Acceleration of Lattice Models for Pricing Portfolios of Fixed-Income Derivatives

Wojciech Michal Pawlak
SimCorp
University of Copenhagen
Copenhagen, Denmark
wmp@di.ku.dk

Marek Hlava
University of Copenhagen
Copenhagen, Denmark
marek.hlava@outlook.com

Martin Metaksov
University of Copenhagen
Copenhagen, Denmark
metaksov@gmail.com

Cosmin Eugen Oancea
University of Copenhagen
Copenhagen, Denmark
cosmin.oancea@diku.dk

Abstract

This paper reports on the acceleration of a standard, lattice-based numerical algorithm that is widely used in finance for pricing a class of fixed-income vanilla derivatives. We start with a high-level algorithmic specification, exhibiting irregular nested parallelism, which is challenging to map efficiently to GPU hardware. From it we systematically derive and optimize two CUDA implementations, which utilize only the outer or all levels of parallelism, respectively. A detailed evaluation demonstrates (i) the high impact of the proposed optimizations, (ii) the complementary strength and weaknesses of the two GPU versions, and that (iii) they are on average 2.4× faster than our well-tuned CPU-parallel implementation (OpenMP+AVX2) running on 104 hardware threads, and by 3-to-4 order of magnitude faster than an OpenMP-parallel implementation using the popular QuantLib library.

CCS Concepts: • **Computing methodologies → Shared memory algorithms**; **Massively parallel algorithms**; **Massively parallel and high per-formance simulations**; Parallel programming languages; • **Applied computing → Economics**.

Keywords: GPGPU (Parallel) Programming, Compilers, Computational Finance, Derivative Pricing

ACM Reference Format:
Wojciech Michal Pawlak, Marek Hlava, Martin Metaksov, and Cosmin Eugen Oancea. 2021. Acceleration of Lattice Models for Pricing Portfolios of Fixed-Income Derivatives. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY '21), June 21, 2021, Virtual, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3460944.3464309

1 Introduction

Pricing is a fundamental component for the *risk management* of any investment portfolio. It applies mathematical modelling and compute-intensive algorithms to approximate the value of financial-derivative instruments. A derivative is a contract that derives its value from other, more basic instruments like fixed-income bonds. Modern derivative portfolios, managed by large institutional investors, usually consist of a large number of such contracts, which differ significantly in their characteristics and cash flows. This diversity has a severe impact on the required computational effort. In recent years, market participants are obliged to track their portfolio risks and report them accurately on a regular and frequent basis. Standard risk analysis combines stress testing against historical data and a simulation of possible future market scenarios, resulting in a big-compute problem, which presents a compelling case for GPU acceleration.

This paper reports on the GPU (and CPU) parallelization of an algorithm that solves a standard pricing model, which is commonly used in practice. Hull-White One-Factor Short-Rate (*HW1F*) model [23] defines the value of an instrument by means of a stochastic differential equation, that represents the random changes in the interest rate over time. The algorithm used to discretise and solve them is a *Trinomial Tree* lattice-based numerical method [24]. Pricing *one instrument* is performed in two main stages: The forward stage builds a tree of bounded width that represents a propagation of the interest rate, which cannot rise indefinitely and reverts to a mean over time, until the maturity of the underlying bond is reached. The backward stage then performs the instrument valuation from maturity back to the current time.

The computational structure comprises two sequential time-series loops that iterate over the height of the tree. Each loop iteration performs several semantically-parallel operations that have the same length as the bounded width of the tree (e.g., the forward step computes each node at the current time step from several neighboring nodes at the previous time step in the tree). Practical use cases require pricing a large portfolio of instruments, which gives raise to an outer level of parallelism.

The main challenge, however, is that in realistic scenarios *the width and height of the trees is highly variant* across the portfolio instruments. This gives raise to a case of irregular nested parallelism that is difficult to map efficiently to

GPU hardware. In particular, related approaches, surveyed in detailed in Section 7, either assume the homogeneous case [15, 20] in which all trees have the same height and width, or acknowledge the problem, but without offering a solution [17]. This paper studies this challenging case of a heterogeneous portfolio, and reports on two parallelization strategies and their subsequent optimizations.

The first strategy follows the common wisdom that states that outer levels of parallelism are more profitable to exploit than inner ones. Section 4 presents such an implementation, named GPU-OUTER, that processes one instrument using one thread, together with a set of optimizations aimed at improving memory footprint and spatial locality. In particular, the high variance of widths and heights across a portfolio introduces two levels of thread divergence on GPU. These levels correspond (i) to the sequential time loop of variant height and (ii) to the inner width-parallel operations, which are sequentialized. GPU-OUTER allows to optimize *one level of divergence, but not both*, e.g., by precomputing the heights of the trees and sorting the portfolio in their decreasing order.

The second strategy is to optimize (i) the height-level of divergence by sorting as before, and (ii) the width level by efficiently exploiting the (irregular) inner level of parallelism at CUDA block (of threads) level, which also allows to maintain most of the data in the fast (shared+register) memory. The idea, described in Section 5, is (i) to *pack instruments into bins*, such that their summed widths fits the size of the CUDA block (bin), and then (ii) to "*flatten*" (merge) the available two-level parallelism. The flattening procedure is highly non-trivial and is a key contribution of this work. In this paper, due to space constraints, we only demonstrate its application to the current pricing method, starting from the nested parallel specification presented in Section 3.[1]

Finally, an evaluation on two Nvidia GPUs shows that:

- The proposed optimizations (data reordering, padding, coalescing global-memory accesses) have high impact: for GPU-OUTER as high as 6.0× and on average 4.1× and for GPU-FLAT as high as 5.8× and on average 2.7×.
- GPU performance is sensitive to both dataset and hardware characteristics.[2] On an older-generation GPU, GPU-OUTER is (moderately) faster than GPU-FLAT by a factor as high as 2.3× and on average 1.8× on large portfolios with constant or randomly-distributed heights and widths. GPU-FLAT is faster in all the other cases: on small portfolios (3.4× on average), or when the distribution of widths/heights is skewed (5.9× on average) or even random on newer GPU hardware (1.8×).[3]
- The best GPU version is faster by a factor as high as 6.7× and on average 2.9× than our tuned multi-threaded and vectorized CPU implementation (OpenMP + AVX2), which at its turn is three-to-four orders of magnitude faster than a parallel OpenMP implementation built on top of the popular QuantLib library [2].

[1] A formalization of the proposed flattening transformation based on rewrite rules is sketched at http://hiperfit.dk/pdf/flat_rwr_hull_white.pdf

[2] In particular, we observe that performance is not portable across different GPU hardware generations on multiple datasets.

[3] To demonstrate that current compiler technology does not support the proposed optimizations/flattening, we also compare with a "matching" GPU implementation written in Futhark [19], which is on average 6.3× slower.

2 Financial Background

Fixed-income instruments like bonds. In this work, we deal with a specific class of bond instruments, that are heavily traded in fixed income markets. Bonds are characterized by a cash flow during their lifetime. We consider a bond paying a (fixed) coupon rate on specified dates up until bond maturity. The value of coupons is most often fixed for fixed-rate bonds, but varies for floating rate notes (FRNs). The payment dates are usually periodic, but we assume they can occur on any future date. However, the market value of a fixed-rate bond is susceptible to fluctuations in interest rates, and carries a significant amount of risk.

Derivative instruments like Options. A derivative (a contract) derives its own value from the value of the underlying asset. A *call or put* vanilla bond option gives an investor the right, but not the obligation, to *buy or sell* the underlying bond for a predetermined strike price at a future exercise date before maturity. Options are typically used for hedging of portfolio risks or market speculation. However, exercise time determines the style of a vanilla options: European can be exercised on one particular date, while Bermudan, on many specific, usually periodical, dates, and American, on any date up until the last exercise date. Analytical formulas exist for an exact valuation of European options, but the other two can only be approximated by numerical methods.

Instrument assumptions. In this work, we deal with a *multi-callable or puttable bond with a fixed or floating coupon.* We focus on Bermudan or American optionality, but capture any bond cash flow. Moreover, we assume that bonds have only one underlying factor, an interest rate of the currency they are traded in. More detailed descriptions can be found in standard literature [10, 21].

Hull-White One-Factor Short Rate Model. One of the most popular short-rate models for interest rate derivative pricing is Hull and White (1990, 1994a) [22, 23], further abbreviated *HW1F.* In this model there is a single source of risk or uncertainty, a short (interest) rate, which is applicable at instantaneous or very short periods of time. The model is based on a stochastic process, which describes a probabilistic evolution of the random short rate over future time. The process also assumes the future interest rates are a function of the initial rates and that their movement is *reverting to the mean.* In this work, we consider a simplified version of the Hull-White extension of the Vasicek model, where volatility σ and mean reversion rate a parameters are constant.

The dynamics of *HW1F* is expressed mathematically by a *Stochastic Differential Equation*:

$$dr = [\theta(t) - ar]dt + \sigma dW \tag{1}$$

The drift (first) part of the equation includes a constant short rate value r, which follows a mean-reverting Ornstein–Uhlenbeck process, i.e. it is pulled back toward a central value with a rate a. $\theta(t)$ is a deterministic function of time chosen to fit the theoretical bond prices to the yield curve observed in the market; it defines the average direction that r moves with rate a at time t. The diffusion (second) part comprises of σ and the stochastic variable dW. In addition, σ determines the absolute level of short rate volatility, while a determines the relative volatilities of long and short rate. A high value a causes short-term rate movements to damp out quickly, so long-term volatility is reduced.

2.1 Hull-White Trinomial Tree

Hull and White (1994a,1996) [23, 24] outline a trinomial tree construction procedure for solving the *HW1F* model. We summarize the main algorithmic steps in this section.

Forward Propagation: Tree Construction. The tree is constructed in a breadth-first manner—the root corresponding to the current time, and the leaves to the bond maturity time—where the nodes at a certain (time) tree level denote possible values for the short rate at that time step. The tree *height* is defined by the remaining time to the maturity of the bond, specified in units denoting a fraction of a year. The tree *width* is determined by a mean reversion rate (the lower the rate the wider the tree), specified as a basis point. The *height* varies across bonds, while the *width* is determined by a calibration of a to market data. Both dimensions also depend on how often we monitor the changes in interest rate. Every bond depends on a *term structure* (*yield curve*), a relationship between interest rates and maturity terms, representing the expectation of market evolution.

The first stage constructs a preliminary tree, where $\theta(t) = 0$ and initial value $r = 0$. We define r as a continuously compounded Δt-period rate. We denote the expected value across a Δt-period $r(t + \Delta t) - r(t)$ as $-ar(t)\Delta t$ and the variance of $r(t+\Delta t)-r(t)$ as $\sigma^2\Delta t$. The time step size is $\Delta t = T/n$, where T is the bond maturity and n is the number of desired time steps. The interest rate step size is then $\Delta r = \sigma\sqrt{3\Delta t}$, which is a theoretical choice. Δt progresses along *height*, while Δr are distributed along the *width*. We denote by (i, j) the tree node for which $t = i\Delta t$ and $r = j\Delta r$, where i denotes the time step along the *height*, and j the rate step along the *width* (the middle node has index $j = 0$).

A *trinomial tree* represents a random propagation of the interest rate in time. In our case, the value of node (i, j) is the Arrow-Debreu state price $Q_{i,j}$, which corresponds to the value of a security that pays 1 if node (i, j) is reached and 0 otherwise. Any derivative depends on an underlying asset, so its value that is uncertain at exercise time can be

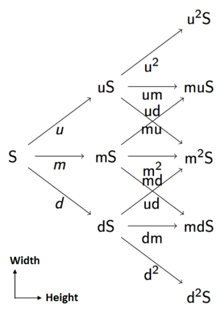

Figure 1. Trinomial-tree construction for a stochastic variable S, as in Boyle (1986) [9].

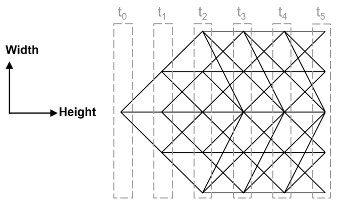

Figure 2. Trinomial tree with a bounded width incorporating the mean reversion phenomenon, i.e., a *HW1F* Tree.

decomposed at each time step as a linear combination of its Arrow-Debreu securities, and thus as a weighted sum of its state prices. In Figure 1, the weights on the edges represent the probabilities of transitions from one value to the other. The tree is considered "trinomial", because the computation of each node value at a current time step i depends on three node values from the previous time step $i - 1$. Essentially, at each node there is a choice u, m and d to branch upward, horizontally and downwards, with probabilities p_u, p_m, and p_d, respectively, where $p_m + p_u + p_d = 1$. A *HW1F tree* is obtained by bounding the tree width to $2 * j_{max} + 1$, where $j_{max} = -0.184/M$ and $M = e^{(-a\Delta t)} - 1$. This is motivated by the empirical observation that the short rate reverts to the mean over time. The current node (i, j) contributes to the computation of three nodes of the next time step $(i + 1)$:

- if $j = -j_{max}$ then horizontally to node j, and upwards to nodes $j + 1$, and $j + 2$;
- if $j = j_{max}$ then horizontally to node j, downwards to nodes $j - 1$, and $j - 2$;
- otherwise horizontally, upwards and downwards to nodes j, $j + 1$ and $j - 1$, respectively.

Figure 2 shows an example, where $j_{max} = 2$, width is 5, height is 6, $i \in [0, 5]$ and $j \in [-2, 2]$.

The second stage adjusts the constructed tree from the first stage to match the initial term structure observed in the market. This is achieved by displacing the nodes (i.e., the Arrow-Debreu state prices Q) at time $i\Delta t$ by an amount α_i, i.e., the new value of $Q_{i,j}$ in the displaced tree is then equal to $Q_{i,j} + \alpha_i$. The value of α_i is determined from a sum

```
iota: int → [n]int                     iota n = [0,...,n-1]
replicate: (n:int)→ α → [n]α replicate n v = [v,...,v]

map:∀n.(α → γ) → [n]α → [n]γ
map f [a_1, ..., a_n] = [f a_1, ..., f a_n]
map2:∀n.(α → β → γ) → [n]α → [n]β → [n]γ
map2 g[a_1,..,a_n][b_1,..,b_n] = [g a_1 b_1, ..., g a_n b_n]

reduce:∀ n. (α → α → α) → α → [n]α → α
reduce ⊙ e_⊙ [a_1,...,a_n] = a_1 ⊙... ⊙ a_n

scan: ∀ n. (α → α → α) → α → [n]α → [n]α
scan^inc ⊙ e_⊙ [a_1,...,a_n] = [a_1,a_1⊙a_2,..,a_1 ⊙...⊙a_n]
scan^exc ⊙ e_⊙ [a_1,...,a_n] = [e_⊙,a_1,..,a_1 ⊙...⊙a_{n-1}]

sgmscan:∀ n. (α → α → α) → α → [n]int → [n]α → [n]α
sgmscan^inc ⊙ e_⊙ [..,1,    0,    ..,  0,  1,   ..  ]
                  [..,a_1^k,a_2^k,  ..,a_n^k, a_1^{k+1},..] =
                  [..,a_1^k,..,Σ^⊙_{i=1..n} a_i^k, a_1^{k+1},..]

scatter: ∀ n,m.[n]α → [m]int → [m]α → [n]α
scatter [a_0, a_1, a_2, a_3,..,a_{n-1}]
        [2, -1,  0,  3]
        [b_0, b_1, b_2, b_3] =
        [b_2, a_1, b_0, b_3,..,a_{n-1}]
```

Listing 1. Parallel Operators of the Functional Notation

of state prices Q across all nodes at the previous time step $(i-1)$ and the bond price $P(0,i)$ with maturity in the current time step i. Our implementation combines the two stages by performing them one after another at each time step to avoid traversing the tree twice.

Backward Propagation: Option Pricing. Once the complete term structure has been calculated at each node, the tree can be used to value a wide range of derivatives. We use a *backward propagation* to value a bond with embedded options from the maturity back to the current time at the root. At each step we adjust for the cash flows, accrued interest or the eventual option exercise and discount the bond price. We end up with the estimated bond price as of the calculation day at the root of the tree. This computation (re)uses the α values computed during forward propagation.

3 Notation & Parallel Specification

Functional Notation. In this paper we use a functional notation, which allows (i) to concisely specify all available parallelism in terms of well known operators, and (ii) to demonstrate at a high level how GPU-FLAT was derived.

We use `[n]`α to denote the type of an array of length `n` and element type α, `[`$a_1,\ldots,a_n$`]` to denote an array literal, and we use `(a,b)` to denote a tuple (record) value. Applying function `f` on two arguments `a` and `b` is written as `(f a b)`. The notation supports the usual unary/binary operators and normalized `let` bindings, which have the form:
`let a =` e_1 `let b =` e_2`... in` e_n and are similar to a block of statements followed by a return denoted by keyword `in`.

In-place updates to array elements are allowed and denoted by `let arr[i] = x`. The notation supports *if* branches `if c then` e_1 `else` e_2, of similar semantics with the C ternary operator `c?` e_1 `:` e_2, and *loop* expressions:
`loop (`x^1`, ..., `x^m`) = (`x_0^1`, ..., `x_0^m`) for i<n do e`.
Here, $x^{1\cdots m}$ are loop-variant variables that are initialized for the first iteration with in-scope variables $x_0^{1\cdots m}$. The loop iterates `i` from `0` to `n-1`, and the result of the loop-body expression `e` provides the values of $x^{1\cdots m}$ for the next iteration. The initialization may be syntactically omitted if in-scope variables $x^{1\cdots m}$ exist; these will provide the initial values.

The notation supports several key *parallel operators*, whose types and semantics are shown in Listing 1: `iota n` creates an integral array with elements from `0` to `n-1`. `replicate n v` creates an array of length `n` whose elements are all `v`. `map` applies its function argument `f` to each element of the input array, resulting in an array of the same length. The function can be declared in the program or can be an anonymous (λ) function—e.g., `map (`λ`x->x+1) arr` adds one to each element of `arr`. `reduce` successively applies an *associative* operator $\odot$ to all elements of its input array, where $e_\odot$ denotes the neutral element of $\odot$. `scan` [5], a.k.a., prefix sum, is similar to `reduce`, except that it produces an array of the same length (`n`) containing all prefix sums of its input array. The *inclusive* scan (`scan`^{inc}) starts with the first element of the array, and the *exclusive* scan (`scan`^{exc}) with the neutral element. Segmented scan (`sgmscan`) has the semantics of a scan applied to each subarray of an irregular array of subarrays. The latter has a flat representation consisting of (i) a flag array made of `0`s and `1`s, where `1` denotes the start position of a subarray, and (ii) a flat array containing all elements of all subarrays in order. E.g., `flag = [1,0,1,0,0,0,1]` would denote an array with 3 rows, having 2, 4 and 1 elements, respectively. `sgmscan`^{inc} `(+) 0 flag [1,2,3,4,5,6,7]` results in `[1,3,3,7,12,18,7]`.

Finally, `scatter x is vs`, updates in place the array `x` at indices contained in `is` with the values of `vs`, except that out-of-bounds indices, such as `-1` are omitted (not updated).

Nested Data-Parallel Specification. Listing 2 sketches the simplified implementation of the algorithm, but still accurately captures the nested-parallel structure. The `main` function (at the bottom) receives a portfolio of instruments and performs a valuation of each by an embarrassingly-parallel `map` operation, that can be easily distributed across threads, GPUs or nodes. Function `valuate` receives an instrument data and computes its price approximation. As in Section 2.1, we denote by arrays `Qs` and α`s` the Arrow-Debreu state prices and their displacements, which are computed during forward propagation, and by `Cs` the bond prices with an embedded optionality computed during backward propagation.

At the start, computation determines the width `w` and height `h` of the trinomial tree (assumed to be performed by the call to function `f1` at line 2), as well as initializes arrays `Qs` of size `w` (`f2`, lines 3-4) and α`s` of size `h` (`f3`, lines 5-7). The two loops of indices `i` and `ii` implement the forward and backward tree propagation, which are sequential in nature.

```
1  let valuate(ins: Instrument) : real =
2    let (w,h) = f1 ins
3    let Qs = replicate w 0.0
4    let Qs[w/2+1] = f2 ins
5    let αs = replicate h 0.0
6    let α_i = f3(ins)
7    let αs[0] = α_i
8    let (_,αs) =
9      loop (Qs,α_i,αs) for i < h-1 do
10       let Qs' =
11         map (λ j ->
12               let q0 = Qs[j]
13               let q1 = if j > 0 then Qs[j-1] else 1.
14               let q2 = if j<w-1 then Qs[j+1] else 1.
15               in  g1(i,j,α_i,q0,q1,q2)
16         ) (iota w)
17       let α_v = reduce (+) 0.0 Qs'
18       let α_i'= g2 α_v ins
19       let αs[i+1] = α_i'
20       in  (Qs', α_i', αs)
21   let Cs = replicate w 100.0
22   let Cs =
23     loop (Cs) for ii < h-1 do
24       let i = h - 2 - ii
25       let α_i = αs[i]
26       in map (λ j ->
27                let c0 = Cs[j]
28                let c1 = if j > 0 then Cs[j-1] else 1.
29                let c2 = if j<w-1 then Cs[j+1] else 1.
30                in  g3(i,j,α_i,c0,c1,c2)
31         ) (iota w)
32   in Cs[w/2+1]
33
34 let main(portfolio:[]Instrument)=map valuate portfolio
```

Listing 2. Nested-Parallel Implementation.

The first loop fills in the values of array αs: First, the map operation of length w (lines 11-16) computes each element at the current breadth level in the tree (time step), i.e., Qs'[j], by aggregating the three different values from the previous breadth level, i.e., Qs[j-1], Qs[j], Qs[j+1]. (Notice that only the current and previous breadth levels, rather than the entire tree, are manifested in memory using arrays Qs' and Qs.) The newly created array Qs' is then summed up, using reduce (line 17), and provides the value of αs[i+1]. The two parallel operators occur inside the outer map that is applied to the whole portfolio, thus giving raise to nested parallelism. Finally, the end values of Qs', α_i' and αs are bound to the loop-variant variables Qs, α_i and αs for the next iteration. The second loop (lines 23- 31) traverses the tree backwards, from the maturity to the present date, and at each step computes the prices using a map. The result is at the tree root Cs[w/2+1].

4 GPU-OUTER and Optimizations

GPU-OUTER is derived by mapping each instrument to one thread, thus sequentializing the inner parallelism available in the valuate function. While most of our CUDA implementation is straightforward, one non-trivial issue refers to the fact that arrays such as Qs and αs need to be *expanded* across all valuations in the portfolio and to be stored in global memory. This section discusses two performance critical optimizations. The first refers to finding a good layout for the expanded arrays, named Qs^{exp} and αs^{exp}, that enables coalesced access to global memory, while minimizing memory pressure. The second refers to diminishing the overhead of one of the two levels of thread divergence.

Naive Expanded-Array Layout. Assuming the portfolio consists of n instruments, a naive layout can be determined by pre-computing (via a map) the width and height of each of the n trees into two arrays ws and hs. Summing these arrays produces the total length of Qs^{exp} and αs^{exp}. Next, one can compute the starting offsets into Qs^{exp} of the logically-local arrays Qs—one for each iteration of the outer map—by applying an exclusive scan operation on ws, which results in an array named Qs_offs. For example, the Qs corresponding to iteration (thread) i of the outer map is represented by the slice Qs^{exp}[Qs_offs[i]:Qs_offs[i+1]] of length ws[i]. Similar thoughts apply to αs^{exp}. The inspector code is presented below, where unzip transforms an array-of-tuples to a tuple-of-arrays:

```
let (ws, hs) = unzip (map f1 portfolio)
let Qs_offs   = scan^exc (+) 0 ws
let len_Qs^exp = Qs_offs[n-1] + ws[n-1]
```

The problem is that this layout results in poor spatial locality, because consecutive threads access global memory with a large stride (equal to the values of w or h), and such uncoalesced access may be prohibitively expensive on GPUs.

Global Padding Enables Coalesced Access. Matters can be improved by computing the maximal width w^{m} and height h^{m} across all n trees and padding each (local) subarray to this size, i.e., Qs^{exp} and αs^{exp} are now two-dimensional arrays of sizes n×w^{m} and n×h^{m}, respectively. Modulo thread divergence (imbalanced) issues, fully coalesced access to global memory is achieved by working with the *transposed* versions of Qs^{exp} and αs^{exp}: the inner array dimension of size n is indexed by the thread (instrument) number, hence consecutive threads would now access consecutive memory locations. The downside is a potential memory-footprint explosion, e.g., under skewed distributions of widths/heights.

Block/Warp-Level Padding. The memory explosion can be remedied by padding at finer granularity, for example, at CUDA thread-block or warp level. Denoting the block (warp) size by B, padding is accomplished (i) by finding the maximal width (and height) for each group of B instruments, (ii) followed by padding to the maximal size within the group, (iii) by computing the start offset of each group via a scan, and (iv) by working with the transposed version of the arrays.

```
let wbs' = reshape (n/B, B) ws
let wb_max = map (reduce max 0) wbs'
let pad_lens = map (λ w -> w*B) wb_max
let blk_offs = scan^exc (+) 0 pad_lens
```

For example, the expanded array for block b is the slice $\texttt{Qs}_b^{exp}$ = $\texttt{Qs}^{exp}$[blk_offs[b] : blk_offs[b+1]], which is seen as a 2D array of shape B×$\texttt{wb}_{max}$[b], i.e., the start-index of logically-local array Qs corresponding to local thread i is (blk_offs[b]+i*$\texttt{wb}_{max}$[b]). To obtain coalesced access, we transpose $\texttt{Qs}_b^{exp}$ of shape $\texttt{wb}_{max}$[b]×B.

Data Reordering. The code in Listing 2 exhibits two levels of divergence. This is because the body of the valuate function is executed (sequentially) by each thread. The recurrences appearing inside valuate are (i) the two forward- and backward-traversal loops of count h, and (ii) the enclosed (inner) map-reduce computations of length w. Since both the height h and width w of the tree varies significantly across instruments, it follows that both parameters are sources of thread divergence and their combination exacerbates it. For example, if two threads executing in lockstep have ($\texttt{w}_1$,$\texttt{h}_1$)=(1,m) and ($\texttt{w}_2$,$\texttt{h}_2$)=(m,1), then their execution takes $O(\texttt{m}^2)$ time, rather than the expected $O(\texttt{m})$ time.

With GPU-OUTER, one of the levels of divergence (but not both) can be optimized by a pre-processing step that sorts the portfolio of instruments after the heights or widths of their corresponding trees. In practice, sorting in the decreasing order of the *widths* (rather than heights) is more beneficial because it improves the degree of coalesced access to frequently-accessed arrays such as Qs and Cs, especially for the version that pads at block/warp level. We report that all pre-processing (inspector) overheads sum up to under 2% of total parallel runtime (on GPU).

5 GPU-FLAT Flattening Nested Parallelism

This section demonstrates how GPU-FLAT, which utilizes both levels of parallelism, was derived from the nested-parallel code of Listing 2, in a way that simultaneously optimizes (i) both levels of divergence and (ii) temporal locality. We keep the discussion here intuitive and specific to the pricing algorithm, but we have sketched a re-write rule based formalization at http://hiperfit.dk/pdf/flat_rwr_hull_white.pdf.

The idea is to first sort the instruments in decreasing order of their heights—thus optimizing the divergence of the time series loops—and then to (bin-)pack them into bins, such that the summed widths of their trees does not exceed the CUDA block size (chosen B=1024) which is thought as the capacity of the bin. The two parallel levels—of the instruments in a bin, and of the inner parallelism inside valuate function—are then flattened and mapped to the CUDA-block level, while the parallelism across bins is mapped on the CUDA grid. On the one hand, this implicitly optimizes the width-level of thread divergence, because the flattened parallelism has roughly the size of the CUDA block (B). On the other hand, temporal locality is also improved because most intermediate data, such as the arrays Qs and Cs (but **not** αs [4]), are maintained in fast scratchpad (shared) memory.

Listing 3 shows the ***flattened code***: the bin array corresponds to a batch of q instruments—whose summed widths is less than B—and the result array contains their expected prices at current time. The flat code is obtained by distributing the (outer) map operation (applied on the bin) around each let statement of the valuate function shown in Listing 2. In essence, distributing the map (i) across a scalar statement results in a map of size q, and (ii) across a parallel operation (necessarily of size width) results in a parallel operation of size $\Sigma_{k=0}^{q-1}\texttt{width}_k$, which is padded to B. For brevity and clarity, the discussion ignores the complications related to padding parallel arrays and to offsets corresponding to the current block; these are tedious but straightforward to add.

Listing 3 starts by computing the widths and heights of the trees of the q instruments (line 2)—in practice these are precomputed by an inspector. For demonstration, we assume that q=2, and the widths and heights are ws=[2,4] and hs=[4,3]. Lines 3-12 compute four helper arrays ($\texttt{B}_w$, flag, $\texttt{out}_{inds}$ and $\texttt{inn}_{inds}$) that are fundamental to the code transformation. Array flag is the flag component in the representation of an irregular array of shape ws, such as Qss. With our example, we expect flag=[1,0,1,0,0,0], i.e., we have two segments of lengths 2 and 4, where a one records the start point of a segment. We first compute the start offset of each segment in array $\texttt{B}_w$=[0,2] by exclusively scanning ws. Then we compute the total number of elements $\texttt{len}_{flat}$=2+4=6, and finally, the scatter operation at line 6 writes ones at the indices in $\texttt{B}_w$=[0,2] into an array of $\texttt{len}_{flat}$=6 zeroes; hence flag=[1,0,1,0,0,0].

Array $\texttt{out}_{inds}$ is intended to record, for each of the width entries associated with an instrument, the index of that instrument in the current bin. As such, we expect $\texttt{out}_{inds}$ = [0,0,1,1,1,1]. This is achieved by inclusive scanning the flag array, which results in [1,1,2,2,2,2], and by subtracting 1 from each element (line 8-9).

Helper array $\texttt{inn}_{inds}$ is the expansion of iota w across the q widths, hence we expect $\texttt{inn}_{inds}$ = [0,1,0,1,2,3]. This is achieved at lines 11-12 by negating the flag array, resulting in [0,1,0,1,1,1], and by applying a segmented scan on the result, i.e., scanning independently the two logical rows of two and four elements, respectively.

Lines 13-18 in Listing 3 are the flattening across q instruments of lines 3-4 in Listing 2—which initializes Qs elements to zeroes and sets index w/2+1 to value f2(ins). The zero-initialization of Qs is translated to a replicate of length $\texttt{len}_{flat}$. The update at index w/2+1 is translated to a scatter on the expanded Qss in which (i) the updated indices are computed at line 16 by summing the offset in Qss of each

[4] The size of αs is not proportional with the Cuda block size, and thus it is not guaranteed to fit in shared memory. As before, αs is padded and transposed at block level to optimize coalescing and global-memory footprint.

```
1   let valuate^bin(q: int, bin: [q]Instrument) : [q]real =
2     let (ws,hs) = map f1 bin
3     let B_w  = scan^exc (+) 0 ws
4     let len_flat= B_w[q-1] + ws[q-1]
5     let tmp = map2 (λs b→ if s==0 then -1 else b) ws B_w
6     let flag= scatter (replicate len_flat 0)
7                       tmp (replicate q 1)
8     let tmp = scan^inc (+) 0 flag
9     let out_inds = map (λ x → x - 1) tmp
10    -- map (λ w → iota w) ws
11    let tmp = map (λ f → 1 - f) flag
12    let inn_inds = sgmscan^inc (+) 0 flag tmp
13    -- map(λw→ replicate w 0) ws
14    let Qss = replicate len_flat 0.0
15    -- map2 (λ Qs w → Qs[w/2+1] = f2 ins) Qss ws
16    let tmp_inds = map2 (λ b w → b + w/2 + 1) B_w ws
17    let tmp_vals = map f2 bin
18    let Qss = scatter Qss tmp_inds tmp_vals
19    -- init regular (transposed) h_max×q matrix αss^T
20    let h_max = reduce max 0 hs
21    let α_is = map f3 bin
22    let αss^T = scatter (replicate (h_max*q) 0.0)
23                        (iota q) α_is
24    -- map-loop interchange;loop count padded to h_max-1
25    let(_,_, αss^T) = loop(Qss, α_is, αss^T)
26      for i < h_max-1 do
27        -- map2 (λis α_i→map (...) is) inn_inds α_is
28        let Qss' =
29          map2(λ j oi → let (b,h) = (B_w[oi],hs[oi]) in
30                 if i ≥ h-1 then Qss[b+j] else
31                 let q0= Qss[b+j]
32                 let q1= if j > 0 then Qss[b+j-1] else 1.
33                 let q2= if j<w-1 then Qss[b+j+1] else 1.
34                 in  g1( i, j, α_is[oi], q0, q1, q2)
35              ) inn_inds out_inds
36        -- map (reduce (+) 0) Qss'
37        let scQs = sgmscan^inc (+) 0.0 flag Qss'
38        let α_vs = map2 (λ b w→ scQs[b+w-1]) B_w ws
39        -- map(λα→α[i+1]=g2(..)) αss
40        let tmp_is = map2 (λ h k → if i≥h-1 then -1
41                                   else (i+1)*q+k
42                          ) hs (iota q)
43        let α_is' = map2 g2 α_vs bin
44        let αss^T = scatter αss^T tmp_is α_is'
45        in (Qss', α_is', αss^T)
46    let Css=replicate len_flat 100 ...--2nd loop not shown
```

Listing 3. Flat-Parallel Implementation.

instrument, denoted by $b \in B_w$, with w/2+1, where $w \in$ws, and (ii) the updated values are the result of mapping f2 on the bin at line 17.

The initialization of αss—the expansion of αs—is simply obtained by padding each row to the maximal height h_{max} of the q instruments—hence total length is $q \times h_{max}$—and by using a scatter to overwrite the first entry in every row with the result of f3. This is implemented in lines 20-23, except that we use αss^T, the transpose of αss, to achieve coalesced access to global memory. Next, the forward loop is padded to count h_{max}-1, the outer map of length q is interchanged inside the loop (always safe), and the outer-map distribution continues on the loop-body statements.

Lines 28-35 correspond to flattening the map that is applied to iota w to compute array Qs' in Listing 2, lines 11-16.

Since the flattened code corresponds to applying the original map simultaneously to all entries of all q instruments, it was translated to a map2 over inn_{inds} and out_{inds}:

- inn_{inds} is the expansion of (iota w) across the q instruments, hence j takes the same values as in Listing 2;
- out_{inds} is used to access values that are the same across the width threads assigned to the current instrument, but are needed by each thread. For example, out_{inds} is used to indirectly index into length-q arrays B_w, hs and α_is in order to compute the start offset b into array Qss, the height h and the α value corresponding to the current instrument, respectively.
- The body of the mapped function is protected by an if condition ($i \geq h-1$) that checks that the tree traversal has not already terminated for the current instrument, which is possible since the count of the transformed loop count was padded to maximal value h_{max}.

The code between lines 37-38 is the flattening across all q instruments of the (original) reduce at line 17 in Listing 2, which sums up the values of array Qs'. First, an inclusive segmented scan is performed on the expanded array Qss', which inherently computes the q corresponding sums in the last entry of each logical subarray of the result scQs. The last entries are then extracted by a map operation of length q: the index of the last entry of the i^{th} subarray is B_w[i]+ws[i]-1, because B_w and ws record the offset and size of each subarray.

Finally, lines 39-44 implement the expansion to update αs[i+1] at line 19 in Listing 2. This is translated to a scatter that updates αss^T at the q flat-indices from row i+1 stored in tmp_is[5] with the values α_is' obtained by applying g2 to all α_vs and corresponding instruments in the bin.

Similar ideas apply for the translation of the backward loop, which is not shown. Our CUDA implementation of GPU-FLAT aggressively fuses the inner-parallel operations and reuses shared memory buffers whenever possible: e.g., Qss, Qss', Css, Css' use the same buffer. Arrays of size q are typically stored in shared memory, while arrays out_{inds} and inn_{inds} in registers. The shortcomings of GPU-FLAT are that (i) it introduces instructional overhead (more complex code) and significant register pressure[6] and (ii) the parallel operations of size q underutilize the block-level parallelism, which is typically significantly larger than q.

6 Experimental Evaluation

Hardware. We present results on two Linux systems:

D1: (**CPU1**) 2×26-core 2-way HT Intel Xeon Platinum 8167M 2.00GHz, 754 GB RAM and NVIDIA **V100** GPU (2560 Volta *FP64* cores) [26], on CUDA 10.1.

[5] If $i \geq h-1$ then the update is omitted (returns -1)—outside original loop.
[6] Nvidia nvcc compiler reports that 74 – 76 registers are used by default per thread and a speedup of up to 1.66× is achieved by limiting it to 32.

D2: (**CPU2**) 2×8-core 2-way HT Intel Xeon E5-2650 v2 2.60GHz , 128 GB RAM and NVIDIA **GTX 780Ti** GPU (2880 Kepler *FP32* Cores), on CUDA 10.1.

The **V100** GPU of **D1** is optimized for double-precision floats (*FP64*), while the **GTX 780Ti** of **D2** is not. We run experiments with the real type instantiated to *FP64* on the **V100** and with *FP32* on the **GTX 780Ti**. Our goal is neither to compare consumer vs. enterprise hardware, nor *FP64* vs. *FP32* performance, but rather to highlight an interesting performance portability issue across different GPU hardware generations, i.e., on some datasets GPU-FLAT is faster than GPU-OUTER on D1 but slower on D2.[7]

Experimental Methodology. We measure total application runtime, excluding host-device memory transfers[8], but including allocations, kernel execution, and all preprocessing steps. Execution times are averaged across 10 runs (standard deviation $< 1.5\%$) and are reported in $GFLOP^{\mathsf{SPEC}}/s$, which counts the number of floating-point operations as they appear in the high level specification (GPU-OUTER).[9]

Datasets. The evaluation uses 7 synthetic datasets that (i) simulate a mix of Bermudan call/put options on top of a bond and (ii) model the instrument distributions in real portfolios used in practice,[10] and (iii) also demonstrate different workload divergence properties and impact of optimizations.

All datasets consist of 100000 valuations, except for **U1**, which uses 3000 and is intended to measure the impact of under-utilizing hardware parallelism by GPU-OUTER. **U1** and **U2** use constant width 259 and height 606 for all trees (no divergence). They model the case of pricing a single instrument against many different market scenarios, where one class of the risk factor parameters varies, e.g. yield curves.

R1, **R2**, and **R3** (**R***) use random distributions of widths and heights in intervals $[7, 511]$ and $[13, 1200]$, respectively. **R1** uses uniform distribution for both, **R2** uses uniform for widths and standard-normal distribution for heights, while **R3** does the opposite. **R*** model an instrument distribution typical for an interest-rate derivative portfolio. The results indicate that such datasets have similar performance.

Finally, **S1** and **S2** (**S***) present skewed distributions. In **S1** 1% of the dataset consists of widths and heights in the interval $[461 - 511]$ and $[1082 - 1200]$, respectively, while the rest has them uniformly distributed in $[7 - 57]$ and $[12 - 131]$. **S2** uses the same figures, but separates skewness over dimensions: 1% combines skewed widths with uniform heights and another 1% the reverse. **S*** model a case often met in practice, where a small set of bonds have much longer maturities or much smaller volatilities than the remaining majority. The maturity of the bond in years can be found by dividing the height with 12. We use the maximal maturity to be 100 years (height=1200), which is used in practice by some pension funds. Regardless, more accurate simulations require a time step smaller than a month, which would increase the height, thus the evaluated scenarios are realistic.

6.1 Performance Results

To start with, we have implemented a multi-threaded version using OpenMP and QuantLib library [2], in which different instruments are priced in parallel. We obtain three-to-four orders of magnitude speedups with both our CPU and GPU implementations, hence we do not discuss it any further.

GPU optimizations. Table 1 and Figure 3 report the performance measured in $GFLOP^{\mathsf{SPEC}}/s$ on machines **D1** and **D2** across 5 datasets, (**R***) and (**S***). GPU-OUTER ($_o$) and GPU-FLAT ($_f$) use *FP64* (^{64}) on **D1** and *FP32* (^{32}) on **D2**. Version V_1 corresponds to a version that does *not* optimize for coalesced global memory accesses, while the remaining versions achieve coalescing by padding and transposing at global (V_2), block (V_3) or warp level (V_4), respectively. Superscripts ^{ns} and ^{ws} correspond respectively to versions without and with data reordering through sorting (in descending order). For GPU-OUTER the trees are sorted by width first, while for GPU-FLAT by height first. Column M^{64} reports the memory footprint in *GB* for *FP64* versions. *FP32* versions use half of the M^{64}. The uniform datasets **U*** are not reported in Table 1, because, due to their constant width/height size, the impact of sorting and block/warp level padding is insignificant.

Important observations for GPU-OUTER are:

- The impact of memory coalescing optimization varies across **U*** and devices. On **D1** it results in a minor 1.1× speedup on **U1**, but a large 20× speedup on **U2**. On **D2** it results in 2.7× for **U1** and 14.1× for **U2**.
- Data reordering and coalescing by padding, have small or even *negative* impact when applied in isolation, because they address independent sources of overhead which "hide" each other. For example, on **D1** and **R1**, the unoptimized version V_{o1}^{ns} is actually 1.5× faster than V_{o1}^{ws}. However, when combined they show high impact: 4.6-5.96× on **R*** and 1.05-2.38× on **S*** datasets.
- Warp-level padding V_{o4}^{ws} achieves coalescing at the cost of a modest 3% increase in memory footprint (M^{64}) w.r.t. V_{o1}^{ns}. Moreover, it is the fastest version on all datasets on both **D1** and **D2**. In comparison, V_{o2}^{ws} increases M^{64} by 1.9× and 12.3× on **R*** and **S*** datasets.

Important observations for GPU-FLAT are:

[7] Unfortunately, we did not have access to a K40 or K80 (older) GPU optimized for *FP64*. We do not discuss the accuracy of *FP32*, because industry would not use it regardless.

[8] The memory-transfer time is less than 1% of total runtime in all tests.

[9] We find $GFLOP^{\mathsf{SPEC}}/s$ better suited to compare across different implementations of the same algorithm, because it represents normalized runtime.

[10] The datasets were suggested by the experts of a leading company developing financial software used by large investment funds. The presented pricing method is used in $\sim$ 70% of clients' use cases. The datasets are intended to have characteristics representative of clients' use cases and also to cover stress cases. Unfortunately, such companies do not share real-world data due to privacy and competitive advantage concerns.

Table 1. GPU-OUTER and GPU-FLAT performance P in *GFLOP*$^{\text{SPEC}}$*/s* and global-memory footprint M^{64} for *FP64* in *GB*. Speedup and memory savings ($\Delta\times$) are ratios between unoptimized (V^{ns}_{o1}/V^{ns}_{f1}) and most optimized (V^{ws}_{o4}/V^{ws}_{f3}) code.

	R1			**R2**			**R3**			**S1**			**S2**		
GPU-OUTER	P^{64}_{D1}	P^{32}_{D2}	M^{64}	P^{64}_{D1}	P^{32}_{D2}	M^{64}	P^{64}_{D1}	P^{32}_{D2}	M^{64}	P^{64}_{D1}	P^{32}_{D2}	M^{64}	P^{64}_{D1}	P^{32}_{D2}	M^{64}
V^{ns}_{o1}	103	29	3.42	84	27	3.42	85	26	3.42	56	13	0.52	92	32	0.52
V^{ws}_{o1}	69	24	3.42	61	23	3.42	60	21	3.42	54	12	0.52	77	24	0.52
V^{ns}_{o2}	97	33	6.55	103	35	6.55	114	44	6.55	48	21	6.40	101	41	6.40
V^{ws}_{o2}	423	114	6.55	427	120	6.55	427	136	6.55	57	23	6.40	189	52	6.40
V^{ns}_{o3}	103	45	6.47	109	49	6.28	120	53	5.98	49	23	0.82	111	52	0.82
V^{ws}_{o3}	461	154	3.62	454	157	3.58	461	154	3.60	58	24	0.56	210	75	0.55
V^{ns}_{o4}	108	45	6.38	114	49	6.14	126	53	5.77	49	23	0.82	115	53	0.82
V^{ws}_{o4}	**474**	**155**	3.53	**469**	**158**	3.51	**477**	**155**	3.52	**59**	**25**	0.54	**211**	**76**	0.53
$\Delta\times$	4.60	5.34	1.03	5.58	5.85	1.03	5.61	5.96	1.03	1.05	1.92	1.04	2.29	2.38	1.02
GPU-FLAT	P^{64}_{D1}	P^{32}_{D2}	M^{64}	P^{64}_{D1}	P^{32}_{D2}	M^{64}	P^{64}_{D1}	P^{32}_{D2}	M^{64}	P^{64}_{D1}	P^{32}_{D2}	M^{64}	P^{64}_{D1}	P^{32}_{D2}	M^{64}
V^{ns}_{f1}	521	69	1.98	529	70	1.98	496	66	1.98	266	37	1.83	118	17	1.83
V^{ws}_{f1}	553	73	1.98	554	73	1.98	525	72	1.98	268	33	1.83	119	15	1.83
V^{ns}_{f2}	517	69	1.98	526	70	1.98	492	66	1.98	266	36	1.83	118	16	1.83
V^{ws}_{f2}	550	73	1.98	552	73	1.98	524	71	1.98	268	33	1.83	119	15	1.83
V^{ns}_{f3}	740	87	1.30	779	91	1.21	710	84	1.26	783	97	0.24	577	76	0.24
V^{ws}_{f3}	**859**	**100**	1.09	**859**	**100**	1.10	**818**	**99**	1.09	**904**	**97**	0.17	**686**	**93**	0.17
$\Delta\times$	1.65	1.45	0.55	1.62	1.43	0.56	1.65	1.50	0.55	3.40	2.62	0.09	5.81	5.47	0.09

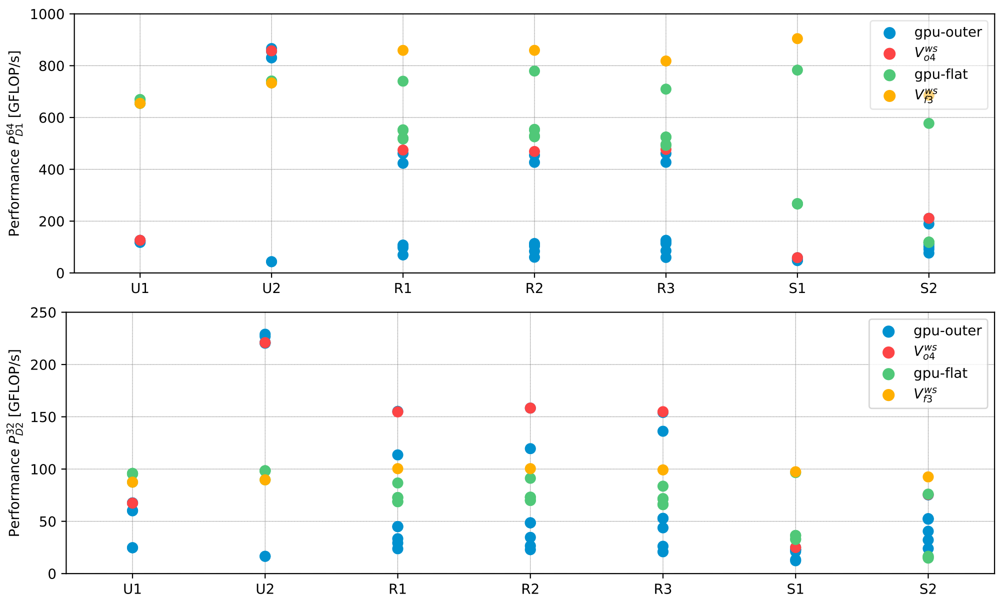

Figure 3. *GFLOP*$^{\text{SPEC}}$*/s FP64* performance on **D1** (upper) and *FP32* on **D2** (lower) for all datasets. V^{ws}_{o4}/V^{ws}_{f3} are most optimized.

- The impact of optimizations on **R*** is smaller than GPU-OUTER: V^{ws}_{f3} is about 1.6× faster and uses 55% of the memory of the unoptimized version (V^{ns}_{f1}).
- The impact of reordering is positive but smallish, e.g., sorting alone produces a speedup as high as 1.2×.
- The impact of optimizations is higher on **S***: V^{ws}_{f3} reduces memory footprint by 11× (w.r.t. V^{ns}_{f1}), and on **S2**, reordering and block-level coalescing (V^{ws}_{f3}) result in speedup higher than 5.5× on **D1** and **D2**.

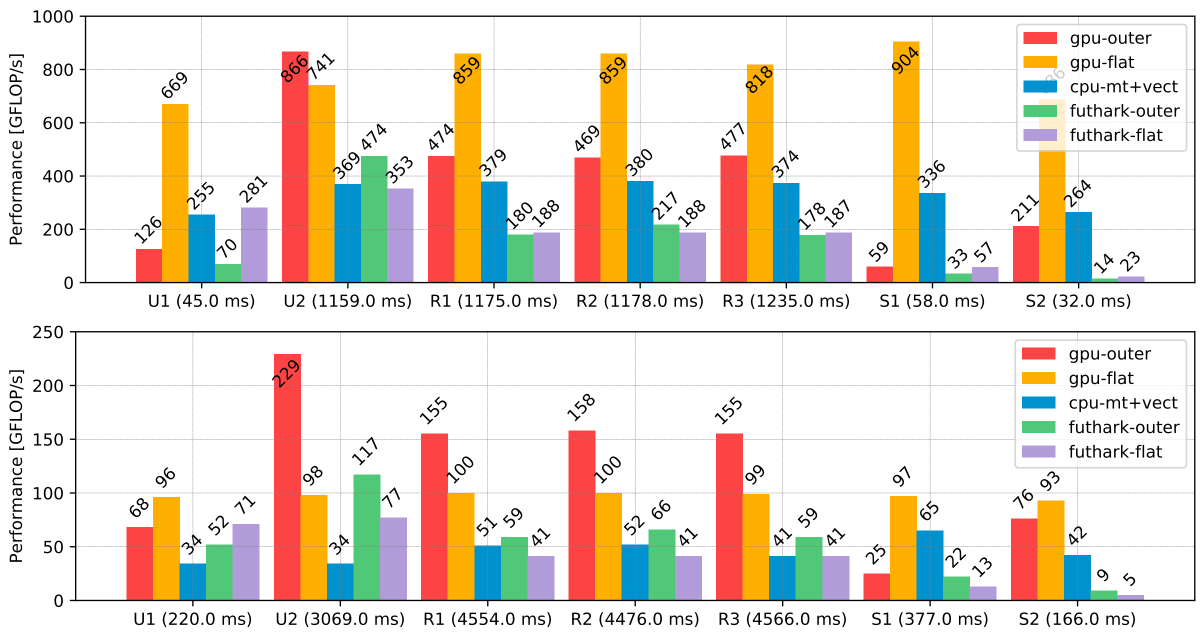

Figure 4. *GFLOP*$^{\text{SPEC}}$*/s* performance for *FP64* on **D1** (upper) and *FP32* on **D2** (lower) for all datasets. Runtimes are shown in *ms*.

GPU-FLAT vs. GPU-OUTER. Figure 4 compares the performance of the best versions of GPU-FLAT (V_{f3}^{ws}) and GPU-OUTER (V_{o4}^{ws}):

- The performance of GPU-FLAT is relatively stable across datasets (669 – 904 *GFLOP*$^{\text{SPEC}}$*/s* on **D1**) while the performance of GPU-OUTER is highly variant.
- On the **R*** datasets, GPU-FLAT is about 1.8× faster than GPU-OUTER on the newer hardware **D1**, but 1.55× slower on **D2**.
- GPU-FLAT is significantly faster than GPU-OUTER (i) on the skewed datasets **S*** because it better optimizes divergence and locality of reference by the use of shared memory, and (ii) on the small dataset **U1** because the outer parallelism of only 3000 valuations is insufficient to utilize the hardware well.
- Newer hardware benefits GPU-FLAT, which outperforms GPU-OUTER on **D1** on all datasets, except for **U2** where it is only 1.17× slower.[11] On **S1**, GPU-FLAT is 15.3× faster on **D1**, but only 3.9× on **D2**.

GPGPU vs. CPU. Figure 4 also compares the best GPU-OUTER and GPU-FLAT configurations with our multi-core implementation using OpenMP multi-threading and AVX2 vectorization, named CPU-MT+VECT. Even though we use powerful CPUs with 104 (**D1**) and 32 (**D2**) hardware threads, the best GPU version is faster than CPU-MT+VECT, with speedups as high as 6.7× on **U2** and 3.8× on **R3**, and on average 2.4× on **D1** and 3.3× on **D2**. Notably, GPU-FLAT is in all cases faster than CPU-MT+VECT, which seems to indicate that GPU hardware is the cost-effective solution for this application.

CUDA vs. Futhark. To demonstrate that current compiler technology does not effectively support the code transformations presented in this work, we compare the performance of GPU-OUTER and GPU-FLAT with a high-level implementation written in the data-parallel language *Futhark* [19]. The results in Figure 4 show that the best CUDA version is faster than Futhark by a factor as high as 29.8× and on average 6.3× on the considered datasets.

7 Related Work and Conclusions

Compiler Techniques. GPU-FLAT builds on Blelloch's seminal work on the flattening code transformation [6, 8] which maps irregular nested parallelism into a sequence of flat-parallel ones, in a manner that preserves the work-depth asymptotic of the source program [7], and which has also been implemented for GPU execution [3]. Our implementation exhibits two key differences: The first one is that flattening pushes all sequential recurrences outside the parallel code, and it introduces many prefix-sum operations that are executed in global memory and thus limit performance gains. Instead, we bin-pack inner parallelism at block level so that it can be efficiently executed in shared memory. The second difference is more subtle and refers to the fact that the original transformation replicates variables that are not bounded (free) in the inner parallel constructs. This may increase the memory footprint, and thus it might prevent the use of shared memory, which is a scarce resource. In comparison, we do not expand such variables, but instead we indirectly access them through auxiliary arrays, such

[11] Using shared memory has higher impact on newer GPU generations because the per-core shared-memory size and bandwidth has grown faster than the global-memory bandwidth.

as $\texttt{out}_{inds}$, $\texttt{inn}_{inds}$, and $\texttt{B}_{w}$ in Listing 3. The latter can be reused between arrays of equal shapes, such as Qss and Css.

GPU-FLAT demonstrates that the common-wisdom strategy of always sequentializing the parallelism in excess is suboptimal, because utilizing inner parallelism may allow to better exploit temporal locality. A similar strategy (and observation) has been used for accelerating a remote-sensing algorithm aimed at detecting landscape changes from satellite data applied at continental scale [16].

Finally, our techniques for optimizing the two-level divergence were inspired by data and iteration reordering transformations aimed at improving locality and communication patterns [12, 28, 30, 33], and by inspector-executor restructuring transformations [29, 31, 34]. However, we are not aware of any compiler framework able to derive the GPU-FLAT version or the optimizations used for GPU-OUTER.

Accelerating Financial Algorithms. In practice, the trinomial tree numerical method is the standard choice for solving the *HW1F* model. It is especially suited for pricing low-dimensional bond instruments, that we focus on in this work, which depend on 1 or 2 underlyings. Its main advantage is its simple deterministic execution path, that enables valuation tractability. In comparison, *Monte Carlo Simulations (MC)* are more general, but also more expensive computationally, and the introduced randomness distorts the understanding of the pricing. We were unable to find work studying GPU acceleration of this exact problem, thus we compare with work aimed at comparable problems, and focus on the main performance inhibitor: the divergence introduced by heterogeneous trees.

Grauer-Gray et al. (2013) [17] adapt QuantLib implementation of Bond and Repo pricing through iterative bootstrapping and enable it for GPUs. Although none of the experiments employs a trinomial tree, the authors report an experiment on a diversified bond portfolio, where they parallelize the computation on the outer level, similarly to GPU-OUTER. They identify, but they do not address the issue of thread divergence. Schabauer et al. (2008) [32] also present an outer-level parallelization scheme for pricing path-dependent interest rate products on bounded trinomial lattices that resembles GPU-OUTER. They use Fortran and MPI for distributed computing across as many as 16 nodes with single-core CPUs and report speedups of up to 13×. However, the evaluation uses a *homogeneous set of trees* (equal dimensions), which does not exhibit divergence, and inner parallelism is not utilized due to high communication costs. Gerbessiotis (2003) [14] describes a distributed implementation of a trinomial tree method for pricing vanilla equity options. He uses MPI to achieve about 16× speedup on 16 dual-core CPUs. The outer level parallelism does not exist, because he addresses the problem of a *single* large trinomial tree computation that has 32768 or 65536 time steps.

GPU parallelisation of a simpler *binomial lattice method* is better represented in the literature: Gui et al. (2013) [18] parallelize the binomial tree using CUDA to price a standard vanilla equity option. The problem is restricted to the *homogenous* case as tree dimensions are *the same* across options, thus no divergence occurs. The approach is to exploit inner parallelism by pricing each option on one thread-block, which is not applicable to variant widths, and is less challenging than the flattening of irregular parallelism used for GPU-FLAT. Suo et al. (2015) [35] use GPUs (via CUDA/OpenCL) to implement a binomial tree, and compare it with a *MC* to price *a single* vanilla equity option. Their optimizations are aimed at unbounded trees, and are not suited for portfolios in which tree dimensions vary. Gerbessiotis (2004) [15] studies the acceleration of a binomial tree for pricing an option using a distributed setup, but they similarly consider a homogeneous portfolio of instruments (no divergence). Zhang et al. (2012) [37] present a hybrid implementation that constructs and traverses a binomial tree on CPU and GPU simultaneously to price a *single* American equity option. Similarly, their technique however does not naturally extend to heterogenous portfolios. Huang and Thulasiram (2005) [20] develop parallel algorithms for pricing path-dependent American exotic options with up to 10 underlyings using the binomial tree. They use multi-core CPUs linked with MPI, and consider the performance changes due to different number of time steps for a variable number of assets. However, the considered options have trees of the same dimensions.

Other work studies different numerical methods to solve the Hull-White pricing model. Theiakos et al. (2015) [36] target GPUs, but price a mortgage contract with one underlying by using a *Finite Difference Method (FDM)* to solve the problem. Dang et al. (2014) [11] use GPUs to price cross-currency interest rate derivatives with many underlyings and path-dependent features. They use *FDM* to solve the high-dimensional system of PDEs. Both approaches assume homogeneous portfolios, hence no divergence. Bernemann et al. (2010) [4] use GPUs, but use *Monte Carlo Simulations (MC)* and more sophisticated Heston Hull-White model with local volatility to price structured equity instruments. *MC* is the only method that can be used to solve this extended model version. Finally, a large body of work was dedicated to GPU acceleration of *MC* used for *pricing derivatives* [25, 27], model *calibration* [1] or *risk management* [13].

Generality. The techniques reported in this paper can be applied to some of the surveyed work. For example, GPU-FLAT can be used to exploit irregular parallelism on both GPU [17, 18] and in a distributed (MPI) setting [15, 32], e.g., by bin-packing valuations across nodes and by mapping inner-parallelism at node level. Similarly, finite-difference methods typically use a *bounded* grid in both spatial and temporal dimensions. Hence our optimisations are likely applicable to solving bulks of PDEs, which are heavily used not only in finance [32, 36], but also in fields such as fluid dynamics, mechanics, acoustics, weather prediction.

Acknowledgments

This research has been partially supported by the Independent Research Fund Denmark grant under the research project *FUTHARK: Functional Technology for High-performance Architectures* and by Innovation Fund Denmark under the research project ref.no. 5189-00224B.

References

[1] Christian Andreetta, Vivien Bégot, Jost Berthold, Martin Elsman, Fritz Henglein, Troels Henriksen, Maj-Britt Nordfang, and Cosmin E. Oancea. 2016. FinPar: A Parallel Financial Benchmark. *ACM Trans. Archit. Code Optim.* 13, 2, Article 18 (2016).

[2] Luigi Ballabio. 2020. QuantLib, A free/open-source library for quantitative finance. https://www.quantlib.org/.

[3] Lars Bergstrom and John Reppy. 2012. Nested Data-parallelism on the GPU. *ICFP '12: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming* 47, 9 (2012), 247–258.

[4] André Bernemann, Ralph Schreyer, and Klaus Spanderen. 2010. Pricing structured equity products on GPUs. In *2010 IEEE Workshop on High Performance Computational Finance*. IEEE, Washington, DC, USA, 1–7.

[5] Guy E. Blelloch. 1989. Scans as Primitive Parallel Operations. *Computers, IEEE Transactions* 38, 11 (1989), 1526–1538.

[6] Guy E Blelloch. 1990. *Vector models for data-parallel computing*. Vol. 75. MIT press Cambridge.

[7] Guy E. Blelloch and John Greiner. 1996. A Provable Time and Space Efficient Implementation of NESL. In *Procs. of ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*. ACM, 213–225.

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

[9] Phelim P. Boyle. 1986. Option Valuation Using a Three-Jump Process. *International Options Journal* 3 (1986), 7–12.

[10] Damiano Brigo and Fabio Mercurio. 2006. *Interest Rate Models - Theory and Practice*. Springer, Berlin, Heidelberg.

[11] Duy Minh Dang, Christina C. Christara, and Kenneth R. Jackson. 2014. Graphics processing unit pricing of exotic cross-currency interest rate derivatives with a foreign exchange volatility skew model. *Concurrency and Computation: Practice and Experience* 26, 9 (2014), 1609–1625.

[12] Chen Ding and Ken Kennedy. 1999. Improving Cache Performance in Dynamic Applications Through Data and Computation Reorganization at Run Time. In *Procs. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99)*. ACM, 229–241.

[13] Matthew Dixon, Jike Chong, and Kurt Keutzer. 2009. Acceleration of Market Value-at-risk Estimation. In *Procs. of Workshop on High Performance Computational Finance (WHPCF '09)*. ACM, 5:1–5:8.

[14] Alexandros V. Gerbessiotis. 2003. Trinomial-tree Based Parallel Option Price Valuations. *Parrallel Algorithms and Applications* 18, 4 (2003).

[15] Alexandros V. Gerbessiotis. 2004. Architecture independent parallel binomial tree option price valuations. *Parallel Comput.* 30, 2 (2004).

[16] Fabian Gieseke, Sabina Rosca, Troels Henriksen, Jan Verbesselt, and Cosmin E. Oancea. 2020. Massively-Parallel Change Detection for Satellite Time Series Data with Missing Values. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 385–396.

[17] Scott Grauer-Gray, William Killian, Robert Searles, and John Cavazos. 2013. Accelerating financial applications on the GPU. In *Procs. of GPGPU-6*. ACM, New York, NY, USA, 127–136.

[18] Yechen Gui, Shenzhong Feng, Gaojin Wen, Guijuan Zhang, Yanyi Wan, and Tao Liu. 2013. High Performance Implementation of Binomial Option Pricing Using CUDA. In *GPU Solutions to Multi-scale Problems in Science and Engineering. Lecture Notes in Earth System Sciences*. Springer, Berlin, Heidelberg, 201–214.

[19] Troels Henriksen, Frederik Thorøe, Martin Elsman, and Cosmin E. Oancea. 2019. Incremental Flattening for Nested Data Parallelism. In *Procs. Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. ACM, New York, NY, USA, 53–67.

[20] Kai Huang and Ruppa K. Thulasiram. 2005. Parallel algorithm for pricing American Asian options with multi-dimensional assets. In *19th International Symposium on High Performance Computing Systems and Applications (HPCS'05)*. IEEE, Washington, DC, USA, 177–185.

[21] John Hull. 2017. *Options, Futures, and Other Derivatives* (10th ed.). Pearson Education, New York, NY, USA.

[22] John Hull and Alan White. 1990. Valuing Derivative Securities Using the Explicit Finite Difference Method. *The Journal of Financial and Quantitative Analysis* 25, 1 (1990), 87–100.

[23] John Hull and Alan White. 1994. Numerical Procedures for Implementing Term Structure Models I: Single-Factor Models. *The Journal of Derivatives* 2, 1 (1994), 7–16.

[24] John Hull and Alan White. 1996. Using Hull-White Interest Rate Trees. *The Journal of Derivatives* 3, 3 (1996), 26–36.

[25] Fredrik Nord and Erwin Laure. 2011. Monte Carlo Option Pricing with Graphics Processing Units. In *Advances in Parallel Computing*. IOS Press, Amsterdam, The Netherlands, 143–150.

[26] NVIDIA. 2017. *NVIDIA Tesla V100 GPU Architecture*. Technical Report. NVIDIA Corporation. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[27] Cosmin E. Oancea, Christian Andreetta, Jost Berthold, Alain Frisch, and Fritz Henglein. 2012. Financial Software on GPUs: Between Haskell and Fortran. In *Procs. of Workshop on Functional High-Performance Computing (FHPC '12)*. ACM, 61–72. https://doi.org/10.1145/2364474.2364484

[28] Cosmin E. Oancea and Alan Mycroft. 2008. Set-Congruence Dynamic Analysis for Thread-Level Speculation (TLS). In *Languages and Compilers for Parallel Computing*, José Nelson Amaral (Ed.). Springer, Berlin, Heidelberg, 156–171.

[29] Cosmin E. Oancea and Lawrence Rauchwerger. 2013. A Hybrid Approach to Proving Memory Reference Monotonicity. In *Languages and Compilers for Parallel Computing (LCPC'11) (LNCS, Vol. 7146)*. Springer, Berlin, Heidelberg, 61–75.

[30] Cosmin E. Oancea, Jason W. A. Selby, Mark Giesbrecht, and Stephen M. Watt. 2005. Distributed Models of Thread-Level Speculation. In *Procs. of International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'05)*. 920–927.

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

[32] Hannes Schabauer, Ronald Hochreiter, and Georg Ch Pflug. 2008. Parallelization of Pricing Path-Dependent Financial Instruments on Bounded Trinomial Lattices. In *Computational Science - ICCS*. Springer, Berlin, Heidelberg, 408–415.

[33] Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. 2003. Compile-time Composition of Run-time Data and Iteration Reorderings. In *Procs. Conference on Programming Language Design and Implementation (PLDI '03)*. ACM, 91–102.

[34] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. 2018. The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code. *Proc. IEEE* 106, 11 (2018), 1921–1934.

[35] Simon Suo, Ruiming Zhu, Ryan Attridge, and Justin Wan. 2015. GPU option pricing. In *WHPCF '15: Proceedings of the 8th Workshop on High Performance Computational Finance*. ACM, New York, NY, USA, 1–6.

[36] Alexios Theiakos, Jurgen Tas, Han van der Lem, and Drona Kandhai. 2015. *Ultra-Fast Scenario Analysis of Mortgage Prepayment Risk*. Technical Report. SSRN, Rochester, NY.

[37] Nan Zhang, Chi-Un Lei, and Ka Lok Man. 2012. Binomial American Option Pricing on CPU-GPU Hetergenous System. *Engineering Letters* 20, 3 (2012), 279–285.