2014 47th Annual IEEE/ACM International Symposium on Microarchitecture

Locality-Aware Mapping of Nested Parallel Patterns on GPUs

HyoukJoong Lee* Kevin J. Brown* Arvind K. Sujeeth* Tiark Rompf†‡ Kunle Olukotun*
*Stanford University: {hyouklee, kjbrown, asujeeth, kunle}@stanford.edu
†Purdue University: {firstname}@purdue.edu, ‡Oracle Labs: {firstname.lastname}@oracle.com

***Abstract*—Recent work has explored using higher level languages to improve programmer productivity on GPUs. These languages often utilize high level computation patterns (e.g., `Map` and `Reduce`) that encode parallel semantics to enable automatic compilation to GPU kernels. However, the problem of efficiently mapping patterns to GPU hardware becomes significantly more difficult when the patterns are nested, which is common in non-trivial applications.**

To address this issue, we present a general analysis framework for automatically and efficiently mapping nested patterns onto GPUs. The analysis maps nested patterns onto a logical multidimensional domain and parameterizes the block size and degree of parallelism in each dimension. We then add GPU-specific hard and soft constraints to prune the space of possible mappings and select the best mapping. We also perform multiple compiler optimizations that are guided by the mapping to avoid dynamic memory allocations and automatically utilize shared memory within GPU kernels. We compare the performance of our automatically selected mappings to hand-optimized implementations on multiple benchmarks and show that the average performance gap on 7 out of 8 benchmarks is 24%. Furthermore, our mapping strategy outperforms simple 1D mappings and existing 2D mappings by up to 28.6x and 9.6x respectively.

I. Introduction

Fixed power budgets combined with the continued demand for higher performance have encouraged the use of graphics processing units (GPUs) for general purpose computing. However, in order to maximize the performance, application programmers are required to understand the low-level hardware details and manually apply optimizations using GPU-specific programming models (e.g., CUDA). To make GPUs more accessible to non-expert programmers, library and compiler developers have extended target independent high-level languages to offload some part of the application to GPUs. In particular, writing applications in terms of parallel patterns is becoming increasingly popular. Examples are Copperhead [1], Nikola [2], and Accelerate [3]. In addition to their higher productivity, parallel patterns also allow compilers to more easily reason about the program by providing more information (e.g., the intrinsic parallelism and data access patterns) than general purpose sequential language constructs. The compiler can then use its own mapping strategy for each pattern onto parallel GPU threads. For example, many systems similarly map a `Reduce` operation using well-known techniques such as utilizing shared memory and avoiding bank conflicts.

In practice, however, applications are typically naturally expressed by composing multiple parallel patterns together. In particular the multidimensional nature of data in many domains (e.g., HPC, machine learning, graph analytics) makes nested parallel constructs quite common. In fact nearly 75% (14 out of 19) of the applications in the Rodinia benchmark suite for heterogeneous accelerators [4] contain kernels with nested parallelism. To illustrate the issue of mapping nested parallel patterns onto GPUs, we show a simple example in Figure 1. $sumCols$ adds elements in each column of a matrix and $sumRows$ adds elements in each row[1]. Both can be represented as a nested pattern; the outer `Map` pattern for iterating over each column (row), and the inner `Reduce` pattern for reducing the elements in a column (row) into a single value.

```
1  // m: matrix of size [R,C]
2  sumCols = m mapCols { c => c reduce { (a,b) => a + b } }
3  sumRows = m mapRows { r => r reduce { (a,b) => a + b } }
```

Fig. 1: Examples of using nested patterns.

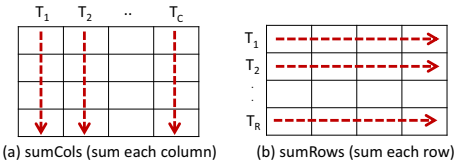


Fig. 2: *1D mapping* (outer loop parallelization) for $sumCols$ and $sumRows$ in Figure 1. T_k corresponds to k^{th} thread on the GPU device, and dotted arrows show the elements each thread touches.

For this nested pattern there exist multiple mapping strategies onto the GPU. One simple mapping strategy is to only parallelize the outer `Map` by assigning one GPU thread[2] for each iteration as shown in Figure 2. We call such strategies that ignore all but one level of parallelism a *1D mapping*. Several systems employ this strategy [5], [6], [2]. Another mapping strategy is to exploit both patterns' parallelism by assigning each iteration of the outer pattern to a thread block and parallelize the inner pattern by threads within the thread block. We call this strategy *thread-block/thread mapping*, which has been previously implemented by Copperhead [1]. Finally, it is also possible to assign each iteration of the outer pattern to a warp and inner iterations to threads within a warp, as presented by Hong et al. [7], which we call *warp-based mapping*. This mapping was originally designed to deal with load imbalance on graph algorithms, where the graph traversal is implemented with a nested pattern.

Unfortunately, none of the three mapping strategies is optimal in general. Figure 3 shows the performance of each mapping on different matrix sizes. Since the number of

[1]Assume the matrix is stored in row-major order.
[2]Section II provides a brief explanation of GPU hardware characteristics and terminology that will be assumed throughout this paper.

1072-4451/14 $31.00 © 2014 IEEE
DOI 10.1109/MICRO.2014.23

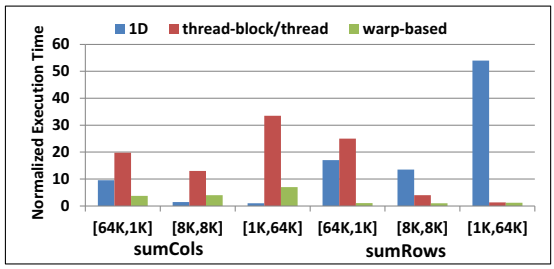


Fig. 3: Performance of $sumCols$ and $sumRows$ on the GPU with different mapping strategies, normalized to the optimal execution time.

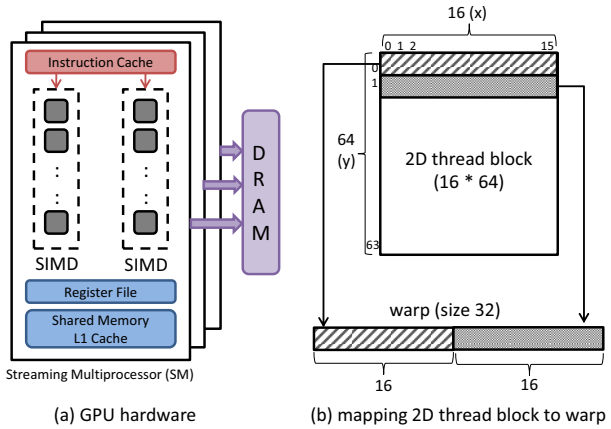


Fig. 4: Overview of GPU hardware and mapping of threads in a two dimensional block on a warp.

elements is the same for the three matrices, ideally all the execution times in Figure 3 should be the same. However, we find a difference of up to 58x. First of all, *1D mapping* shows low performance for the $sumCols$ calculation on a matrix of size (64k,1k) since it launches only 1k threads, which is not enough to fully utilize the GPU resources and hide memory latency. The $sumRows$ calculation is even more problematic, especially on the (1k,64k) matrix. In addition to the problem of resource underutilization the memory access pattern is not optimal for GPUs. As shown in Figure 2 (b), adjacent threads are accessing non-adjacent memory locations, which wastes memory bandwidth. *thread-block/thread mapping* experiences the overhead of too many thread blocks when assigning thread blocks to the 64k dimension. In addition, since threads in the block are accessing elements in a column for $sumCols$, memory requests from adjacent threads cannot be coalesced. *warp-based mapping* shows good performance on $sumRows$ but not on $sumCols$. The reason is again due to suboptimal memory access patterns, similar to *thread-block/thread mapping*, since the mapping always assigns the inner pattern to adjacent threads (threads in a warp).

This example clearly shows that there does not exist a fixed mapping strategy that works well in general, but rather the mapping must be flexibly changed for each case. In this paper we specifically address this problem. We present an analysis technique for automatically mapping nested parallel patterns on GPUs, which provides comparable performance to hand optimized GPU code. The analysis uses the notion of logical dimensions (e.g., *x*,*y*,*z*) for nested patterns and tries to assign adjacent threads to a logical dimension that can maximize data locality and resource utilization. To enable this, we exploit hardware characteristics specific to GPUs (e.g., memory coalescing across threads in a warp) as well as the input/output stencils of the patterns, calculating performance estimate scores for each potential mapping and selecting the one with the best score. The analysis is general enough to generate effectively the same mappings as in the previous work and can also flexibly change the mapping parameters for different use cases. We also present compiler optimizations that are coupled with the mapping analysis. For example, in order to avoid the cost of dynamic memory allocations from inner patterns (e.g., a `Map` nested within another `Map`), we pre-allocate the space for the entire kernel and adapt the data layout to best match the mapping decision (e.g., row-major vs. column-major matrices) such that the memory accesses are optimized.

Our specific contributions are as follows:

- We present an analysis for automatically mapping nested parallel patterns onto GPUs that maximizes locality and resource utilization. The analysis is more general and flexible than previous mapping strategies which only work well in certain cases (Section IV).
- We present compiler optimizations that interact with the mapping analysis to further improve performance. The cost of dynamic memory allocations from inner patterns can be avoided and shared memory can also be efficiently utilized for data sharing and prefetching (Section V).
- We implemented a set of benchmark applications and show that our mapping analysis and optimizations generate efficient GPU code that is comparable to manually optimized CUDA code and performs better than alternative strategies (Section VI).

The rest of this paper is organized as follows. Section II briefly presents relevant background information on GPU hardware and programming models. In Section III we describe the intermediate representation (IR) of applications as the input to our analysis. Section IV explains in detail the analysis and the decision process for efficiently mapping nested patterns on GPUs. Section V presents additional optimizations to eliminate dynamic allocations and efficiently utilize shared memory. Section VI presents performance results and Section VII presents related work. We conclude in Section VIII.

II. GPU Background

This section briefly explains the components of GPU hardware and programming models that are useful for understanding the following sections. Each vendor uses different terminology for similar components, but for this paper we will use those from NVIDIA GPUs and the CUDA programming model.

GPU Hardware: Figure 4 (a) shows the typical structure of GPU hardware. A set of arithmetic units are grouped

Pattern	Description	Example
map	construct a new collection by applying a pure function to every element	in `map` {e => e + 1}
zipWith	construct a new collection by applying a pure function to every pair of elements	inA `zipWith`(inB) {(eA,eB) => eA + eB}
foreach	mutates existing collections by applying an effectful function to every element	inA `foreach` {e => if(e>0) inB(e) = true}
filter	filter elements in a collection by applying a boolean predicate	in `filter` {e => e > 0}
reduce	reduce elements from a collection by applying an associative binary function	in `reduce` {(e1,e2) => e1 + e2}
groupBy	group elements in a collection based on the keys computed from a key function	in `groupBy` {e => e.id}

TABLE I: Supported parallel patterns and their usage examples.

together to form a SIMD execution unit. Each assembly instruction defines its operation over multiple data elements called a warp (32 in NVIDIA GPUs), whereas each individual element is called a thread. To hide memory latency, multiple warps can be scheduled on a single hardware SIMD unit and context switches occur between warps for certain instructions such as memory load. Multiple SIMD units typically share resources such as the memory controller, local software-controlled shared memory, and sometimes an L1 cache. This group of hardware is called a streaming multiprocessor (SM) for NVIDIA GPUs. In order to improve DRAM memory bandwidth utilization, the memory controller is often implemented to coalesce adjacent memory requests from adjacent threads in a warp into a single bulk request. Finally to provide even higher FLOPs and bandwidth, multiple SMs are included on a GPU device (e.g., 14 for NVIDIA C2050), and all share a global device memory.

GPU Programming Model: Since GPU compute units are hierarchical (e.g, SIMD cores and SMs), programming models expose both individual threads as well as local groups of threads (called a thread block in CUDA) to programmers. A GPU kernel is written by describing the execution of each thread, and the kernel is launched with a number of threads in a block and a number of blocks. Threads within a block can be synchronized with an API call (__syncthreads()), while threads across different blocks cannot be naturally synchronized. Each thread and block has a unique logical ID, which are mapped to warps in consecutive order and therefore GPU programmers should write kernels that issue adjacent memory requests from adjacent threads to coalesce memory requests. CUDA also allows IDs to be multi-dimensional (e.g., threadIdx.x, blockIdx.y). Since SIMD units on GPUs are actually one-dimensional, multi-dimensional IDs must be mapped to one-dimensional IDs before execution. Currently in CUDA this linearization is deterministic such that indices in dimension x vary the fastest, followed by dimension y, etc. Therefore, multi-dimensional thread blocks are essentially syntactic sugar for a fixed linearization strategy (the user could also linearize the kernel manually completely equivalently). Figure 4 (b) shows how threads in a two dimensional block are mapped onto a warp.

III. Intermediate Representation (IR)

In this section, we explain the intermediate representation (IR) that we use as input to our analysis. Our IR is based on previous parallel pattern and data parallel languages [1], [5], [8], [9], [10], and can be generated from a compiler front-end. Since our analysis and optimizations are not tied to a specific front-end but rather generally applicable to any high-level language that encodes parallel pattern information in the IR, it is easy to integrate our framework into existing / new languages that perform GPU code generation, without modifying their language.

```
1  nodes map { n =>
2    nbrsWeights = n.nbrs map { w =>
3      getPrevPageRank(w) / w.degree
4    }
5    sumWeights = nbrsWeights reduce { (a,b) => a + b }
6    ((1 - damp) / numNodes) + damp * sumWeights
7  }
```

Fig. 5: Snippet of PageRank implemented in our language using nested patterns. For each node `n` (outer `map`), calculate its neighbors' weights (inner `map`) and aggregate them (inner `reduce`).

Expressions in the IR are either basic sequential statements (allocations, control flow, primitive arithmetic, logical tests) or instances of a parallel pattern. Table I lists the parallel patterns currently supported. Each pattern requires a user-defined function to specify the body of the computation. This function can include any of the language constructs, thereby allowing patterns to be nested. The structured nature of these patterns provides information like the inherent parallelism, internal communication, and write-sets that are required for the mapping analysis. Having this high-level information directly from the IR is one of the major differences from other systems without patterns (e.g., using polyhedral analysis [11], [12]) that cannot easily optimize patterns such as `Filter`.

For data structures, we currently support scalar types, arrays, and structs. Structs can compose other primitive data structures, which allows a rich set of higher level data structures in the language or in the application. For example, a graph data structure can be implemented with a struct of three arrays (CSR format).

To demonstrate our work, we implemented a data parallel language that provides a thin wrapper around the IR described above. Figure 5 shows pseudocode demonstrating how the PageRank [13] algorithm can be implemented in our language using nested patterns.

IV. Mapping Analysis

In this section, we present an analysis that determines an efficient mapping for nested patterns. A mapping result is generated for each *level* of a nested pattern. A *level* defines how deep each pattern is from the outermost enclosing pattern, starting from level 0 for the outermost pattern and incrementing the number for each nest. For example, Figure 5 shows a

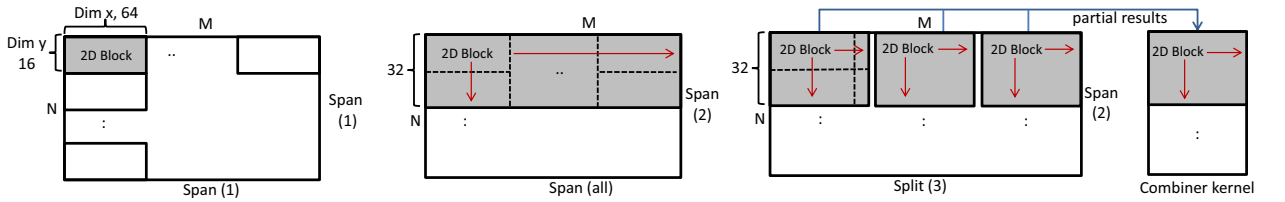


Fig. 6: Example of assigning different Span/Split types on each dimension. The gray area specifies the index domain a block covers. (a) *Span(1)* on both dimensions (b) *Span(all)* on Dim x and *Span(2)* on Dim y (c) *Split(3)* on Dim x and *Span(2)* on Dim y. The number of blocks launched are (a) $M/64 \times N/16$ (b) $N/32$ (c) $3 \times N/32$

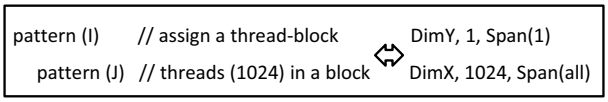
two level nested pattern with a `Map` pattern at level 0 and a `Map` and `Reduce` pattern both at level 1.

A. Mapping Parameters

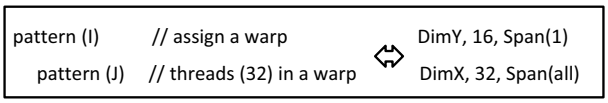
A mapping decision for a nested pattern consists of the following three parameters at each level.

- **Dimension:** A unique logical dimension (e.g., x, y, etc) is assigned to each nest level.[3] Logical indices for each dimension are eventually translated to thread indices on the GPU. Since GPU hardware coalesces adjacent memory requests issued from adjacent threads (threads in a warp), the translation should be done such that a dimension that contains adjacent memory requests is assigned threads with adjacent indices. This translation can be done either manually or by utilizing the multidimensional kernel abstraction of the GPU programming model.
- **Block size:** Block size specifies the number of threads for a given dimension in a CUDA thread block. The total number of threads in one thread block is then simply the product of the number of threads in each dimension. There is a maximum number of thread blocks that can run in parallel within each SM, and therefore the resources may be underutilized if the thread block size is too small (e.g., less than 64).
- **Degree of Parallelism:** We refer to degree of parallelism (DOP) as the number of parallel computations enabled by a particular mapping. For a given dimension and block size decision, the DOP can be controlled by specifying its *Span*, the portion of the index domain covered by the block. Figure 6 shows a block of size 64×16 and the two dimensional index domain of size $M \times N$. The gray area indicates the index domain a block covers. We define *Span(n)* to mean that a single thread covers n points in the index domain. For example, if we have a two dimensional domain with disjoint accesses one common parallelization strategy is to assign one element in the domain to each thread, launching $M/64 \times N/16$ blocks as shown in Figure 6 (a). We therefore specify this by assigning *Span(1)* for both dimensions. Since all the elements in the index domain are parallelized, the DOP is $M \times N$. By increasing the span factor to *Span(n)* in one of the dimensions, thereby assigning n elements to each thread, we can decrease the DOP by a factor of n. The reason for doing this is because (1) there is a limit on the number of blocks that can be launched and (2) threads can reuse the result of statements inside the outer pattern while iterating over the inner pattern multiple times.
  A useful special case with minimal DOP is *Span(all)*, where only one block is launched for the dimension. In this case all the indices in the dimension are covered by threads in a single block. There are two instances where we assign *Span(all)* to a specific dimension. The first is when the size of the dimension is not known at the time of kernel launch (e.g., when the size of an inner pattern is calculated dynamically). The second is when the data accesses for the pattern in that dimension requires a global synchronization to produce final results (e.g., `Reduce`), since synchronization across thread blocks is not supported on GPUs. Figure 6 (b) shows a different assignment of *Span* types on the same index domain as in Figure 6 (a). DOP for this case is $64 \times (N/2) = 32N$. Since *Span(all)* can reduce DOP too much to saturate the GPU we introduce *Split(n)*, which splits the dimension into n blocks at the cost of inter-block synchronization. *Split(n)* can only be applied when *Span(all)* was assigned to satisfy the synchronization requirements, and requires launching a subsequent combiner kernel to globally synchronize partial results from each split section. Figure 6 (c) shows an example of applying *Split(3)* to Figure 6 (b) on dimension x, which increases DOP by a factor of 3.

[3]Logical dimensions are not necessarily tied to multidimensional thread blocks of GPU programming models since the number of logical dimensions can be arbitrary depending on the nest level. This allows easily exposing more parallelism with more logical dimensions.



(a) Thread-block / thread mapping and its equivalent mapping



(b) Warp-based mapping and its equivalent mapping

Fig. 7: Effectively equivalent mappings to strategies in previous work. (a) *thread-block/thread mapping* (b) *warp-based mapping*. I and J represent the size of each pattern.

B. Mapping Coverage and Flexibility

Figure 7 shows how the mapping strategies from previous work discussed in Section I (*thread-block/thread mapping* and

Weight	Scope	Constraint
Hard	Local	For patterns that require global synchronization (e.g., `Reduce`), assign *Span(all)*
Hard	Global	Pick the most conservative span for all patterns for each level (*Span(all)* is more conservative than *Span(n)*)
Soft	Local	For patterns that generate sequential memory requests, assign *Dim(x)* and block size multiple of $WARP\_SIZE$
Soft	Global	Threads in block (combining all dimensions) is greater than MIN_BLOCK_SIZE (64)

TABLE II: Examples of constraints used for our mapping analysis.

warp-based mapping) are covered by the mapping space of our analysis.

Thread-block/thread mapping: Figure 7 (a) shows an example of mapping each iteration of the outer pattern to a thread block and the inner pattern to threads within the thread block (MAX_BLOCK_SIZE 1024). The effectively equivalent mapping based on our parameters is specified on the right, with the DOP of $I \times min(J, MAX\_BLOCK\_SIZE)$ since the parallelism of the inner loop is limited to MAX_BLOCK_SIZE.

Warp-based mapping: Figure 7 (b) shows an example of mapping each iteration of the outer pattern to a warp and the inner pattern to threads within the warp. And the effectively equivalent mapping shown on the right[4] has DOP of $I \times min(J, WARP\_SIZE)$.

Discussion: Re-expressing *thread-block/thread mapping* and *warp-based mapping* in terms of the general parameters of our analysis, we see that despite their apparent differences, both strategies are fundamentally quite similar. We can emulate either strategy using the same logical dimension and span mapping just by varying the block size.

As the similarity in their mappings implies, both strategies have similar problems. First, they do not take into account the access pattern of the kernel and may result in poor locality. For example, if the memory accesses are sequential relative to the indices of the outer pattern (e.g., $sumCols$ example in Figure 1), then both mappings cannot coalesce memory requests since the inner pattern indices are already assigned to warps. With our analysis, just switching the dimension assignment of the patterns allows coalescing. Second, since the block size of the inner dimension is fixed, both mappings can suffer from resource underutilization when the loop sizes are skewed (e.g., inner loop is too small while outer loop is large). With our analysis, the DOP can be easily controlled by changing the span type and block size parameters so that enough threads can be launched. Finally, if the pattern was triply nested, both *thread-block/thread mapping* and *warp-based mapping* can only take advantage of two levels of parallelism while our analysis only needs to add one more logical dimension to exploit the parallelism at all levels.

These examples clearly show the flexibility of our mapping analysis. Rather than having a fixed mapping strategy as *thread-block/thread mapping* and *warp-based mapping*, we allow the mapping parameters (dimension, block size, span/split) to be flexibly changed to optimize locality, parallelism, and resource utilization. Our mapping parameters also provide a better view of the similarities and differences between different mapping strategies. Finally, our mapping parameters are general enough to cover various mapping strategies in previous work. Therefore, our mapping parameters can be used by other compiler or auto-tuners to explore the mapping space.

[4]For the block size of the outer pattern, other choices are also possible, but 16 was chosen to have enough total threads in a block.

C. Constraints

We use an analysis to add a variety of constraints which inform the automatic selection of an efficient mapping. While traversing the IR, mapping constraints are added based on the memory accesses within the pattern as well as the type and size of the pattern. We also add additional constraints to model restrictions in the hardware and CUDA programming model. We then calculate a score for each potential mapping based on the constraints it satisfies, and choose the mapping with the highest score. Constraints can be grouped by two orthogonal categories: scope (**Local / Global**) and weight (**Hard / Soft**). Table II shows examples of constraints and their categories.

Local Constraints are constraints applied to a specific pattern in a nested pattern structure. For example, when the memory accesses in a pattern are found to be sequential, we add a local constraint that the corresponding pattern should be mapped to the logical dimension *x* (the fastest varying dimension by convention), and the size of the block in dimension *x* should be a multiple of $WARP\_SIZE$ to maximize memory coalescing.

Global Constraints are constraints that need to consider multiple patterns together. For example, when there are multiple patterns at the same level, the span type for the level should be the most conservative one required for any pattern at that level(e.g., *Span(1)* and *Span(all)* at a same level results in *Span(all)* for that level).

Hard Constraints are constraints that must be satisfied for correct execution. Restrictions arising from the target GPU or programming model are often added as hard constraints, such as the maximum number of threads for a block and for each dimension.

Soft Constraints provide performance hints; the execution will still be correct even when soft constraints are violated. We therefore associate a weight with each soft constraint and satisfying a soft constraint increases the total score of the mapping. We selected a set of common optimizations GPU experts often apply in order to maximize bandwidth utilization, avoid thread divergence, and provide enough parallelism. Each soft constraint has an intrinsic weight associated with it that indicates the relative importance of the constraint based on our observations on the performance impact. In particular, the applications written using parallel patterns are often bandwidth limited, and therefore we assign the highest intrinsic weight on the soft constraint that allows memory coalescing. The intrinsic weight of each soft constraint is then multiplied by the number of times the related code will be executed (e.g., enclosing loop size(s)), and discounted by any enclosing

```
1  Pattern1 with i in Domain(0,I) {
2    array1D(i)     #weight: α*I
3    Pattern2 with j in Domain(0,J) {
4      array2D(i,j)   #weight: α*I*J
5  } }
```

Fig. 8: Constraints with different derived weights based on the size of patterns. `I` and `J` represents the size of each pattern and α represents the intrinsic weight of the constraint.

branches (e.g., if it will only be executed in the `Then` branch of an `IfThenElse` statement). Therefore, the mapping decision prioritizes the more important soft constraints within the same loop nest level but a deeper loop nest still has the most power to gear the decision. When the size of a pattern is not known as a constant value during analysis, a default size is assumed (1000 by default), but users can provide the size information from the application to enable better optimization. Note that in many cases the absolute size values are actually not necessary since inner patterns will always execute strictly more times than outer patterns, and our strategy naturally gives higher priority to more deeply nested patterns.

Figure 8 shows a simple example of calculating soft constraint weights with nested patterns. Both `Pattern1` and `Pattern2` add a constraint that it should be assigned to dimension *x* because of the array accesses in line 2 and 4, respectively, but it is not possible to satisfy both. However, since `array2D(i,j)` (line 4) is accessed more often than `array1D(i)` (line 2) by factor of J, the constraint on `Pattern2` will end up with a higher derived weight, thereby giving the mapping that assigns `Pattern2` to dimension *x* a higher total score than the mapping that assigns `Pattern1`.

D. Search for an Efficient Mapping

Once all of the constraints have been added during IR traversal, a search process for an efficient mapping begins, as described in Algorithm 1. The input to the search process is a set of constraints added for each level ($CSet$). First, all of the candidate mappings are created by constructing the possible combinations that satisfy hard constraints ($CandSet$). For *span* parameters, only *Span(1)* or *Span(all)* are assigned initially, which are replaced with *span(n)* or *split(k)* when manipulating DOP later. The search space is exponential to the level of loop nests with the base of $|DimSet| * |SizeSet| * |SpanSet|$ (less than 100), and for typical loops (1 to 3 levels) it takes less than a few seconds for brute-force search to find an efficient mapping.

For each candidate mapping that satisfies hard constraints, we iterate over the soft constraints in $CSet$ and add the weight of the constraint to the mapping score if the constraint is satisfied. If the score is higher than the previous best score, we assign the current mapping and score to be the best one. When the score is the same as the previous best score, the one with higher DOP value is selected, and if the DOP values are also the same, we pick randomly. After all candidates are enumerated, we check DOP criteria (procedure $ControlDOP$) on the best scored mapping. This procedure is parameterized by two values, MIN_DOP and MAX_DOP, which specify

Algorithm 1 Efficient Mapping Search Process

```
1:  Input: CSet: Map[level, Set[Constraint]]
2:  DimSet ::= {x , y , z , w , .. }
3:  SizeSet ::= { 1, 2, 4, 8, .. , 1024 }
4:  SpanSet ::= { Span(1), Span(all) }
5:
6:  procedure CONTROLDOP(mapping)
7:      if CURRENT_DOP of mapping < MIN_DOP then
8:          k = MIN_DOP / CURRENT_DOP
9:          Span(all) ↦ Split(k)
10:     if CURRENT_DOP of mapping > MAX_DOP then
11:         n = CURRENT_DOP / MAX_DOP
12:         Span(1) ↦ Span(n)
13:
14: # returns Map[level, (dim,size,span/split)]
15: procedure SEARCH
16:     CandSet = {(level,(dim,size,span)) | level ∈ CSet.keys,
17:     dim ∈ DimSet, size ∈ SizeSet, span ∈ SpanSet}
18:     bestScore = 0; bestMapping = NULL;
19:     # filter out candidates by hard constraints
20:     for mapping ← candidate filtered by hard constraints do
21:         score = 0
22:         for constraint ← soft constraints do
23:             if mapping satisfies constraint then
24:                 score += constraint.weight
25:         if score > bestScore then
26:             bestScore = score; bestMapping = mapping;
27:     # control degree of parallelism
28:     ControlDOP (bestMapping);
```

the minimum and maximum number of threads that should be launched for a particular device. For example, Tesla K20c GPU has 13 SMs with maximum of 2048 threads per SM, and therefore we set MIN_DOP of $13 * 2048$ to provide enough parallel threads and MAX_DOP of $100 * MIN\_DOP$ to limit the number of thread blocks. If the DOP is lower than MIN_DOP, we increase it by replacing *Span(all)* to *Split(k)*, and if the DOP is greater than MAX_DOP we decrease it by replacing *Span(1)* with *Span(n)*. Note that *span(all)* contributes to DOP not in terms of its loop size but in terms of the block size, making DOP calculation less sensitive to heuristic values (e.g., 1000 for statically unknown loop size). Also, once the mapping decision that determines the generated code structure is made (e.g, dim *x*,*y*,... or *span/split*), certain parameters are adjusted at runtime using the actual loop size. For example, the block sizes and *span/split* factors can be dynamically changed without recompiling the code. Having both static decision (global) and dynamic decision (local) minimizes the runtime decision overhead while allowing fine tuning based on dynamic values.

E. Code Generation

Once the mapping decision has been made, the code generation process is relatively simple. The code generator has a set of high-level templates for each pattern. Just having a fixed template for each pattern is not sufficient since different mapping decisions may require different code structures, not just changing the launching parameters (e.g., number of threads). For example, Figure 9 shows the generated code for $sumRows$ in Figure 1, and the mapping decision specified at the top. For

```
1  // Level 0: [DimY, size 64, span(1)]
2  // Level 1: [DimX, size 32, span(all)]
3  __global__ kernel(double *m, int cols, double *out) {
4    int y = threadIdx.y + blockIdx.y * blockDim.y;
5    __shared__ double smem[64][32];
6    double local_sum = 0.0;
7
8    for (int cidx = threadIdx.x; cidx < cols; cidx += 32)
9      local_sum += m[y*cols + cidx];
10   smem[threadIdx.y][threadIdx.x] = local_sum;
11   __syncthreads();
12
13   /* reduce 32 values in smem[threadIdx.y][*] */
14
15   if(threadIdx.x == 0) out[y] = smem[threadIdx.y][0];
16 }
```

Fig. 9: Generated CUDA code for $sumRows$.

```
1  collection map { e =>                //size M
2    // requires allocation for each e
3    res = e map { // some function }  //size N
4
5    // uses res
6    res reduce { // some function }
7  }
```

Fig. 10: Example of dynamic allocation for inner patterns. Each parallel thread must perform a local allocation to compute `res`.

this mapping decision, the inner reduce pattern in Figure 9 uses shared memory to combine data across threads (line 13, well known warp synchronous programming technique, omitted for brevity), which would not be needed if each inner reduce pattern was not parallelized. If the mapping for the reduce pattern was *split(k)*, global memory is allocated for partial outputs and another kernel is generated for the final reduce operation across blocks. Therefore, depending on the mapping decision our code generator selects the appropriate template for the pattern and generates complete CUDA code.

V. Optimizations

In this section, we describe compiler optimizations that further improve the performance of nested patterns. These optimizations are well-known and often manually applied by GPU programming experts, and here we show how they can instead be applied automatically as well as how they interact with the mapping analysis.

A. Dynamic Memory Allocation

A common characteristic of functional parallel pattern languages is that they tend to allocate more memory than hand-optimized code. Therefore when parallel patterns are nested, inner patterns may require dynamic memory allocations. Figure 10 shows a nested pattern where both the outer pattern and the inner pattern are `Map`, which is a natural way people use nested patterns. Since the outer pattern is parallelized, possibly launching thousands of threads, the overhead of calling `malloc` per thread to store the result of the inner `Map` is significant.

However, when the size of the allocation is the same across all of the outer loop iterations, we can avoid the cost of dynamic allocations by preallocating the space for the entire outer pattern (i.e., size $M \times N$) at once before launching the kernel. In addition, we can also choose an efficient layout for the allocation based on the mapping decision. Consider two possible mapping decisions for the code example in Figure 10. When dimension *y* is assigned for the outer `Map` pattern and dimension *x* for the inner `Map`, the optimal layout of the allocation is shown in Figure 11 (a). On the other hand, when the dimension assignments are reversed (outer `Map` on dimension *x*), the optimal layout for the allocation is Figure 11 (b). Therefore, by rewriting accesses in the kernel to use the proper offset and stride values for each mapping decision as shown in the figure, the memory accesses can be coalesced for both situations. We currently only apply this transformation when the inner allocation(s) do not escape the kernel. It can also be applied when the allocation escapes, but then either subsequent kernels must be rewritten to conform to the selected data layout or the data must be copied back to the expected layout, which is not always faster in general.

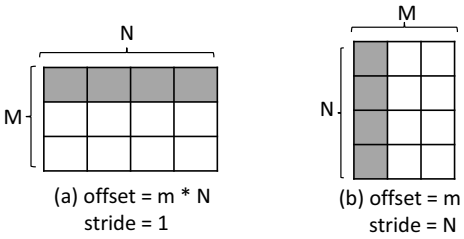


Fig. 11: Two different layouts for the allocations in Figure 10 optimized for different mapping results. The gray area specifies the area assigned for the first iteration of the outer `Map`, and `m` is the iteration index.

Note that this optimization provides a way to satisfy more soft constraints for a mapping. Freedom in the physical layout of the preallocated memory relaxes the constraints that were imposed by the logical access patterns. Therefore, the mapping analysis prioritizes other non-flexible constraints when searching for an efficient mapping and then determines the physical layout after the mapping.

B. Shared Memory

When patterns are imperfectly nested (i.e., memory accesses exist outside the innermost pattern), any multidimensional kernel will have idle threads when computing the outer level(s). Considering the example in Figure 8, a simple two-dimensional strategy is to launch $I * J$ threads to cover the index domain. While all $I * J$ threads are active to compute line 4, only I threads are needed to compute line 2. Therefore, during this portion of the execution the remaining $I * J - I$ threads will be idle. This is typically accomplished by adding a guard in the CUDA code so that only one thread in a dimension computes the outer level operations and then all the threads synchronize before using the result of that thread's computation. In addition, for situations such as Figure 8 where multiple levels have memory accesses that could in theory be coalesced, but the mapping strategy can only satisfy one level, the memory accesses for the outer level computation will be suboptimal.

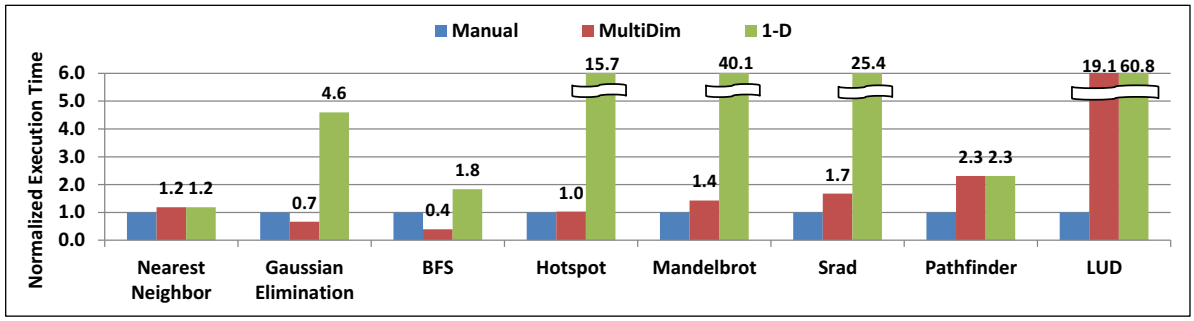


Fig. 12: Performance of Rodinia benchmark applications compared to manually optimized and one-dimensional implementations. Execution times are normalized to manual.

We can solve both of these problems by exploiting the GPU's software-controlled shared memory, which provides fast random access within a thread block. When the analysis detects imperfectly nested loops, we automatically implement a form of prefetching. Multiple threads in one dimension all read a contiguous chunk of the data at the outer level and store it in shared memory. This solves the underutilization problem because we fetch data for multiple outer loop iterations simultaneously using multiple threads. In addition, by using dimension *x* to perform the prefetch, this also results in coalesced memory accesses for both the outer pattern (which was assigned to dimension *y*) and the inner pattern (assigned to dimension *x*). Once the kernel enters the inner parallel computation, each thread can access any element of the loaded chunk quickly from shared memory and the kernel performs multiple iterations of the outer loop without accessing main memory again.

VI. Experiments

A. Rodinia Benchmarks

In this section we present performance results of multiple benchmarks and applications running on a system with a multi-core CPU and GPU. For evaluation, we implemented applications in our data parallel language described in Section III. The compiler traverses the IR to perform the mapping analysis as described in Section IV and then emits CUDA kernels corresponding to the optimal mapping. We refer to this implementation as "*MultiDim*" in the experimental results.

B. Methodology

Our experiments were run on a Dell Precision T7500n with two quad-core Xeon 2.67 GHz processors, 96GB of RAM, and an NVIDIA Tesla K20c. CUDA v5.0 is used to compile the generated kernels. For each experiment we measure the total execution time across all kernels, ignoring the cost of transferring the input data to the GPU's main memory except in Section VI-E.

C. Simple Nested Patterns

We first quickly wrap up our running example of $sumRows$ and $sumCols$ from Sections I. As previously discussed, fixed execution strategies are sensitive to both the data dimensions as well as the data access patterns. However our analysis automatically selects a strategy that is optimal for each variant. The execution times in Figure 3 are all normalized to our strategy's execution time, which is the same value for every data size for both $sumRows$ and $sumCols$. Note that this is the ideal case since the total number of data elements is held constant, and when mapped correctly all main memory accesses are sequentially adjacent.

We selected a set of applications from the Rodinia benchmark suite [4] that contains nested parallelism. We rewrote each application shown using our data parallel language and present the execution time of the automatically chosen mapping strategy. We compare our results to a *1D mapping* strategy as well as to the hand-optimized implementations provided by Rodinia. To generate the *1D mapping* implementations, we used our same applications and compiler, and just added a directive that forces the compiler to ignore all but the outermost level of parallelism.

Applications in Rodinia consist of one or two levels of parallelism. Nearest Neighbor contains only one dimension of parallelism, and we include it here as a baseline to see how efficient our generated code is to hand-written CUDA in general. As shown in Figure 12, our generated code is 20% slower than manually-optimized. The primary reason is that we currently generate class wrappers for multidimensional arrays that have multiple extra fields (e.g., offset, stride), and the physical index is dynamically calculated at each access from the logical index and the fields. The manual code on the other hand uses raw pointers and avoids excess index calculations. We believe this can be optimized by specializing the data structures we use to minimize the overhead. Since this is a one-dimensional application the *1D mapping* strategy of course produces identical performance.

For Gaussian Elimination and Breadth First Search (BFS), our compiler actually generated better code than the hand-optimized versions provided by Rodinia. For Gaussian Elimination, one of the two-level nested patterns was not written to coalesce memory accesses in the manual version, whereas our analysis figured it out automatically and assigned the proper dimension for each level. BFS also benefits from our analysis as the manual code was only written to take advantage of the top-level parallelism (each node in the graph but not the neighbors of each node), equivalent to the *1D mapping*. Our analysis, however, chooses to parallelize both the outer and

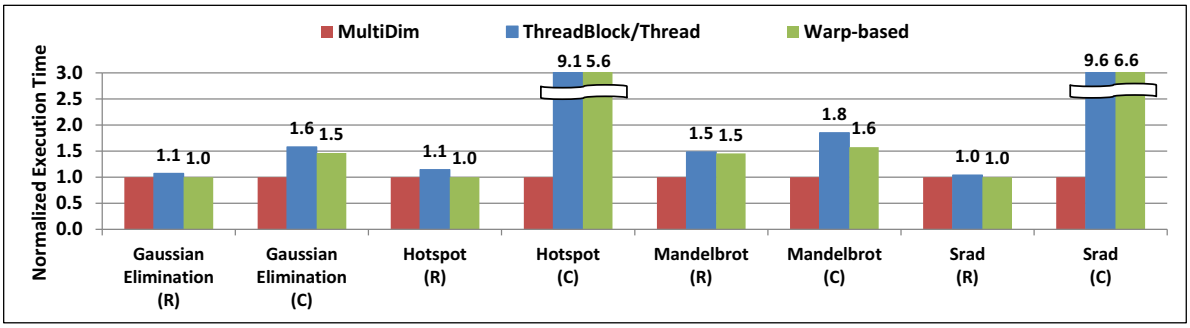


Fig. 13: Performance of a subset of Rodinia benchmark applications compared to previous two-dimensional strategies. Execution times are normalized to our *MultiDim* results.

the inner patterns. This produces better load balancing across nodes with varying numbers of neighbors, and therefore better overall performance. These two examples clearly show that expert GPU programmers can make incorrect decisions. We believe such mistakes are more likely in more complicated applications, and therefore we see the benefit of our automated approach.

Hotspot, Mandelbrot, and Srad all perform very poorly with a *1D mapping* strategy compared to the optimized code. Only parallelizing the outer pattern results in the kernel under-utilizing the GPU's resources. By exploiting the parallelism at multiple levels, however, the size of each individual pattern does not need to be very large as long as the total inherent parallelism in the application (the product of the parallelism at each level) is large enough to fully utilize the GPU's resources. Also parallelizing only the outer pattern can yield non-coalesced memory accesses depending on how the application is written. Whereas our analysis is tolerant to how the application is written by assigning the appropriate dimension to produce coalesced accesses. As shown in Figure 12, our analysis produces code that performs comparably to the manually-optimized code for each of the three applications.

Pathfinder and LUD show significant performance differences even when using our analysis. Both applications have a common computation pattern. They are iterative applications where each iteration performs a stencil-based data-parallel operation, and the input to each iteration is the result from the previous iteration. The manually-optimized code combines multiple iterations of data-parallel operations into a single kernel call and uses the GPU's shared memory to store intermediate results. Since computing each new value requires reading multiple overlapping previous values (the stencil), the implementation trades off work duplication for fewer main memory accesses in order to combine multiple iterations into one CUDA kernel. This is described in more detail by Rodinia [4]. The stencil uses are quite complicated and it is unclear how to automatically infer the stencil from the application and automatically map it to shared memory without more domain-level or application-level knowledge. In addition we currently have a one-to-one mapping of outer-level parallel patterns to kernels, and do not attempt to make any trade-off between fusing multiple patterns into a single kernel at the cost of work duplication (this is obviously not always correct in general).

D. Comparison to Fixed Two-Dimensional Strategies

We next compare the performance of our multidimensional mapping strategy to the *thread-block/thread mapping* and *warp-based mapping* strategies on the Rodinia benchmarks in Figure 13. In order to isolate the performance differences caused by the strategies themselves, we use our same compiler to generate CUDA kernels but manually select the mappings listed in Figure 7. Among the applications in Figure 12, we selected a subset that can be naturally written in two ways; a row-major traversal order (R) and a column-major traversal order (C). [5] For the row-major implementations all three strategies give relatively similar performance, but our strategy performs at least as well as the others in all cases, providing up to 50% speedup over the fixed strategies. The main reason for the performance improvement is that our strategy exploits all available parallelism in all dimensions, whereas the other strategies have a fixed degree of parallelism in the inner dimension. For the column-major traversal, the performance difference is much more significant since the fixed strategies cannot automatically adapt to properly coalesce memory accesses, which produces between 1.5x and 9.6x slowdown compared to the *MultiDim* strategy. Therefore, our flexible mapping strategy provides significant benefit to application programmers since they are no longer required to write their application in a specific way to maximize the performance on different targets.

E. Application Case Studies

In these experiments, we demonstrate that our multidimensional mapping strategy is also applicable to and performs well on real-world applications, shown in Figure 14. We compare the kernels generated from the multidimensional mapping both to a simple one-dimensional mapping and to the original optimized multi-core CPU reference implementations.

We first consider QPSCD HogWild!, which is a quadratic programming solver implemented using a lock-free version of stochastic coordinate descent [14]. The application is stochastic such that the main outer pattern is over random rows, while

[5]For the rest of the benchmarks, our compiler performs at least as well as fixed strategies; warp-based mapping was within 10%, thread-block/thread mapping was 3-4x slower in some cases.

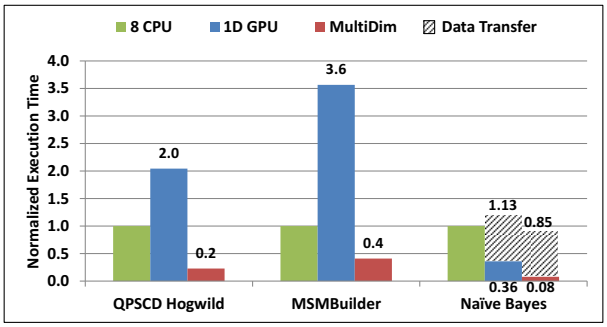


Fig. 14: Performance of real-world applications compared to multi-core CPU and one-dimensional implementations. Execution times are normalized to multi-core.

the inner pattern iterates over a row sequentially. Since the outer access pattern is random, parallelizing only the outer pattern (*1D mapping*) results in memory requests that cannot be coalesced, leading to the performance worse than the CPU. In contrast, *MultiDim* assigns the sequential inner pattern to dimension *x*, which results in a speedup of 4.38x over the multi-core implementation and 8.95x over *1D mapping*.

The next application we consider is MSMBuilder [15], an open source software package for accelerating molecular dynamics simulations and building Markov State Models (MSMs). We implemented the performance-critical trajectory clustering portion of this code and applied *MultiDim* to it. The reference multi-core C++ implementation we compare against is heavily hand-optimized (including using manual SSE3 intrinsics). The results for MSMBuilder look similar to QPSCD, but for different reasons. In this application one of the nested patterns has a relatively small domain in each dimension (around 100 elements each) therefore the *1D mapping* strategy greatly underutilizes the GPU's hardware. *MultiDim* however parallelizes over the product of the domain, producing significantly better hardware utilization, and achieves a speedup of 2.4x over the multi-core implementation and 8.7x over *1D mapping*.

The last application is a spam document classifier using the Naïve Bayes model. Given a training dataset (a matrix, each row representing words in a document), the application calculates both the number of words per document and the number of documents for each spam/non-spam word. Therefore there exists different data access patterns on the same training data. *1D mapping* can only satisfy one of the access patterns while *MultiDim*'s flexibility enables data accesses from all the GPU kernels in the application to be optimized, achieving a speedup of 12.5x over the multi-core implementation and 4.5x over *1D mapping*. This application includes significant overhead for the input data transfer cost, whereas the iterative nature of the previous two applications amortized this overhead. As shown in Figure 14, *MultiDim* is 15% better than multi-core when including the input data transfer time.

F. Optimizing Dynamic Memory Allocations

All previous results utilized the optimizations presented in Section V where applicable, and now we study the impact

```
1  // m: matrix of size [R,C]
2  // v: vector of size [C]
3  sumWeightedCols = m mapCols { c =>
4    temp = c zipWith v { (a,b) => a * b }
5    temp reduce { (a,b) => a + b }
6  }
```

Fig. 15: Modified version of $sumCols$ to multiply a weight vector before the sum. The `ZipWith` pattern allocates memory for its output, which occurs per outer pattern iteration.

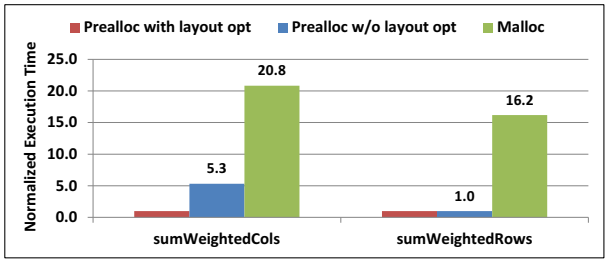


Fig. 16: Performance impact of optimizing dynamic allocations. Execution times are normalized to the fully optimized execution

of these optimizations in isolation on a micro-benchmark. Figure 16 shows the impact of optimizing the usage of dynamically allocated memory in nested patterns. For this experiment, $sumRows$ and $sumCols$ is slightly modified to multiply a weight vector before reducing each row or column, where $sumWeightedCols$ is shown in Figure 15. In this example multiplying the weight vector logically creates a new allocation within each outer pattern iteration. As discussed in Section V-A, we can eliminate the thread-local allocations by preallocating memory for the entire kernel upfront and assigning each thread a unique region that it can use within the kernel. For a fixed allocation layout strategy (we chose row-major for this experiment), this optimization provides over 16x speedup for $sumWeightedRows$, but only 4x speedup for $sumWeightedCols$ since the access patterns to the temporary memory are not properly aligned. Each row of the allocated matrix is given to each outer pattern iteration, which cannot be coalesced since the analysis assigns dimension x to the outer pattern. However, by using the mapping information we can choose the proper layout based on the access patterns, as shown in Figure 11 (b), and this provides an additional 5x speedup for $sumWeightedCols$. After choosing the optimal layout of temporary memory for each version, both $sumWeightedRows$ and $sumWeightedCols$ execute in the same amount of time for a given input size as expected.

G. Performance and Mapping Scores

Figure 17 shows the performance and the mapping score for various mappings generated from our framework. The Mandelbrot benchmark is used for this experiment, and we chose the output matrix to be skewed (50,20K) to show the flexibility of our framework. We can see a region of high scored mappings (region A) that provide the best performance, and our framework manages to produce a mapping in this region. The reason for the region of similar performance is

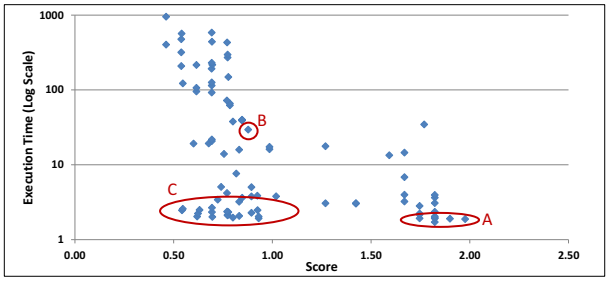


Fig. 17: Performance and score of various mappings generated from our framework. (A: best performance region, B: warp-based mapping, C: false negatives)

that minor variations in the mapping decision (e.g., block size of [128,4] or [256,2]) only have a small impact on the performance. We can also see that even when the size is skewed, which is not known at compile time, the system can dynamically select the kernel launch parameters based on the runtime size to maximize performance. In contrast, warp-based mapping (region B) shows poor performance due to its fixed strategy, which leads to resource underutilization.

Although our analysis selects a mapping in the best performance region by trying to satisfy soft constraints as much as possible, there are still false negatives (region C: low score and high performance). The main reason is that we used fixed intrinsic values for the soft constraints and the scoring calculation only considers loop counts and branching behavior. In order to have a more sophisticated scoring policy, the intrinsic values need to be fine-tuned based on the application characteristics such as the arithmetic intensity and memory access patterns. There has been recent work on analytical models for GPU performance estimation [16], [17], and integrating such a model into our framework is a subject of future work.

VII. Related Work

Compiling high-level languages to GPUs: Many previous systems have chosen to automatically target GPUs from a higher-level programming environment. Nikola [2] is an array language embedded in Haskell which uses type-directed techniques and parallel patterns to automatically translate from Haskell to CUDA. Nystrom et al. [6] use a library-based approach for translating Scala programs to OpenCL via Java bytecode translation. Thrust [5] is a C++ library that simplifies many of the low-level details of CUDA programming and also provides a data-parallel pattern API that automatically creates and launches CUDA kernels. Other systems designed specifically to simplify the CUDA programming model include hiCUDA [18] and CUDA-lite [19]. Yang et al. [20] presents a compiler that optimizes naive GPU programs. Sponge [21] is a compiler which generates CUDA code from the StreamIt [22] programming language. Udupa et al. [23] also target StreamIt for GPUs. Jablin et al. [24] focused on optimizing CPU-GPU communication in auto-parallelized C/C++ code using runtime APIs. Lime can compile to both Verilog for FPGA accelerators [25] and OpenCL for GPU and multi-core execution [26]. Delite [8] is a compiler infrastructure for building parallel DSLs that is based on parallel patterns and targets multi-core CPUs and GPUs. Dandelion [27] also automatically generates code for both CPUs and GPUs from a data-parallel subset of C#. All of these systems are similar in that they do not present a strategy for automatically handling nested parallel patterns efficiently.

Nested parallelism on GPUs: Copperhead [1] generates CUDA code automatically from a data-parallel subset of Python, and is capable of exploiting nested parallelism by mapping the outer parallel computation to CUDA thread blocks and the inner parallel computation threads within a thread block. However, as discussed in this paper such a fixed mapping strategy is not always sufficient to maximize performance. In addition the strategy is limited to a single level of nesting / can only exploit two dimensions of parallelism. Hong et al. [7] take a more domain-specific approach and optimize specifically for nested graph computations on datasets with high skew, whereas our system can replicate their strategy in a more general framework. NESL [28] flattens nested parallelism, which can be an effective strategy to improve load balancing across the (formerly) inner loop. However, for hardware designs that are naturally hierarchical like GPUs flattening is not necessarily the best strategy and incurs additional overheads. Nested parallelism on GPUs has also been investigated using polyhedral models [11], [12], tiling, and mapping different loops onto threads and thread blocks. However, the applications are written using affine loops, which makes it difficult to recover high-level semantic knowledge like parallel patterns, and therefore limits their possible mapping strategies (e.g., a reduction or filter using multiple kernel launches cannot be generated). CUDA-NP [29] tries to parallelize inner loops using OpenMP style pragmas and automatically generates different mappings, but it still requires programmers to write CUDA code.

VIII. Conclusion

In this paper we presented an analysis framework for automatically and efficiently mapping nested parallel patterns onto GPUs. Our analysis maps nested patterns onto a logical multidimensional domain space and includes parameters to control the GPU block size and degree of parallelism. We then add constraints based on the types of patterns and internal memory accesses and find the mapping that best satisfies the given constraints in the search space. Previous strategies for executing nested patterns on GPUs can be seen as using a fixed set of parameters in our available search space, whereas our system can adapt to run a broader range of nested patterns efficiently on GPUs. We also presented additional optimizations to eliminate dynamic memory allocations and utilize the GPU's shared memory to improve the performance of the selected mapping strategy. We showed that the performance of our automatically selected mappings is competitive with hand-optimized implementations in most cases and up to 28x faster than a simple 1D mapping. Finally we show experimentally that our strategy always performs at least as well (and in certain cases significantly better) than previous solutions due to its inherent greater flexibility.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers, Nithin George (EPFL), and Wonchan Lee (Stanford) for their comments and valuable suggestions. This work is supported by DARPA Contract-Air Force, Xgraphs; Language and Algorithms for Heterogeneous Graph Streams, FA8750-12-2-0335; Army Contract AHPCRC W911NF-07-2-0027-1; NSF Grant, BIGDATA: Mid-Scale: DA: Collaborative Research: Genomes Galore - Core Techniques, Libraries, and Domain Specific Languages for High-Throughput DNA Sequencing, IIS-1247701; NSF Grant, SHF: Large: Domain Specific Language Infrastructure for Biological Simulation Software, CCF-1111943; Dept. of Energy- Pacific Northwest National Lab (PNNL)-Integrated Compiler and Runtime Autotuning Infrastructure for Power, Energy and Resilience-Subcontract 108845; NSF Grant- EAGER- XPS:DSD:Synthesizing Domain Specific Systems-CCF-1337375; Stanford PPL affiliates program, Pervasive Parallelism Lab: Oracle, AMD, Intel, NVIDIA, Huawei, SAP Labs. Authors also acknowledge additional support from Oracle. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

REFERENCES

[1] B. Catanzaro, M. Garland, and K. Keutzer, "Copperhead: compiling an embedded data parallel language," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, ser. PPoPP. New York, NY, USA: ACM, 2011, pp. 47–56.

[2] G. Mainland and G. Morrisett, "Nikola: embedding compiled GPU functions in Haskell," in *Proceedings of the third ACM Haskell symposium on Haskell*, ser. Haskell '10. New York, NY, USA: ACM, 2010, pp. 67–78.

[3] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier, "Optimising purely functional GPU programs," in *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '13. New York, NY, USA: ACM, 2013, pp. 49–60. [Online]. Available: http://doi.acm.org/10.1145/2500365.2500595

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, Oct 2009, pp. 44–54.

[5] J. Hoberock and N. Bell, "Thrust: C++ template library for CUDA," 2009.

[6] N. Nystrom, D. White, and K. Das, "Firepile: run-time compilation for GPUs in Scala," in *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, ser. GPCE. New York, NY, USA: ACM, 2011, pp. 107–116.

[7] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, ser. PPoPP, 2011, pp. 267–276.

[8] K. J. Brown, A. K. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "A heterogeneous parallel framework for domain-specific languages," ser. PACT, 2011.

[9] A. Prokopec, P. Bagwell, and T. R. abd Martin Odersky, "A generic parallel collection framework," ser. Euro-Par, 2010.

[10] S. L. P. Jones, R. Leshchinskiy, G. Keller, and M. M. T. Chakravarty, "Harnessing the multicores: Nested data parallelism in Haskell," in *FSTTCS*, 2008, pp. 383–414.

[11] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for CUDA," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 54:1–54:23, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2400682.2400713

[12] M. Amini, O. Goubier, S. Guelton, J. O. Mcmahon, F.-x. Pasquier, G. PÃl'an, and P. Villalon, "Par4all: From convex array regions to heterogeneous computing," in *Second International Workshop on Polyhedral Compilation Techniques*, ser. IMPACT 2012, 2012.

[13] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999, previous number = SIDL-WP-1999-0120. [Online]. Available: http://ilpubs.stanford.edu:8090/422/

[14] F. Niu, B. Recht, C. Ré, and S. J. Wright, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent," *Advances in Neural Information Processing Systems*, vol. 24, pp. 693–701, 2011.

[15] G. R. Bowman, X. Huang, and V. S. Pande, "Using generalized ensemble simulations and Markov state models to identify conformational states," *Methods*, vol. 49, no. 2, pp. 197 – 201, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1046202309000978

[16] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 152–163. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555775

[17] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for GPU architectures," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '10. New York, NY, USA: ACM, 2010, pp. 105–114. [Online]. Available: http://doi.acm.org/10.1145/1693453.1693470

[18] T. D. Han and T. S. Abdelrahman, "hiCUDA: a high-level directive-based language for GPU programming," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2009, pp. 52–61.

[19] S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W. H. Wen-mei, "CUDA-lite: Reducing GPU programming complexity," in *Languages and Compilers for Parallel Computing*. Springer, 2008, pp. 1–15.

[20] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU compiler for memory optimization and parallelism management," in *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10. New York, NY, USA: ACM, 2010, pp. 86–97. [Online]. Available: http://doi.acm.org/10.1145/1806596.1806606

[21] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke, "Sponge: portable stream programming on graphics engines," in *ACM SIGPLAN Notices*, vol. 46, no. 3. ACM, 2011, pp. 381–392.

[22] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," in *Compiler Construction*. Springer, 2002, pp. 179–196.

[23] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil, "Software pipelined execution of stream programs on GPUs," in *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*. IEEE, 2009, pp. 200–209.

[24] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic CPU-GPU communication management and optimization," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 142–151, 2011.

[25] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "Lime: a Java-compatible and synthesizable language for heterogeneous architectures," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA. New York, NY, USA: ACM, 2010, pp. 89–108.

[26] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink, "Compiling a high-level language for GPUs: (via language support for architectures and compilers)," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, ser. PLDI '12, 2012, pp. 1–12.

[27] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly, "Dandelion: a compiler and runtime for heterogeneous systems," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 49–68.

[28] L. Bergstrom and J. Reppy, "Nested data-parallelism on the GPU," in *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '12. New York, NY, USA: ACM, 2012, pp. 247–258. [Online]. Available: http://doi.acm.org/10.1145/2364527.2364563

[29] Y. Yang and H. Zhou, "CUDA-NP: Realizing nested thread-level parallelism in GPGPU applications," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. New York, NY, USA: ACM, 2014, pp. 93–106. [Online]. Available: http://doi.acm.org/10.1145/2555243.2555254