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

WOJCIECH MICHAL PAWLAK, Department of Computer Science, University of Copenhagen, Denmark and SimCorp Technology Labs, SimCorp A/S
COSMIN EUGEN OANCEA, Department of Computer Science, University of Copenhagen, Denmark

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 2.9× faster than our well-tuned CPU-parallel implementation (OpenMP+AVX2), 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-performance simulations**; Parallel programming languages; • **Applied computing** → **Economics**; • **Computer systems organization** → *Multicore architectures.*

Additional Key Words and Phrases: High-Performance Computing, Parallel (GPU) Programming, Compilers, Computational Finance, Derivative Pricing

ACM Reference Format:
Wojciech Michal Pawlak and Cosmin Eugen Oancea. 2019. Acceleration of Lattice Models for Pricing Heterogeneous Portfolios of Fixed-Income Derivatives. *ACM Trans. Arch. Code Optim.* 37, 4, Article 111 (August 2019), 22 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Pricing is a fundamental computational component for the *risk management* of any investment portfolio. It applies mathematical modeling and compute-intensive algorithms to accurately approximate the value of any instrument that is actively traded in the financial markets. A derivative is an example of such an instrument: it is a contract that derives its value from other, more basic, instruments like fixed-income bonds. Modern portfolios, managed by large institutional investors, consist usually of a large number of heterogeneous contracts.[1] In recent years, financial markets have become highly regulated as a consequence of the Global Financial Crisis of 2007-2008. Market participants are now obliged to track their portfolio risks and report them accurately on a regular

[1] By heterogeneous we mean that the internal mechanics of contracts may differ—there are several *types* of contracts such as European, American, Bermudan—but more importantly, that the computational characteristics differ across contracts of the same type, for example because they use different underlying instruments, and are exercised/monitored at different dates.

Authors' addresses: Wojciech Michal Pawlak, wmp@di.ku.dk, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark, SimCorp Technology Labs, SimCorp A/S, Copenhagen, Denmark; Cosmin Eugen Oancea, cosmin.oancea@diku.dk, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark.

XXXX-XXXX/2019/8-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

and frequent basis. The standard risk analysis combines stress testing against historical data and against a simulation of possible future market scenarios. The intensive testing of large-scale, heterogeneous portfolios gives raise to a challenging big-compute problem and makes a compelling case for applying parallel programming on highly-parallel hardware, such as GPUs. In this sense, accelerated computation paves the way for real-time (on-demand) decision making, for a cost-efficient hardware infrastructure, and for improving the accuracy of financial modeling.

This paper reports on the GPU (and CPU) parallelization of a standard pricing algorithm, which is commonly used in practice, that models the value of an instrument by means of a system of differential equations. These equations represent the changes in the interest and are defined by the Hull-White One-Factor Short-Rate (*HW1F*) model [10]. The algorithm that we use to discretise and solve them is a *Trinomial Tree* lattice-based numerical method. The algorithm used for pricing *one instrument* has two main stages: The forward stage builds a tree of bounded width[2], representing the propagation of the interest rate until the maturity date of the underlying bond is reached, and the backward stage then performs the instrument valuation from maturity back to the current time. The computational structure thus consists of two sequential time-series loops, whose counts are the height of the tree, in which each iteration performs several semantically-parallel operations which have the same length—given by the bounded width of the tree.[3]

Practical scenarios require pricing a (large) portfolio of instruments, which can be in principle straightforwardly parallelized by mapping the pricing of each instrument to a different thread. All current approaches related to the acceleration of this pricing method []:

- have considered the simple case in which all trees that have the same width and height, and
- have utilized only the outer parallelism—i.e., of the instruments in the batch—while sequentializing the inner parallelism available in the valuation of one instrument.

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. This paper studies this difficult case, and reports on two parallelization strategies and subsequent optimizations.

The first strategy follows the conventional wisdom that says that outer levels of parallelism are more profitable to exploit than inner ones. As such, given a big enough portfolio, one should sequentialize the inner parallelism and perform one instrument valuation per thread. Section 4 presents such an implementation, named GPU-OUTER, together with a set of optimizations aimed at improving memory footprint and spatial locality (i.e., coalesced accesses to global memory).

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*; for example, by precomputing the widths (or heights) of the instrument' trees, and by sorting the portfolio in decreasing order of their widths (or heights).

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 thread-block level, which also allows to maintain most of the data in fast (shared+register) memory.

The idea is (i) to *pack instruments into bins*, such that their summed widths fit the size of the CUDA block (bin), and then (ii) to "*flatten*" (merge) the available two-level parallelism—i.e., one level corresponds to the instruments in a bin, and the second level corresponds to the width-length parallel operations that appear in the implementation of each instrument.

[2] The model uses a tree of bounded width, because in practice the interest rate tends to revert to the mean value over time.
[3] For example, in the forward step, each node in the next level of the tree is computed from three nodes belonging to the previous level of the tree.

The flattening step is highly nontrivial and is a key contribution of this work. We document it by presenting in Section 3 an initial—simple ad clear—nested-parallel specification that uses operators such as `map` and `reduce`. We then demonstrate in Section 5 how flattening is performed for the specific application that is the subject of this paper, resulting in a program named GPU-FLAT. For completeness and generality, Appendix A formalizes the proposed transformation by a set of inference (rewrite) rules that can be integrated in the repertoire of a data-parallel compiler.

A detailed experimental 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 4.0× and on average 3.15× and for GPU-FLAT as high as 8.4× and on average 2.63×.
- GPU performance is sensitive to dataset characteristics: GPU-OUTER is faster than GPU-FLAT by a factor as high as 2.7× and on average 1.59× on large portfolios with constant or randomly-distributed heights and widths, while GPU-FLAT is as high as 9.5× and on average 3.7× faster on small portfolios or when the distribution of heights and widths is skewed.
- The best GPU version is faster by a factor as high as 5.3× and on average 2.9× than our tuned multicore and vectorized CPU implementation (OpenMP+AVX2), named CPU-MT+VECT. At its turn, our CPU implementation is three-to-four orders of magnitude faster than an OpenMP-parallel implementation build on top of the popular QuantLib library [**?**].

Add citation for QuantLib, as an electronic resource and cite it above.

2 FINANCIAL BACKGROUND AND ALGORITHM

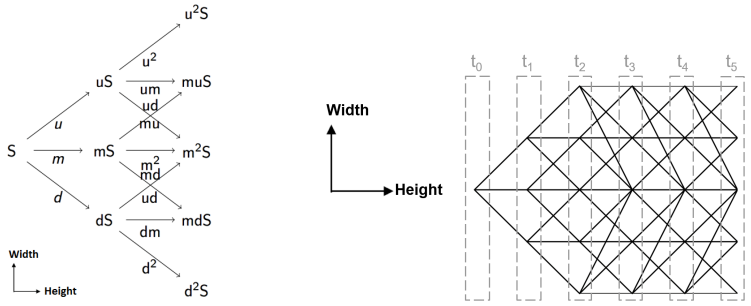
2.1 Option as a Derivative and Bond as an Underlying Asset.

We consider bonds paying a fixed coupon rate up until maturity, They are are traded more frequently as they offer larger premiums. A *call* or *put* bond option is a derivative contract that gives an investor the right, but not the obligation, to, respectively, buy or sell a bond for a predetermined strike price at a future exercise date before the maturity of the bond. Options are used for hedging of the portfolio risks or market speculation. A *call* is used to benefit from a decline in interest rates and a respective increase in bond prices, while a *put* from the inverse situation. Vanilla options vary in when they can be exercised; European, on one particular date, Bermudan, on many specific dates, and American, on any date. Analytical formulas exist for an immediate and exact valuation of European options. The value of the latter two can only be approximated using numerical methods. We refer to the financial literature [9] for further details.

In this work, we deal with a *multi-callable or puttable fixed-rate bond.* Our versions are able to capture any bond cash flow. We focus on Bermudan or American embedded optionality.

2.2 Hull-White One-Factor Short Rate Model (*HW1F*).

A stochastic process describes a probabilistic evolution of a random variable over time. We consider a one-factor interest rate model, where there is a single source of uncertainty, a short rate. It is an interest rate, which is applicable at instantaneous or very short period of time. We consider a simplified version of the Hull-White extension of the Vasicek model with constant volatility σ and mean-reversion rate a. The dynamics of *HW1F* are expressed by a stochastic differential equation (SDE) of form $dr = [\theta(t) - ar]dt + \sigma dW$. The drift (first) part in the equation includes a constant short rate value r. The short rate 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 and defining the average direction that r moves at time t and rate a. The diffusion (second) part comprises of σ and



(a) A trinomial tree. (b) A Hull-White *HW1F* Tree.

Fig. 1. 1a shows tree construction for a stochastic variable S. 1b is a trinomial tree with a bounded width incorporating the mean-reversion phenomenon.

the stochastic variable dW. *HW1F* is appreciated by the practitioners for its ability to reproduce an arbitrary market yield curve and enable a model parameter calibration to the prices currently observed in the markets.

2.3 Trinomial Tree as a Numerical Method

. A trinomial tree represents a random propagation of the interest rate in time. Nodes represent the values, while edges the probabilities of transitions from one value to the other. This tree is "trinomial", because the computation of each value at a current time step depends on three values of the previous (or next) time step. It is defined by the choices u, m, d, and probabilities of moving to the upper p_u, middle p_m and bottom node p_d, where $pm = 1 - pu - pd$. See Figure 1a. We follow the propagation procedure outlined in [10] and [11]. The *height* is defined by the remaining time to maturity of the bond, specified as a year fraction. The *width* is determined by a mean-reversion rate (the lower the rate the wider the tree). It follows that the height varies across bonds, while the width is determined in the model calibration. On top of that, both dimensions depend on the frequency of pricing days/time steps. Every bond depends on a *term structure* (or a *yield curve*), a relationship between interest rates and maturity terms. It represents a market expectation of interest rate evolution, and is adjusted daily. The *HW1F* tree has a bounded width dimension and is subdivided into two time regions, the "normal" region (from time 0 to j_{max}) and the "switched" region (from time j_{max} onwards). See Figure 1b, where $j_{max} = t_2$, width is 5, height is 6, $i \in [0, 5]$ and $j \in [-2, 2]$. The "switched" region is further subdivided into upward-, normal-, and downward-switching regions. i denotes the time step along the height, and j the node along the width. In the upward (downward) region, upward (downward) switching prevails, i.e. from node (i, j) nodes $(i + 1, j)$, $(i + 1, j + 1)$, $(i + 1, j + 2)$ (nodes $(i + 1, j)$, $(i + 1, j - 1)$, $(i + 1, j - 2)$) are reached, respectively, while In the normal region, to $(i + 1, j)$, $(i + 1, j + 1)$ and $(i + 1, j - 1)$. We refer to [5, 9] for a further description of the model and the numerical method.

Wojciech: please introduce the arrays `Qs`, α`s` and `Cs` in the financial description, so that I can refer to them directly in the core of the paper.

3 NOTATION AND NESTED-PARALLEL SPECIFICATION

This chapter first introduces a functional notation that is then used, in section 3.2 to present the nested-parallel structure of the pricing algorithm that is the subject of this paper.

In comparison to a lower-level loop-based notation, such as CUDA, the functional notation has the advantage that (i) it enables a concise specification of all available (nested) parallelism in terms of well-known data-parallel operators such as map, reduce, scan, and (ii) it allows to demonstrate at a high level the rewrite rules that were applied for deriving GPU-FLAT.

3.1 Functional Notation

We use `i32` and `real` to denote the type of a 32-bit integer and (single- or double-precision) floating-point numbers, respectively, we use `[N]`α to denote the type of an array whose elements have type α, `[`a_1`,...,`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` (without parenthesis or commas).

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 are written as `let arr[i] = x`.[4] The notation supports (i) if condition `if c then` e_1 `else` e_2, which have similar semantics with the C ternary operator `c?` e_1 `:` e_2, and (ii) loop expressions:

loop $(x^1, \ldots, x^m) = (x_0^1, \ldots, x_0^m)$ **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 executes iterations `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 part may be syntactically omitted—i.e., `loop (`x^1`, ..., `x^m`) for i < n do e` is legal—in which case the initialization refers to in-scope variables bearing the same name ($x^{1\cdots m}$).

Most importantly, the notation supports several key-parallel operators, whose types and semantics are shown in listing 1: `iota` applied to integer `n` creates the array with elements from `0` to `n-1` (i.e., an iteration space), and `replicate n v` creates an array of length `n` whose elements are all `v`. A `map` operation applies its function argument `f` to each element of the input array, resulting into an array of 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`. Similarly, `map2` applies its function argument to corresponding elements from its two input arrays. `reduce` successively applies a binary-*associative* operator $\odot$ to all elements of its input array, where $e_\odot$ denotes the neutral element of $\odot$. `scan` [4] (a.k.a., parallel-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}) starts 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 zeroes and ones, where a one denotes the start position of a subarray, and (ii) by a length-matching flat array containing in order all elements of all subarrays. For example, `flags = [1,0,1,0,0,0,1]` would denote an array with three rows, having two, four and one elements, respectively, and `sgmscan`^{inc} `(+) 0 flags [1,2,3,4,5,6,7]` results in array `[1,3,3,7,12,18,7]`. Segmented scan can be implemented in terms of a scan with a modified operator [4], e.g., for the inclusive one:

λ(f1,v1) (f2,v2) $\rightarrow$ **if** f2 != 0 **then** (f1|f2, v2) **else** (f1|f2, v1$\odot$v2)

The last operator, `(scatter x is vs)` updates in place the array `x` at indices contained in array `is` with the values contained in array `vs`, except that out-of-bounds indices are ignored (not updated). For example, in listing 1, value b_1 was not written in the result because its index `-1` is out of bounds.

[4] In place updates can be supported without affecting language purity by means of a uniqueness type mechanism [7].

```
1  iota : (n: i32) → [n]i32
2  iota n = [0,...,n-1]
3
4  replicate : (n: i32) → α → [n]α
5  replicate n v = [v,...,v]
6
7  map : ∀n.(α → γ) → [n]α → [n]γ
8  map f [a_1, ..., a_n] = [f a_1, ..., f a_n]
9
10 map2: ∀n.(α → β → γ) → [n]α → [n]β → [n]γ
11 map2 g [a_1,...,a_n] [b_1,...,b_n] =
12               [g a_1 b_1, ..., g a_n b_n]
13
14 reduce: ∀n.(α → α → α) → α → [n]α → α
15 reduce ⊙ e_⊙ [a_1,...,a_n] = a_1 ⊙ ... ⊙ a_n
16
17 scan: ∀n.(α → α → α) → α → [n]α → [n]α
18 scan^inc ⊙ e_⊙ [a_1,...,a_n] =
19           [a_1, a_1⊙a_2, ..., a_1 ⊙ ... ⊙a_n]
20 scan^exc ⊙ e_⊙ [a_1,...,a_n] =
21           [e_⊙, a_1, ..., a_1 ⊙ ... ⊙a_{n-1}]
22
23 sgmscan : ∀n.(α → α → α) → α →
24                  [n]i32 → [n]α → [n]α
25 sgmscan^inc ⊙ e_⊙
26   [..., 1,  0,     ..., 0,  1,   ...]
27   [..., a_1^k, a_2^k,   ..., a_n^k, a_1^{k+1}, ...] =
28   [..., a_1^k,...,a_1^k ⊙ ... ⊙a_n^k, a_1^{k+1}, ...]
29
30 scatter: ∀n,m.[n]α → [m]i32 → [m]α → [n]α
31 scatter [a_0, a_1, a_2, a_3,...,a_{n-1}]
32         [2, -1,  0,  3]
33         [b_0, b_1, b_2, b_3] =
34         [b_2, a_1, b_0, b_3,...,a_{n-1}]
```

Listing 1. Data-Parallel Operators Semantics

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

Listing 2. Nested-Parallel Implementation.

3.2 Simplified Nested Data-Parallel Specification

Listing 2 sketches the (simplified) implementation of the pricing algorithm, but which 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 as argument and computes its price approximation. Computation starts by determining the width `w` and height `h` of the trinomial tree (at line 2), and by initializing arrays `Qs` of size `w` (lines 3-4) and α`s` of size `h` (lines 5-7). The two sequential loops of indices `i` and `ii` implement the forward and backward tree propagation.

The first loop fills in the values of array α`s`: First, the `map` operation of length `w` (at lines 10-15) computes each element at the current breath level in the tree, i.e., `Qs'[j]`, by aggregating the three different values belonging to the previous breath level (time step) i.e., `Qs[j-1]`, `Qs[j]`, `Qs[j+1]`. (Please notice that only the current and previous breadth levels—rather than the entire tree—are manifested in memory by means of arrays `Qs'` and `Qs`.) The newly created array `Qs'` is then summed up—by means of the `reduce` operation at line 16—and provides the value of α`s[i+1]`. Please notice that both parallel operations are inner to the outer `map` operation, which is applied to the whole portfolio, thus giving raise to nested parallelism. Finally, the resulted values of `Qs'`, α`_i'` and α`s` are bound to the loop-variant variables `Qs`, α`_i` and α`s` for the next iteration.

The second loop traverses the tree backwards, from the maturity to the present date, and at each step it computes the prices associated to the current breadth level by a similar `map` operation. The price of the instrument today is at the root of a tree, corresponding to `Cs[w/2+1]` (after the loop).

4 OUTER-PARALLEL VERSION 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 the low-level (CUDA) implementation is straightforward, one non-trivial issue refers to the fact that arrays such as `Qs` and α`s` need to be *expanded* across all valuations in the portfolio and to be stored in global memory—because there is no suitable statically-known upper bound for their length, and as such they cannot be stored in CUDA-private memory. This section discusses two performance critical optimizations:

The first refers to finding a good layout for the expanded arrays, named $\texttt{Qs}^{exp}$ and $\alpha\texttt{s}^{exp}$, that enables coalesced access to global memory, while minimizing the memory pressure.

The second refers to diminishing the overhead of one of the two levels of thread divergence, which is due to the fact that the per-thread computation is carried out both across the (breadth) width `w` and the height `h` of the tree, where both `w` and `h` vary significantly across threads (instruments).

4.1 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 $\texttt{Qs}^{exp}$ and $\alpha\texttt{s}^{exp}$. Next, one can compute the starting offsets into $\texttt{Qs}^{exp}$ of the logically-local arrays `Qs`—declared in the `valuate` function executed by 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 logical arrays `Qs` corresponding to iteration (thread) `i` of the outer `map` is represented by the slice $\texttt{Qs}^{exp}$`[Qs_offs[i]:Qs_offs[i+1]]` of length `ws[i]`. Similar thoughts apply to $\alpha\texttt{s}^{exp}$. The inspector code is presented bellow, where `unzip` transforms an array-of-tuples to a tuple-of-arrays, i.e., `unzip [(`a_1`,`b_1`),...,(`a_n`,`b_n`)] = ([`a_1`,...,`a_n`], [`b_1`,...,`b_n`])`

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

In practice, the implementation of the parallel inspector above fuses the `map` and the two `scan` operations, leading to efficient code, especially if this builds on a single-pass scan implementation [13]. The problem however is that this layout results in poor spatial locality, because consecutive threads access global memory with a largish stride—given by the values of `w` (or `h`)—leading thus to accessing global memory in an uncoalesced fashion, which can be prohibitively expensive.

4.2 Global Padding Enables Coalesced Access

Matters can be improved by computing the maximal width $\texttt{w}^m$ and height $\texttt{h}^m$ across all `n` trees and padding each *local* array to this size, i.e., $\texttt{Qs}^{exp}$ and $\alpha\texttt{s}^{exp}$ are now two-dimensional arrays of sizes $\texttt{n}\times\texttt{w}^m$ and $\texttt{n}\times\texttt{h}^m$, respectively. Modulo thread divergence (imbalanced) issues, fully coalesced access to global memory can be achieved by working with the *transposed* versions of $\texttt{Qs}^{exp}$ and $\alpha\texttt{s}^{exp}$: the inner array dimension of size `n` is indexed by the thread (instrument) number, hence consecutive threads would now access consecutive memory locations, thus achieving coalesced access to global memory. The main downside is a potential explosion in the memory footprint, for example in the case when the distribution of width or height values is skewed.

4.3 Block/Warp-Level Padding: Coalesced Access at Small Memory Overhead

The memory explosion can be optimized by padding at finer granularity, for example at CUDA thread-block or warp level. Denoting by B the block (or warp) size, this is accomplished by finding the maximal width (and height) for each group of B trees (instruments) and then by padding to the maximal size of the group, and working with the transposed version of the arrays, as before.

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

The implementation of the parallel inspector is shown above: (i) the precomputed array of widths (heights) is reshaped as an $\frac{n}{B} \times B$ two-dimensional array, then (ii) a segmented reduce operation finds the maximal width of each group, (iii) the padded lengths of each group is obtained by multiplying the maximal widths by B, and (iv) the start offsets into the expanded array for each block (warp) are computed by an exclusive-scan operation. 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 two-dimensional 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]). In order to obtain coalesced access, the implementation manifests the transpose of $\texttt{Qs}_b^{exp}$ of shape $\texttt{wb}_{max}$[b] × B.

4.4 Data Reordering Optimizes One Level of Thread Divergence

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 the two forward- and backward-traversal loops of count h and 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 constructs are sources of thread divergence and their combination further 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 lockstep execution will take $O(\texttt{m}^2)$ time, rather than the expected $O(\texttt{m})$.

With the GPU-OUTER implementation, 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* is more beneficial than the heights because it improves the coalescing of the version performing block/warp level padding, and the arrays of size w, such as Qs, are accessed more frequently than the ones of size h, such as αs. We conclude by noticing that all pre-processing (inspector) overheads are small, summing up to under 2% of the total runtime.

5 GPU-FLAT: FLATTENING TWO-LEVEL PARALLELISM

This section demonstrates how the GPU-FLAT implementation was derived from the nested-parallel code of Listing 2. While we keep the discussion here intuitive and specific to the trinomial-pricing algorithm, Appendix A formalizes the transformation by means of a set of inference (rewrite) rules, which can be integrated in the repertoire of a compiler.

Essentially, GPU-FLAT utilizes both levels of parallelism, which allows to optimize in the same time (i) temporal locality and (ii) both levels of divergence. The idea is to first sort the options in decreasing order of their heights—thus optimizing the divergence of the forward/backward traversal loops—and then to (bin-)pack the input (instruments) into bins, such that the summed widths of their trees does not exceed the CUDA block size—we chose the maximal size 1024—which is thought as the capacity of the bin. The two parallel levels—of the instruments in a bin, and of

```
let valuate^bin (q: i32, bin: [q]Instrument) : [q]real =
  let (ws,hs) = map f1 bin
  let B_w  = scan^exc (+) 0 ws
  let len_flat= B_w[q-1] + ws[q-1]
  let tmp = map2 (λs b → if s == 0 then -1 else b) ws B_w
  let flag= scatter (replicate len_flat 0) tmp (replicate q 1)
  let tmp = scan^inc (+) 0 flag
  let out_inds = map (λ x → x-1) tmp
  -- map (λ w → iota w) ws
  let tmp = map (λ f→ 1-f) flag
  let inn_inds = sgmscan^inc (+) 0 flag tmp
  -- map(λw→ replicate w 0) ws
  let Qss = replicate len_flat 0.0
  -- map2 (λ Qs w → Qs[w/2+1] = f2(ins) ) Qss ws
  let tmp_i= map2(λ b w → b + w/2 + 1) B_w ws
  let tmp_v= map f2 bin
  let Qss = scatter Qss tmp_i tmp_v
  -- init regular (transposed) h_max×q matrix αss^T
  let h_max = reduce max 0 hs
  let α_is = map f3 bin
  let αss^T = scatter (replicate (h_max*q) 0.0) (iota q) α_is

  -- map-loop interchange; loop count padded to h_max-1
  let(_,_,αss^T)=loop(Qss,α_is,αss^T)
    for i < h_max-1 do
      -- map2 (λis α_i→map (...) is) inn_inds α_is
      let Qss' = map2 (λ j oi → let (b,h) = (B_w[oi], hs[oi])
                                in  if  i ≥ h-1 then Qss[b+j]
                                    else let q0 = Qss[b+j]
                                         let q1 = if j > 0 then Qss[b+j-1] else 1.
                                         let q2 = if j<w-1 then Qss[b+j+1] else 1.
                                         in  g1( i, j, α_is[oi], q0, q1, q2)
                      ) inn_inds out_inds
      -- map (reduce (+) 0) Qss'
      let scQs = sgmscan^inc (+) 0.0 flag Qss'
      let α_vs = map2 (λ b w→ scQs[b+w-1]) B_w ws
      -- map(λα→α[i+1]=g2(..)) αss
      let tmp_i = map2 (λ h k → if i≥h-1 then -1 else (i+1)*q + k) hs (iota q)
      let α_is' = map2 g2 α_vs bin
      let αss^T = scatter αss^T tmp_i α_is'
      in  (Qss', α_is', αss^T)
  let Css = replicate len_flat 100. ... -- second loop is not shown (similar)
```

Listing 3. Flat-Parallel Implementation.

the inner parallelism inside `valuate` function—are then combined (flattened) and mapped at 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 flatten parallelism has roughly the size of the CUDA block. On the other hand, temporal locality is also optimized because the data created by inner-parallel operations (inside `valuate`)—such as the arrays `Qs`, `Cs`—is maintained in fast (scratchpad/shared) memory.[5] The downsides are that the flattening transformation introduces instructional overhead, and shared-memory/register pressure.

5.1 Flat-Parallel Version in Fast Memory

Listing 3 shows the code resulted by applying the flattening transformation: the `bin` array corresponds to a batch of `q` instruments—whose summed widths is less than the CUDA-block size—and the result is an array of length `q` or real numbers denoting the prices of those instruments at current time. The flat code is obtained by distributing the (outer) `map` operation—i.e., over the `q` instruments of the bin—around each `let` statement of the original `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 CUDA-block size. For brevity and clarity of exposition, our discussion ignores the complications related to padding parallel arrays to CUDA-block size and to the non-trivial offsets in the arrays of all instruments and of α`s` corresponding to the sub-arrays of the current block—these are tedious but straightforward to add.

Listing 3 starts by computing the widths and heights,[6] of the trees of the `q` instruments (line 2). For demonstration, we assume that `q=2`, and the widths and heights are `ws=[2,4]` and `hs=[4,3]`. Lines 3-11 compute three helper arrays (`flag`, $\texttt{out}_{inds}$ and $\texttt{inn}_{inds}$) that are used in the code transformation. The first array `flag` is the flag component in the flat-representation of an irregular array of shape `ws`, such as `Qss`. We recall that the flag arrays is required by the segmented scan operations; an irregular array of shape `[2,4]` has two rows of lengths `2` and `4`, respectively, and its flag array marks with `1` the start of each subarray and has otherwise `0` elements. It follows that we expect `flag` to be equal to `[1,0,1,0,0,0]`. This is computed by applying an exclusive scan on `ws`, resulting in $\texttt{B}_w$`=[0,2]`, then by computing the total number of elements $\texttt{len}_{flat}$`=2+4=6`, and finally, by the `scatter` operation at line 6 that 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]`, as expected.

The second array $\texttt{out}_{inds}$ is supposed 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 obtained element (lines 7-8).

The final 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 10-11 by negating the flag array, resulting in `[0,1,0,1,1,1]`, and applying an inclusive-segmented scan on the result under the flag array `flag`, i.e., scanning independently the two logical rows of two and four elements, respectively.

Lines 12-17 in listing 3 are the flattening across `q` instruments of the lines 3-4 in listing 2—which initializes `Qs` elements to zeroes and sets index `w/2+1` to value `f2(opt)`. The zero-initialization of `Qs` is translated to a `replicate` of length $\texttt{len}_{flat}$—i.e., the summed widths of the `q` instruments—and the update at index `w/2+1` is translated to a `scatter` on the expanded `Qss` in which

- the updated indices are computed at line 15 by summing the offset in `Qss` of each instrument, denoted by `b`$\in$$\texttt{B}_w$, with `w/2+1`, where `w`$\in$`ws`, and
- the updated values are the result of mapping `f2` on the `q` instruments at line 16.

The initialization of α`ss`—the expansion of α`s`—is simply obtained by padding each row to the maximal height $\texttt{h}_{max}$ of the `q` instruments—hence total length is `q`$\times$$\texttt{h}_{max}$—and by using a `scatter` to overwrite the first entry in every row with the result of calling `f3`. This is implemented in lines 19-21, except that αss^T—the transpose of αss—is used in order to achieve coalesced access

[5] The array α`s` is maintained as before in global memory—because it is not guaranteed to fit—and it is padded and transposed at block level to optimize coalescing and memory footprint, as before.

[6] In practice the widths and heights are precomputed by an inspector which is sliced out of the original code. This is because, as preliminary steps, the instruments are first sorted after their heights in order to optimize the divergence of the sequential loops, and then the widths are necessary for packing instruments into bins.

to global memory. Next, the forward loop is padded to count $\mathtt{h}_{max}$, the outer `map` of length `q` is interchanged inside the loop, and distribution continues on the loop-body statements.

Lines 27-33 correspond to flattening the `map` that is applied to `iota w` to compute array `Qs'` in listing 2, lines 10-15. 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 $\mathtt{inn}_{inds}$ and $\mathtt{out}_{inds}$:

- $\mathtt{inn}_{inds}$ is precisely the expansion of `(iota w)` across the `q` instruments, hence `j` takes the same values as in listing 2;
- $\mathtt{out}_{inds}$ is used to access values that are the same across the width threads assigned to process the current instrument, but are needed by each thread—we recall that the $\mathtt{out}_{inds}$ values record the index of each instrument in each of the width entries associated with it. For example $\mathtt{out}_{inds}$ is used to (indirectly) index into length-`q` arrays $\mathtt{B}_w$, `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—because the loop count was padded to maximal value $\mathtt{h}_{max}$. If so, then the input value of `Qss` is directly returned.

The code between lines 35-36 is the flattening across all `q` instruments of the (original) `reduce` at line 16 in listing 2, which sums up the values of array `Qs'`. This is implemented by first performing an inclusive segmented scan on the expanded array `Qss'`, which by definition, computes the `q` corresponding sums in the last entry of each logical subarray of the result `scQs`. Then these last entries are extracted by a `map` operation of length `q`; the index of the last entry of the $\mathtt{i}^{th}$ subarray is $\mathtt{B}_w$`[i]+ws[i]-1`, because $\mathtt{B}_w$ and `ws` record the offset and the size of each subarray, respectively.

Finally, lines 38-40 implement the expansion of the update to α`s[i+1]` at line 18 in listing 2. This is translated to a `scatter` that updates αss^T at the `q` flat-indices belonging to row `i+1` (stored in `tmp_i`) with the values `tmp_v` obtained by applying `g2` to all α`_vs` and batched instruments. Please note that if the loop index `i` is greater or equal than the logical loop count `h-1` then the return index is `-1`, hence the update is ignored.

Similar ideas apply for the translation of the backward loop, which is not shown. Our CUDA implementation of GPU-FLAT fuses aggressively 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 the shared memory (since they save register space), and arrays $\mathtt{out}_{inds}$ and $\mathtt{inn}_{inds}$ are hold in registers. The shortcomings of GPU-FLAT are that (i) it introduces instructional overhead—i.e., the code is more complex than the nested-parallel one, (ii) it introduces significant register pressure[7] and that (iii) the parallel operations of size `q` underutilize the block-level parallelism (which is typically much larger than `q`).

6 EXPERIMENTAL EVALUATION

6.1 Experimental Methodology.

We run the experiments on two Linux systems: **D1**: 26-core 2-way HT Intel Xeon Platinum 8167M CPU@2.00GHz (**CPU1**), 754 GB RAM and NVIDIA Tesla **V100** SXM2 GPU (2688 Volta *FP64* cores, 32 GB HBM2) using CUDA 10.0. **D2**: 2×8-core 2-way HT Intel Xeon CPU E5-2650 v2@2.60GHz (**CPU2**), 128 GB RAM and NVIDIA GeForce **GTX 780Ti** GPU (2880 Kepler *FP32* Cores, 4 GB GDDR5) using CUDA 9.2.

We measure total application runtime, including all overheads—e.g., host-device memory transfers and preprocessing steps such as data reordering and bin-packing. Execution times are averaged

[7] nvcc compiler reports that 74 – 76 registers per thread are used by default and a speedup of up to 1.66× is achieved by limiting the number of registers to 32.

	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}	192	38	1.67	176	39	1.67	174	38	1.67	44	8	0.25	145	24	0.25
V^{ws}_{o1}	117	35	1.67	99	32	1.67	99	32	1.67	53	15	0.25	104	34	0.25
V^{ns}_{o2}	105	37	3.23	113	39	3.23	123	50	3.23	27	7	3.19	101	26	3.19
V^{ws}_{o2}	455	117	3.23	469	123	3.23	480	142	3.23	56	27	3.19	199	90	3.19
V^{ns}_{o3}	107	50	3.22	115	56	3.14	125	59	3.04	27	7	2.36	99	26	2.35
V^{ws}_{o3}	469	147	1.91	484	157	1.86	493	155	1.85	59	28	0.25	**232**	**96**	0.28
V^{ns}_{o4}	121	51	3.14	130	56	3.02	142	60	2.84	27	7	1.10	111	27	1.10
V^{ws}_{o4}	**543**	**148**	1.73	**558**	**157**	1.72	**552**	**155**	1.72	**59**	**28**	0.25	225	95	0.25
Speedup ×	2.8	3.9		3.2	4.0		**3.2**	**4.0**		1.3	3.5		1.6	4.0	
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}	459	59	0.95	464	66	0.95	435	57	0.95	260	27	0.91	115	11	0.91
V^{ws}_{f1}	459	54	0.95	507	54	0.95	415	62	0.95	231	32	0.91	105	14	0.91
V^{ns}_{f2}	456	59	0.95	461	66	0.95	**439**	57	0.95	261	27	0.91	115	12	0.91
V^{ws}_{f2}	**462**	54	0.95	506	54	0.95	413	62	0.95	230	32	0.91	106	14	0.91
V^{ns}_{f3}	399	**81**	0.61	394	**89**	0.56	384	77	0.59	433	79	0.11	300	39	0.20
V^{ws}_{f3}	385	79	0.51	**581**	79	0.51	342	**88**	0.51	**560**	**95**	0.08	**532**	**92**	0.08
Speedup ×	1.0	1.4		1.4	1.3		1.0	1.5		2.2	3.5		**4.6**	**8.4**	

Table 1. Summary of GPU-OUTER and GPU-FLAT performance in $GFLOP^{SPEC}/s$.

across 5 runs and are reported in $GFLOP^{SPEC}/s$, which counts the number of float operations as they appear in the high level specification (GPU-OUTER). We argue that $GFLOP^{SPEC}/s$ is better suited than *GFLOP/s* for comparing across different implementations of the same algorithm, because it represents normalized runtime (while *GFLOP/s* may report the slower version as having better performance).

6.2 Datasets.

The evaluation uses seven synthetic datasets that model realistic distributions of instruments in portfolios, and exhibit different workload-divergence properties. 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 607 across all trees (no divergence). **R1**, **R2**, and **R3** (**R***) use random distributions of heights and widths in intervals [13, 1200] and [7, 511], respectively. **R1** uses uniform distribution for both, **R2** uses uniform and standard-normal distributions for widths and heights, respectively, while **R3** does the opposite. The idea is that normal distribution results in significantly less divergence than uniform distribution (for the widths in **R3** and heights in **R2** than **R1**). Finally, **S1** and **S2** (**S***) present skewed distributions : in **S1** 1% of the dataset consists of widths and heights in [461 − 512] and [1082 − 1200], respectively, while the rest has widths and heights uniformly distributed in [7 − 58] and [12 − 131]. **S2** presents the same figures, but separates skewness over dimensions: 1% combines skewed widths with uniform heights and another 1% the reverse.

6.3 Performance Results

in $GFLOP^{SPEC}/s$ are reported in Table 1 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, and V_2, V_3, V_4 to versions

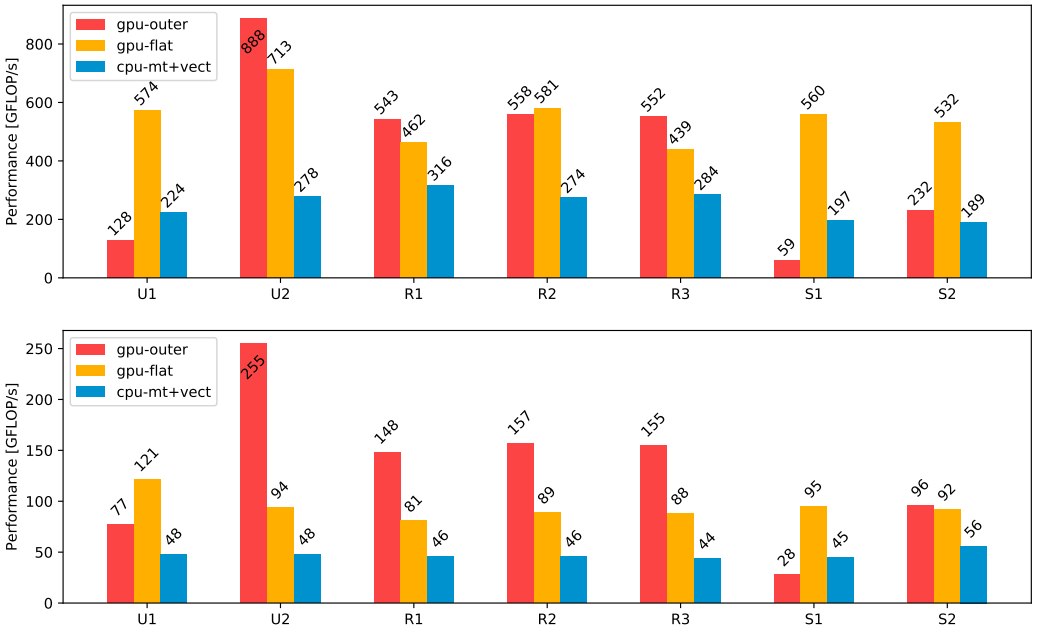


Fig. 2. Best *FP64* (upper) and *FP32* (lower) performance across all datasets on **D1**/**CPU1** and **D2**/**CPU2**, respectively.

in which coalescing was achieved by padding and transposing at global, block or warp level, respectively. ^{ns} and ^{ws} correspond to versions without and with data reordering (sorting). For GPU-OUTER the trees are sorted by width first, for GPU-FLAT by height first. Column M^{64} reports the memory footprint for *FP64* runs in *GB*. *FP32* versions use half the M^{64} of *FP64*.

Main observations for GPU-OUTER are:

- **U1** and **U2** are not reported, because, due to their constant width/height size, sorting and block/warp level padding have no effect; coalescing results in 1.1× and 23.8× speedups on **D1** and 2.7× and 10.1× on **D2**.
- The two optimizations, data reordering and coalescing by padding, have small or even *negative* impact when applied in isolation, e.g., V_{o1}^{ns} is faster than V_{o1}^{ws} and all V_{ok}^{ns} $(k = 2 \ldots 4)$ on **D1** on **R***. However, when combined they have significant impact (as high as 4×) on all datasets.
- Warp-level padding V_{o4}^{ws} achieves coalescing at the cost of a modest 1.04× increase in M^{64} wrt. V_{o1}^{ns}, and is the fastest (except **S2** on **D1**). In comparison, V_{o2}^{ws} increases M^{64} by 1.9× and 12.8× on (**R***) and (**S***), respectively.

Important observations for GPU-FLAT are:

- We observe relatively-stable performance ranging between 439−581 and 81−95 $GFLOP^{\text{SPEC}}/s$ on **D1** and **D2**, respectively.
- The impact of the optimizations on **R*** is small, with reordering by height having the highest 1.4× impact. In several cases the best performance is achieved without sorting, because our bin-packing heuristics are ineffective in filling the bins, leading to larger ratios of idle threads inside blocks.

- GPU-FLAT performs well on **S***. Reordering and coalescing have a significant impact: on average 4.7×, and as high as 8.4× on **S2/D2**, and V_{f3}^{ws} reduces M^{64} by 11× wrt. V_{f1}^{ns}. Performance of GPU-FLATis highly impacted by the shared memory size, e.g., on **S1** it yields a significant 9.5× speedup over GPU-OUTER on **D1**, but only 3.5× on **D2** (half the memory size).

We report an *FP64* arithmetic intensity of 3.91 on **R1**, which is well below the 8.73 peak ratio for **D1** and makes the algorithm memory-bound. The same is valid for *FP32* on **D2**, where $7.83 < 15.88$. Figure 2 compares the best GPU-OUTER and GPU-FLAT configurations and our CPU-MT+VECT, that uses OpenMP multi-threading and AVX2 vectorization. GPU-OUTER is faster than GPU-FLAT on the **U2** and **R***, and is slower on **U1**, because the 3000 valuations underutilize hardware parallelism, and on **S***, as it GPU-FLAT optimizes better the two-level divergence. However, on the newer **D1** hardware, GPU-FLAT significantly narrows the gap (e.g., only 1.25× slower on **U2**). Even though we use powerful CPUs with 32 and 52 hardware threads, the GPU versions are faster than the CPU-MT+VECT, with speedups as high as 3.2× and 5.3× (**U2**) and on average 2.5× on **D1** and 3.1× on **D2**. We also implement an OpenMP version using QuantLib library [8] and obtain 3-to-4 order of magnitude speedups over it (not shown).

7 RELATED WORK AND CONCLUSION

7.1 Accelerating Financial Algorithms

A large body of work was dedicated to GPU acceleration of *Monte-Carlo Simulations* used for (i) *pricing derivatives* [12, 14, 15], (ii) model *calibration* [1] or (iii) *risk management*[6]. Beyond that, we are not aware of any comprehensive study or publicly-available GPU-accelerated lattice model code, in particular for *HW1F* and callable bonds. Moreover, the acceleration approach used in this work is general and can be applied to other compute-intensive financial algorithms or engineering applications.

7.2 Compiler Techniques

Our implementation draws inspiration from a number of compiler techniques. The GPU-FLAT version builds on the flattening transformation [2, 3], which maps irregular nested parallelism into a sequence of flat-parallel ones. The key difference is that flattening pushes all sequential recurrences outside the parallel code, and the many prefix-sum operations introduced in global memory limit performance gains. Instead, we take advantage of the tree widths being less than the block size 1024 in all practical cases, and we efficiently bin-pack inner parallelism to operate in shared memory. Finally, our techniques for optimizing the two-level divergence by sorting after the tree dimensions was inspired by data reordering transformations aimed at improving locality and communication patterns [17], and by inspector-executor restructuring transformations [16]. Achieving coalesced accesses by working with arrays in transposed form and the derivation of differently optimized implementations from the same high-level specification was inspired by work on the Futhark language and compiler [8]. However, we are not aware of any compiler framework that is able to generate the GPU-FLAT code version from the nested-parallel specification of figure 2. We believe that the work presented in this paper may provide useful insights into how to integrate such techniques into a compiler.

REFERENCES

[1] C. Andreetta, V. Bégot, J. Berthold, M. Elsman, F. Henglein, T. Henriksen, M.-B. Nordfang, and C. E. Oancea. 2016. FinPar: A Parallel Financial Benchmark. *ACM Trans. Archit. Code Optim.* 13, 2, Article 18 (2016), 27 pages.

[8] QuantLib - a free/open-source library for quantitative finance - http://quantlib.org/

[2] L. Bergstrom and J. Reppy. 2012. Nested Data-parallelism on the GPU. *SIGPLAN Not.* 47, 9 (Sept. 2012), 247–258. https://doi.org/10.1145/2398856.2364563
[3] G.E. Blelloch, J.C. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee. 1994. Implementation of a Portable Nested Data-Parallel Language. *Journal of parallel and distributed computing* 21, 1 (1994), 4–14.
[4] Guy E. Blelloch. 1989. Scans as Primitive Parallel Operations. *Computers, IEEE Transactions* 38, 11 (1989), 1526–1538.
[5] D. Brigo and F. Mercurio. 2006. *Interest Rate Models - Theory and Practice: With Smile, Inflation and Credit* (2nd ed.). Springer, Berlin ; New York.
[6] M. Dixon, J. Chong, and K. Keutzer. 2009. Acceleration of Market Value-at-risk Estimation. In *Proceedings of the 2nd Workshop on High Performance Computational Finance (WHPCF '09)*. ACM, New York, NY, USA. https://doi.org/10.1145/1645413.1645418
[7] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. 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)*. ACM, New York, NY, USA, 556–571. https://doi.org/10.1145/3062341.3062354
[8] T. Henriksen, F. Thorøe, M. Elsman, and C. Oancea. 2019. Incremental Flattening for Nested Data Parallelism. In *Procs. Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. ACM, 53–67.
[9] J. Hull. 2017. *Options, Futures, and Other Derivatives* (10th ed.). Pearson Ed.
[10] J. Hull and A. White. 1994. Numerical Procedures for Implementing Term Structure Models I: Single-Factor Models. *The Journal of Derivatives* 2, 1 (1994), 7–16.
[11] J. Hull and A. White. 1996. Using Hull-White Interest Rate Trees. *The Journal of Derivatives* 3, 3 (1996), 26–36. https://doi.org/10.3905/jod.1996.407949
[12] A. Lee, C. Yau, M.B. Giles, A. Doucet, and C.C. Holmes. 2010. On the Utility of Graphics Cards to Perform Massively Parallel Simulation of Advanced Monte Carlo Methods. *Journal of Computational and Graphical Statistics* 19, 4 (2010), 769–789.
[13] Duane Merrill and Michael Garland. March 2016. *Single-pass Parallel Prefix Scan with Decoupled Look-back, NVR-2016-002.* Technical Report. NVIDIA Corporation.
[14] F. Nord and E. Laure. 2011. Monte Carlo Option Pricing with Graphics Processing Units. In *In Procs. Int. Conf. ParCo.*
[15] C. E. Oancea, C. Andreetta, J. Berthold, A. Frisch, and F. Henglein. 2012. Financial Software on GPUs: Between Haskell and Fortran. In *Procs. Work. Functional High-Perf. Computing (FHPC '12)*. ACM, 61–72.
[16] C. E. Oancea and L. Rauchwerger. 2013. A Hybrid Approach to Proving Memory Reference Monotonicity. In *Int. Languages and Compilers for Parallel Computing (LCPC'11) (LNCS)*, Vol. 7146. 61–75.
[17] M. M. Strout, L. Carter, and J. Ferrante. 2003. 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)*. ACM, New York, NY, USA, 91–102. https://doi.org/10.1145/781131.781142

A INFERENCE RULES FOR TWO-LEVEL FLATTENING

This section provides (the gist of) the formal inference rules that implement the two-level flattening transformation, which were used in section 5 to derive the GPU-FLAT code version. The transformation is applied on code exhibiting exactly two-levels of parallelism, in which the outer level is a map of length q, whose (lambda) body respects the following restrictions:

- all inner-parallel operations—such as map, reduce, scan, scatter—have the same length, denoted w, for the same outer-map iteration, albeit w may vary across iterations;
- there exists at most one (inner) sequential loop whose count is variant to the outer map, and such a loop is not contained in an inner parallel operation;
- most computation is carried out by the inner-parallel operations, i.e., all sequential loops should contain inner parallelism;
- for simplicity of exposition, we assume that all inner arrays are one dimensional—this can be achieved by a pre-processing step that flattens the array indexing.

We also assume that the lengths of the parallel operations for each of the q iterations of the outer map, have been pre-computed by slicing out the computation of w into a simpler map construct. Figure 4 shows the code for computing the helper arrays, discussed in section 5, where q denotes the number of outer-map iterations packed in the current bin, and shp denotes the parallel lengths in the current bin. We assume that this code has been already inserted, at the beginning of the translation. (To note, the code allows empty parallel lengths, i.e., elements of shp are allowed to be zero.) We briefly recount the rationale for each helper variable:

- $\texttt{len}_{flat}$ is the sum of the inner-parallel lengths of the q packed outer-map iterations, i.e., the size of the flat-parallel operation;
- $\texttt{B}_{shp}$ is an array of length q, which records the start index in the flat-parallel array representation of each of the q logical sub-arrays (segments);
- $\texttt{inn}_{inds}$ is an array of length $\texttt{len}_{flat}$ that records the inner iteration space of each inner-parallel construct. For example, in figure 4, the first sub-array has length 3, therefore the first three elements of $\texttt{inn}_{inds}$ are `0,1,2`; the last sub-array has length 2, therefore the last two elements of $\texttt{inn}_{inds}$ are `0,1`.
- $\texttt{out}_{inds}$ is an array of length $\texttt{len}_{flat}$, which records the index of the outer-map iterations for each of the inner-parallel elements. For example, the first sub-array has length 3, therefore the first three elements of $\texttt{out}_{inds}$ are `0,0,0`; the third sub-array has length 2, therefore the last two elements of $\texttt{out}_{inds}$ are `2,2`. The rationale is that scalar variables that were variant to the outer map are expanded in the transformed code to length-q arrays, and accesses to these scalars from inner map operations is translated by using $\texttt{out}_{inds}$ to indirectly index into these expanded arrays.
- flag is the flag array (also of length $\texttt{len}_{flat}$) that is used by segmented scan, and which records with an one the start of each sub-arrray and has zero elements otherwise.

A.1 The Translation Context Σ

The context of the translation—denoted Σ and summarized in Figure 3—is represented by a record containing the following fields:

- $\mathcal{H}$ is a record containing helper variables for translation:
  - w the original-program scalar variable w containing the length of the inner-parallel operation;
  - q the number of outer-map iterations packed in the current bin;
- i,o are the scalar variables used by the translation as formal arguments of the lambda body of a flat map, and that take values from $\texttt{inn}_{inds}$ and $\texttt{out}_{inds}$, respectively;

```
1  let helper (q:int) (shp: [q]int) :
2      (int,[q]int,[]int,[]int,[]int) =
3    -- Assume shp = [3,1,2]
4    let inds = scan^exc (+) 0 shp
5    let B_shp = map2(λs k→ if (s <= 0)
6                           then -1 else k
7                    ) shp inds
8    let len_flat= B_shp[q-1] + shp[q-1]
9    -- B_shp : i32 = [0,3,4], len_flat = 6
10   let flag = scatter (replicate len 0)
11                      B_shp (replicate q 1)
12   -- flag:[len_flat]i32 = [1,0,0,1,1,0]
13
14   let inn_inds = sgmscan^exc (+) 0 flag
15                       (replicate len_flat 1)
16   -- inn_inds:[len_flat]i32 =[0,1,2,0,0,1]
17   let tmps    = scan^inc (+) 0 flags
18   let out_inds = map (-1) tmps
19   -- out_inds:[len_flat]i32 =[0,0,0,1,2,2]
20
21   in  (len_flat,B_shp,inn_inds,out_inds,flag)
```

Listing 4. Helper Function for Flattening

$\mathcal{V}^o$ *and* $\mathcal{V}^t$: *set of variable names of the original and target program, respectively.*

$$\Sigma = < \mathcal{H}, \mathcal{V}^{map}_{inv}, \mathcal{A}^{par}_{var}, \mathcal{S}_{var}, \mathcal{A}^{seq}_{var}, \mathcal{L} >$$

$$\mathcal{H} =< \mathtt{w}, \mathtt{q}, \mathtt{i}, \mathtt{o}, \mathtt{shp}, \mathtt{len}_{flat}, \mathtt{B}_{shp}, \mathtt{inn}_{inds}, \mathtt{out}_{inds}, \mathtt{flag} >$$

$$\mathcal{H} \in \mathcal{V}^o \times \mathcal{V}^t \times \ldots \times \mathcal{V}^t$$

$$\mathcal{V}^{map}_{inv} \in Set(\mathcal{V}^{o/t})$$

$$\mathcal{A}^{par}_{var} \in \mathcal{V}^o \mapsto \mathcal{V}^t$$

$$\mathcal{S}_{var} \in \mathcal{V}^o \mapsto \mathcal{V}^t$$

$$\mathcal{A}^{seq}_{var} \in \mathcal{V}^o \mapsto \mathcal{V}^t \times \mathcal{V}^t$$

$$\mathcal{L} = < \mathtt{j}, \mathtt{n}^o, \mathtt{n}^t_{max} > \in \mathcal{V}^{o/t} \times \mathcal{V}^o \times \mathcal{V}^t$$

Fig. 3. The Structure of the Transformation Context Σ.

  - shp, $\mathtt{len}_{flat}$, $\mathtt{B}_{shp}$, $\mathtt{out}_{inds}$, $\mathtt{inn}_{inds}$ and flag have been discussed before, and are assumed available in the translation process.
- $\mathcal{V}^{map}_{inv}$ represent the set of original program variables that are invariant to the outer map (free variables). The translation will directly reuse these names and may insert also variables of the transformed program in this set.
- $\mathcal{A}^{par}_{var}$ is a finite map that binds each (inner) parallel array—of symbolic length w—in the original code to their corresponding expansion (across q iterations) in the transformed code; these array are stored in fast (shared) memory.
- $\mathcal{A}^{seq}_{var}$ is a finite map that binds each sequential array—i.e., that are computed sequentially and are assumed to be of length other than w—in the original code to their expansion in the transformed code. In this case, the array expansion is performed by padding to the maximal size of the q subarray, and the expanded arrays are stored in global memory. The lookup returns the expanded array name together with its maximal (padded) row/segment length.
- $\mathcal{S}_{var}$ is a finite map that binds each scalar variable in the original program that is variant within the outer map to its corresponding expanded-array variable in the translation.
- $\mathcal{L}$ is a record containing the information of the sequential loop that has irregular count across the q-packed iterations of the outer map:
  - field j denotes the original-loop index, whose name is reused in the translation;
  - field $\mathtt{n}^o$ denotes the original-program variable storing the loop count. The lookup $\mathtt{ns}^t = \mathcal{S}_{var}(\mathtt{n}^o)$ must succeed inside such a loop, and the result $\mathtt{ns}^t$ is the array expansion of the scalar $\mathtt{n}^o$ in the translated program;
  - field $\mathtt{n}^t_{max}$ denotes the transformed-program variable recording the maximal count across the q packed iterations of the outer map—i.e., $\mathtt{n}^t_{max}$ = `reduce max 0` $\mathtt{ns}^t$;

  Whenever the translation encounters such a loop, this information is updated; otherwise, outside such loops, filed $\mathtt{n}^o$ is set to a dummy value that fails the $\mathcal{S}_{var}$ lookup.

The translation process assumes that the initial context have the fields $\mathcal{H}$ and $\mathcal{V}^{map}_{inv}$ already filled, while the other are unset. The translation requires that the input program is normalized to

A-normal form: **let** bindings can be seen as a block of statements followed by a sequence of result variables, where the statement can be:

- unary/binary operations in the form of three-address code;
- a parallel operation, whose lambda body is also normalized;
- sequential loops whose body is also normalized and the loop initializers are variables.

For readability and clarity of exposition:

- The resulted program is not normalized and variables are not necessarily uniquely named.
- The translation rules use the fields of record $\Sigma.\mathcal{H}$ freely—i.e., omitting the $\Sigma.\mathcal{H}$ prefix.
- The discussion ignores the complications related to padding and to the non-trivial offsets at which the sequential arrays of the current bin are located inside flat (global-memory) arrays that extends across all bins; these details are tedious but straightforward to add.

A.2 Formalizing the Translation By Inference Rules

The core inference (rewrite) rules of our translation are formalized in figures 4 and 5. The rules allow inferences of the form

- $\Sigma \vdash_{out} e \Rightarrow e'$, which are read "in context Σ, the source expression e appearing outside the lambda function of an inner-parallel map operation can be translated into the target expression e'."
- $\Sigma \vdash_{inn} e \Rightarrow e'$, which are read "in context Σ, the source expression e appearing inside the lambda function of an inner-parallel map operation can be translated into the target expression e'." In this case, the variables $\mathcal{H}.\mathtt{o}$ and $\mathcal{H}.\mathtt{i}$ are assumed available in the declaration of the enclosing `map` lambda function, and are taking values from $\mathcal{H}.\mathtt{out}_{inds}$ and $\mathcal{H}.\mathtt{inn}_{inds}$.

In an inference rule, the part below the line specifies the translation (the conclusion), i.e., this source expression is translated to this target expression. The part above the line contains the premises necessary for the translation to fire, including (i) generation of fresh names, (ii) lookup operations into finite maps (that need to succeed for the rule to fire), (iii) recursive application of inference rules, (iv) creation of new contexts, and (v) abbreviating subexpression so that they fit in the space of the conclusion.

We discuss the inference rules $\vdash_{out}$ from Figure 4:

G0 refers to a binary operation $\odot$ between two scalar variables, in which one $\mathtt{v}_1^o$ is invariant to the outer map and the other $\mathtt{v}_2^o$ is variant. The array-expanded translation of the latter, $\mathtt{vs}_2^t$, is obtained after a successful lookup in $\Sigma.\mathcal{S}_{var}$. The bound expression is translated to a **map** that applies $\odot$ to the invariant $\mathtt{v}_1^o$ and each element of $\mathtt{vs}_2^t$. The result is stored in fresh variable $\mathtt{vs}^t$, a new context Σ' is created by extending $\Sigma.\mathcal{V}_{var}$ with the new binding $\mathtt{v}^o \mapsto \mathtt{vs}^t$, and the let-body expression e^o is recursively translated in the new context Σ'. If both operands were invariant, then the original binding would have been left unmodified; if both were variant, then **map2** would have been used to combine elements from both expanded arrays.

G1 refers to translating an array-indexing expression $\mathtt{Q}^o\mathtt{[ind}^o\mathtt{]}$, where $\mathtt{ind}^o$ is outer-map variant and $\mathtt{Q}^o$ is a parallel array. The expanded arrays of the translation $\mathtt{Qs}^t$ and $\mathtt{inds}^t$ are looked up, and the indexing expression is translated to a map that takes values `k∈{0,...,q-1}` and in which $\mathtt{Qs}^t$ is indexed by $\mathtt{inds}^t\mathtt{[k]}$ to which we add $\mathtt{B}_{shp}\mathtt{[k]}$—the start offset of each parallel sub-array. Finally, the context is extended with the result array, and the let-body expression is recursively translated (into e^t). Indexing a sequential array is similar: the translation and its row-padded size are looked up $(\mathtt{Ys}^t, \mathtt{m}_{max}) = \Sigma.\mathcal{A}^{seq}_{var}(\mathtt{Y}^o)$ and the indexing operation inside the map becomes: $\mathtt{Ys}^t\mathtt{[m}_{max}\mathtt{*k + inds}^t\mathtt{[k]]}$.

*Specialized-Flattening Rules for code outside inner-***map** *constructs* $\boxed{\Sigma \vdash_{out} e^o \Rightarrow e^t}$

$$\frac{\mathsf{v}_1^o \in \Sigma.\mathcal{V}_{inv}^{map},\quad \mathsf{v}_2^o \notin \Sigma.\mathcal{V}_{inv}^{map},\quad \mathsf{vs}_2^t = \Sigma.\mathcal{S}_{var}(\mathsf{v}_2^o),\quad \mathsf{vs}^t \text{ fresh name},\\ \Sigma' = \Sigma \textit{ with } \{\mathcal{S}_{var} = \Sigma.\mathcal{S}_{var} \cup \{\mathsf{v}^o \mapsto \mathsf{vs}^t\}\} \qquad \Sigma' \vdash_{out} e^o \Rightarrow e^t}{\Sigma \vdash_{out} \textbf{let } \mathsf{v}^o = \mathsf{v}_1^o \odot \mathsf{v}_2^o \textbf{ in } e^o \Rightarrow \textbf{let } \mathsf{vs}^t = \textbf{map } (\lambda \mathsf{v}_2^t \rightarrow \mathsf{v}_1^o \odot \mathsf{v}_2^t)\ \mathsf{vs}_2^t \textbf{ in } e^t} \tag{G0}$$

$$\frac{\mathsf{Qs}^t = \Sigma.\mathcal{A}_{var}^{par}(\mathsf{Q}^o),\quad \mathsf{inds}^t = \Sigma.\mathcal{S}_{var}(\mathsf{ind}^o)\quad (\text{i.e., both lookups succeed}),\\ \mathsf{k}, \mathsf{vs}^t \text{ fresh name},\quad e_{map}^t = \textbf{map } (\lambda\, k \rightarrow \mathsf{Qs}^t[\mathsf{B}_{shp}[\mathsf{k}] + \mathsf{inds}^t[\mathsf{k}]])\ (\textbf{iota } \mathsf{q}),\\ \Sigma' = \Sigma \textit{ with } \{\mathcal{S}_{var} = \Sigma.\mathcal{S}_{var} \cup \{\mathsf{v}^o \mapsto \mathsf{vs}^t\}\},\qquad \Sigma' \vdash_{out} e^o \Rightarrow e^t}{\Sigma \vdash_{out} \textbf{let } \mathsf{v}^o = \mathsf{Q}^o[\ \mathsf{ind}^o\ ] \textbf{ in } e^o \Rightarrow \textbf{let } \mathsf{vs}^t = e_{map}^t \textbf{ in } e^t} \tag{G1}$$

$$\frac{\mathsf{Qs}^t = \Sigma.\mathcal{A}_{var}^{par}(\mathsf{Q}^o),\quad \mathsf{inds}^t = \Sigma.\mathcal{S}_{var}(\mathsf{ind}^o),\quad \mathsf{vs}^t = \Sigma.\mathcal{S}_{var}(\mathsf{v}^o),\quad \Sigma' \vdash_{out} e^o \Rightarrow e^t,\\ \mathsf{j} = \Sigma.\mathcal{L}.\mathsf{j},\quad \mathsf{ns}^t = \Sigma.\mathcal{S}_{var}(\Sigma.\mathcal{L}.n^o),\quad (\text{i.e., inside a loop}),\quad \mathsf{k}^t \text{ fresh name},\\ e_{inds}^t = \textbf{map2 } (\lambda\, \mathsf{k} \rightarrow \textbf{if } \mathsf{j} < \mathsf{ns}^t[k] \textbf{ then } \mathsf{inds}^t[k] + \mathsf{B}_{shp}[\mathsf{k}] \textbf{ else } -1)\ (\textbf{iota } q)}{\Sigma \vdash_{out} \textbf{let } \mathsf{Q}^o[\ \mathsf{ind}^o\ ] = \mathsf{v}^o \textbf{ in } e^o \Rightarrow \textbf{let } \mathsf{Qs}^t = \textbf{scatter } \mathsf{Qs}^t\ e_{inds}^t\ \mathsf{vs}^t \textbf{ in } e^t} \tag{G2}$$

$$\frac{\mathsf{n} = \Sigma.\mathcal{H}.\mathsf{w},\quad \mathsf{v}^o \notin \mathcal{V}_{inv}^{map},\ \mathsf{vs}^t = \Sigma.\mathcal{S}_{var}(\mathsf{v}^o),\quad e_{rep}^t = \textbf{map}(\lambda \mathsf{o} \rightarrow \mathsf{vs}^t[\mathsf{o}])\ \mathsf{out}_{inds}\\ \mathsf{Qs}^t \text{ fresh name},\quad \Sigma' = \Sigma \textit{ with } \{\mathcal{A}_{var}^{par} = \Sigma.\mathcal{A}_{var}^{par} \cup \{\mathsf{Q}^o \mapsto \mathsf{Qs}^t\}\},\quad \Sigma' \vdash_{out} e^o \Rightarrow e^t}{\Sigma \vdash_{out} \textbf{let } Q^o = \textbf{replicate } \mathsf{n}\ \mathsf{v}^o \textbf{ in } e^o \Rightarrow \textbf{let } \mathsf{Qs}^t = e_{rep}^t \textbf{ in } e^t} \tag{G3}$$

$$\frac{\mathsf{m}^o \notin \Sigma.\mathcal{V}_{inv}^{map},\quad \mathsf{m}^o \neq \Sigma.\mathcal{H}.\mathsf{w},\quad \mathsf{ms}^t = \Sigma.\mathcal{S}_{var}(\mathsf{m}^o),\quad \mathsf{v}^o \notin \Sigma.\mathcal{V}_{inv}^{map},\quad \mathsf{vs}^t = \Sigma.\mathcal{S}_{var}(\mathsf{v}^o),\\ \Sigma' = \Sigma \textit{ with } \{\mathcal{V}_{inv}^{map} = \Sigma.\mathcal{V}_{inv}^{map} \cup \{\mathsf{m}_{max}\},\ \mathcal{A}_{var}^{seq} = \Sigma.\mathcal{A}_{var}^{seq} \cup \{\mathsf{X}^o \mapsto (\mathsf{Xs}^t, \mathsf{m}_{max})\}\}\\ \mathsf{k}, \mathsf{m}_{max} \text{ fresh names},\quad e_{red}^t = \textbf{reduce max } 0\ \mathsf{ms}^t,\quad \Sigma' \vdash_{out} e^o \Rightarrow e^t,\\ e_{map}^t = \textbf{map } (\lambda k \rightarrow \mathsf{vs}^t[\mathsf{k}/\mathsf{m}_{max}])\ (\textbf{iota } (\mathsf{q}{*}\mathsf{m}_{max}))}{\Sigma \vdash_{out} \textbf{let } X^o = \textbf{replicate } \mathsf{m}^o\ \mathsf{v}^o \textbf{ in } e^o \Rightarrow \textbf{let } \mathsf{m}_{max} = e_{red}^t \textbf{ in let } \mathsf{Xs}^t = e_{map}^t \textbf{ in } e^t} \tag{G4}$$

$$\frac{\Sigma.\mathcal{H}.\mathsf{w} = \textbf{length } \mathsf{Q}^o,\quad \mathsf{Qs}^t = \Sigma.\mathcal{A}_{var}^{par}(\mathsf{Q}^o),\quad \mathsf{vs}^t, \mathsf{Qs}_{scn} \text{ fresh name},\\ \Sigma' = \Sigma \textit{ with } \{\mathcal{S}_{var} = \Sigma.\mathcal{S}_{var} \cup \{\mathsf{v}^o \mapsto \mathsf{vs}^t\}\},\quad \Sigma' \vdash_{out} e^o \Rightarrow e^t,\\ e_{scn}^{sgm} = \textbf{segscan } (\odot)\ 0_{\odot}\ \mathsf{flag}\ \mathsf{Qs}^t,\\ e_{map}^{pck} = \textbf{map2 } (\lambda\ b\ s \rightarrow \textbf{if } s == 0 \textbf{ then } 0_{\odot} \textbf{ else } \mathsf{Qs}_{scn}[b + s - 1])\ \mathsf{B}_{shp}\ \mathsf{shp}}{\Sigma \vdash_{out} \textbf{let } v^o = \textbf{reduce } (\odot)\ 0_{\odot}\ \mathsf{Q}^o \textbf{ in } e^o \Rightarrow \textbf{let } \mathsf{Qs}_{scn} = e_{scn}^{sgm} \textbf{ in let } \mathsf{vs}^t = e_{map}^{pck} \textbf{ in } e^t} \tag{G5}$$

$$\frac{\Sigma.\mathcal{H}.\mathsf{w} = \textbf{length } \mathsf{Q}^o,\quad \mathsf{Qs}^t = \Sigma.\mathcal{A}_{var}^{par}(\mathsf{Q}^o),\quad \mathsf{Xs}^t \text{ fresh name},\\ \Sigma' = \Sigma \textit{ with } \{\mathcal{A}_{var}^{par} = \Sigma.\mathcal{A}_{var}^{par} \cup \{\mathsf{X}^o \mapsto \mathsf{Xs}^t\}\},\quad \Sigma' \vdash_{out} e^o \Rightarrow e^t,\\ e_{scn}^{sgm} = \textbf{segscan } (\odot)\ 0_{\odot}\ \mathsf{flag}\ \mathsf{Qs}^t}{\Sigma \vdash_{out} \textbf{let } X^o = \textbf{scan } (\odot)\ 0_{\odot}\ \mathsf{Q}^o \textbf{ in } e^o \Rightarrow \textbf{let } \mathsf{Xs}^t = e_{scn}^{sqm} \textbf{ in } e^t} \tag{G6}$$

$$\frac{\Sigma.\mathcal{H}.\mathsf{w} = \textbf{length } \mathsf{Q}^o,\quad \mathsf{Qs}^t = \Sigma.\mathcal{A}_{var}^{par}(\mathsf{Q}^o),\quad \mathsf{flgs}_Q^t = \Sigma.\mathcal{A}_{var}^{par}(\mathsf{flgs}_Q^o),\\ \mathsf{Xs}^t, \text{ fresh name},\quad \Sigma' = \Sigma \textit{ with } \{\mathcal{A}_{var}^{par} = \Sigma.\mathcal{A}_{var}^{par} \cup \{\mathsf{X}^o \mapsto \mathsf{Xs}^t\}\},\\ e_{scn}^{sgm} = \textbf{segscan } (\odot)\ 0_{\odot}\ \mathsf{flgs}_Q^t\ \mathsf{Qs}^t,\quad \Sigma' \vdash_{out} e^o \Rightarrow e^t}{\Sigma \vdash_{out} \textbf{let } X^o = \textbf{segscan } (\odot)\ 0_{\odot}\ \mathsf{flg}_Q^o\ \mathsf{Q}^o \textbf{ in } e^o \Rightarrow \textbf{let } \mathsf{Xs}^t = e_{scn}^{sqm} \textbf{ in } e^t} \tag{G7}$$

Fig. 4. Flattening rules for the code outside inner-parallel constructs (assumed to have all the same length).

G2 refers to an in-place update to an element of a parallel array $\texttt{Q}^o\texttt{[ind}^o\texttt{]}$, where the index is variant. The update occurs inside a loop of index `j` and variant count $\texttt{n}^o$, which is translated to array $\texttt{ns}^t$—i.e., please note that the rule fires only if the lookup $\Sigma.\mathcal{S}_{var}(\Sigma.\mathcal{L}.\texttt{n}^o)$ succeeds. This case is translated by a **scatter** operation in which the to-be-updated indices into $\texttt{Qs}^t$ are computed similarly to **G1**, except that they are guarded by condition $\texttt{if j < ns}^t\texttt{[k]}$. If this condition does not hold, then the current sub-array has logically finished the execution of the loop, and index `-1` is returned instead, which will prompt the **scatter** to ignore the updates to such sub-arrays. For updating in-place a sequential array, the indexing is computed similarly to **G1**, i.e., $\texttt{m}_{max}\texttt{*k + inds}^t\texttt{[k]}$, and the guard is also inserted.

G3 refers to a replicate-based initialization of a parallel array—i.e., the length of the original array is `w`—where the replicated variable $\texttt{v}^o$ is variant, and has the array-expanded translation $\texttt{vs}^t$. This is translated by a **map** in which the corresponding value for the elements of a sub-array is taken by indirectly accessing $\texttt{vs}^t$ with the sub-array indices stored in helper array $\texttt{out}_{inds}$, i.e., $\texttt{vs}^t\texttt{[o]}$, where $\texttt{o} \in \texttt{out}_{inds}$. Finally, the context is extended with the result array ($\texttt{Q}^o \mapsto \texttt{Qs}^t$), and the let-body expression e^o is recursively translated to e^t.

G4 is similar to **G3** except that the **replicate** initializes a sequential array, i.e., of length $\texttt{m}^o$ different than `w`. The translation of $\texttt{m}^o$ is the array $\texttt{ms}^t$, and its maximal element $\texttt{m}_{max}$ is computed by a **reduce**. The result array $\texttt{Xs}^t$ is padded to have logically `q` rows of length $\texttt{m}_{max}$, and the corresponding values of sub-array elements are selected from $\texttt{vs}^t$ by using the regular indexing $\texttt{k / m}_{max}$ where $\texttt{k} \in \{\texttt{0,...,q*m}_{max}\texttt{-1}\}$.

G5 refers to flattening a reduce operation—on the original parallel array $\texttt{Q}^o$ of size `w`—nested inside the outer map. This is translated by (i) performing a segmented scan on the translated array $\texttt{Qs}^t$, and (ii) by selecting the last element of each sub-array (segment) by means of a map (see e^{pck}_{map}). The index of the last element is `b+s-1`, where $\texttt{b} \in \texttt{B}_{shp}$ represents the start position of each sub-array, and $\texttt{s} \in \texttt{shp}$ represents the length of each sub-array. The $\texttt{B}_{shp}$, `shp`, and flag array (for segmented scan) are taken from $\Sigma.\mathcal{H}$.

G6 refers to flattening a scan operation on an original parallel array $\texttt{Q}^o$ of size `w`. This is translated (by definition) to a segmented scan where the flag array is taken from $\Sigma.\mathcal{H}$.

G7 refers to a segmented scan on an original parallel array $\texttt{Q}^o$ of size `w`. This is translated to a segmented scan, where the flag array is the translation of the original flag array—i.e., $\texttt{flgs}^t_Q = \Sigma.\mathcal{A}^{par}_{var}(\texttt{flgs}^o_Q)$—because the original flag array is necessarily a parallel array (length `w`).

We now discuss the inference rules $\vdash_{out}$ from Figure 5:

G8 refers to an inner-map operation, necessarily of length `w`, which is applied directly to parallel arrays. The rule normalizes/rewrites the input map into one that is applied to one array: the iteration space, which is given by `iota w = [0,...,w-1]`. Then, essentially it relies on rule **G9** to perform the translation.

G9 refers to a map operation applied (only) to `iota w`, that occurs inside a sequential loop of index `j` and variant loop count $\texttt{n}^o$, which is translated to array $\texttt{ns}^t$. The original map is translated to a **map2** operating on helper arrays $\texttt{inn}_{inds}$ and $\texttt{out}_{inds}$, and the original lambda expression e^o_k is translated by the $\vdash_{inn}$ inference rules to e^t_k. These rules requires that $\mathcal{H}.\texttt{i}$ and $\mathcal{H}.\texttt{o}$ are available; to this extent $\mathcal{H}.\texttt{i}$ is set to the original formal argument of the lambda `k`—which was taken values from `iota w`, hence it preserves semantics since in the translation it feeds from $\texttt{inn}_{inds}$—and $\mathcal{H}.\texttt{o}$ is used as the second formal argument of **map2**'s lambda, feeding as expected from $\texttt{out}_{inds}$. Finally, an **if** guard is inserted to return a dummy value for the sub-arrays that have logically finished executing their loops (similar to **G2**).

G10 refers to the treatment of a loop whose count $\texttt{n}^o$ is variant to the outer map. The (original) loop-variant variable is assumed to be a parallel array $\texttt{Q}^o$ whose size is assumed invariant

Specialized-Flattening Rules for code outside inner **map** *constructs* $\boxed{\Sigma \vdash_{out} e^o \Rightarrow e^t}$

$$\frac{\Sigma.\mathcal{H}.\mathtt{w} = \textbf{length}\ \mathtt{A}^o, \quad \mathtt{k}\ \text{fresh name}, \quad \Sigma \vdash_{out} \textbf{let}\ \mathtt{Q}^o = \textbf{map}\ (\lambda \mathtt{k} \rightarrow \textbf{let}\ \mathtt{a} = \mathtt{A}[\mathtt{k}]\ \textbf{in}\ e^o_a)\ (\textbf{iota}\ \mathtt{w})\ \textbf{in}\ e^o_{let}\ \Rightarrow\ e^t_{all}}{\Sigma \vdash_{out} \textbf{let}\ \mathtt{Q}^o = \textbf{map}\ (\lambda \mathtt{a} \rightarrow e^o_a)\ \mathtt{A}^o\ \textbf{in}\ e^o_{let}\ \Rightarrow\ e^t_{all}} \tag{G8}$$

$$\frac{\begin{array}{c}\Sigma' = \Sigma\ with\ \{\mathcal{H}.\mathtt{i} = \mathtt{k}\}, \quad \Sigma' \vdash_{inn} e^o_k \Rightarrow e^t_k, \quad \mathtt{ns}^t = \Sigma.\mathcal{S}_{var}(\Sigma.\mathcal{L}.n^o), \quad \mathtt{j} = \Sigma.\mathcal{L}.\mathtt{j}, \\ e^t_{map} = \textbf{map2}\ (\lambda \mathtt{k}\ \mathtt{o} \rightarrow \textbf{if}\ j < \mathtt{ns}^t[\mathtt{o}]\ \textbf{then}\ e^t_k\ \textbf{else}\ \mathtt{dummy})\ \mathtt{inn}_{inds}\ \mathtt{out}_{inds}, \\ \mathtt{Qs}^t\ \text{fresh name}, \quad \Sigma'' = \Sigma\ with\ \{\mathcal{A}^{par}_{var} = \Sigma.\mathcal{A}^{par}_{var} \cup \{\mathtt{Q}^o \mapsto \mathtt{Qs}^t\}\}, \quad \Sigma'' \vdash_{out} e^o\ \Rightarrow\ e^t\end{array}}{\Sigma \vdash_{out} \textbf{let}\ \mathtt{Q}^o = \textbf{map}\ (\lambda \mathtt{k} \rightarrow e^o_k)\ (\textbf{iota}\ \mathtt{w})\ \textbf{in}\ e^o\ \Rightarrow\ \textbf{let}\ \mathtt{Qs}^t = e^t_{map}\ \textbf{in}\ e^t} \tag{G9}$$

$$\frac{\begin{array}{c}\Sigma.\mathcal{H}.\mathtt{w} = \mathtt{length}\ \mathtt{Q}^o, \quad \mathtt{Qs}^t_0 = \Sigma.\mathcal{A}^{par}_{var}(\mathtt{Q}^o_0), \quad \mathtt{ns}^t = \Sigma.\mathcal{S}_{var}(\mathtt{n}^o), \\ \mathtt{Qs}^t, \mathtt{n}_{max}\ \text{fresh names}, \quad \Sigma' = \Sigma\ with\ \{\mathcal{A}^{par}_{var} = \Sigma.\mathcal{A}^{par}_{var} \cup \{\mathtt{Q}^o \mapsto \mathtt{Qs}^t\}\} \\ \Sigma'' = \Sigma'\ with\ \{\mathcal{L} = <\mathtt{j}, \mathtt{n}^o, \mathtt{n}_{max}>,\ \mathcal{V}^{map}_{inv} = \Sigma.\mathcal{V}^{map}_{inv} \cup \{\mathtt{n}_{max}\}\} \\ e^t_{red} = \textbf{reduce}\ \textbf{max}\ 0\ \mathtt{ns}^t, \quad \Sigma'' \vdash_{out} e^o_{body}\ \Rightarrow\ e^t_0, \quad e^t_0 = \textbf{let}\ \mathtt{Qs}^t_r = e^t_r\ \textbf{in}\ \mathtt{Qs}^t_r \\ e^t_{inds} = \textbf{map2}\ (\lambda\ \mathtt{o}\ \mathtt{i} \rightarrow \textbf{if}\ \mathtt{j} < \mathtt{ns}^t[\mathtt{o}]\ \textbf{then}\ \mathtt{i}\ \textbf{else}\ -1)\ \mathtt{out}_{inds}\ (\textbf{iota}\ \mathtt{len}_{flat}) \\ e^t_{body} = \textbf{let}\ \mathtt{Qs}^t_r = e^t_r\ \textbf{in scatter}\ \mathtt{Qs}^t\ e^t_{inds}\ \mathtt{Qs}^t_r\end{array}}{\begin{array}{c}\Sigma \vdash_{out} \textbf{loop}\ (\mathtt{Q}^o) = (\mathtt{Q}^o_0)\ \textbf{for}\ \mathtt{j} < \mathtt{n}^o\ \textbf{do}\ e^o_{body}\ \Rightarrow \\ \textbf{let}\ \mathtt{n}_{max} = e^t_{red}\ \textbf{in loop}\ (\mathtt{Qs}^t) = (\mathtt{Qs}^t_0)\ \textbf{for}\ \mathtt{j} < \mathtt{n}_{max}\ \textbf{do}\ e^t_{body}\end{array}} \tag{G10}$$

Specialized-Flattening Rules for code inside inner-map constructs $\boxed{\Sigma \vdash_{inn} e^o \Rightarrow e^t}$

$$\frac{\mathtt{v}^o\ \text{has scalar type}, \quad e^t_k = \begin{cases}\mathtt{vs}^t[\mathtt{o}], & \text{if } \mathtt{vs}^t = \Sigma.\mathcal{S}_{var}(\mathtt{v}^o) \\ \mathtt{v}^o, & \text{otherwise}\end{cases}}{\Sigma \vdash_{inn} \mathtt{v}^o\ \Rightarrow\ e^t_k} \tag{G11}$$

$$\frac{\mathtt{Qs}^t = \Sigma.\mathcal{A}^{par}_{var}(\mathtt{Q}^o), \quad \Sigma \vdash_{inn} \mathtt{v}^o_{ind}\ \Rightarrow\ e^t_{ind}}{\Sigma \vdash_{inn} \mathtt{Q}^o[\ \mathtt{v}^o_{ind}\ ]\ \Rightarrow\ \mathtt{Qs}^t[\ \mathtt{B}_{shp}[\mathtt{o}] + e^t_{ind}\ ]} \tag{G12}$$

$$\frac{(\mathtt{X}^t, \mathtt{h}_{max}) = \Sigma.\mathcal{A}^{seq}_{var}(\mathtt{X}^o), \quad \Sigma \vdash_{inn} \mathtt{v}^o_{ind}\ \Rightarrow\ e^t_{ind}}{\Sigma \vdash_{inn} \mathtt{X}^o[\ \mathtt{v}^o_{ind}\ ]\ \Rightarrow\ \mathtt{X}^t[\ \mathtt{h}_{max} * \mathtt{o} + e^t_{ind}\ ]} \tag{G13}$$

Fig. 5. Flattening rules for the code outside (G8-G10) and inside (G11-13) inner-map constructs.

to the loop. The count of the translated loop is padded to $\mathtt{n}_{max}$: the maximal value of $\mathtt{ns}^t$, which is the array-expansion translation of scalar $\mathtt{n}^o$. A fresh variable is created for the translation of $\mathtt{Q}^o$, denoted $\mathtt{Qs}^t$, and the context is updated with the new binding $\mathtt{Q}^o \mapsto \mathtt{Qs}^t$ and the loop information. The body of the original loop e^o_{body} is recursively translated in the newly created context Σ'' and its result is assumed to be variable $\mathtt{Qs}^t_r$. This variable contains the correct loop result for the sub-arrays that have not yet finished the loop execution, but contains dummy results for the ones that have already finished execution—see for example rule **G9**. The latter values are stored in $\mathtt{Qs}^t$, and an extra step is necessary to put together the correct values for all subarrays. The **map2** operation computes the indices of all elements belonging to loop-active sub-arrays and -1 for the rest; these indices are fed to a **scatter**

operation which updates $\texttt{Qs}^t$ at the positive indices to the values of $\texttt{Qs}^t_r$ (and preserves the other values). The **scatter** operation adds negligible overhead in practice because it can be fused in most cases with the parallel computation of $\texttt{Qs}^t_r$ and with the computation of indices, hence neither needs to be manifested in memory. The case when the loop-variant variable is a scalar requires a similar scatter. The case when the loop variant symbols is a sequential array, the scatter operation is not necessary, because such array can only be updated in place, and rule **G2** already ensures that loop-inactive sub-arrays are not updated.

We finally discuss the inference rules $\vdash_{inn}$ from Figure 5, which assume that the code of interest is inside the lambda-expression of an inner **map** operation:

G11 refers to the translation of a scalar variable $\texttt{v}^o$. If the scalar is variant to the outer map—i.e., lookup $\texttt{vs}^t = \Sigma.\mathcal{S}_{var}(\texttt{v}^o)$ succeeds—than the translation is $\texttt{vs}^t\texttt{[o]}$, where o takes values from $\texttt{out}_{inds}$ in the enclosing map. Otherwise, $\texttt{v}^o$ is used as it is because it is either invariant to the outer map or it is a variable local to the lambda body of the inner map.

G12 refers to indexing into an original parallel array, where the index is stored in scalar variable $\texttt{v}^o_{ind}$. This is translated by selecting from the translated array the element at index $\texttt{B}_{shp}\texttt{[o]+ind}^t$, where $\texttt{ind}^t$ is the translation of $\texttt{v}^o_{ind}$ by rule **G11**, and $\texttt{B}_{shp}\texttt{[o]}$ stores the start offset of the current sub-array—both $\texttt{B}_{shp}$ and o are taken from $\Sigma.\mathcal{H}$.

G13 refers to indexing into a sequential array $\texttt{X}^o$ at index $\texttt{v}^o_{ind}$. The translation $\texttt{Xs}^t$ and the maximal row length $\texttt{h}_{max}$ are obtained by a lookup in $\Sigma.\mathcal{A}^{seq}_{var}$, and the start offset of the current sub-array is computed by $\texttt{h}_{max} * \texttt{o}$.