Re-Write Rules for Flattening in Fast Memory the Parallelism of Hull-White Trinomial Pricing

COSMIN E. OANCEA, University of Copenhagen, Denmark
WOJCIECH M. PAWLAK, SimCorp, Denmark

This document sketches a formalization of the re-write rules used for optimizing the parallel GPU execution of an implementation of the Hull-White One-Factor Short-Rate pricing model, which uses the trinomial-tree lattice-based method for numerically solving the underlying stochastic differential equation.

The proposed code transformation is a specialization of Blelloch's flattening transformation, which maps irregular nested parallelism into a sequence of flat-parallel constructs, such that the work-depth asymptotics of the original program is respected.

Our transformation applies on code with two levels of parallelism: the first level corresponding to a `map` operation, and the second level corresponding to multiple parallel operations—potentially nested within sequential loops—but which have the same length for the same iteration of the outer `map`.

The main difference with respect to the classical approach is that our approach does not replicate the variables that are bound in the outer `map` but free in the inner parallel constructs. Instead, such variables are found by indirectly accessing some auxiliary arrays which are reused across equal-shape arrays and across sequential recurrences (loops). This technique reduces the footprint of scratchpad/fast memory, whose efficient use is paramount to achieving efficient GPU execution.

Additional Key Words and Phrases: parallelism, compilers, flattening transformation, dependence analysis, dynamic analysis, GPU, computational finance, derivative pricing.

1 INTRODUCTION

This document sketches a formalization by means of re-write rules of a specialized instance of Blelloch's flattening code transformation [Blelloch and Greiner 1996; Blelloch et al. 1994].

The transformation was motivated by the work [Pawlak et al. 2021] on optimizing for GPU execution of a standard pricing algorithm, namely the Hull-White One-Factor Short-Rate (*HW1F*) model [Hull and White 1994]. The latter defines the value of a financial instrument by means of a stochastic differential equation, that represents the random changes in the interest rate over time, which is solved numerically by means of a *Trinomial Tree* lattice-based numerical method [Hull and White 1996].

In this algorithm, pricing *one* financial instrument has two main stages. The forward stage builds a tree of bounded width, representing the propagation of the interest rate until the maturity of the underlying bond is reached. [1] 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 as the bounded width of the tree. For example, the forward step computes each node at the current level (time step) from three nodes at the previous level (time step) in the tree.

[1] The model uses a tree of bounded width, because in practice the short-interest rate tends to revert to the mean value over time.

Authors' addresses: Cosmin E. Oancea, University of Copenhagen, Copenhagen, 2100, Denmark, cosmin.oancea@diku.dk; Wojciech M. Pawlak, SimCorp, Copenhagen, 2100, Denmark, Wojciech.M.Pawlak@simcorp.com.

2021. /2021/5-ART1 $15.00
https://doi.org/

Practical use cases require pricing of a (large) portfolio of instruments, which gives raise to an outer level of parallelism, as instruments can be priced independently of each other.

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 parallelization approaches either assume the homogeneous case [Gerbessiotis 2004; Huang and Thulasiram 2005] in which all trees have the same height and width, or acknowledge the problem, but do not offer a solution [Grauer-Gray et al. 2013].

This document considers the challenging case of a heterogeneous portfolio, in which the heights and widths of various instrument trees may differ significantly. At a very high-level, (at least) two parallelization strategies make sense.

The first strategy follows the common wisdom that states 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. The problem with this approach is that the high variance of widths and heights across a portfolio instruments introduces two levels of thread divergence on GPU. These levels correspond

(1) to the sequential time loop of variant height and
(2) to the inner width-parallel operations, which are sequentialized.

This strategy allows to optimise *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 exploit both levels of parallelism, even when the outer parallelism is enough to fully utilize hardware. This allows to better exploit temporal locality by maintaining most of the intermediate arrays in the fast (shared + register) memory, and to optimize *both* levels of divergence. The height-level of divergence is optimized by sorting as before. To optimize the width-level of divergence the idea is

(1) *to pack instruments into bins*, such that their summed widths fits the size of the thread-block (bin), and then
(2) to "flatten" (merge) the available two-level parallelism—the first 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.

The flattening step is non-trivial and is the main focus of this document: In Section 2.2, we document it by presenting an initial simplified nested-parallel specification that uses operators such as `map` and `reduce`. In Section 3, we give the gist of the technique by demonstrating how the flat-parallel program is obtained, named GPU-FLAT.

Section 4 is the core contribution of this document that corresponds to sketching a formalization of the proposed transformation by a set of inference (rewrite) rules, which can possibly be integrated in the repertoire of a data-parallel compiler.

Please be advised that the material other than Section 4 has been already presented elsewhere [Pawlak et al. 2021]; we recount it here for convenience, i.e., to provide the intuition for the rewrite rules and to make this document self contained.

2 NOTATION AND NESTED-PARALLEL SPECIFICATION

This chapter introduces first a functional notation that is then used, in Section 2.2, to present the nested-parallel structure of the pricing algorithm. In comparison to a lower-level loop-based notation, such as CUDA, the functional notation has the advantage that it (i) enables a concise specification of all available (nested) parallelism in terms of well-known data-parallel operators

such as map, reduce, and scan, and (ii) allows to demonstrate at a high level the rewrite rules, that are applied for deriving GPU-FLAT.

2.1 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, [N]α to denote the type of an array, whose elements have type α, [a_1,...,a_n] to denote an array literal, and (a,b) to denote a tuple (record) value. Function f applied to 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. [2] The notation supports (i) conditional if statement if c then e_1 else e_2 with semantics similar to the C ternary operator c? e_1 : e_2, and (ii) sequential loop expression:

$$\textbf{loop}\ (x^1, \ldots, x^m) = (x_0^1, \ldots, x_0^m)\ \textbf{for}\ i < n\ \textbf{do}\ e.$$

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 in an array of same length. The function can be declared in the program or can be an anonymous (λ) function—e.g., map (λx->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 [Blelloch 1989] (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 ($\texttt{scan}^{inc}$) starts with the first element of the array, and the *exclusive* scan ($\texttt{scan}^{exc}$) 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 composed of 0s and 1s, where 1 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, flag = [1,0,1,0,0,0,1] denotes an array with three rows, having two, four and one elements, respectively, and $\texttt{sgmscan}^{inc}$ (+) 0 flag [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 [Blelloch 1989], e.g., for the *inclusive* one:

$$\lambda(f1, v1)\ (f2, v2) \rightarrow \textbf{if}\ f2 != 0\ \textbf{then}\ (f1 \mid f2,\ v2)\ \textbf{else}\ (f1 \mid f2,\ v1 \odot v2)$$

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 is not written in the result because its index -1 is out of bounds.

[2] In place updates can be supported without affecting language purity by means of a uniqueness type mechanism [Henriksen et al. 2017].

```
iota : (n: i32) → [n]i32
iota n = [0,...,n-1]

replicate : (n: i32) → 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]i32 → [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,...,a_1^k⊙...⊙a_n^k, a_1^{k+1}, ...]

scatter: ∀n,m.[n]α → [m]i32 → [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. 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.

2.2 Simplified Nested Data-Parallel Specification

Listing 2 sketches the (simplified) implementation of the pricing algorithm, which nevertheless 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 an 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 an array α`s`: First, the `map` operation of length `w` (at lines 10-15) computes each element across width at the current time step (height) level in the tree, i.e., `Qs'[j]`, by aggregating the three different values belonging to the previous time step level, i.e., `Qs[j-1]`, `Qs[j]`, `Qs[j+1]`. Note that only the current and previous time step elements across width 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]`. Note that both parallel operations occur inside 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 width 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).

3 GPU-FLAT: FLATTENING TWO-LEVEL PARALLELISM

This section demonstrates how the GPU-FLAT implementation is derived from the nested-parallel code of Listing 2. While we keep the discussion here intuitive and specific to the trinomial-pricing algorithm, Section 4 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 simultaneously optimize (i) the 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 do not exceed the thread-block size—we choose the maximal size 1024—which is considered 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 combined (flattened) and mapped at the thread-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 thread-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` and `Cs`—is maintained in fast (scratchpad/shared) memory. [3] The downsides are that the flattening transformation introduces instructional overhead, and shared-memory/register pressure.

3.1 Flat-Parallel Version in Fast Memory

Listing 3 shows the code that is obtained from applying the flattening transformation: the `bin` array corresponds to a batch of `q` instruments—whose summed widths is less than the thread-block size—and the result is an array of length `q` of 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 of size `width` results in a parallel operation of size $\Sigma_{k=0}^{q-1}$`width`$_k$, which is padded to thread-block size. For brevity and clarity of exposition, our discussion ignores the complications related to (i) padding parallel arrays to thread-block size and to (ii) using 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,[4] 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`, `out`$_{inds}$ and `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

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

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

```
1  let valuate^bin (q: i32, 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) tmp (replicate q 1)
7    let tmp = scan^inc (+) 0 flag
8    let out_inds = map (λ x → x-1) tmp
9    -- map (λ w → iota w) ws
10   let tmp = map (λ f→ 1-f) flag
11   let inn_inds = sgmscan^inc (+) 0 flag tmp
12   -- map(λw→ replicate w 0) ws
13   let Qss = replicate len_flat 0.0
14   -- map2 (λ Qs w → Qs[w/2+1] = f2(ins) ) Qss ws
15   let tmp_i= map2(λ b w → b + w/2 + 1) B_w ws
16   let tmp_v= map f2 bin
17   let Qss = scatter Qss tmp_i tmp_v
18   -- init regular (transposed) h_max×q matrix αss^T
19   let h_max = reduce max 0 hs
20   let α_is = map f3 bin
21   let αss^T = scatter (replicate (h_max*q) 0.0) (iota q) α_is
22
23   -- map-loop interchange; loop count padded to h_max-1
24   let(_,_,αss^T)=loop(Qss,α_is,αss^T)
25     for i < h_max-1 do
26       -- map2 (λis α_i→map (...) is) inn_inds α_is
27       let Qss' = map2 (λ j oi → let (b,h) = (B_w[oi], hs[oi])
28                                 in  if  i ≥ h-1 then Qss[b+j]
29                                     else let q0 = Qss[b+j]
30                                          let q1 = if j > 0 then Qss[b+j-1] else 1.
31                                          let q2 = if j<w-1 then Qss[b+j+1] else 1.
32                                          in  g1( i, j, α_is[oi], q0, q1, q2)
33                       ) inn_inds out_inds
34       -- map (reduce (+) 0) Qss'
35       let scQs = sgmscan^inc (+) 0.0 flag Qss'
36       let α_vs = map2 (λ b w→ scQs[b+w-1]) B_w ws
37       -- map(λα→α[i+1]=g2(..)) αss
38       let tmp_i = map2 (λ h k → if i≥h-1 then -1 else (i+1)*q + k) hs (iota q)
39       let α_is' = map2 g2 α_vs bin
40       let αss^T = scatter αss^T tmp_i α_is'
41       in  (Qss', α_is', αss^T)
42   let Css = replicate len_flat 100. ... -- second loop is not shown (similar)
```

Listing 3. Flat-Parallel Implementation.

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 B_w=`[0,2]`, then by computing the total number of elements len_{flat}=2+4=6, and finally, by the `scatter` operation at line 6 that writes 1s at the indices in B_w=`[0,2]` into an array of len_{flat}=6 0s; hence `flag=[1,0,1,0,0,0]`, as expected.

The second array out_{inds} records, for each of the width entries associated with an instrument, the index of that instrument in the current bin. Thus, we expect 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 inn_{inds} is the expansion of `iota w` across the `q` widths, hence we expect 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∈$\texttt{B}_w$, with w/2+1, where w∈ws, and
- the updated values are the result of mapping f2 on the q instruments at line 16.

The initialization of αss—the expansion of αs—is simply obtained by padding each row to the maximal height $\texttt{h}_{max}$ of the q instruments—hence total length is q×$\texttt{h}_{max}$—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 to global memory. Next, the forward loop is padded to count $\texttt{h}_{max}$, the outer map of length q is interchanged inside the loop, and the outer-map 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 is translated to a map2 over $\texttt{inn}_{inds}$ and $\texttt{out}_{inds}$:

- $\texttt{inn}_{inds}$ is precisely the expansion of (iota w) across the q instruments, hence j takes the same values as in Listing 2;
- $\texttt{out}_{inds}$ 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 $\texttt{out}_{inds}$ values record the index of each instrument in each of the width entries associated with it. For example, $\texttt{out}_{inds}$ is used to (indirectly) index into length-q arrays $\texttt{B}_w$, 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≥h-1) that checks that the tree traversal has not already terminated for the current instrument, because the loop count is padded to maximal value $\texttt{h}_{max}$. 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 $\texttt{i}^{th}$ subarray is $\texttt{B}_w$[i]+ws[i]-1, because $\texttt{B}_w$ and ws record the offset and the size of each subarray, respectively.

Finally, lines 38-40 implement the expansion of the update to αs[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. 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 $\texttt{out}_{inds}$ and $\texttt{inn}_{inds}$ are held 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 [5] and that (iii) the parallel operations of size q underutilize the block-level parallelism, which is typically much larger than q.

4 INFERENCE RULES FOR TWO-LEVEL FLATTENING

This section provides (the gist of) the formal inference rules that implement the two-level flattening transformation, which is used in Section 3 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 3, 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 are 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-array and has zero elements otherwise.

4.1 The Translation Context Σ

The context of the translation—denoted Σ and summarized in Figure 1—is represented by a record containing the following fields:

[5] Nvidia 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.

```
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 flag
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

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

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

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

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

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

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

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

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

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

Fig. 1. The Structure of the Transformation Context Σ.

- $\mathcal{H}$ is a record containing helper variables for translation:
  `w` is the original-program scalar variable `w` containing the length of the inner-parallel operation;
  `q` is 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 $\mathtt{inn}_{inds}$ and $\mathtt{out}_{inds}$, respectively.
  The other variables—`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.
- $\mathcal{V}^{map}_{inv}$ represents the set of original program variables that are invariant to the outer map (free variables). The translation directly reuses these names and may also insert 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.

4.2 Formalizing the Translation By Inference Rules

The core inference (rewrite) rules of our translation are formalized in figures 2 and 3. 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 sub-expression so that they fit in the space of the conclusion.

We discuss the inference rules $\vdash_{out}$ from Figure 2:

G0 refers to a binary operation $\odot$ between two scalar variables, in which one $\mathtt{v}^o_1$ is invariant to the outer map and the other $\mathtt{v}^o_2$ is variant. The array-expanded translation of the latter, $\mathtt{vs}^t_2$, 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 $\mathtt{v}^o_1$ and each element of $\mathtt{vs}^t_2$. 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 are invariant, then the original binding is left unmodified. If both are variant, **map2** is 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

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

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

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

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

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

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

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

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

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

Fig. 2. Flattening rules for the code outside inner-parallel constructs (assumed to have all the same length).

which $\texttt{Qs}^t$ is indexed by $\texttt{inds}^t\texttt{[k]}$ to which we add $\texttt{B}_{shp}\texttt{[k]}$—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 $(\texttt{Ys}^t, \texttt{m}_{max}) = \Sigma.\mathcal{A}^{seq}_{var}(\texttt{Y}^o)$ and the indexing operation inside the map becomes: $\texttt{Ys}^t\texttt{[m}_{max}\texttt{*k + inds}^t\texttt{[k]]}$.

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 prompts 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 $\texttt{k} \in \{\texttt{0},\ldots,\ \texttt{q*m}_{max}\texttt{-1}\}$.

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 3:

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 require 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 takes values from `iota w`, hence it preserves semantics since in the

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

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

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

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

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

translation it feeds from $\mathtt{inn}_{inds}$—and $\mathcal{H}.\mathtt{o}$ is used as the second formal argument of **map2**'s lambda, feeding as expected from $\mathtt{out}_{inds}$. 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 $\mathtt{n}^o$ is variant to the outer map. The (original) loop-variant variable is assumed to be a parallel array $\mathtt{Q}^o$ whose size is assumed invariant 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 $\texttt{Qs}^t$, 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 3, 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 $\texttt{h}_{max} * \texttt{o}$.

5 COMPILER RELATED WORK AND CONCLUSIONS

Our implementation draws inspiration from a number of compiler techniques. The GPU-FLAT version builds on the flattening transformation [Blelloch 1990; Blelloch and Greiner 1996; Blelloch et al. 1994], which maps irregular nested parallelism into a sequence of flat-parallel ones, and has been also implemented for GPU execution [Bergstrom and Reppy 2012].

There are two key differences here: 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 Cuda-block level so that temporarily locality can be efficiently exploited by maintaining and reusing arrays in/from shared (scratchpad) memory.

The second difference is more subtle and refers to the fact that the traditional flattening transformation replicates variables that bound in the outer `map` operation but are free in the inner parallel constructs. This may lead to memory explosion, which might prevent the use of shared memory, which is a scarce resource. In comparison, our procedure does not expand such variables, but instead indirectly accesses them by means of auxiliary arrays, such as $\texttt{out}_{inds}$, $\texttt{inn}_{inds}$, $\texttt{B}_w$ in Listing 3. The latter can be seen as part of the shape representation of an irregular array of arrays, which is reused between similarly-shaped arrays, such as `Qss` and `Css`. Furthermore, such auxiliary arrays are invariant to the sequential loop, hence they are created once and the overhead is amortized across the execution of many loop iterations.

The pricing algorithm that motivated our flattening approach is part of a class of applications, whose best parallelization strategy goes against the common-wisdom of always sequentializing

the parallelism in excess of what the hardware can support, because this reduces the per-thread memory footprint and allows to better exploit locality of reference. Another example that benefits from a similar optimization strategy has been studied in the remote-sensing field, where GPU acceleration enabled to apply at continental scale an algorithm that analyses satellite data to detect landscape changes such as deforestation [Gieseke et al. 2020]—but there, the flattening step is straightforward.

Our techniques for optimizing the two-level divergence were inspired by data reordering transformations aimed at improving locality of reference by dynamically (and systematically) reorganizing the array elements in the order in which they are accessed in loop iterations [Ding and Kennedy 1999; Strout et al. 2003, 2018]. Such transformation typically use an inspector-executor model, which has also been used (i) to compute sufficient conditions for loop parallelization [Oancea and Rauchwerger 2013, 2015; Rus et al. 2003] or (ii) to restructure a partially-parallel loop into a sequence of parallel waves [Rauchwerger et al. 1995]. In the context of software thread-level speculation [Dang et al. 2002; Rauchwerger and Padua 1999]—which is another technique that extracts partial loop parallelism at runtime—similar work was aimed at optimizing at runtime the layout of the speculative storage [Oancea and Mycroft 2008], or at optimizing communication overheads by optimistic execution of remote calls [Oancea et al. 2005] in a distributed-component framework [Chicha et al. 2004; Oancea and Watt 2005]. Finally, other work optimizes locality by a memory-centric approach that prefers executions of work-items from the current memory partition and delays the others [Oancea et al. 2009].

However, we are not aware of any compiler framework that is able to derive the GPU-FLAT version of the code. For example the code transformations of the polyhedral Pluto compiler [Bondhugula et al. 2008; Verdoolaege et al. 2013] prioritize optimization of locality of references over the degree of parallelism, but it does not support the flattening transformation (based on scans). The data-parallel language Futhark [Henriksen et al. 2020, 2017] supports flattening, but only of regular parallelism, i.e., the parallel sizes of inner operations cannot vary across different iterations of an outer map. Flattening of irregular can be performed manually in Futhark, but it results in code that maintains all intermediate arrays in global memory, and is significantly slower than GPU-FLAT. While Futhark's incremental-flattening transformation [Henriksen et al. 2019] can create code versions that utilize shared memory, it does so only when all inner-parallel constructs have the same size, and this is not the case of GPU-FLAT, which has some parallel operation of size len_{flat} and others of size q.

The work presented in this document may provide useful insights into how to integrate such a technique into the repertoire of a compiler.

REFERENCES

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.

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

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

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.

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.

Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 101–113. http://doi.acm.org/10.1145/1375581.1375595

Y. Chicha, M. Lloyd, C. Oancea, and S. M. Watt. 2004. Parametric Polymorphism for Computer Algebra Software Components. In *Proc. 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC '04)*. Mirton Publishing House, 119–130.

Francis Dang, Hao Yu, and Lawrence Rauchwerger. 2002. The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops. In *International Parallel and Distributed Processing Symposium (PDPS)*. 20–29.

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.

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

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.

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.

Troels Henriksen, Sune Hellfritzsch, Ponnuswamy Sadayappan, and Cosmin Oancea. 2020. Compiling Generalized Histograms for GPU. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '20)*. IEEE Press, Article 97, 14 pages.

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 *Procs. 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. 556–571.

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.

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.

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.

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

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.

Cosmin E. Oancea, Alan Mycroft, and Stephen M. Watt. 2009. A New Approach to Parallelising Tracing Algorithms. In *Proceedings of the 2009 International Symposium on Memory Management (ISMM '09)*. Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/1542431.1542434

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.

Cosmin E. Oancea and Lawrence Rauchwerger. 2015. Scalable Conditional Induction Variables (CIV) Analysis. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15)*. IEEE Computer Society, Washington, DC, USA, 213–224. http://dl.acm.org/citation.cfm?id=2738600.2738627

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.

Cosmin E. Oancea and Stephen M. Watt. 2005. Parametric Polymorphism for Software Component Architectures. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 147–166. https://doi.org/10.1145/1094811.1094823

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)*. ACM, 12. https://doi.org/10.1145/3460944.3464309

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.

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

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

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.

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.

Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4, Article 54 (2013), 54:1–54:23 pages. http://doi.acm.org/10.1145/2400682.2400713