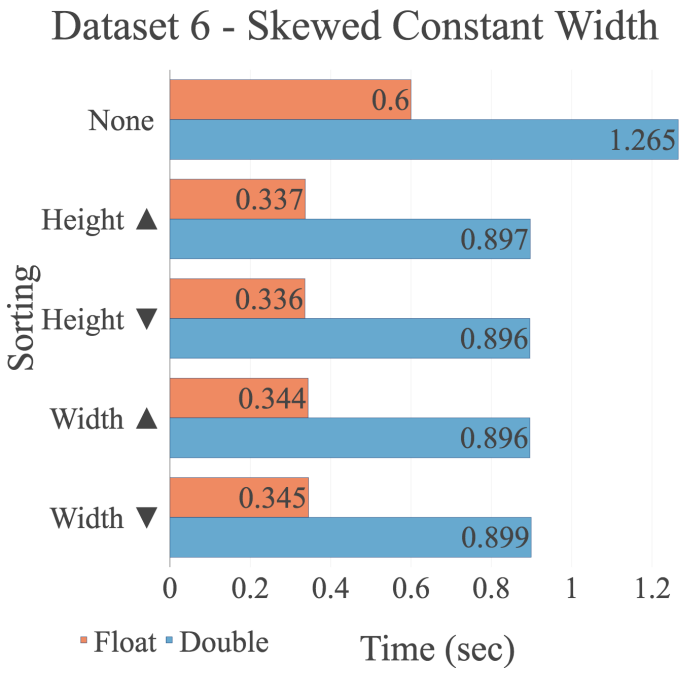

Dataset 6 - Skewed Constant Width
Block size
512
1024
0.471
1.44
0.336
0.896
0 0.2 0.4 0.6 0.8 1 1.2 1.4
Float Double
Time (sec)

C.3 Sorting Experiments (Best Runtimes)

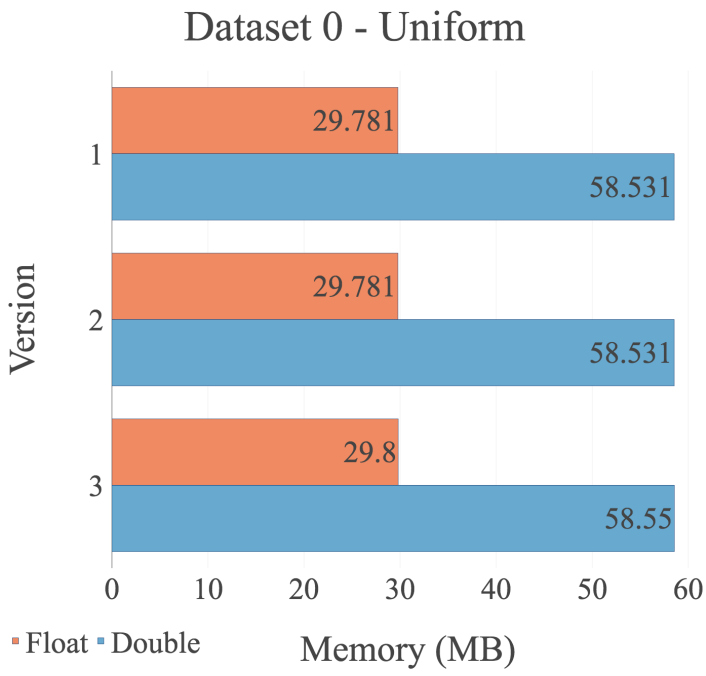


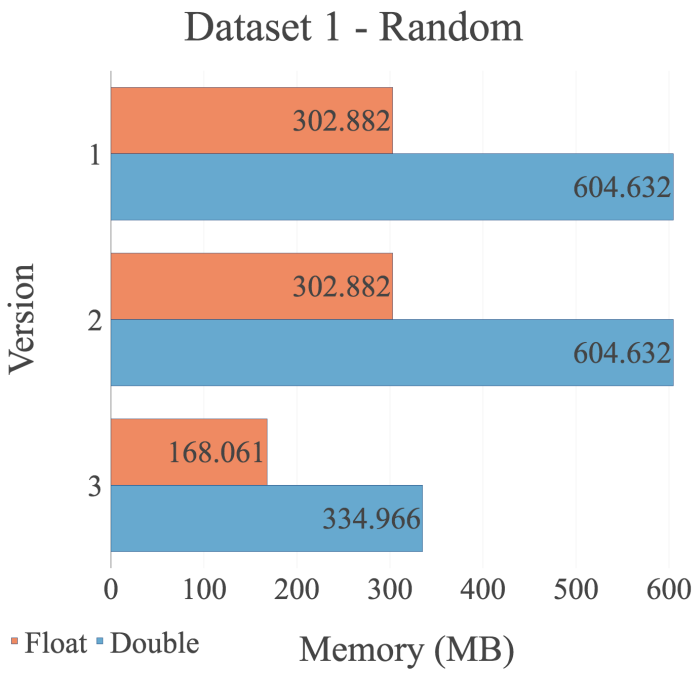

Dataset 4 - Skewed
None
Height ▲
Height ▼
Width ▲
Width ▼
Sorting
0.778
1.59
0.452
1.149
0.45
1.147
0.464
1.151
0.462
1.147
0 0.2 0.4 0.6 0.8 1 1.2 1.4 1.6
Float Double
Time (sec)

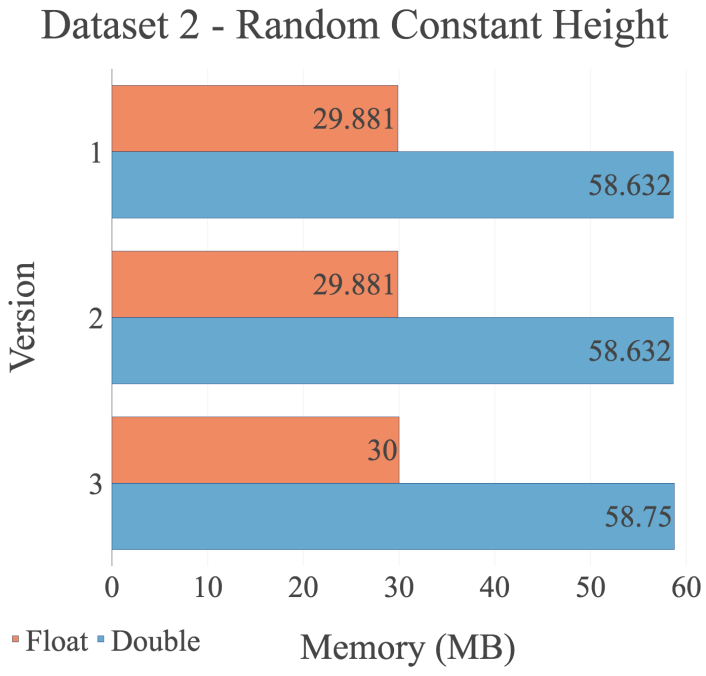

Dataset 5 - Skewed Constant Height
None
Height ▲
Height ▼
Width ▲
Width ▼
Sorting
0.769
1.484
0.342
0.921
0.334
0.91
0.366
0.94
0.365
0.939
0 0.2 0.4 0.6 0.8 1 1.2 1.4
Float Double
Time (sec)

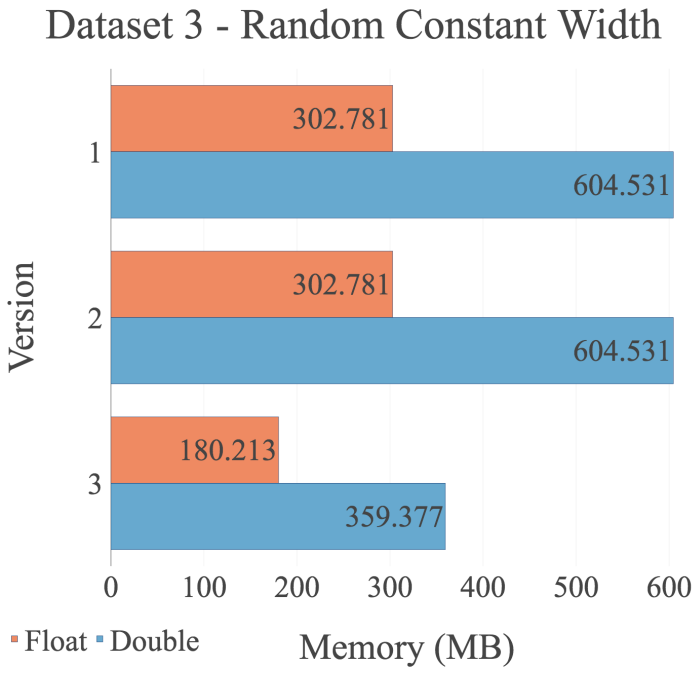

Dataset 6 - Skewed Constant Width
None
Height ▲
Height ▼
Width ▲
Width ▼
Sorting
0.6
1.265
0.337
0.897
0.336
0.896
0.344
0.896
0.345
0.899
0 0.2 0.4 0.6 0.8 1 1.2
Float Double
Time (sec)

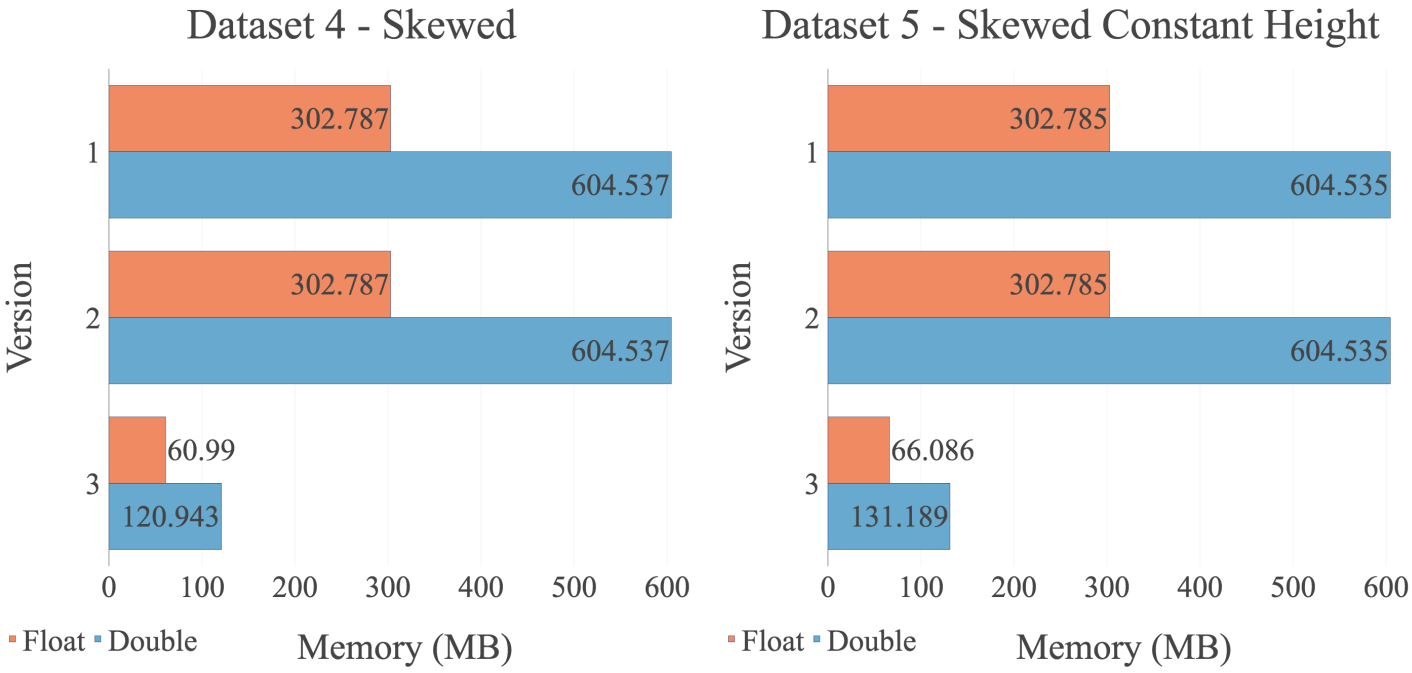
C.4 Version Experiments (Average Memory)

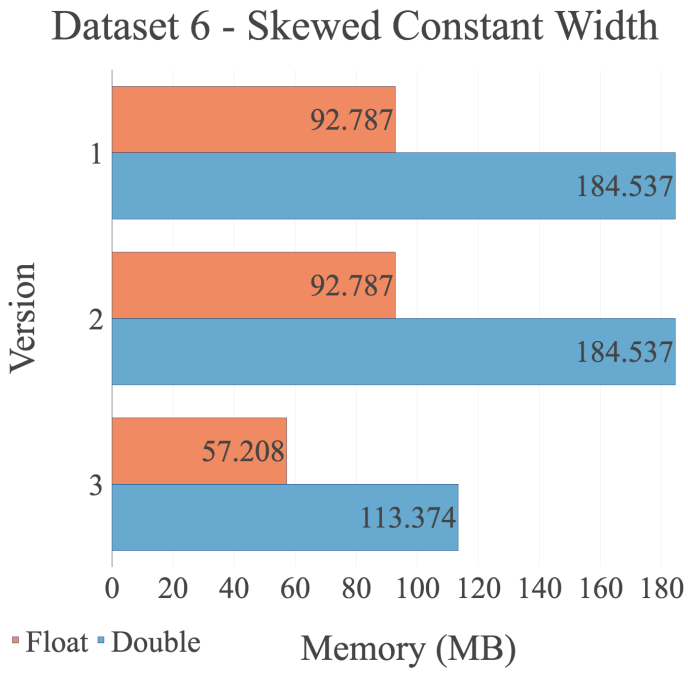






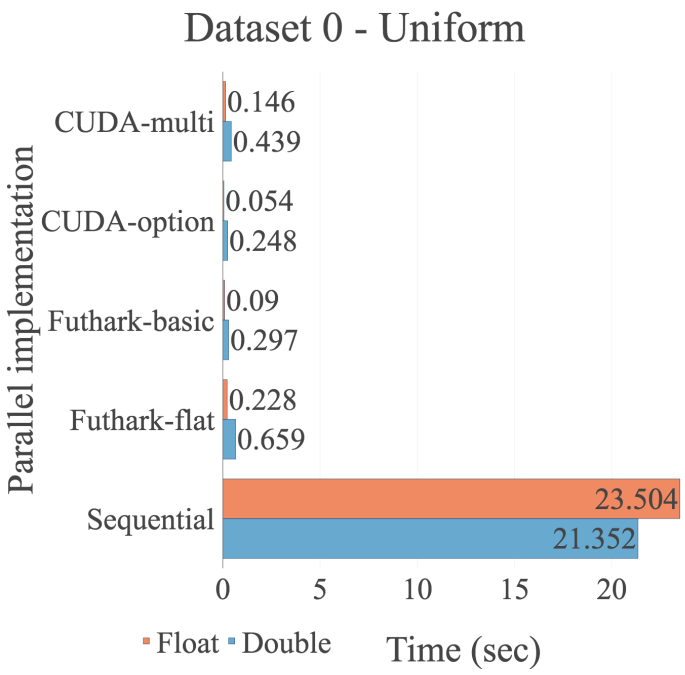


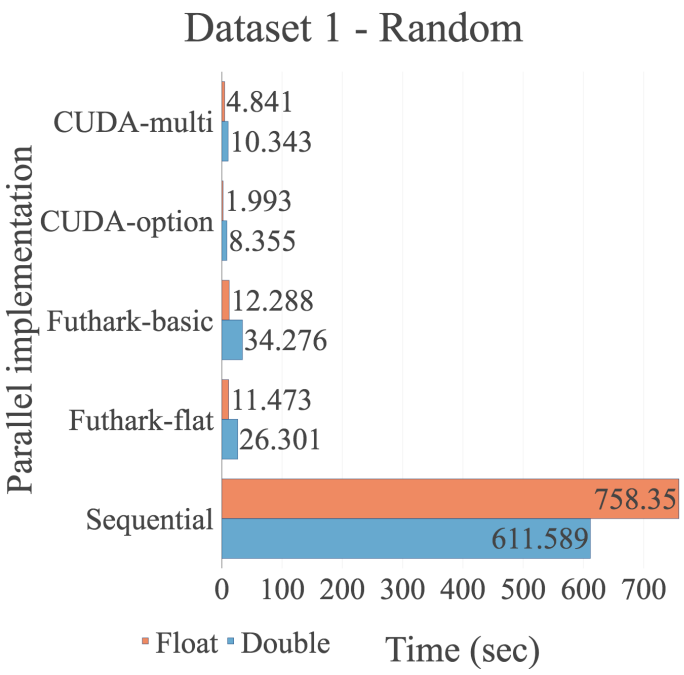

Dataset 4 - Skewed
302.787
604.537
302.787
604.537
60.99
120.943
Version
1
2
3
0
100
200
300
400
500
600
Float
Double
Memory (MB)
Dataset 5 - Skewed Constant Height
302.785
604.535
302.785
604.535
66.086
131.189
Version
1
2
3
0
100
200
300
400
500
600
Float
Double
Memory (MB)

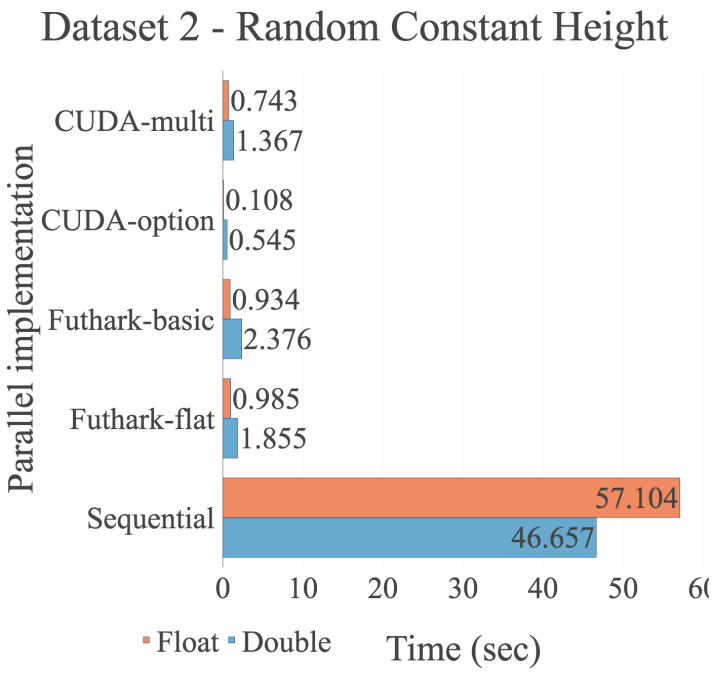

Dataset 6 - Skewed Constant Width
92.787
184.537
92.787
184.537
57.208
113.374
Version
1
2
3
0
20
40
60
80
100
120
140
160
180
Float
Double
Memory (MB)

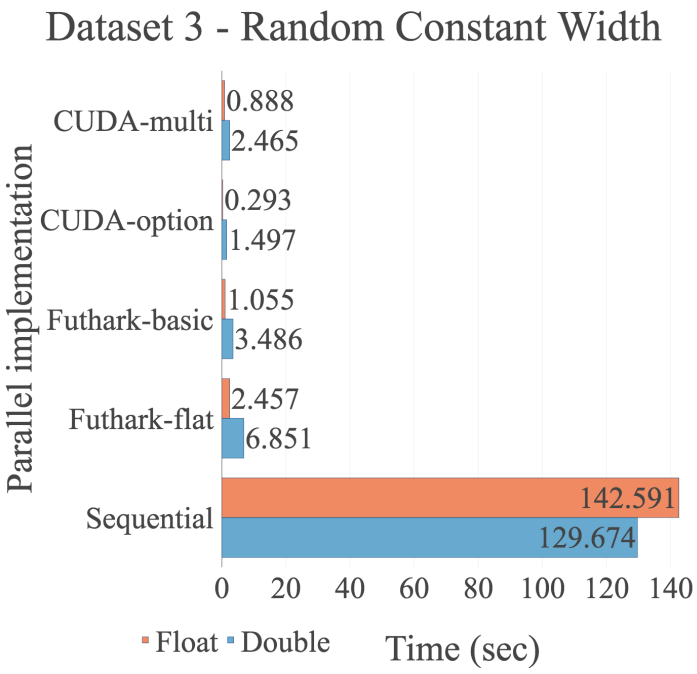
Appendix D. Implementations Experiments

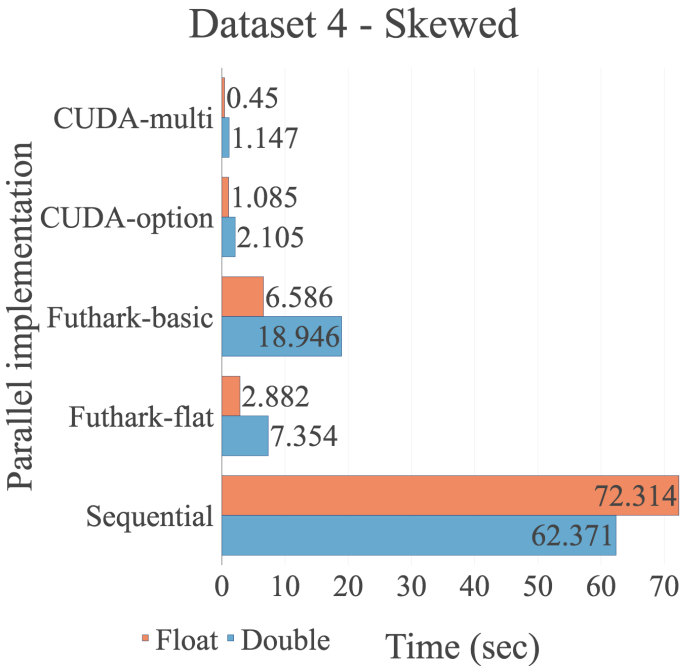
D.1 Parallel Implementations (Best Runtimes)

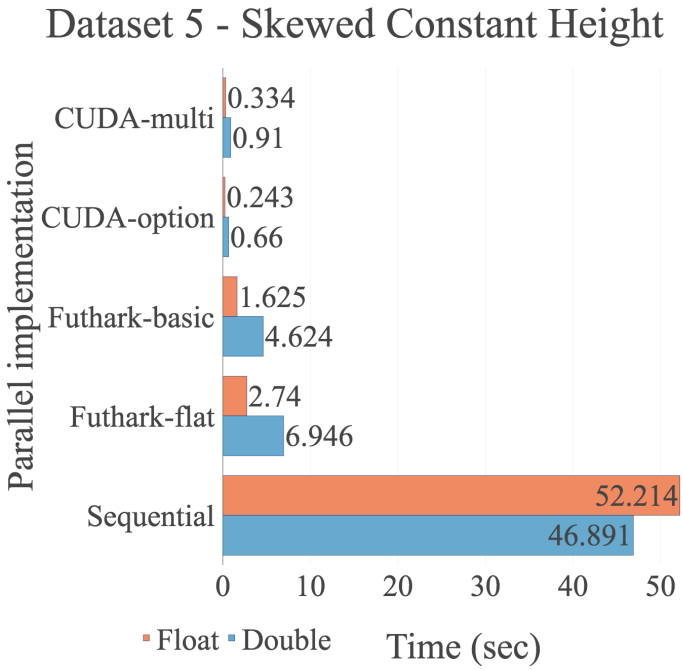


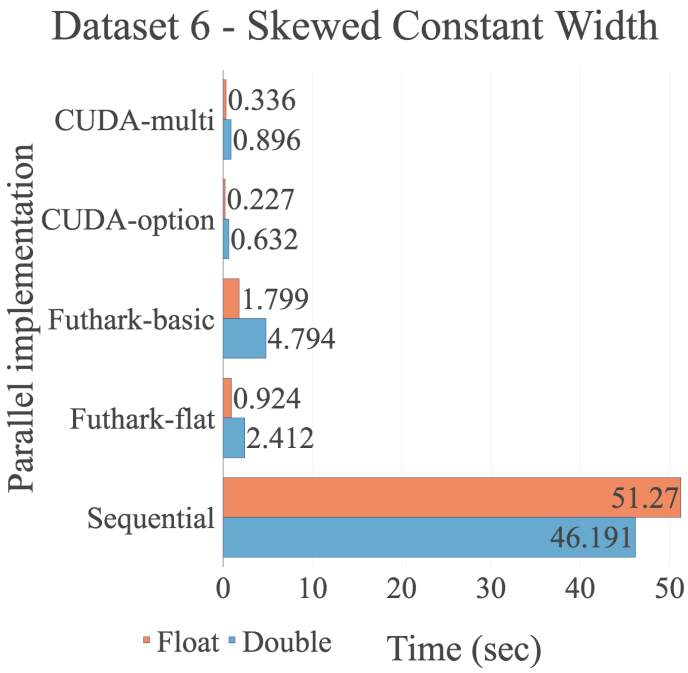




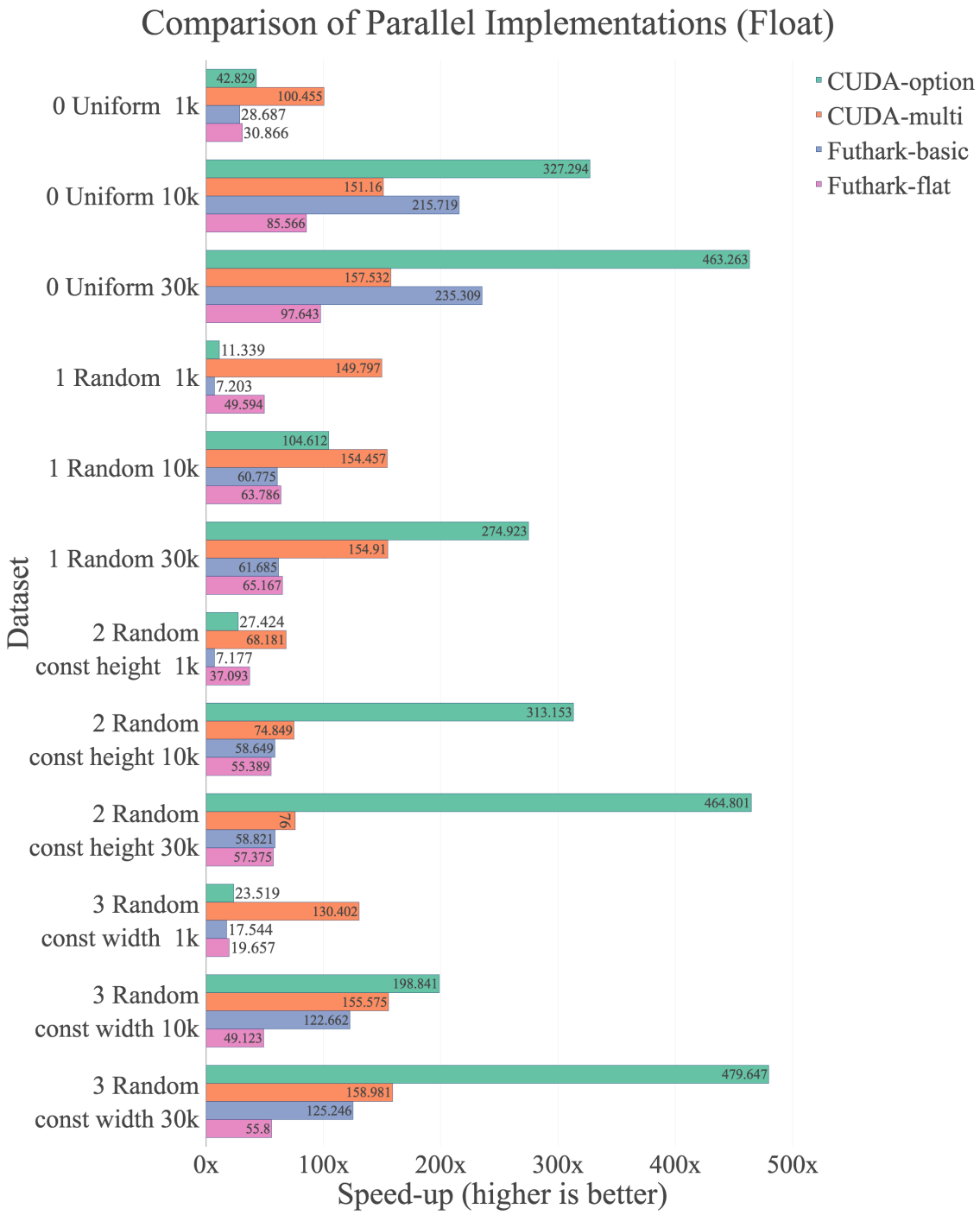



Dataset 4 - Skewed
Parallel implementation
CUDA-multi
CUDA-option
Futhark-basic
Futhark-flat
Sequential
0.45
1.147
1.085
2.105
6.586
18.946
2.882
7.354
72.314
62.371
0 10 20 30 40 50 60 70
Float Double
Time (sec)


Dataset 5 - Skewed Constant Height
Parallel implementation
CUDA-multi
CUDA-option
Futhark-basic
Futhark-flat
Sequential
0.334
0.91
0.243
0.66
1.625
4.624
2.74
6.946
52.214
46.891
0 10 20 30 40 50
Float Double
Time (sec)


Dataset 6 - Skewed Constant Width
Parallel implementation
CUDA-multi
CUDA-option
Futhark-basic
Futhark-flat
Sequential
0.336
0.896
0.227
0.632
1.799
4.794
0.924
2.412
51.27
46.191
0 10 20 30 40 50
Float Double
Time (sec)

D.2 Speed-up on Small Datasets (Float)



D.3 Speed-up on Small Datasets (Double)