Optimized Two-Level Parallelization for GPU Accelerators using the Polyhedral Model

Jun Shirako
Rice University, USA
shirako@rice.edu

Akihiro Hayashi
Rice University, USA
ahayashi@rice.edu

Vivek Sarkar
Rice University, USA
vsarkar@rice.edu

Abstract

While GPUs play an increasingly important role in today's high-performance computers, optimizing GPU performance continues to impose large burdens upon programmers. A major challenge in optimizing codes for GPUs stems from the two levels of hardware parallelism, blocks and threads; each of these levels has significantly different characteristics, requiring different optimization strategies.

In this paper, we propose a novel compiler optimization algorithm for GPU parallelism. Our approach is based on the polyhedral model, which has enabled significant advances in program analysis and transformation compared to traditional AST-based frameworks. We extend polyhedral schedules to enable two-level parallelization through the idea of *superposition*, which integrates separate schedules for block-level and thread-level parallelism. Our experimental results demonstrate that our proposed compiler optimization framework can deliver 1.8× and 2.1× geometric mean improvements on NVIDIA Tesla M2050 and K80 GPUs, compared to a state-of-the-art polyhedral parallel code generator (PPCG) for GPGPUs.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers

Keywords Program transformations, parallelization, data locality optimizations, polyhedral model, GPUs, memory coalescing, CUDA code generation

1. Introduction

Graphics processing units (GPUs) are increasingly popular for high-performance computing because they can enable significant performance and energy efficiency improvements for certain classes of applications. However, current GPU programming models such as CUDA [33] and OpenCL [25] are too complex for non-expert programmers to exploit the full capability of GPUs since they require orchestrating low-level operations such as memory allocations/optimizations on GPUs and data transfers between CPUs and GPUs. Even expert programmers find optimizing programs for GPUs in general to be a time-consuming, non-trivial affair. Additionally, performance tuning with such low-level programming models is often device-specific, thereby reducing performance portability. When considering software productivity, a more attractive approach would allow users to express parallelism and locality as simply as possible in a platform-independent manner, and to rely on tools such as optimizing compilers to optimize/customize their code for specific target systems.

A breakthrough in the last decade, the polyhedral compilation model has provided significant advancements in the unifications/-generalizations of affine loop transformations and powerful code generations for the practical class of programs [1, 7, 15, 23, 42]. The benefit of this unified formulation can be seen (for example) in the Pluto algorithm [7], which has been successfully extended and specialized to integrate SIMD constraints [26]. In contrast with pure polyhedral approaches, the PolyAST [42] framework, a hybrid approach of polyhedral and AST-based transformations, supports a cache-aware affine transformation algorithm that is guided by AST-based cost models.

With respect to polyhedral compilations for GPU architectures, we briefly summarize a few frameworks that have been developed recently. C-to-CUDA [5] is an end-to-end polyhedral framework targetting GPUs that extends the Pluto algorithm with additional scheduling constraints for memory coalescing and managing the data transfer between global and shared memories. The R-Stream compiler [29, 47] employs the Pluto scheduling algorithm with several extensions for GPUs, including balance of fusion, parallelism and SIMD efficiency. Although R-Stream supports several GPU-specific optimizations, most of them are explained as individual AST-based transformations and it is not clear how such optimizations are integrated in their polyhedral framework. PPCG [49], which is widely-recognized as a state-of-the-art polyhedral compiler for GPUs, uses a modified version of Pluto algorithm implemented in the ISL library [48]. PPCG generates highly optimized GPU codes with strong shared/private memory optimizations. However, despite those extensions to GPUs, the affine scheduling algorithms used in such frameworks are fundamentally equivalent to those proposed for CPUs and/or general computing systems. In particular, there has not been much attention paid in polyhedral scheduling algorithms to modeling the two distinct levels of parallelism in GPU devices in a manner that closely resembles the GPU's architectural characteristics.

In this paper, we introduce a new polyhedral scheduling algorithm specialized for GPU parallelism. A major challenge in optimizing codes for GPUs stems from their two levels of hardware parallelism: blocks and threads, containing significantly different characteristics and therefore requiring different optimization strategies. Due to the lack of support for inter-block synchronization, synchronization-free parallelism is mandatory at the outer block level, while maximization of coalesced memory accesses is crucial for efficient thread-level parallelism at the inner level, even at the cost of inter-thread synchronizations within a block. To coordinate different optimization strategies in a unified manner, we extend polyhedral schedules to enable two-level parallelization through

CC'17, February 5–6, 2017, Austin, TX, USA
© 2017 ACM. 978-1-4503-5233-8/17/02...$15.00
http://dx.doi.org/10.1145/3033019.3033022

the idea of *superposition*, which constructs separate schedules independently for block-level and thread-level parallelism; and integrates into a final schedule. As a model to capture GPU's architectural characteristics, we employ an analytical memory cost model named *Distinct Line (DL) model*, which was originally proposed for cache/TLB analysis. We extend the DL model to GPU memory warps and coalesced accesses analysis. Our thread-level scheduling strategy assigns the highest priority to maximizing coalesced memory accesses via the extended DL model, while the block-level strategy aims at coarser-grained synchronization-free parallelism. To summarize, this paper makes the following contributions:

- Proposal of superposition to enable two-level parallelization for GPUs, seamlessly integrating separate schedules for block-level and thread-level hardware parallelism.
- A cost-based scheduling algorithm to maximize coalesced memory accesses for GPU thread-level parallelism, via extensions of an analytical memory cost model for CPU cache/TLB.
- Detailed performance evaluations of the proposed optimizer using PolyBench 3.2 and SPEC ACCEL, showing 1.8 geometric mean improvement over PPCG on NVIDIA Tesla M2050 with 448 CUDA cores and 2.1 geometric mean improvement over PPCG on NVIDIA Tesla K80 with 2496 CUDA cores.

The paper is organized as follows. Section 2 contains background information on GPUs and CUDA, and motivates the problem. Section 3 provides an overview of our framework. Section 4 introduces our extensions to enable two-level parallelization in the polyhedral model, and Section 5 discusses the algorithms to find the best composition of transformations and parallelizations based on these extensions. Section 6 presents experimental results to evaluate our approach using PolyBench 3.2 on the two GPU systems. Sections 7 and 8 summarize related work and our conclusions.

2. Background and Motivation

2.1 GPUs and CUDA Programming Model

NVIDIA GPU architecture consists of global memory and an array of *streaming multiprocessors* (SMXs). Each SMX comprises many single- and double- precision cores, special function units, and load/store units to execute hundreds of threads concurrently. L1 cache, read-only cache, and shared memory are shared among these cores/units to improve data locality within a single SMX. Also, global memory data requested from each SMX are cached by L2.

CUDA [33] is a standard parallel programming model for NVIDIA GPUs. In CUDA, **kernels** are C functions that will be executed on GPUs. A **block** is a group of **threads** executed on the same SMX and is organized in a collection of blocks called a **grid** that corresponds to a single kernel invocation. All blocks within a grid are indexed either 1- or 2-D array. Similarly, all threads within each block are indexed 1-, 2-, or 3-D array. While **barrier synchronizations among threads** in the same block are allowed, no support exists for inter-block (global) barrier synchronizations. Hence, programmers need to prepare kernels separately for global barriers. For memory optimizations, the programmer and compiler must utilize registers and shared memory for improving data locality. Also, it is worth mentioning that global memory accesses for adjacent memory locations are coalesced into a single memory transaction if consecutive global memory locations are accessed by an amount of consecutive threads (normally 32 threads) and the starting address is aligned. This is called **memory access coalescing** and can be performed by programmers and compilers.

2.2 Motivating Example

We used two benchmarks from the PolyBench suite, jacobi-2d-alt and doitgen, to motivate our approach. The input sequential C code fragments are shown in Figures 1 and 4, respectively. We applied a state-of-the-art polyhedral compiler, PPCG [49], which enables maximal data locality, tilability, and GPU parallelization. The resulting codes parallelized via CUDA code generation[1] are shown in Figures 2 and 5, while the output CUDA kernels generated by our proposed framework are shown in Figures 3 and 6. For readability, we omitted data transfers among global and shared memories and register optimizations in Figures 2, 3, 5, and 6.

2-D Jacobi variant: As shown in Figure 1, jacobi-2d-alt is a variant of jacobi-2d in which the `i`-dimension of the 2-D space has no dependences. This variant has two levels of loop parallelism: `i`-loops with synchronization-free forall and `j`-loops with fine-grained forall requiring synchronizations. The scheduling phase in PPCG is based on the Pluto algorithm, and uses the same schedule for blocks and threads. PPCG maps the outermost band (a set of consecutive parallel loops in the schedule) onto both block-level and thread-level GPU parallelism [49]. Figure 2 shows the outermost parallel `i`-loop is tiled into chunks with 32 iterations, which are mapped to blocks while iterations per chunk are mapped to threads. On the other hand, the inner fine-grained parallelism of `j`-loops is ignored and executed sequentially within a thread. Although the generated code enables coarse-grained parallelization without synchronization, it misses the opportunity for coalesced memory accesses on both arrays `A` and `B` because contiguous threads access the arrays with a non-unit stride of 2000.

In contrast, our framework can select different levels of parallelism individually for block-level and thread-level schedules via superposition. In Figure 3, the synchronization-free parallelism of the `i`-loops is mapped to a 1-D block space while both `i`-loops and `j`-loops are mapped to a 2-D thread space, enabling coalescing memory accesses on `A` and `B` at the cost of inter-thread synchronizations. Table 1 shows the kernel execution time for PPCG and our flow on NVIDIA Tesla M2050 (labeled as Fermi) and Tesla K80 (labeled as Kepler). The results show that the code generated by our approach is faster than that of PPCG by factors of 2.5 on Fermi and 3.3 on Kepler. Note that shared memory and register optimizations, omitted in the figures, were enabled for both PPCG and our framework when evaluating the codes. This result is due to the fact that inter-thread synchronizations are relatively lightweight while memory coalescing has a large impact on GPU performance.

	jacobi-2d-alt		doitgen		gemver	
Systems	Fermi	Kepler	Fermi	Kepler	Fermi	Kepler
PPCG	22.22×	1.77×	0.60×	0.29×	73.26×	147.34×
our flow	55.02×	5.77×	124.12×	76.08×	103.79×	177.08×

Table 1: Performance comparison (speedup vs. GCC)

Doitgen: The second motivating example is doitgen whose original code is shown in Figures 4. As with jacobi-2d-alt, PPCG computes the schedule of given kernel and maps the outermost band to blocks and threads. All statements `S`, `T` and `U` are fully fused and synchronization-free foralls, `r`-loop and `q`-loop, are tiled and mapped to the 2-D block/thread spaces (lines 2–5 in Figure 5). These scheduling and mapping are optimal for block-level, but not for thread-level due to non-coalesced memory accesses.

In our approach (Figure 6), the same parallelism as that of PPCG is selected for block-level, while selecting different thread-level schedule such that statement `U` is distributed from `S` and `T` to keep `p`-loops parallel. Note that our extended DL model considered `p`-loop as the most profitable loop for memory coalescing and guided thread-level scheduling that way. As a consequence, `r`,`q`,`p`-loops are mapped to the 3-D thread space and enable good coalesced accesses while keeping synchronization-free foralls for block-level, resulted in large performance gains shown in Table 1.

[1] PPCG can output C, CUDA, and OpenCL code.

```
  for (t = 0; t < T; t++) {
    for (i = 1; i < N-1; i++)
      for (j = 1; j < N-1; j++)
S:        B[i][j] = fact * (A[i][j]
            + A[i][j-1] + A[i][j+1]);
    for (i = 1; i < N-1; i++)
      for (j = 1; j < N-1; j++)
T:        A[i][j] = B[i][j];
  }
```

Figure 1: Input jacobi-2d-alt code (variant where i-loop is synchronization-free forall)

```
__global__ void kernel0(...) {
  int b0 = blockIdx.x;  // i-tile (blk-x)
  int t0 = threadIdx.x; // i      (thrd-x)
  for (int c0 = 32 * b0; c0 < N-1; c0+=1048576) // blk-x
    for (int c1 = 0; c1 < T; c1+=32)
      for (int c2 = 2 * c1; c2 <= ...; c2+=32) {
        if (N >= t0 + c0 + 2 && t0 + c0 >= 1)
          for (int c4 = ...; c4 <= ...; c4+=1)
            for (int c5 = ...; c5 <= ...; c5+=1) {
              if (N + 2 * c1 + 2 * c4 >= c2 + c5 + 2)
S:              B[(t0+c0)*2000+(-2*c1+c2-2*c4+c5)] = ...
              if (c2 + c5 >= 2 * c1 + 2 * c4 + 2)
T:              A[(t0+c0)*2000+(-2*c1+c2-2*c4+c5-1)] = ...
}     }     }
```

Figure 2: jacobi-2d-alt using PPCG

```
__global__ void kernel0(...) {
  int c1, c3, c5, c7, c9;
  c1 = blockIdx.x;                         // i-tile (blk-x)
  c3 = 32 * c1 + threadIdx.y;              // i      (thrd-y)
  if (c3 >= 1 && c3 <= N-2) {
   for (c5 = 0; c5 <= T-1; c5++) {         // t
     for (c7 = 0; c7 <= (N-2)/32; c7++) { // j-tile
       c9 = 32 * c7 + threadIdx.x;         // j      (thrd-x)
       if (c9 >= 1 && c9 <= N-2)
S:      B[c3*2000+c9] = fact * (A[c3*2000+c9] +
            + A[c3*2000+c9-1] + A[c3*2000+1+c9]);
     }
     __syncthreads();
     for (c7 = 0; c7 <= (N-2)/32; c7++) { // j-tile
       c9 = 32 * c7 + threadIdx.x;         // j      (thrd-x)
       if (c9 >= 1 && c9 <= 1998)
T:      A[c3*2000+c9] = B[c3*2000+c9];
     }
     __syncthreads();
}}}
```

Figure 3: jacobi-2d-alt using our flow

```
  for (r = 0; r < NR; r++)
    for (q = 0; q < NQ; q++)  {
      for (p = 0; p < NP; p++)  {
S:        sum[r][q][p] = 0;
        for (s = 0; s < NP; s++)
T:          sum[r][q][p] +=
              A[r][q][s] * C4[s][p];
      }
      for (p = 0; p < NR; p++)
U:        A[r][q][p] = sum[r][q][p];
    }
```

Figure 4: Input doitgen code

```
__global__ void kernel0(...) {
  int b0 = blockIdx.y;  // r-tile (blk-y)
  int b1 = blockIdx.x;  // q-tile (blk-x)
  int t0 = threadIdx.y; // r      (thrd-y)
  int t1 = threadIdx.x; // q      (thrd-x)
  for (int c0 = 32 * b0; c0 < NR; c0+=8192)      // blk-y
    for (int c1 = 32 * b1; c1 < NQ; c1+=8192) {  // blk-x
      for (int c2 = 0; c2 < NP; c2+=32) {
        for (int c3 = c2; c3 <= ...; c3+=32) {
         if (NR >= t0 + c0 + 1)
          for (int c5 = t1; c5 <= ...; c5+=16) { // thrd-x
           if (c2 >= 32) { ... } else
            for (int c7 = 0; c7 <= ...; c7+=1) {
S:           sum[((t0+c0)*256+(c1+c5))*256+(c3+c7)] = 0;
T:           sum[((t0+c0)*256+(c1+c5))*256+0] += ...
            }
           ...
           if (c2 + 31 >= NP)
            for (int c7 = ...; c7 <= ...; c7+=1)
U:           A[((t0+c0)*256+(c1+c5))*256+(-NP+c3+c7)] = ...
        ...
}
```

Figure 5: doitgen using PPCG

```
__global__ void kernel0(...) {
  int c1, c3, c5, c7, c9, c11, c13, c15;
  c1 = blockIdx.y;                      // r-tile (blk-y)
  c3 = blockIdx.x;                      // q-tile (blk-x)
  c5 = c1 * 4 + threadIdx.z;            // r      (thrd-z)
  c7 = c3 * 4 + threadIdx.y;            // q      (thrd-y)
  for (c9 = 0; c9 <= (NP-1)/32; c9++) { // p-tile
    c11 = c9 * 32 + threadIdx.x;        // p      (thrd-x)
S:  sum[c5*65536+c7*256+c11] = 0;
    for (c13 = 0; c13 <= (NP-1)/32; c13++)        // s-tile
      for (c15 = c13*32; c15 <= c13*32+31; c15++) // s
T:      sum[c5*65536+c7*256+c11] +=
          A[c5*65536+c7*256+c15] * C4[c15*256+c11];
  }
  __syncthreads();
  c5 = c1 * 4 + threadIdx.z;            // r      (thrd-z)
  c7 = c3 * 4 + threadIdx.y;            // q      (thrd-y)
  for (c9 = 0; c9 <= (NR-1)/32; c9++) { // p-tile
    c11 = c9 * 32 + threadIdx.x;        // p      (thrd-x)
U:  A[c5*65536+c7*256+c11] = sum[c5*65536+c7*256+c11];
} }
```

Figure 6: doitgen using our flow

gemver: Finally, our framework supports (intra-block) thread-level reductions, which is implemented via a template of the parallel tree-based reduction on shared memory [31]. Table 1 shows this reduction support delivers additional performance improvement relative to the PPCG versions.

To summarize, our polyhedral scheduling algorithm enables two-level parallelism for GPUs via superposition, which allows block-level and thread-level schedules to be computed individually with different optimization strategies: maximizing synchronization-free parallelism for blocks and coalesced memory accesses for threads. Furthermore, our framework supports thread-level reduction parallelism to increase the amount of available parallelism and opportunities for coalesced memory accesses. These extensions deliver considerable performance improvements over a state-of-the-art polyhedral compiler for GPU accelerators.

3. Overview

3.1 Polyhedral Compilation Framework

The polyhedral model is a flexible representation for collections of (imperfectly) nested loops. Loop nests amenable to this algebraic representation are called *Static Control Parts* (SCoPs) and represented in the SCoP format [36], where each statement consists of three elements: iteration domain, access relation, and scattering function. SCoPs require their loop bounds, branch conditions, and array subscripts to be affine functions of iterators and global parameters.

Iteration domain, $\mathcal{D}^S$: The iteration domain of a statement S enclosed by m loops is represented by an m-dimensional polytope. Each element $\vec{i} \in \mathcal{D}^S$ represents a unique dynamic instance of statement S. As an example, the iteration domain of statement `S` in Figure 1 is:
$\mathcal{D}^S = \{(t,i,j) \in \mathbb{Z}^3 \mid 0 \le t \le \mathrm{T}-1 \,\wedge\, 1 \le i,j \le \mathrm{N}-2\}$.

Access relation, $\mathcal{A}^S(\vec{i})$: Each array reference in a statement is abstracted as an access relation, which maps a statement instance $\vec{i}$ to one or more array elements to be read/written[2] [52]. This mapping is typically expressed as a set of affine expressions of loop iterators and global parameters (access functions). In Figure 1, the write access relation for statement `S` is: $\mathcal{A}^S(\vec{i}) = (B,i,j)$.

Scattering function, $\Theta^S(\vec{i})$**:** In general, any composition of iteration- and statement- reordering sequential loop transformations (e.g., permutation, skewing, distribution, fusion) as well as parallelizing transformations can be specified by a *scattering function*. As described below, the scattering function can include two components, a *schedule* which orders statement instances in time, and a *space-mapping* which distributes statement instances across processors [6].

- **Schedule:** The sequential execution order of a program is captured by the schedule, which maps each statement instance to a logical time-stamp vector, expressed as a multidimensional affine function of $\vec{i}$. Statement instances are executed according to the increasing lexicographic order of their time-stamps, while statement instances with the same time-stamp can be executed in parallel.
  The schedules that represent the sequential execution order of statements `S` and `T` in Figure 1 are: $\Theta^S(\vec{i}) = (0,t,0,i,j,0)$ and $\Theta^T(\vec{i}) = (0,t,1,i,j,0)$. The first two dimensions $(0,t)$ of $\Theta^S(\vec{i})$

[2] A scalar variable is considered as a degenerate case of an array.

and of $\Theta^T(\vec{i})$ indicate that t is the outermost loop. The 0 or 1 value in the next dimension indicates that, for the same iteration of the t-loop, all instances of `S` are executed before any instance of `T`. The next i, j values indicate that each of `S` and `T` is enclosed in an i-j loop nest. The first and last 0 values in the time-stamps are effectively no-ops, and would only come into play if there were more statements in this example.

- **Space-mapping:** In the case of space-mapping, the number returned by the scattering function for a given statement instance is the identifier of the logical processor that executes the instance. As with schedules, the space-mapping can be a multidimensional vector (Section 4.1).

As shown in above examples, any dimension of a scattering function may contain loop iterators. A dimension is called a *loop dimension* if it contains one or more iterators; otherwise it is called *scalar dimension*.

Dependence Polyhedra, $\mathcal{D}^{S\to T}$: The dependences between pairs of statements S and T are captured by dependence polyhedra — the subset of pairs $(\vec{i},\vec{i'}) \in \mathcal{D}^S \times \mathcal{D}^T$ that participate in a dependence [7]. Given two statement instances $\vec{i}$ of S and $\vec{i'}$ of T, $\vec{i'}$ is said to depend on $\vec{i}$ if: 1) they access the same array location, i.e., $\mathcal{A}^S(\vec{i}) = \mathcal{A}^T(\vec{i'})$; 2) at least one of them is write access; and 3) $\vec{i}$ has a lexicographically smaller schedule than $\vec{i'}$, i.e., $\Theta^S(\vec{i}) \prec \Theta^T(\vec{i'})$.

3.2 Overview of Framework (PolyAST+GPU)

```
Algorithm 1: End-to-end optimization flow
  Input  : Source program
           $P_1 := (Scop,\ Dependence\ Polyhedra)$
1 begin
2   $P_2$ := threads and blocks transformation($P_1$);
3   $P_3$ := single thread transformation($P_2$);
4   $P_4$ := shared memory and register optimization($P_3$);
  Output: Parallelized and optimized program $P_4$
```

Algorithm 1[3] shows the overview of our end-to-end optimization framework called PolyAST+GPU, developed as an extension to the PolyAST hybrid compilation framework for combining polyhedral and AST-based transformations [42]. The input to our framework is the polyhedral representations of a source program and dependence information. While this paper focuses on sequential input programs with standard polyhedral dependence analyzers [8, 23, 48], supporting parallel input programs in our framework should be a straightforward extension by leveraging recent work on polyhedral dependence analysis for parallel programs [3, 9].

After dependence analysis, PolyAST+GPU selects legal compositions of sequential and parallelizing loop transformations as schedules that are computed at both the block-level and thread-level for GPU execution. The two schedules are then *superposed* into the final schedule, which can be given to a standard polyhedral code generator such as CLooG [6]. In this paper, we target CUDA as our output code and the code generation phase of PolyAST+GPU internally uses CLooG. The current implementation (not algorithmic) limitation is that tile sizes and GPU thread sizes must be the same to simplify code generation.

After block-level and thread-level transformations, our framework performs additional transformations for sequential code regions by a single thread, including loop tiling to enhance data locality. The single thread transformations are encoded in the thread-level schedules. We use the PolyAST transformation algorithm for this step. As the final step in the optimization flow, shared memory and register optimizations are performed. We employ a memory optimization approach similar to PPCG [49] and C-to-CUDA [5] for this step: 1) individual tiles are identified after superposition; 2) array elements to be used/modified within each tile are computed; and 3) such elements with temporal reuse and/or non-coalesced accesses are transferred to/from shared memory or registers based on reuse patterns, i.e., inter-thread or intra-thread reuse.

We present details for the superposition of schedules (i.e., scattering functions) in Section 4, and the scheduling algorithm via superposition in Section 5.

[3] Step 1 (threads & blocks transformation) is a novel contribution of this paper, while steps 2 and 3 reuse past work from PolyAST [42] and PPCG [49].

4. Superposition for GPU 2-level Parallelism

In this section we first introduce the composition of schedule and space-mapping, which respectively represent transformations and parallelism, in a single scattering function. To handle different optimization strategies for blocks and threads, we compute two scattering functions per statement: one at the block-level and the other at the thread-level. We use the idea of superposition to seamlessly integrate the two levels of optimizations. To the best of our knowledge, PolyAST+GPU is the first polyhedral-based compilation system to support different transformations at the GPU block and thread levels in a unified code generation framework.

4.1 Composition of Schedule and Space-mapping

In the polyhedral model, the selection of transformations and parallelization is encoded via scattering functions. To capture both in a single form, we employ the composition of schedule dimensions and space-mapping dimensions in a scattering function, analogous to the time/space-partition in affine partitioning [30]. In this work specific to GPUs, we annotate space-mapping dimensions with subscripts x, y, and z, which represent dimensions in thread/block space, to differentiate these dimensions from schedule dimensions.

Let us look at the thread-level parallelization in Figure 3, which can be represented by the following pseudo code.

```
  for (t = 0; t < T; t++) {
    pfor_y (i = ...; i < ...; i++)
      pfor_x (j = ...; j < ...; j++)
S:      B[i][j] = fact * (A[i][j] + A[i][j-1] + A[i][j+1]);
    barrier;
    pfor_y (i = ...; i < ...; i++)
      pfor_x (j = ...; j < ...; j++)
T:      A[i][j] = B[i][j];
    barrier;
  }
```

Note that `pfor_x`/`pfor_y` is a two-dimensional parallel loop whose iterations are mapped to the *x*/*y*-dimension of the thread space, and the barrier is an inter-thread synchronization. The scattering functions for statements S and T are:

$$\Theta^S(\vec{i}) = (0,t,0,i_y,j_x,0), \quad \Theta^T(\vec{i}) = (0,t,1,i_y,j_x,0)$$

The first three dimensions $(0,t,0)$ of $\Theta^S(\vec{i})$ and $(0,t,1)$ of $\Theta^T(\vec{i})$ are schedule dimensions that capture the sequential execution order specified by the t-loop and the two barriers, while the space-mapping dimensions (i_y, j_x) capture the thread-level parallelism of the `pfor_y`/`pfor_x` loops.

4.2 Superposition of Scattering Functions

This section presents superposition, which enables different transformations to be performed at the block and thread level, and also allows inter-thread synchronization (barriers) to be expressed within a block. We use two kinds of scattering functions for superposition:

- **Outer scattering function,** $\Theta^{S_{out}}(\vec{i})$ is a many-to-one function that can assign multiple statement instances $\vec{i}$ of S to the same

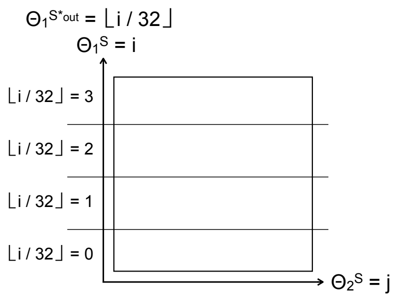


Figure 7: Relation of $\Theta^{S^\star_{out}}(\vec{i}) = (\lfloor i/32 \rfloor)$ to $\Theta^S(\vec{i}) = (i, j)$

value (e.g., to identify the thread block in which the instances will be executed).

- **Regular scattering function,** $\Theta^S(\vec{i})$ is a one-to-one function that assigns each instance $\vec{i}$ of S a unique value (e.g., to denote the schedule and space-mapping within a block).

These two scattering functions are inclusive – i.e., loop iterators in the outer scattering function may also appear in the regular scattering function – and defined independently. They are superposed into a single function in the manner of loop tiling: $\Theta^{S_{out}}(\vec{i})$ to specify individual tiles and $\Theta^S(\vec{i})$ to specify iterations per tile. Let $\Theta^{S^\star_{out}}(\vec{i})$ denote the outer scattering function with specific tile sizes: TL_1, TL_2, $\cdots$. The k-th dimension of $\Theta^{S^\star_{out}}(\vec{i})$ is defined as[4]:

$$\Theta_k^{S^\star_{out}}(\vec{i}) = \lfloor \Theta_k^{S_{out}}(\vec{i})/TL_k \rfloor \quad (if\ k\ is\ a\ loop\ dimension)$$

$$\Theta_k^{S^\star_{out}}(\vec{i}) = \Theta_k^{S_{out}}(\vec{i}) \qquad (otherwise\ scalar\ dimension)$$

For statements S and T in Figure 3, the regular scattering functions for the thread-level are those shown in Section 4.1, and the outer scattering functions, which are used at the block-level, are shown below. The corresponding 2-D space is shown in Figure 7. We used 32 as the tile size here.

$$\Theta^{S_{out}}(\vec{i}) = (0, i_x), \quad \Theta^{T_{out}}(\vec{i}) = (0, i_x)$$

$$\Theta^{S^\star_{out}}(\vec{i}) = (0, \lfloor i/32 \rfloor_x), \quad \Theta^{T^\star_{out}}(\vec{i}) = (0, \lfloor i/32 \rfloor_x)$$

The outer and regular scattering functions may have different affine combinations of loop iterators (e.g., different loop permutation orders and fusion/distribution structures) and different parallelism choices (e.g., a loop may be parallel at the outer-level, and serial at the regular-level) so long as Lemma 1 below is satisfied. Finally, the superposition of outer and regular scattering functions is defined by concatenation as follows:

$$\Theta^{S^\star}(\vec{i}) = (\Theta^{S^\star_{out}}(\vec{i}),\ \Theta^S(\vec{i}))$$

In our running example, the superposed scattering functions are:

$$\Theta^{S^\star}(\vec{i}) = (0, \lfloor i/32 \rfloor_x, 0, t, 0, i_y, j_x, 0),$$

$$\Theta^{T^\star}(\vec{i}) = (0, \lfloor i/32 \rfloor_x, 0, t, 1, i_y, j_x, 0)$$

The space-mapping dimension $\lfloor i/32 \rfloor_x$ indicates that each group of 32 iterations is mapped as a chunk to the x-dimension of the block space. The individual blocks contain the thread-level parallelism, (i_y, j_x) described in Section 4.1. Figure 3 shows the generated code from $\Theta^{S^\star}(\vec{i})$ and $\Theta^{T^\star}(\vec{i})$.

We now state an important lemma that establishes a sufficient condition for the legality of superposition.

[4] In codegen, $\Theta_k^{S^\star_{out}}(\vec{i})$ is replaced by a new iterator (e.g., $ii = \lfloor i/32 \rfloor$) and expressed as inequality constraints (e.g., $32 \times ii \le i \le 32 \times ii + 31$).

LEMMA 1 (Legality constraints of superposition).
Superposition is legal (i.e., satisfies all dependences) if all outer scattering functions are fully permutable and regular scattering functions satisfy all dependences.

Proof. For any dependence of interest, $\mathcal{D}^{S \to T}$, the precondition of Lemma 1 ensures:

$$\forall k : \Theta_k^{T_{out}}(\vec{i'}) - \Theta_k^{S_{out}}(\vec{i}) \ge 0, \ \ (\vec{i}, \vec{i'}) \in \mathcal{D}^{S \to T} \qquad (1)$$

$$\Theta^T(\vec{i'}) - \Theta^S(\vec{i}) \succ \vec{0}, \ \ (\vec{i}, \vec{i'}) \in \mathcal{D}^{S \to T} \qquad (2)$$

We prove that the superposed scattering functions also satisfy all dependences: $\Theta^{T^\star}(\vec{i'}) - \Theta^{S^\star}(\vec{i}) \succ \vec{0}, \ (\vec{i}, \vec{i'}) \in \mathcal{D}^{S \to T}$. In this proof, we consider classical 2d+1 encoding[5] for outer scattering functions with specific tile sizes:

$$\Theta^{S^\star_{out}}(\vec{i}) = (\Theta_1^{S_{out}}(\vec{i}),\ \lfloor \Theta_2^{S_{out}}(\vec{i})/TL_2 \rfloor,\ \Theta_3^{S_{out}}(\vec{i}),\ \cdots)$$

Given a dependence $(\vec{i}, \vec{i'}) \in \mathcal{D}^{S \to T}$, expression (1) ensures:

$$\forall j : \Theta_{2j-1}^{T^\star_{out}}(\vec{i'}) - \Theta_{2j-1}^{S^\star_{out}}(\vec{i}) = \Theta_{2j-1}^{T_{out}}(\vec{i'}) - \Theta_{2j-1}^{S_{out}}(\vec{i}) \ge 0$$

$$\forall j : \Theta_{2j}^{T^\star_{out}}(\vec{i'}) - \Theta_{2j}^{S^\star_{out}}(\vec{i}) = \lfloor \Theta_{2j}^{T_{out}}(\vec{i'})/TL_{2j} \rfloor - \lfloor \Theta_{2j}^{S_{out}}(\vec{i})/TL_{2j} \rfloor$$
$$\ge \lfloor \Theta_{2j}^{S_{out}}(\vec{i})/TL_{2j} \rfloor - \lfloor \Theta_{2j}^{S_{out}}(\vec{i})/TL_{2j} \rfloor = 0$$

These expressions are summarized as:

$$\forall k : \Theta_k^{T^\star_{out}}(\vec{i'}) - \Theta_k^{S^\star_{out}}(\vec{i}) \ge 0, \ \ (\vec{i}, \vec{i'}) \in \mathcal{D}^{S \to T} \qquad (3)$$

Finally, $\Theta^{S^\star}(\vec{i}) = (\Theta^{S^\star_{out}}(\vec{i}),\ \Theta^S(\vec{i}))$, expressions (2) and (3) ensure:

$$\Theta^{T^\star}(\vec{i'}) - \Theta^{S^\star}(\vec{i}) \succ \vec{0}, \ \ (\vec{i}, \vec{i'}) \in \mathcal{D}^{S \to T}$$

□

4.3 GPU-specific Aspects of Superposition

We assume the target GPUs have no support for inter-block synchronizations while inter-thread synchronization is available in the form of barriers within a thread block. Under these assumptions, the scattering functions for GPU kernels take the following form in our approach.

Outer scattering function (block-level): consists of a scalar schedule dimension to specify the sequential execution order of the invoked GPU kernel, followed by space-mapping dimensions to specify block indexes (y, x):

$$\Theta^{S_{out}}(\vec{i}) = (kernel\_id,\ block_y,\ block_x)$$

Statements with the same schedule dimension ($kernel\_id$) are enclosed in the same GPU kernel. Different GPU kernel calls are sequentially invoked in the increasing order of the schedule dimensions[6]. Since all loop dimensions are space-mapping, i.e., parallel loops with no dependence, the permutability constraint in Lemma 1 is guaranteed to hold.

Regular scattering function (thread-level): consists of schedule dimensions to specify the sequential execution order of the thread block, followed by space-mapping dimensions to specify thread indexes (z, y, x), and inner schedule dimensions to specify single thread execution order:

$$\Theta^S(\vec{i}) = (sche\_dim,\ ...,\ thread_z,\ thread_y,\ thread_x,\ sche\_dim,\ ...)$$

The schedule dimensions outside the space-mapping indicate the inter-thread synchronization points (barriers) at which to invoke

[5] In 2d+1 encoding, scalar and loop dimensions are interleaved: odd and even dimensions respectively corresponds to scalar and loop dimensions.

[6] Inter-kernel parallelism is a subject for future work.

`__syncthreads`. The inner schedule dimensions encode per-thread loop transformations, including additional loop tiling for improved intra-thread locality.

5. Parallelization Algorithm for GPUs

This section discusses the optimization algorithms used to find the block-level and thread-level scattering functions for a given GPU device. The block-level optimization policy aims to find the coarsest-grained synchronization-free parallelism, while the thread-level optimization policy aims to maximize coalesced memory accesses with guidance from the DL memory cost model extended to GPUs.

5.1 Target Affine Form of Scattering Function

The primary contribution of PolyAST, the underlying framework of PolyAST+GPU, was to restrict affine forms such that the program transformations can focus on implementing good data locality while preserving SIMD parallelism [42]. As with SIMD execution, efficient GPU thread-level parallelization requires good spatial data locality to expose coalesced memory accesses. Thus, we employ the same affine form in our target scattering functions for both vector and GPU thread parallelism, which looks as follows:

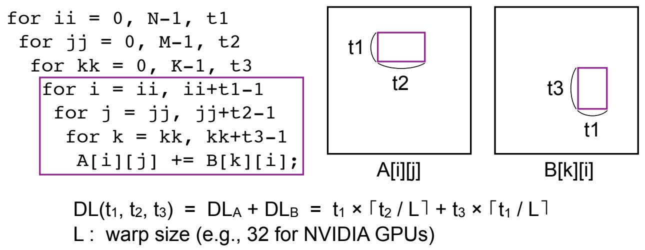
$$\Theta^S(\vec{i}) = \begin{pmatrix} 0 & 0 & \dots & 0 & \beta_1 \\ \alpha_{1,1} & \alpha_{1,2} & \dots & \alpha_{1,d} & c_1 \\ \vdots & & & \vdots & \\ 0 & 0 & \dots & 0 & \beta_{d'} \\ \alpha_{d',1} & \alpha_{d',2} & \dots & \alpha_{d',d} & c_{d'} \\ 0 & 0 & \dots & 0 & \beta_{d'+1} \end{pmatrix} \cdot \begin{pmatrix} i_1 \\ i_2 \\ \vdots \\ i_d \\ 1 \end{pmatrix}$$

where $\forall k \in \{1..d'\}, \sum_{i=1}^{d} |\alpha_{k,i}| = 1$. This encoding is reminiscent of classical 2d+1 encoding [26, 38]: odd rows are scalar dimensions to model multidimensional statement interleaving (i.e., multidimensional fusion/distribution/code motion), while even rows are loop dimensions to model loop permutation, reversal, and multidimensional retiming (i.e., index set shifting) by factor of c_i. Note that loop skewing, which affects array subscript expressions and possibly degrades spatial data locality, is avoided in this affine form due the constraint, $\sum_{i=1}^{d} |\alpha_{k,i}| = 1$.

Although the above form can be directly used for thread-level scattering functions, some modifications are needed at the block level. As described in Section 4, a block-level scattering function is partial and not a one-to-one mapping; the number of rows can be fewer than $2d+1$. Further, to implement better locality and coarser granularity for stencil algorithms, several novel loop tiling approaches have been proposed [4, 16, 17, 21]. We conjecture that many such tiling approaches can be encoded in our block-level scattering functions; however, validating this conjecture is a subject for future work. For simplicity, we restrict our attention to rectangular tiling in this paper.

5.2 Extending DL Model to Coalesced Memory Access

The Distinct Lines (DL) model was designed to estimate the number of distinct cache lines, accessed in a sequential loop nest [12, 40]. In the PolyAST framework, the DL model was used for loop permutation order analysis and loop fusion profitability analysis, so that the best loop orders and fusion decisions are respectively found by minimizing the cache-based CPU memory access cost for the given loop nests [42]. The fundamental idea behind the DL model is to estimate the total number of distinct cache lines accessed within a given loop nest as a function of loop lengths or tile sizes, and to convert that cost to a normalized per-iteration value. Assuming that each distinct line is kept in cache until the last use, the DL value can be used as the per-iteration cache miss count, i.e., memory access cost. In this paper, we extend the DL model to estimate the total number of memory transactions within a given parallel loop nest designed for GPU execution, by assuming: 1) accesses to continuous data elements per segment are coalesced into a single memory transaction; and 2) such elements are kept and reused on shared memory through explicit data transfers and/or implicit cache operations. Figure 8 shows a simple case with two array references enclosed in a triply nested tiled loops, where the DL value is expressed as a function of tile sizes, $DL(t_1, t_2, t_3)$.



```
for ii = 0, N-1, t1
 for jj = 0, M-1, t2
  for kk = 0, K-1, t3
   for i = ii, ii+t1-1
    for j = jj, jj+t2-1
     for k = kk, kk+t3-1
      A[i][j] += B[k][i];
```

DL(t1, t2, t3) = DLA + DLB = $t_1 \times \lceil t_2 / L \rceil + t_3 \times \lceil t_1 / L \rceil$
L : warp size (e.g., 32 for NVIDIA GPUs)

Figure 8: Example of DL for tiled loop nest

For arrays A and B, $t_1 \times t_2$ and $t_3 \times t_1$ elements are respectively accessed in a tile, and the shared memory optimization phase is assumed to insert data transfers such that the array elements accessed in the tile are prefetched from global memory as fully coalesced transactions. The per-iteration memory access cost of a given loop is defined as:

$$mem\_cost(t_1, t_2, \cdots, t_d) = \frac{Cost_{trans} \times DL(t_1, t_2, \cdots, t_d)}{t_1 \times t_2 \times \cdots \times t_d}$$

$Cost_{trans}$ represents the cost of a single memory transaction. Under the assumption that the shared/cache memory keeps any data until the last use, $Cost_{trans} \times DL$ represents the total memory transaction cost. Note that the DL model supposes proper applications of loop tiling in the latter phase. Although the applicability of tiling depends on loop dependences and other transformations, optimistic assumptions are generally acceptable in profitability analyses.

In our optimization algorithms, we employ the DL model with these extensions to GPU memory transactions. Our permutation order analysis first computes the partial derivative of the per-iteration memory cost with respect to tile size t_i, i.e., $\partial mem\_cost/\partial t_i$, which represents the variation rate of memory cost when increasing t_i. $\partial mem\_cost/\partial t_i < 0$ indicates that increasing t_i causes a decrease in memory cost, and the loop with the most negative $\partial mem\_cost/\partial t_i$ is considered to be the most profitable loop in terms of memory cost minimization. We refer a parallel loop with the minimum $\partial mem\_cost/\partial t_i$ to *coalescing parallelism*. In our approach, $\partial mem\_cost/\partial t_i$ is used as the priority for permutation among parallel loops - i.e., the ascending order of $\partial mem\_cost/\partial t_i$ is the most profitable loop permutation order (from inner to outer). Therefore, coalescing parallelism is naturally mapped to the innermost space-mapping dimension, i.e., $thread_x$.

5.3 Detection of Loop Parallelism

Our framework detects loop parallelism in the following manner. Given a set of statements *StmtSet* which are enclosed by a loop at level-k, we compute dependence distance Δ for each dependence polyhedra $\mathcal{D}^{S \to T}$ among *StmtSet*:

$$\Delta = \Theta_k^T(\vec{i'}) - \Theta_k^S(\vec{i}), \ \ (\vec{i}, \vec{i'}) \in \mathcal{D}^{S \to T}$$

Let *ActSet* denote the set of active – i.e., unsatisfied by the outer dimensions – dependence distances and *RedSet* denote the set of distances whose dependence polyhedra are covered by reduction

Algorithm 2: Thread-level transformation

```
Input  : StmtSet : set of statements $S, T, \cdots$,
         PoDG : polyhedral dependence graph
1  begin
2      SccSet := compute SCCs of PoDG;
3      $SccSet_{coalesce} := \emptyset$;
4      for each $Scc \in SccSet$ do
5          repeat
6              combo := iterator ids that is untested and
               prioritized by DL model (Section 5.2);
7              for each $S \in Scc$ do
8                  $j := combo^S$;          // an iterator id for S
9                  $\Theta_k^S(\vec{i}) := \alpha \cdot i_j + c$;  // a loop dimension
10             if detected loop kind for $\Theta_k^S(\vec{i})$, $S \in Scc$ is
               forall/reduction (Section 5.3) then
11                 keep $\Theta_k^S(\vec{i})$ as a parallel loop on Scc;
12             else if detected loop kind is sequential $\wedge$
13             no sequential loop is detected on Scc then
14                 keep $\Theta_k^S(\vec{i})$ as a sequential loop on Scc;
15         until all combinations for combo are tested;
16         if coalesced parallelism exists in parallel loops on
           Scc (Section 5.2) then
17             par_loops := all parallel loops that are
               directly/indirectly enclosing Scc;
18             sort par_loops based on DL model and prune
               extra loops; keep par_loops on Scc.
19             $SccSet_{coalesce} := \{SccSet_{coalesce}, Scc\}$;
20         else
21             apply Algorithm2(Scc, PoDG); // recursive
22     apply thread_level_fusion($SccSet_{coalesce}$, PoDG);
Output: Thread-level schedules
```

computations based on commutativity and associativity [42]. As with the classical dependence vector analysis, the loop enclosing *StmtSet* is classified as follow.

- if $\forall\Delta \in ActSet: \ \Delta = 0 \ \Rightarrow \ forall$
- else if $\forall\Delta \in ActSet: \ \Delta = 0 \vee \Delta \in RedSet \ \Rightarrow \ reduction$
- else if $\forall\Delta \in ActSet: \ \Delta \geq 0 \ \Rightarrow \ sequential$
- else $\ \Rightarrow$ *dependence violation due to illegal schedules*

5.4 Transformation Algorithms

We describe our thread and block level transformations in the combined form of SCC tree structure and loop dimensions, analogous to the schedule tree [50]. Such forms can be formally converted into the 2d+1 schedule by extracting scalar dimensions from the SCC tree structure [18, 50].

Thread-level transformation Algorithm 2 describes the thread-level scheduling algorithm, with the primary goal of mapping the parallel loop dimension with the best coalesced accesses (coalescing parallelism) to the innermost thread dimension ($thread_x$) for each statement. In this regard, fusing multiple loops should be avoided if doing so introduces loop dependences that interfere with exposing coalesced parallelism. Algorithm 2 is applied recursively to the input statements from outside to in. At each level, it distributes the statements into individual Strongly Connected Components (SCCs) [24], and then identifies parallel loops that directly enclose each SCC. Algorithm 2 traverses the SCCs in depth-first order to identify the SCCs with coalescing parallelism. Let $SCC_{coalesce}$ denote such a SCC. If $SCC_{coalesce}$ is nested in outer SCCs, the parallel loops enclosing the outer SCCs are pushed down to $SCC_{coalesce}$ level, analogous to the parallel loop chunking transformations [32, 41]. Based on the extended DL model in Section 5.2, the loops to be mapped to $thread_x$, $thread_y$, and $thread_z$ dimensions are selected per $SCC_{coalesce}$. Finally, loop fusion is performed among the set of $SCC_{coalesce}$s using a greedy algorithm such that a pair of SCCs is fused only if doing so 1) improves data locality and computation granularity; and 2) does not destroy any coalescing parallelism.

Figure 9 shows the output for jacobi-2d-alt in Figure 1. Although the i_y loop dimension was originally found on Scc_0, it is pushed down to the inner SCCs with j_x loop dimension, i.e., coalescing parallelism. In this example, loop fusion is not applied between inner Scc_{00} and Scc_{01} because the fusion would destroy the coalescing parallelism.

Algorithm 3: Block-level transformation

```
Input  : StmtSet : set of statements $S, T, \cdots$,
         PoDG : polyhedral dependence graph
1  begin
2      SccSet := compute SCCs of PoDG;
3      $SccSet_{outer} := \emptyset$;
4      for each $Scc \in SccSet$ do
5          compute parallel loops on Scc as with thread-level
           transformation (lines 5–15 in Algorithm 2);
6          if forall parallel loops on Scc exist then
7              remove reduction and extra foralls from Scc;
8              $SccSet_{outer} := \{SccSet_{outer}, Scc\}$;
9          else
10             apply Algorithm3(Scc, PoDG); // recursive
11     apply block_level_fusion($SccSet_{outer}$, PoDG);
Output: Block-level schedules
```

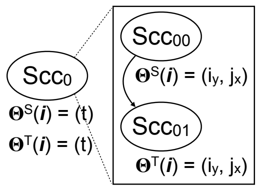


Figure 9: Thread-level output

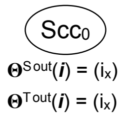


Figure 10: Block-level output

Block-level transformation In contrast to the thread-level transformations that aim to maximize coalescing parallelism, the block-level transformations aim to maximize coarse-grained parallelism. The details of the block-level algorithm, which has a similar structure (but different optimization goal) to the thread-level algorithm, is shown in Algorithm 3. The block-level algorithm traverses the SCCs in depth-first order so as to identify SCCs with the outermost forall parallelism, which is mapped to block dimensions $block_x$ and $block_y$. The block-level loop fusion is applied to such SCCs if fusion does not destroy any outermost forall parallelism.

Figure 10 shows the output of the block-level transformation for jacobi-2d-alt, where the i_x loop dimension is kept at the outermost level and mapped to $block_x$.

Benchmark	Suites	Description	Parallelism	Data Size	Hand CUDA?	Seq. GFLOP/s	
2mm	PolyBench [37]	2 Matrix Multiplications (D=A.B; E=C.D)	Forall	2,048×2,048	✓ [2]	0.34	0.48
3mm		3 Matrix Multiplications (E=A.B; F=C.D; G=E.F)	Forall	2,048×2,048	✓ [2]	0.27	0.37
atax		Matrix Transpose and Vector Multiplication	Forall+Reduction	8,192×8,192	✓ [2]	2.66	1.89
bicg		BiCG Sub Kernel of BiCGStab Linear Solver	Forall+Reduction	8,192×8,192	✓ [2]	2.14	2.18
covariance		Covariance Computation	Forall	2,000×2,000	✓ [2]	1.35	1.71
doitgen		Multiresolution Analysis Kernel (MADNESS)	Forall	256×256×256	none	1.04	4.29
gemm		Matrix-multiply C=alpha.A.B+beta.C	Forall	2,048×2,048	✓ [2]	0.41	0.61
gemver		Vector Multiplication and Matrix Addition	Forall+Reduction	8,192	none	0.61	0.57
gesummv		Scalar, Vector and Matrix Multiplication	Forall+Reduction	8,192×8,192	✓ [2]	2.59	2.19
jacobi-2d		5-Point 2-D Jacobi Stencil Computation	Forall	2,000×2000, T = 1,000	none	1.99	5.21
jacobi-2d-alt		3-Point 2-D Jacobi Stencil Computation	Forall	2,000×2000, T = 200	none	0.85	18.14
mvt		Matrix Vector Product and Transpose	Forall+Reduction	8,192×8,192	✓ [2]	0.25	0.23
symm		Symmetric matrix-multiply	Forall	2,048×2,048	none	0.23	0.12
syr2k		Symmetric rank-2k operations	Forall	2,048×2,048	✓ [2]	2.67	1.66
omriq	SPEC ACCEL^{TM} [43]	3-D MRI reconstruction (`computeQ` kernel)	Forall	32,768	✓ [44, 45]	n/a	n/a
sp-xsolve-1		Scalar Penta-diagonal Solver Kernel 1	Forall	5 × 255 × 256 × 256	none	n/a	n/a
sp-xsolve-3		Scalar Penta-diagonal Solver Kernel 3	Forall	5 × 255 × 256 × 256	none	n/a	n/a

Table 2: Evaluated benchmarks in PolyBench (data type is float) and SPEC ACCEL; Seq. GFLOP/s on Xeon & POWER8

6. Experimental Results

This section presents the results of an experimental evaluation of our compiler on two single-node platforms with GPUs. For both platforms, the GPU's error-correcting code (ECC) feature was turned on in our experiments.

6.1 Experimental Protocol

Machines: The first platform consists of a multicore Intel Xeon CPU connected to two NVIDIA Tesla M2050 (Fermi) devices via PCI-Express. We only used one of the two devices in our experiments. The platform has a 12-core Intel Xeon X5660 running at 2.82GHz with a total main memory size of 48GB. Each NVIDIA Tesla M2050 has 14 SMs, each with 32 CUDA cores, running at 1.15GHz with approximately 2.5GB of global memory. The second platform consists of a multicore IBM POWER8 CPU (S822L) and an NVIDIA Tesla K80. The platform has two 12-core IBM POWER8 CPUs (3.02GHz with a total 256GB of main memory). Each core is capable of running eight SMT threads, resulting in 192 CPU threads per platform. The NVIDIA K80 GPU has 13 SMXs, each with 192 CUDA cores, operating at up to 875MHz with 12GB of global memory, and is connected to the POWER8 via PCI-Express.
Benchmarks: Table 2 lists the 14 benchmarks from PolyBench/C 3.2 [37] and 3 benchmarks from SPEC ACCEL^{TM} [43] that were used in the experiments. For "Data Size", Table 2 only shows the largest array size evaluated. "Hand CUDA?" indicates whether reference CUDA implementations from the PolyBench/GPU and Parboil suite [2, 14, 45] exists. Finally, "Seq. GFLOP/s" shows performance for sequential CPU executions on Intel Xeon (left) and IBM POWER8 (right).
Experimental variants: Each benchmark was evaluated by comparing the following versions relative to a sequential CPU execution of the original C version. We ran each variant three times and reported the fastest run. Thanks to the exclusive use of machines, the performance numbers are quite stable with small variations.

- **PolyAST**: Optimized OpenMP C code generated using the PolyAST [42], from which our framework is derived.
- **CUDA reference**: Reference CUDA implementations from the PolyBench/GPU and Parboil suite [2, 45] if available.
- **PPCG**: Optimized CUDA code obtained using PPCG, a state-of-the-art C-to-CUDA optimizer and code generator. (version: 0.05).
- **PolyAST+GPU**: Optimized CUDA code generated using the optimization framework described in this paper. Currently, our framework generates kernel CUDA code using PolyAST+GPU, and generates host code using PPCG.

For the Intel Xeon and NVIDIA Tesla M2050 machine, we used the GNU Compiler Collection (`gcc`) 4.4.7 with the `-O3` option for a sequential and parallel C versions, and the NVIDIA CUDA Compiler (`nvcc`) 7.0.27 with the `-O3 -arch sm_20` options for all CUDA variants. For the IBM POWER8 and NVIDIA Tesla K80 machine, we used `gcc` 4.8.4 with the `-O3` option, and `nvcc` 7.5.17 with the `-O3 -arch sm_35` options.

Performance was measured in terms of elapsed microseconds from the start of the first kernel to the completion of the last kernel. We used the C library call `gettimeofday` for CPU, and CUDA Driver API `cudaEventElapsedTime` for GPU experiments. Since **our primary focus is on kernel optimization, our measurements only include kernel execution time on the CPU (for the sequential and parallel versions) or on the GPU including kernel invocation overhead (for all CUDA variants)**[7].

Optimized selection of tile sizes (i.e., grid/block sizes) can be an important optimization for GPUs. However, neither PPCG nor PolyAST+GPU support auto-tuning capability for such sizes; they are specified by users as compile-time tuning parameters. In the experiments below, We experimented with a range of square tile sizes[8] (e.g., 8×8, 16×16, 32×32) for each benchmark, and reported PPCG results using the best tile size for PPCG and our results using the best tile size for our optimizer. We also collected the performance numbers using the default tile size of PPCG, 32×32, which gave the geometric mean speedup factors of: 35.2 for PPCG and 76.2 for PolyAST+GPU on Xeon + Tesla M2050; and 30.8 for PPCG and 78.8 for PolyAST+GPU on POWER8 + Tesla K80. As shown in Figures 11 and 12, the tile size tuning generated 1.1× to 1.5× performance improvements in geometric mean.

6.2 Overall Performance Improvements

Figures 11 and 12 show speedup values relative to the sequential C version on a log scale, respectively for the Intel Xeon CPUs with NVIDIA Tesla M2050 GPU and the IBM POWER8 CPUs with NVIDIA Tesla K80 GPU.

For most benchmarks, PolyAST+GPU selected the same block-level schedules as PPCG, but different thread-level schedules, i.e., different parallelism and transformations, to achieve better mem-

[7] Overheads from CUDA Driver API, such as GPU memory allocation and data transfer between the host and the GPU are negligible for these benchmarks, relative to the kernel execution time, though they can be significant in other examples.

[8] PPCG supports option `--tile-size=<int>` to specify square tile size.

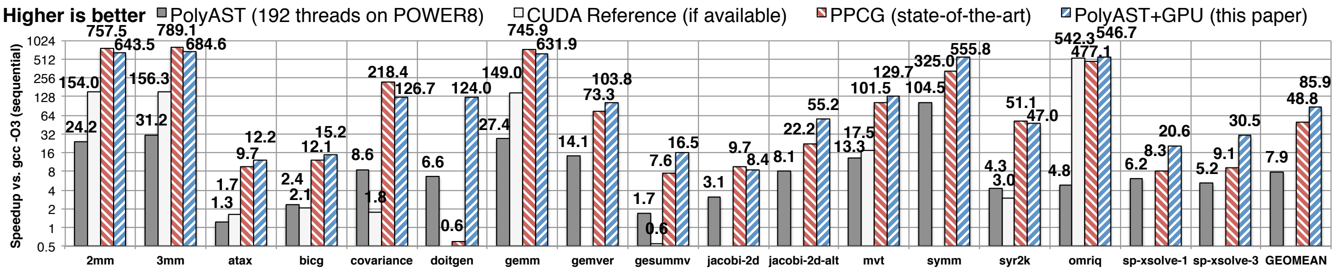


Figure 11: Performance improvements (log scale) over sequential C on the Intel Xeon + NVIDIA Tesla M2050 platform.

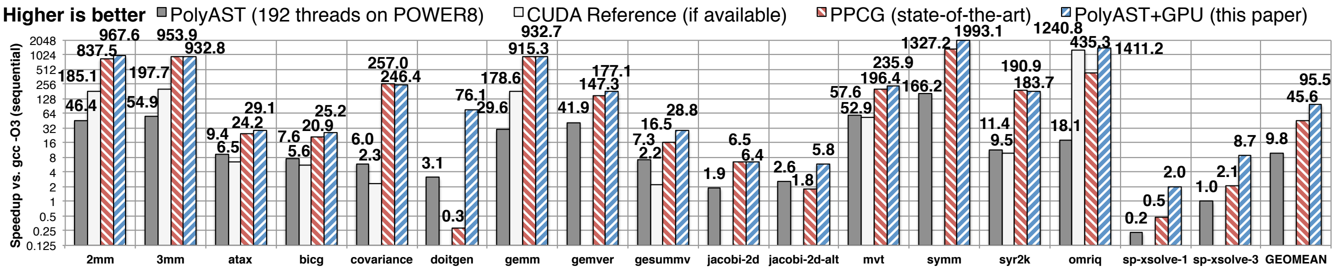


Figure 12: Performance improvements (log scale) over sequential C on the IBM POWER8 + NVIDIA Tesla K80 platform.

ory coalescing and/or reduction parallelism. For doitgen, jacobi-2d-alt, symm, sp-xsolve-1 and sp-xsolve-3, the thread-level schedules selected by PolyAST+GPU achieved better coalesced memory accesses than those by PPCG, at the cost of additional inter-thread (intra-block) synchronizations. For atax, bicg, gemver, gesummv and mvt, PolyAST+GPU benefits from thread-level reduction to increase amount of parallelism and coalesced memory accesses, in addition to the differences in thread-level schedules. As a consequence, PolyAST+GPU delivered 1.8× and 2.1× geometric mean improvements over PPCG, on Tesla M2050 and Tesla K80 GPUs respectively. These performance improvements mainly stem from: superposition of schedules that allows different optimizations to be performed at the block-level and thread-level; and the use of different cost models to guide the block/thread-specific transformations and parallelization (i.e., schedules). In contrast, PPCG and PolyAST+GPU selected the same block and thread schedules for 2mm, 3mm, covariance, gemm, jacobi-2d, and syr2k, where PPCG generated similar or better performance relative to PolyAST+GPU. An important difference exists in the code generation capabilities of the two frameworks: PPCG's code generation can select block and thread counts independently from tile sizes, and thereby improve global memory bandwidth. We plan to incorporate this code generation extension in future work.

Comparing with CUDA reference versions (PolyBench/GPU and Parboil benchmarks), PolyAST+GPU delivered better performance for all benchmarks on both systems. Note that PolyBench/GPU employs straightforward GPU parallelization strategies without complicated shared memory management, while PPCG and PolyAST+GPU automatically generate optimized code that exploits shared memory transfer code from the source code. On the other hand, Parboil mriq has highly tuned implementations including loop tiling, register and constant memory optimizations, which dramatically reduce the required bandwidth to off-chip memory [44]. Interestingly, PolyAST+GPU applied similar optimizations, e.g., loop tiling, register and shared memory enhancements, and achieved almost the same performance on Tesla M2050 and even better performance than Parboil mriq on Tesla K80. A key difference between the variants is that the Parboil version divided the main computation (`computeQ`) into a sequence of GPU kernel invocations due to the constant memory optimization, while the shared memory optimization in PolyAST+GPU enclosed the main computation in a single kernel invocation, thereby reducing the kernel invocation overhead. While PPCG and PolyAST+GPU selected the same transformation (schedule), PolyAST+GPU shows better performance on the both platform because PPCG did not utilize shared memory for this benchmark.

The selection of preferred computing resource between CPUs and GPUs is an interesting question on accelerator-equipped many-core systems [20]. Although the experiments using PolyAST+GPU and PolyAST in this paper did not show interesting trade-off, automatic selection of the best resource in addition to individual performance optimizations is an important future challenge for performance portability.

6.3 Performance Breakdown using CUDA Profiler

As discussed earlier in this paper, memory access coalescing and parallel reduction are the keys to improving GPU performance. We executed two of the benchmarks, doitgen to show the efficiency of memory coalescing and gemver to show the impact of parallel reduction, on both the Tesla M2050 and K80 with the CUDA profiler [34] enabled. Space limitations prevent us from including similar results for other benchmarks, but we conclude that a large part of performance improvement for PolyAST+GPU can be attributed to memory coalescing and/or parallel reduction.

6.3.1 Impact of Memory Coalescing

We focused on collecting measurements for gld_transactions_per_request and gst_transactions_per_request. For the doitgen benchmark, our measurements show that PolyAST+GPU achieved an average number of memory transactions per request of 1.0 for both loads and stores on the both platforms, which is ideal indicating that almost all memory accesses were coalesced. However, PPCG achieved an average transaction count of 31.4 on M2050 and 31.9

on K80 for loads and 31.5 on M2050 and 32.0 on K80 for stores, which is much worse (32 is the largest possible value).

6.3.2 Impact of Parallel Reduction

For the third kernel of gemver benchmark, achieved_occupancy and stall_exec_dependency are collected to show the impact of parallel reduction. Because a reduction loop needs to be executed sequentially by each thread in a block with PPCG, this can 1) increase memory and/or arithmetic latency due to a sequence of dependent instructions and also 2) make the GPU less busy. PolyAST+GPU achieved an average achieved occupancy of 65.3% on M2050 and 96.4% on K80, whereas PPCG achieved an average occupancy of 14.3% on M2050 and 20.0% on K80, which means PolyAST+GPU keeps the GPU busier than PPCG. Additionally, our measurements show that PPCG achieved an average percentage of stalls occurring due to dependent instructions of 63.2% on M2050 and 20.6% on K80 due to sequential reduction.

6.4 Additional Experience with Hand-tuned Code

Proc	Variants	Intel/Fermi	IBM/Kepler
CPU	Sequential	62951.4 ms	42495.8 ms
CPU	PolyAST (parallel)	2296.1 ms	1435.7 ms
GPU	PPCG	84.4 ms	46.4 ms
GPU	PolyAST+GPU	99.6 ms	45.5 ms
GPU	CUDA hand-tuned	100.1 ms	48.1 ms

Table 3: Absolute Performance comparison (sgemm)

Our primary focus is on performance portability, i.e., users can develop simple/high-level code and compilers can optimize/customize it for complex target systems such as GPUs. However, performance evaluations against hand-optimized GPU code is also important to understand how close to such well-tuned implementations the compiler-driven approach can deliver. Therefore, in addition to the case of Parboil mriq in Section 6.2, this section discusses the performance differences among 1) PolyAST, 2) PPCG, 3) PolyAST+GPU, and 4) hand-optimized CUDA programs. For hand-optimized CUDA versions, we evaluated a hand-tuned 2048×2048 matrix multiply CUDA code available from the CUDA SDK [51][9] . Table 3 shows absolute performance numbers for these variants on the Intel Xeon with Tesla M2050 and IBM POWER8 with Tesla K80.

Based on results shown in Table 3, PolyAST+GPU the performance is comparable to the hand-tuned matrix multiply CUDA code and much faster than PolyAST. It is worth mentioning that optimizations performed by PolyAST+GPU is very similar to the hand-tuned variant (e.g. loop tiling, register and shared memory enhancements). While PPCG and PolyAST+GPU selected the same transformation (schedule), PPCG shows better performance on Tesla M2050 due to the difference in code generations (details shown in Section 6.2).

7. Related Work

There is an extensive body of literature on the polyhedral compilation framework for GPUs [5, 29, 47, 49] as we discussed in Section 1. Beside end-to-end transformation frameworks, Fauzia et al. [11] proposed an approach to analyze non-coalesced accesses via dynamic trace and remap thread block geometry for better accesses. Braak et al. [46] also proposed a static optimization tool for the thread block geometry, as a part of the NVCC compiler. These approaches focus on GPU parallelism mapping while our framework additionally supports general loop transformations. Pradelle et al. [39] introduced two new operators for polyhedral compilers: focalisation and defocalisation, which largely reduce the complexity of multi-level loop tiling targetting deep hardware hierarchy, e.g., multi-level cache.

Many previous studies aim to facilitate GPU programming by providing high-level abstractions of GPU programming. They often introduce directives and/or language constructs expressing parallelism for semi-/fully- automated code generations and optimizations for GPUs. OpenACC [35] is a widely-recognized directive-based programming model for heterogeneous systems. OpenMPC[27] transforms extended OpenMP programs into CUDA applications. For JVM-based languages, many approaches [10, 19, 22, 28] provide high-level abstractions of GPU programming. Velociraptor [13] compiles MATLAB and Python to GPUs. A GPGPU compiler [53] optimizes CUDA programs by performing several optimization techniques including memory access vectorization and coalescing. Many of them rely on AST-based optimizations. In contrast, our approach takes a sequential C program with `SCoP` directives and performs fully automatic loop transformation and codegen using the polyhedral model. However, it is worth mentioning that these AST-based optimizations can also be applied afterwards to attain further performance improvements.

8. Conclusions and Future Work

In this paper, we proposed a new polyhedral compilation framework for optimizing GPU kernels. To enable efficient exploitation of two levels of hardware parallelism in the GPU, blocks and threads, we introduced the concept of superposition of polyhedral schedules so that different loop transformations and parallelization can be performed for GPU blocks and threads. This approach supports a larger optimization space than existing approaches such as PPCG, which currently uses the same schedule for blocks and threads. Our approach uses different optimization strategies, i.e., different cost models, at the thread and block levels to guide the selection of transformations/parallelization (schedules) at those levels. Our experimental results demonstrate the effectiveness of our approach relative to a state-of-the-art polyhedral optimizer for GPUs, PPCG. The primary focus of this paper is on performance portability, so that users can develop simple/high-level code and compilers can optimize/customize it for complex target systems such as GPUs. We believe that compilers can help non-expert users achieve better productivity with respect to performance portability. For future work, we plan to extend PolyAST+GPU with support for special loop tilings [4, 16, 17, 21] for stencil algorithms, inter-kernel parallelization, and custom code generation for host code. We also plan to extend our infrastructure by migrating from the CLooG code generator to ISL.

Acknowledgments

This work is partially supported by the Intel Corporation with matching funds from the NSF under the Innovation Transition (InTrans) Program (CCF-1436827). We are also grateful to IBM Canada Lab for a CAS Fellowship award that partially supported this work. Finally, we would like to thank all members of the Habanero Extreme Scale Software research group at Rice University for their ongoing feedback on this research.

References

[1] The Polyhedral Compiler Collection. http://www.cs.ucla.edu/~pouchet/software/pocc/.

[2] PolyBench/GPU Implementation of PolyBench codes for GPU processing. http://web.cse.ohio-state.edu/~pouchet/software/polybench/GPU/.

[9] The code is slightly modified for sgemm.

[3] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S. v. Haastregt, A. Kravets, A. Lokhmotov, R. David, and E. Hajiyev. Pencil: A platform-neutral compute intermediate language for accelerator programming. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 138–149, Oct 2015. doi: 10.1109/PACT.2015.17. URL `http://ieeexplore.ieee.org/document/7429301/`.

[4] V. Bandishti, I. Pananilath, and U. Bondhugula. Tiling stencil computations to maximize parallelism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 40:1–40:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. ISBN 978-1-4673-0804-5. URL `http://dl.acm.org/citation.cfm?id=2388996.2389051`.

[5] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 244–263, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-11969-7, 978-3-642-11969-9. doi: 10.1007/978-3-642-11970-5_14. URL `http://dx.doi.org/10.1007/978-3-642-11970-5_14`.

[6] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2229-7. doi: 10.1109/PACT.2004.11. URL `http://dx.doi.org/10.1109/PACT.2004.11`.

[7] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proc. of PLDI '08*, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375595. URL `http://doi.acm.org/10.1145/1375581.1375595`.

[8] CANDL. CANDL: Data dependence analysis tool in the polyhedral model. `http://icps.u-strasbg.fr/~bastoul/development/candl`.

[9] P. Chatarasi, J. Shirako, and V. Sarkar. Polyhedral optimizations of explicitly parallel programs. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 213–226, 2015. URL `http://ieeexplore.ieee.org/document/7429307/`.

[10] 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*, PLDI '12, pages 1–12, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9. doi: 10.1145/2254064.2254066. URL `http://doi.acm.org/10.1145/2254064.2254066`.

[11] N. Fauzia, L.-N. Pouchet, and P. Sadayappan. Characterizing and enhancing global memory data coalescing on GPUs. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 12–22, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4799-8161-8. URL `http://dl.acm.org/citation.cfm?id=2738600.2738603`.

[12] J. Ferrante, V. Sarkar, and W. Thrash. On Estimating and Enhancing Cache Effectiveness. *Proc. LCPC 91*, 589:328–343, 1991.

[13] R. Garg and L. Hendren. Velociraptor: An embedded compiler toolkit for numerical programs targeting cpus and gpus. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 317–330, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2809-8. doi: 10.1145/2628071.2628097. URL `http://doi.acm.org/10.1145/2628071.2628097`.

[14] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, , and J. Cavazos. Auto-tuning a High-Level Language Targeted to GPU Codes. In *Proceedings of Innovative Parallel Computing (InPar '12)*, 2012.

[15] T. Grosser, A. Größlinger, and C. Lengauer. Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters*, 22(4), 2012. URL `http://dblp.uni-trier.de/db/journals/ppl/ppl22.html#GrosserGL12`.

[16] T. Grosser, A. Cohen, P. H. J. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege. Split tiling for GPUs: Automatic parallelization using trapezoidal tiles. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 24–31, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2017-7. doi: 10.1145/2458523.2458526. URL `http://doi.acm.org/10.1145/2458523.2458526`.

[17] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege. Hybrid hexagonal/classical tiling for GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 66:66–66:75, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2670-4. doi: 10.1145/2544137.2544160. URL `http://doi.acm.org/10.1145/2544137.2544160`.

[18] T. Grosser, S. Verdoolaege, and A. Cohen. Polyhedral ast generation is more than scanning polyhedra. *ACM Trans. Program. Lang. Syst.*, 37 (4):12:1–12:50, July 2015. ISSN 0164-0925. doi: 10.1145/2743016. URL `http://doi.acm.org/10.1145/2743016`.

[19] A. Hayashi, M. Grossman, J. Zhao, J. Shirako, and V. Sarkar. Accelerating Habanero-Java Programs with OpenCL Generation. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '13, pages 124–134, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2111-2. doi: 10.1145/2500828.2500840. URL `http://doi.acm.org/10.1145/2500828.2500840`.

[20] A. Hayashi, K. Ishizaki, G. Koblents, and V. Sarkar. Machine-Learning-based Performance Heuristics for Runtime CPU/GPU Selection. In *Proceedings of the Principles and Practices of Programming on The Java Platform*, PPPJ '15, pages 27–36, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3712-0. doi: 10.1145/2807426.2807429. URL `http://doi.acm.org/10.1145/2807426.2807429`.

[21] J. Holewinski, L.-N. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 311–320, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1316-2. doi: 10.1145/2304576.2304619. URL `http://doi.acm.org/10.1145/2304576.2304619`.

[22] K. Ishizaki, A. Hayashi, G. Koblents, and V. Sarkar. Compiling and optimizing java 8 programs for gpu execution. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 419–431, 2015. URL `http://ieeexplore.ieee.org/document/7429325/`.

[23] ISL. Integer set library. `http://isl.gforge.inria.fr`.

[24] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. ISBN 1-55860-286-0.

[25] KHRONOS GROUP. OpenCL 2.1 Provisional API Specification, Version 2.1, 2013. `https://www.khronos.org/registry/cl/`.

[26] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan. When Polyhedral Transformations Meet SIMD Code Generation. volume 48, pages 127–138, New York, NY, USA, June 2013. ACM. doi: 10.1145/2499370.2462187. URL `http://doi.acm.org/10.1145/2499370.2462187`.

[27] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-7559-9. doi: 10.1109/SC.2010.36. URL `http://dx.doi.org/10.1109/SC.2010.36`.

[28] A. Leung, O. Lhoták, and G. Lashari. Automatic parallelization for graphics processing units. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 91–100, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-598-7. doi: 10.1145/1596655.1596670. URL `http://doi.acm.org/10.1145/1596655.1596670`.

[29] A. Leung, N. Vasilache, B. Meister, M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin. A mapping path for multi-GPGPU accelerated computers from a portable high level programming ab-

straction. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, pages 51–61, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-935-0. doi: 10.1145/1735688.1735698. URL http://doi.acm.org/10.1145/1735688.1735698.

[30] A. W. Lim, G. I. Cheong, and M. S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, pages 228–237, New York, NY, USA, 1999. ACM. ISBN 1-58113-164-X. doi: 10.1145/305138.305197. URL http://doi.acm.org/10.1145/305138.305197.

[31] Mark Harris. Optimizing parallel reduction in CUDA, 2007. http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf.

[32] V. K. Nandivada, J. Shirako, J. Zhao, and V. Sarkar. A Transformation Framework for Optimizing Task-Parallel Programs. *ACM Trans. Program. Lang. Syst.*, 35(1):3:1–3:48, Apr. 2013. ISSN 0164-0925. doi: 10.1145/2450136.2450138. URL http://doi.acm.org/10.1145/2450136.2450138.

[33] NVIDIA Corporation. CUDA C PROGRAMMING GUIDE 7.0, 2014. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[34] NVIDIA Corporation. PROFILER USER'S GUIDE 7.5, 2015. http://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf.

[35] OpenACC forum. The OpenACC Application Programming Interface, Version 2.0, 2013. http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf.

[36] OpenScop. Openscop specification and library. http://icps.u-strasbg.fr/ bastoul/development/openscop/.

[37] PolyBench. The polyhedral benchmark suite. http://www.cse.ohio-state.edu/~pouchet/software/polybench/.

[38] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-7559-9. doi: 10.1109/SC.2010.14. URL http://dx.doi.org/10.1109/SC.2010.14.

[39] B. Pradelle, B. Meister, and M. Baskaran. Scalable hierarchical polyhedral compilation. In *The 45th International Conference on Parallel Processing*, Paris, France, August 2016. URL https://www.researchgate.net/publication/230759922_Joint_Scheduling_and_Layout_Optimization_to_Enable_Multi-Level_Vectorization?ev=prf_pub.

[40] V. Sarkar. Automatic Selection of High Order Transformations in the IBM XL Fortran Compilers. *IBM J. Res. & Dev.*, 41(3), May 1997.

[41] J. Shirako, J. M. Zhao, V. K. Nandivada, and V. N. Sarkar. Chunking parallel loops in the presence of synchronization. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 181–192, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-498-0. doi: 10.1145/1542275.1542304. URL http://doi.acm.org/10.1145/1542275.1542304.

[42] J. Shirako, L.-N. Pouchet, and V. Sarkar. Oil and water can mix: An integration of polyhedral and ast-based transformations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 287–298, Piscataway, NJ, USA, 2014. IEEE Press. ISBN 978-1-4799-5500-8. doi: 10.1109/SC.2014.29. URL http://dx.doi.org/10.1109/SC.2014.29.

[43] SPEC. Spec accel benchmark. https://www.spec.org/accel/.

[44] S. S. Stone, J. P. Haldar, S. C. Tsao, W. m. W. Hwu, B. P. Sutton, and Z. P. Liang. Accelerating advanced mri reconstructions on gpus. *J. Parallel Distrib. Comput.*, 68(10):1307–1318, Oct. 2008. ISSN 0743-7315. doi: 10.1016/j.jpdc.2008.05.013. URL http://dx.doi.org/10.1016/j.jpdc.2008.05.013.

[45] J. A. Stratton, C. Rodrigrues, I.-J. Sung, N. Obeid, L. Chang, G. Liu, and W.-M. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, Urbana, Mar. 2012.

[46] G. J. van den Braak, B. Mesman, and H. Corporaal. Compile-time GPU memory access optimizations. In *International Conference on Embedded Computer Systems*, 2010.

[47] N. Vasilache, B. Meister, M. Baskaran, and R. Lethin. Joint scheduling and layout optimization to enable multi- level vectorization. In *IMPACT-2: 2nd International Workshop on Polyhedral Compilation Techniques, Paris, France, January*, Paris, France, Jan 2012. URL https://www.researchgate.net/publication/230759922_Joint_Scheduling_and_Layout_Optimization_to_Enable_Multi-Level_Vectorization?ev=prf_pub.

[48] S. Verdoolaege. isl: An Integer Set Library for the Polyhedral Model. In K. Fukuda, J. Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 299–302. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-15581-9. doi: 10.1007/978-3-642-15582-6_49. URL http://dx.doi.org/10.1007/978-3-642-15582-6_49.

[49] 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.*, 9(4):54:1–54:23, Jan. 2013. ISSN 1544-3566. doi: 10.1145/2400682.2400713. URL http://doi.acm.org/10.1145/2400682.2400713.

[50] S. Verdoolaege, S. Guelton, T. Grosser, and A. Cohen. Schedule Trees. In *IMPACT - 4th Workshop on Polyhedral Compilation Techniques, associated with HiPEAC*, Vienna, Austria, Jan. 2014. ACM. URL https://hal.inria.fr/hal-00911894.

[51] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 31:1–31:11, Piscataway, NJ, USA, 2008. IEEE Press. ISBN 978-1-4244-2835-9. URL http://dl.acm.org/citation.cfm?id=1413370.1413402.

[52] D. G. Wonnacott. *Constraint-based Array Dependence Analysis*. PhD thesis, College Park, MD, USA, 1995. UMI Order No. GAX96-22167.

[53] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU compiler for memory optimization and parallelism management. *SIGPLAN Not.*, 45(6):86–97, June 2010. ISSN 0362-1340. doi: 10.1145/1809028.1806606. URL http://doi.acm.org/10.1145/1809028.1806606.