Characterizing and Enhancing Global Memory Data Coalescing on GPUs

Naznin Fauzia Louis-Noël Pouchet P. Sadayappan

The Ohio State University

{fauzia,pouchet,saday}@cse.ohio-state.edu

Abstract

Effective parallel programming for GPUs requires careful attention to several factors, including ensuring coalesced access of data from global memory. There is a need for tools that can provide feedback to users about statements in a GPU kernel where non-coalesced data access occurs, and assistance in fixing the problem. In this paper, we address both these needs. We develop a two-stage framework where dynamic analysis is first used to detect and characterize uncoalesced accesses in arbitrary PTX programs. Transformations to optimize global memory access by introducing coalesced access are then implemented, using feedback from the dynamic analysis or using a model-driven approach. Experimental results demonstrate the use of the tools on a number of benchmarks from the Rodinia and Polybench suites.

Keywords GPU, PTX, coalescing, dynamic analysis, locality, program transformation, polyhedral compilation

1. Introduction

Parallel programming is hard and programming GPUs is even harder. In order to achieve high performance, it is essential to address many aspects, such as avoidance/minimization of control divergence among threads, ensuring sufficiently high degrees of parallelism to effectively mask memory latency, and achieving coalesced access to global memory. In contrast to shared-memory parallel programs for CPUs, where stride-1 access to memory by each thread is very efficient, for effective utilization of memory bandwidth on GPUs, adjacent threads must access adjacent data elements in global memory. Thus coalesced access generally implies that a single thread will not access contiguous memory locations in adjacent iterations of a loop. Therefore many programs directly converted from OpenMP to CUDA without a fundamental change to the loop structure exhibit uncoalesced access to global memory.

While attempts have been made to develop tools to ease the development of GPU applications [5–7, 20, 22–24, 36], many existing CUDA applications still suffer from uncoalesced accesses. Thus, there is a strong need for tools to assist application developers develop codes that exhibit a high fraction of coalesced accesses. Unless the programmer is able to detect the problem, other optimization tools depending on programmer input (e.g., [13, 21, 33]) are of little help. If uncoalesced access is detected, the programmer can then seek to transform the code.

Existing static transformation approaches to enhance coalesced access are only applicable to restricted classes of programs. In this paper, we overcome the limitations of these purely static approaches by combining the benefits of dynamic analysis with static transformations. When dynamic analysis on traces generated from the program detects uncoalesced accesses, some recommendations are made depending on the overall memory access pattern. In many cases, program transformations implementing a different yet semantically equivalent thread geometry can increase data access coalescing. In this work we target arbitrary PTX codes for the dynamic analysis and key program transformations, thereby covering both CUDA and OpenCL programs. We also propose a dedicated CUDA program transformation framework for a subset of CUDA programs which relies on polyhedral analysis to enable more aggressive program restructuring for memory coalescing when applicable. This paper makes the following contributions:

- A dynamic analysis tool for analyzing arbitrary PTX codes for precisely characterizing run-time uncoalesced memory accesses, along with suggestions for potential improvement strategies.
- A PTX/CUDA transformation framework that implements a remapping of work among threads to improve global memory access coalescing.

The rest of the paper is organized as follows. Sec. 2 contains background information on GPU global memory access and the PTX intermediate representation. Sec. 3 elaborates on the design and implementation of our dynamic analysis, and Sec. 4 presents our program transformation approach. Sec. 5 presents an experimental evaluation, and related work is discussed in Sec. 6 before concluding.

2. Background and Overview

2.1 GPU Architecture

Computation GPUs are designed for high computational throughput. GPUs typically contain hundreds of cores (streaming processors) arranged in tightly coupled groups of 8-32 scalar processors per streaming multi-processor (SMs). Parallel threads are grouped into thread blocks that are scheduled on a SM and cannot migrate. Threads are spawned in 1-, 2-, or 3-dimensional rectangular groups of cooperative threads, called blocks (CUDA) or work-groups (OpenCL). A 1-, 2- or 3-dimensional grid of blocks is used to schedule the thread blocks. Both the size and number of thread blocks are fixed when launching a GPU kernel and cannot be changed after the threads have launched. We note $\vec{G} = (bdim_x, bdim_y, bdim_z, tdim_x, tdim_y, tdim_z)$ the geometry of the thread space: the sizes of a thread block in each dimension are denoted $tdim_x$, $tdim_y$ and $tdim_z$. In the case of a 2D or 1D geometry for a thread block, we simply set $tdim_z$ or $tdim_z$ and $tdim_y$ to 1. The grid of thread blocks also has a 3D geometry, the dimension in each dimension is typically computed from the total number of threads (e.g., problem size) divided by the thread block size in the dimension. A thread in the computation is uniquely identified by $\vec{t} = (b_x, b_y, b_z, t_x, t_y, t_z)$ a vector of 6 integers, where each component can range between 0 and the size in $\vec{G}$ from the corresponding component.

Memory The GPU memory hierarchy consists of global memory (shared across thread blocks), shared memory (shared only among the threads in a single block), local memory and registers. Global memory is the largest in terms of size, but also the slowest. Shared memory is faster than global memory but limited in size. In modern GPUs, each Streaming Multiprocessor (SM) has 64KB of fast memory that can be partitioned between L1 cache for global memory and shared memory. The 64KB space can be either divided into two 32KB partitions, or 48KB and 16KB. Global memory coalescing (described in Sec. 2.2) leads to efficient usage of the available bandwidth between global memory and shared memory or L1 cache.

2.2 Global Memory Coalescing

When a kernel is launched on a GPU, it is executed by all the threads in parallel. A typical scenario is to have a global memory reference in the kernel that is executed by all threads, but requesting different memory addresses for each thread, as shown in Lst. 1.

```
__global__ void kernel(float* a) {
  int tid = threadIdx.x;
  a[tid] = 1.0;
}
```

Listing 1: CUDA code

These memory requests are grouped into a number of memory transactions by the GPU in the current scheduling unit for a thread block to maximize the bandwidth usage. That is, the memory transaction is computed based on the region of data requested by a set of active threads. When consecutive threads access consecutive global memory region (as in Lst. 1) then a single transaction may be implemented, and accesses are *coalesced*. With modern GPUs (compute capability 1.2 or higher) consecutive threads are no longer required to get coalesced access, it is enough that the set of data accessed by the set of threads considered (e.g., threads having the same t_y, t_z but different t_x) is a contiguous chunk of memory [31].

When the data region accessed by threads is not contiguous (e.g., for an access `a[tid * N]` instead of `a[tid]` in Lst. 1, leading to a poor spatial data locality), then it is not possible anymore to pack the data request into a single, large transaction: the reference leads to *uncoalesced* accesses. Up to one transaction per thread will be needed, dramatically reducing the effective bandwidth. Accessing non-contiguous memory from the global memory incurs significant performance penalties [31]. One approach to possibly hide the effects uncoalescing is to use the shared memory for caching the accesses (see Sec. 4.4), however this requires complex code restructuring. In this work we first take the approach of changing the thread geometry (i.e., which threads will end up being grouped together at the time of issuing the memory transactions) to improve spatial data locality and global memory coalescing before resorting to shared memory usage.

2.3 Overview of the Framework

Fig. 1 depicts the overall steps of our approach, which consists of 4 stages: instrument, execute, analyze, and transformation.

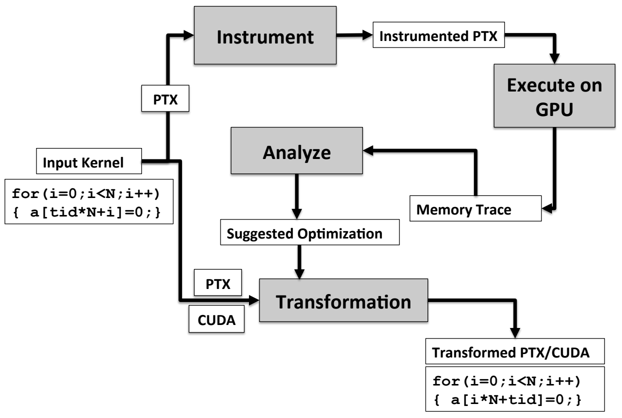

Figure 1: Overall Flow Chart of Our Approach

We use Parallel Thread Execution (PTX), an intermediate language for kernels designed to run efficiently on NVIDIA GPUs [30]. High-level language compilers like nvcc [31] or clang [34] can generate PTX instructions from CUDA or OpenCL codes. First, the input PTX code is instrumented and executed on a GPU to generate its memory traces. We use Ocelot [18], a compiler framework for PTX code analysis on heterogeneous systems, to instrument PTX codes. Next, an analysis is performed on the memory traces to characterize coalesced and uncoalesced accessed, with recommendation of possible transformations based on the dynamic analysis results (Sec. 3). Finally, based on the recommendation the code may be transformed. An small and always-applicable set of thread geometry permutations can be applied on arbitrary PTX programs (Sec. 4.2), and the dynamic analysis can be re-run on the resulting program to observe whether coalescing has improved on the working dataset. Alternatively a more powerful geometry transformation framework but which operates only a subset of CUDA programs can be applied to maximize coalesced accesses (Sec. 4.3).

3. Dynamic Analysis of Uncoalesced Accesses

3.1 Instrumentation and Execution

The input to the dynamic analysis is the PTX code of a kernel we wish to investigate. First, the PTX kernel is instrumented by inserting a function call to a device function that stores all the necessary information about the memory access after each global memory access to compute and store a trace of the program. The memory trace of an instrumented PTX code is generated simply by executing it on the GPU. The trace can be dependent on the values of the input data, and so the results may vary with different input datasets. In such cases, it is recommended to perform the dynamic analysis with a variety of representative input datasets, determining such sets is out of the scope of the present paper. An example of a trace excerpt is shown in Lst. 2:

```
#tx  ty  tz  static_id  Load(1)/Store(2)  Address  Dyn_id
0  0  0  31   1  30066082304  0
0  0  0  34   2  30066081792  0
0  0  0  31   1  30066082320  1
0  0  0  34   2  30066081796  1
0  0  0  31   1  30066082336  2
0  0  0  34   2  30066081800  2
0  1  0  31   1  30066082320  0
0  1  0  34   2  30066081796  0
0  1  0  31   1  30066082336  1
0  1  0  34   2  30066081800  1
0  1  0  31   1  30066082352  2
0  1  0  34   2  30066081804  2
```

Listing 2: Trace

where $(tid_x, tid_y, tid_z, static\_id, type, address, dynamic\_id)$ is the tuple forming a trace entry. *static_id* is the unique identifier of the memory instruction in the PTX code generating this trace entry. *dynamic_id* is the unique identifier of the *instance* of the static instruction, for a particular $(tid_x, tid_y, tid_z, static\_id)$ value. Dynamic ids are needed to model loops in the kernel code iterating memory instructions. The tuple $(tid_x, tid_y, tid_z, static\_id, dynamic\_id)$ is necessarily unique in a trace. The actual memory address accessed by this instruction as well as the type of access (load or store) is also captured. In this work only trace entries with identical $(static\_id, dynamic\_id)$ values may execute in parallel and are candidate for coalescing analysis.

3.2 Dynamic Analysis Algorithm

Our analysis algoritm detailed in Alg. 1 scans through the memory trace file to characterize memory access patterns and produce meaningul, per-instruction statistics about the access strides and the potential for coalescing. The algorithm takes as input W the scheduled block size (e.g., $W = 16$ for a half warp for compute capability lower than 2.0).

Algorithm 1: Memory Trace Analysis Algorithm

```
Input  : $T$: Memory trace, viewed as a map
         T[$static\_id$][$dynamic\_id$][$tid_y$][$tid_z$][$tid_x$] = $addr$
         $W$: Warp scheduling size
Output: Report on coalescing and possible optimization strategy
begin
    for all unique static_id in T do
        $(min\_stride, max\_stride, avg\_stride, allStr) \leftarrow (\infty, 0, 0, 0)$
        $AllAddrs \leftarrow$ emptyVector()
        for all unique dynamic_id in T[static_id] do
            // Take the trace entries corresponding to candidate
            // accesses for coalescing.
            $M \leftarrow$ T[$static\_id$][$dyn\_id$][*]
            // Build the linearized list of memory addresses accessed.
            $i \leftarrow 0$
            for all $(tid_y, tid_z, tid_x)$ in M in lexicographic order do
                $V[i] \leftarrow$ M[$tid_y$][$tid_z$][$tid_x$]
                $i \leftarrow i + 1$
            $c \leftarrow 0$
            while $c < i$ do
                // Sort $W$ consecutive elements of $V$ to compute the sorted
                // list of memory addresses accessed by $W$ consecutive
                threads.
                $V[c..c+W] \leftarrow$ sortByIncreasingValue($V[c..c+W]$)
                for $j \in [0..W-1]$ do
                    $stride \leftarrow V[c+j+1] - V[c+j]$
                    $min\_stride \leftarrow min(min\_stride, stride)$
                    $max\_stride \leftarrow max(max\_stride, stride)$
                    $avg\_stride \leftarrow avg\_stride + stride$
                $avg\_stride \leftarrow avg\_stride / W$
                $c \leftarrow c + W$
            $AllAddrs \leftarrow concat(AllAddrs, V)$
        // Check if the entire memory space accessed is contiguous.
        $AllAddrs \leftarrow$ sortByIncreasingValue($AllAddrs$)
        for $i \in [0..AllAddrs.size - 1]$ do
            $allStr \leftarrow max(allStr, AllAddrs[i+1] - AllAddrs[i])$
        // Produce report and suggestions.
        Print(static_id, load/store type)
        Print(min_stride,max_stride,avg_stride)
        if $max\_stride \leq sizeof(data\_type)$ then
            Print("coalesced")
        else
            Print("uncoalesced")
            if $allStr > sizeof(data\_type)$ then
                Print("accesses cannot be all coalesced")
            else
                Print("Suggest thread geometry transformations")
                if instruction is a load then
                    Print("Suggest also shared memory usage")
```

4. Compiler Transforms for Data Coalescing

In this section we present a compiler framework to improve data coalescing. We first present an approach that uses our dynamic analysis to empirically drive a re-scheduling of the CUDA threads, that is a change of the thread block geometry in Sec. 4.2.1. This approach operates on arbitrary CUDA/PTX programs. We then present a purely compile-time approach that only requires very basic static analysis of the references in a CUDA/PTX program to compute a new thread block geometry aimed at reducing the number of uncoalesced accesses in Sec. 4.2.2. To address cases of uncoalesced accesses which require using loops from the thread code to formulate a new thread geometry, we focus on a subset of CUDA programs that can be handled with the polyhedral compilation framework [6] and discuss our method in Sec. 4.3. Finally, complementary transformations for load optimization is presented in Sec. 4.4.

4.1 Overview

An uncoalesced access arises from non-consecutive data being accessed by threads in the same warp along the t_x dimension. A rescheduling of the threads here is analogous to a loop permutation: for instance to permute dimensions t_x and t_y, we (1) update the geometry to become $\vec{G} = (bdim_y, bdim_x, bdim_z, tdim_y, tdim_x, tdim_z)$; and (2) substitute each occurrence of `threadIdx.x` by `threadIdx.y` (and conversely) in the CUDA/PTX program.

Our dynamic analysis presented previously provides two key pieces of information to drive the profitability of a program transformation: which reference is uncoalesced (and its stride), and how often a reference is executed. To improve data coalescing, we seek a program transformation essentially based on finding a new geometry for the threads, such that accesses are consecutive in memory along the newly computed t_x dimension.

4.2 Computing a New Thread Block Geometry

This transformation stage takes two inputs: (1) the original geometry, and (2) the AST of the thread code. CUDA programs have the key property of allowing any bijective transformation of the thread geometry. That is, all inter-block and inter-thread dimensions are fully data-parallel and hence interchangeable without violating program semantics. We remark that at this stage we do not require any additional property of the code such as having affine control or data-flow: we simply exploit the parallelism readily available through the thread geometry, and seek an alternative geometry with improved coalescing of data accesses.

4.2.1 Empirical search using dynamic analysis

Given $\vec{t}$ the vector identifying the threads. An uncoalesced access on a reference R arises typically because for two consecutive values of t_x, the same reference accesses non-consecutive data. Our first approach seeks a permutation of thread block dimensions so that the fastest varying one does not incur non-unit stride accesses by R. An iterative approach to test all possibilities is effective and straightforward in this case: only 3 possibilities exist (one for each of the 3 original thread block dimensions when used as the t_x component of the geometry), they are all semantically correct, so one can implement the 3 alternatives and run the dynamic analysis on each of the 3 cases. The rest of the permutation, that is which of the original thread block dimensions will be used as the new t_y and t_z dimensions, can be chosen arbitrarily as it will not affect coalescing.

Then, the dynamic analysis presented previously is run on each of the three cases, the result is inspected and the configuration providing the lowest number of uncoalesced accesses is retained. In our experiments, this simple approach successfully solved uncoalesced accesses for the benchmarks Gaussian Elimination and Cell. We remark that this approach implicitly assumes that the test data set used during dynamic analysis is representative of the typical control-flow for the program.

4.2.2 Model-driven geometry transformation

Another approach that does not rely on dynamic analysis is possible when the reference can be successfully characterized using standard static analysis. Contrary to the previous empirical approach, this model-driven framework requires a static analysis of all the references in the CUDA/PTX program to gather information for the cost model. The objective and constraints do not change: we are seeking a permutation of the geometry, and support arbitrary CUDA/PTX programs as input.

Given an array reference R `A[pos]`[1], where `pos` is the expression used to index the array, we first perform static analysis on `pos` (possibly inspecting the entire kernel code) to uncover key properties on the relationship between thread ids in each dimension and the value of `pos`. Precisely, we analyze each sub-expression involved in the computation of `pos`, so as to determine:

1. if `pos` is of the form `x + b`, where x is a thread id and b is invariant to x; otherwise `pos` is of the form `b`.
2. If b above is of the form `y*c + d` where y is a thread id, and c is greater than 1.
3. The list of all thread ids used to compute `pos`.

In other words, we perform static analysis of the expression `pos` for the purpose of finding (1) which thread id, if any, occurs without any multiplier (it will therefore be suitable for coalesced accesses along this dimension); (2) which thread id, if any, occurs with a non-unit multiplying factor (it will not be suitable for coalesced accesses); and (3) which thread id is used to compute the reference (any thread id not involved will be suitable for stride-0 accesses). Standard dataflow analysis can be used, in particular computing

[1] We only discuss the case of linearized array references

reaching definitions for `pos` [2] and the read/write sets of these definitions.

We denote $\vec{c} = (c_x, c_y, c_z)$ a vector of 3 Booleans, such that if c_x is set to 1 then the thread dimension t_x matches x in the pattern $x + b$ above, 0 otherwise. Similarly for c_y when t_y matches x in the pattern $x + b$, etc. We denote $\vec{u} = (u_x, u_y, u_z)$ another vector of 3 Booleans, with the semantics that u_x is set to 1 if t_x matches y in the pattern $b = y * c + d$ above, and so on for the other coordinates. Finally we denote $\vec{z} = (z_x, z_y, z_z)$ the vector where z_x is set to 1 if t_x is used to compute the value `pos`, and so on.

We are now equipped to formulate an Integer Linear Program whose optimization provides us with the thread dimension to use for t_x. To achieve this, we implement the solution in the form of a permutation matrix for the three thread block coordinates. We first introduce 9 Boolean variables $p_{i,j}$, one for each of the elements in a 3×3 permutation matrix which models the 3D thread geometry permutation we need to apply to maximize coalesced accesses. To ensure it is a true permutation matrix, we add the constraint $\sum_i p_{i,j} = 1$, one for each of the three values for j, and $\sum_j p_{i,j} = 1$, for each value of i, that is a total of six constraints. Then, for each reference R, we add constraints to capture the cost of large-stride accesses (that is, t_x would be true in $\vec{u}$), stride-1 accesses (t_x is true in $\vec{c}$) and stride-0 accesses (t_x is true in $\vec{z}$). We use equal cost of 1 for stride-0 and stride-1 accesses, and a fictitious large value N for the cost of large-stride accesses (N must be greater than the number of references). We get for a reference R:

$$C_R = \sum_{i=1}^{3} c(i).p_{1,i} + \sum_{i=1}^{3} z(i).p_{1,i} + \sum_{i=1}^{3} N.u(i).p_{1,i}$$

where only the cost associated to the dimension used as "inner-most" (the first row of the permutation matrix, capturing the output t_x dimension) is being modeled. The final optimization problem, where $p_{i,j}$ are Boolean unknowns, is then:

$$\begin{aligned} & P = \sum_R C_R \\ \text{minimize} \quad & P \quad \text{s.t.} \sum_i p_{i,j} = 1,\ \forall j \wedge \sum_j p_{i,j} = 1,\ \forall i \end{aligned}$$

Because of the constraints on P to output a permutation matrix, we are guaranteed to find a permutation that minimizes the number of large-stride references. If multiple solutions with identical cost exist, one may be selected randomly. Further refinement can be achieved by weighting each cost by the number of times the reference is accessed. The optimal solution is implemented as the geometry permutation encoded in the permutation matrix is applied to the CUDA/PTX program, altering both the thread block geometry and substituting thread ids in the code as necessary.

4.3 Geometry and Thread Code Transformations

Threads are grouped into the same warp according to their threadIdx.x. Therefore, these threads should read/write consecutive global memory locations as much as possible. Listing 3 shows a common example where each thread is assigned a complete row of a matrix, which leads to strided memory access of threads in the warp.

```
__global__
void kernel(float *A, float value){
  int tx = threadIdx.x + blockIdx.x*blockDim.x;
  for (int i = 0; i < N; i++){
R:    A[tx][i] = value;
  }
}
```

Listing 3: Inefficient Global Memory Store Example

Loop `i` is a parallel loop and therefore we can re-distribute the stores among the threads to ensure coalesced access. That is, we can use the `i` loop *in the thread code* as an additional source of parallel threads, and compute a new 2D geometry in place of the original 1D one, such that accesses will be coalesced for t_x: in this example this amounts to (1) making i the t_x dimension, updating b_x correspondingly to capture N threads; and (2) make the original t_x dimension t_y in the transformed code.

The example above highlights a key issue when transforming codes for coalescing: that *intra-thread loops may need to be analyzed and transformed into CUDA threads* to ensure proper coalescing without data layout transformations. Indeed, in our benchmark suite the Rodinia Myocyte and all PolyBench/GPU kernels are programmed using 1-dimensional thread block geometry, with parallel loops inside the kernel code. Such programs require deep static analysis and transformation of both the geometry and kernel codes to implement efficient data accesses.

To address this problem, we now present a polyhedral-based framework to transform CUDA programs by lifting loops inside the kernel code and producing a multi-dimensional thread geometry. Because of the need for precise analysis, contrary to the previous sections this framework applies only to a specific subset of CUDA kernels: those whose control- and data-flow can be exactly characterized at compile-time using the polyhedral model [6, 17], and which do not contain any synchronization primitives. We remark that this framework uses the CUDA kernel source code as input, in place of the PTX representation. Algorithm 2 outlines our polyhedral optimization flow, and is detailed in the following.

4.3.1 Program representation

To represent a CUDA program in the polyhedral model, we first resort to classical AST-to-polyhedron conversion (function `extractPolyhedralRepresentation`). First we compute, for each syntactic statement in the program, its iteration domain. This set captures all the statement run-time

Algorithm 2: Polyhedral optimization flow

```
   Input : $AST$: AST of the CUDA kernel code;
           $G$: original thread geometry;
   Output: $AST, G$: Output new CUDA code and geometry
1  begin
2      $Poly$ ←extractPolyhedralRepresentation(AST,G);
3      $C$ ←computeLegalityConstraints($Poly$);
4      $C_p$ ←computeParallelismConstraints($Poly$);
5      $C$ ←$C \cap C_p$;
6      $P$ ←computeCostForAllRefs($Poly$);
7      $sol$ ←minimize $P$ s.t. $C$;
8      $AST$ ←polyhedralCodegen($Poly$,$AST$,$sol$);
9      $AST, G$ ←postProcessingAndTiling($AST$);
10     return $AST, G$;
```

instances, with an integer set bounded by affine inequalities. We then integrate the thread block geometry by viewing it as a loop nest with 3 loops, each iterating from 0 to $tdim_x$, etc. for the other 2. For instance for statement `R` above, its iteration domain $\mathcal{D}_S$ is:

$$\begin{aligned}\mathcal{D}_S \quad = \quad & \{(t_x,t_y,t_z,i) \in \mathbb{Z}^4 \mid \\ & 0 \le t_x < tdim_x \wedge 0 \le i < N \wedge t_y = t_z = 1\}\end{aligned}$$

Access functions describe the location of the data accessed by a statement instance. In static control parts, memory accesses are performed through array references (a scalar variable being a particular zero dimensional case of an array). We restrict ourselves to subscripts that are affine expressions of surrounding loop counters and global parameters. For instance, the subscript function of a read reference `A[i][k]` surrounded by 3 loops i, j and k is simply $f_A(i,j,k) = (i,k)$.

The execution order of the dynamic instances of statements captured in the iteration sets is described using a scheduling function Θ^{S_i} for each statement S_i. A schedule is a function which associates a logical execution date (a timestamp) to each instance of a given statement. In the case of multidimensional schedules, this timestamp is a vector. In the target program, statement instances will be executed according to the increasing lexicographic order of their timestamp. To construct a full program description, we build a collection of schedules $\Theta = \{\Theta^{S1}, \ldots, \Theta^{Sn}\}$, that is a list of the per-statement scheduling functions.

In this work, we restrict the scheduling function to model only the original program order and any possible permutations of the loop dimensions, including intra-thread loops and thread block dimensions. For homogeneity if two statements are not surrounded by the same number of loops, artificial one-time loops are introduced so that all statement iteration domains and schedules have the same dimensionality for the program.

4.3.2 Computing the program transformation

Our objective to find a transformation of the program is extremely analogous to Sec. 4.2.2: we formulate an ILP integrating the cost of accesses for each reference, seeking a solution in the form of a permutation matrix. But we also need to now take into account (1) legality conditions, as not all permutation of intra-thread loops may be semantically correct; and (2) parallelism conditions, as the intra-thread dimensions can be permuted with geometry dimensions only if they are parallel loops. Finally, to generate correct CUDA codes we must comply with the maximal thread block sizes as specified by the GPU and resort to a complementary tiling phase if needed.

The function `computeLegalityConstraints` computes the legality conditions that constrain the coefficients of the permutation matrix. The convex set of semantics-preserving schedules can be built in the traditional manner, linearizing the constraints provided by each dependence polyhedron using the Farkas Lemma [17, 32] to end up with affine inequalities bounding the schedule coefficients (i.e., the permutation matrix) so that each solution satisfying the inequalities necessarily preserves the program semantics. The Ponos tool automatically computes these constraints from the polyhedral representation [1, 32]. The sets of statement instances between which there is a data dependence relationship are modeled as equalities and inequalities describing a *dependence polyhedron* [16]. All dependence polyhedra can be automatically extracted from the program polyhedral representation, for instance using the Candl tool [1].

The function `computeParallelismConstraints` encodes a system of constraints on the schedule coefficients so that for two instances $\vec{x}_R$ and $\vec{x}_S$ in dependence, the constraint $\Theta_R(\vec{x}_R) = \Theta_S(\vec{x}_S)$ is enforced for all and only thread geometry dimensions and that thread geometry dimensions are not one-time loops. Indeed, for the rest of the program (kernel code), only the semantics-preserving condition $\Theta_R(\vec{x}_R) \preceq \Theta_S(\vec{x}_S)$ needs to be enforced. This condition is the classical sync-free parallelism condition [6, 26] to be enforced in a scheduling function, and is also linearized using the Farkas Lemma [17]. The final set of constraints C is computed as the intersection of the legality and parallelism constraints.

The function `computeCostForAllRefs` first analyzes each reference to compute the various $\vec{c}$, $\vec{u}$ and $\vec{z}$ vectors based on the reference access functions. We remark that these vectors are extended to have one element per surrounding dimension (that is, the thread geometry dimensions and any surrounding loop, including one-time loops), instead of only 3. We then formulate an ILP to find the coefficients of a permutation matrix (the linear part of the scheduling functions), in a manner analogous to Sec. 4.2.2. The permutation matrix size is adapted to properly model all dimensions, and we encode the cost constraints not only for the t_x dimension (i.e., $p_{1,j}$ the coefficients of the permutation matrix corresponding to the output t_x dimension) but also for all intra-thread loops (i.e., $p_{3+k,j}$, for each $k \in [1..d]$ with d intra-thread loops). We then minimize this ILP to obtain *sol*, the coefficients of the permutation matrix subject to all constraints.

The function `polyhedralCodegen` embeds the permutation *sol* into a complete schedule Θ and implements the transformation on the polyhedral representation of the CUDA code, by generating a code scanning iteration domains following the order specified by the schedule [8]. As we have modeled the geometry dimensions as standard loops in the polyhedral representation, a final post-processing stage needs to be performed to translate the loops in CUDA geometry and kernel loop syntax, the function `postProcessingAndTiling` performs this task. Because of hardware GPU constraints, we may need to apply tiling along the thread block dimensions to ensure the new $tdim_x$, $tdim_y$ and $tdim_z$ do not exceed their maximal allowed size, and compute accordingly the associated *bdim* sizes. Conversely, thread block dimensions which have become kernel loops need to be surrounded by new kernel loops iterating across all original thread blocks along these dimensions.

Returning to Listing 3, (t_x, t_y, t_z, i) gets permuted as (i, t_x, t_y, t_z), where a one-time loop t_z is used inside the thread code (in other words, after code generation there is no loop inside the thread code), and loop i becomes the new t_x dimension. Because i has N iterations, a strip-mining of i is performed, updating $bdim_x$ and i, so that the new dimension t_x has a size not exceeding the hardware geometry constraints, with $bdim_x \times tdim_x = N$.

4.4 Other Static Transformations

Listing 4 describes a common inefficient access pattern where a single thread is used to compute a reduction operation on a row of a matrix.

```
__global__
void kernel(float *A, float *x, float *tmp){
 int i = blockIdx.x*blockDim.x+threadIdx.x;
 if (i < N){
  for(j=0; j < N; j++){
   tmp[i] += A[i * N + j] * x[j];
  }}}
```

Listing 4: Inefficient Global Memory Load Example

This straightforward implementation leads to poor performance, because of uncoalesced accesses. We propose an effective work re-distribution strategy to improve the performance of such kernels.

During the dynamic analysis phase, such inefficiency is detected when we find a thread performs two contiguous loads followed by a store to a single location, but only one of the load operations is uncoalesced. It is likely, altough not guaranteed, that the thread is performing a reduction operation. To ensure correctness, we find and analyze the loads and the store in the original unoptimized PTX during the static transformation phase. The reduction operation is detectable at this point as it appears as a sequence of ld.global (data load from global memory), ld.global, fma (fused multiplication and add) and st.global (data store in global memory) operations in the PTX code. The optimization pass then proceeds with the transformation.

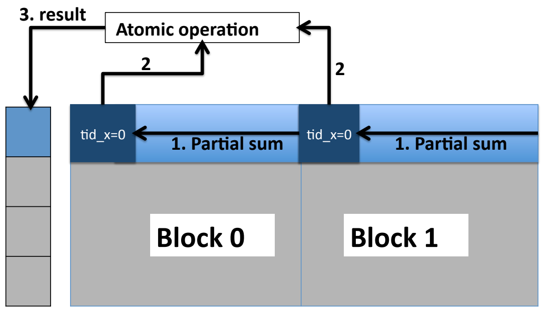

Figure 2: Partial Sum Method for Load Optimization

The transformation converts the reduction operation into a series of intermediate partial result computations, followed by atomic operations. The overall idea is depicted in Figure 2. The matrix is logically divided into blocks, where each block is loaded in a coalesced fashion by a thread block into a shared memory buffer. The block size needs to be a perfect square (e.g., 256) so that T is an integer. Another run-time requirement is teh use of N^2/T^2 blocks, so that each thread just loads one element from the NxN matrix. We allocate a buffer in the shared memory to hold data from global memory, where the size of the buffer is equal to the thread block size. The multiplication operation is performed on these data and the results are stored back in the shared buffer, replacing the the old values. After syncthreads, a subgroup of threads with ids $a*T+0$, where $a = 0, 1, ...T-1$ computes the partial sum of row a of the shared memory. Following another syncthreads, we need to perform an atomic addition across the blocks to get the final result from the partial results.

The transformation steps are shown in Algorithm 3. The declaration of the 2-D shared array is inserted in the first basic block. Lines 4-10 are the computations inserted and stored in registers. For simplicity, we skipped using PTX level computations in the algorithm. From the $inst_{ID}$, we find the array base addresses of the two ld.global and the st.global instructions that surround the fma. The original PTX instructions are moved out of the loop that contained them. We replace them with new instructions. Then the uncoalesced load instruction is transformed into a coalesced load by using the new computed offset $index1$. The ld.global is now followed by a st.shared (store in shared location). The offset of the second load operation is also replaced by $index2$. The multiplication is inserted after the second load and the result is stored back in the shared buffer. Following a syncthreads instruction, we add a code block that checks whether the thread has an ID that equals $a*T+0$ for some a. Inside

Algorithm 3: Transform Consecutive Load Operations by Single Thread

```
Input  : $inst_{ID}$: Static ID of the instruction to transform;
         $T$: Shared Memory dimension $TxT$;
         $C$: Input PTX code;
Output: $C'$: Output PTX code
1  begin
2      $C' \longleftarrow C$ ;
3      Allocate shared memory $S[TxT]$ in $C'$;
4      $i \longleftarrow threadID/T$ ;
5      $j \longleftarrow threadID\%T$ ;
6      $t_1 \longleftarrow blockID * BLOCK\_SIZE$ ;
7      $t_2 \longleftarrow (blockID * T)\%N$ ;
8      $index1 \longleftarrow t_1 + t_2 + i * N + j$ ;
9      $index2 \longleftarrow (blockID * T + j)\%N$ ;
10     $index3 \longleftarrow (blockID * T * T)/N + i$ ;
11     $load\_inst$ ⟵PTX instruction for $inst_{ID}$;
12     $A \longleftarrow$ Base address of $load\_inst$ ;
13     $x \longleftarrow$ Base address of the next load instruction;
14     $y \longleftarrow$ Base address of the store instruction;
15     Find loop containing $load\_inst$;
16     Move reduction instructions out of loop ;
       // Coalesced read from global memory
17     Load $A[index1]$ into $S[i][j]$ ;
18     $S[i][j] \longleftarrow S[i][j] * x[index2]$ ;
19     Syncthreads ;
20     if $j = 0$ then
           // Compute partial sum for block
21         for $k = 0$ to $T$ do
22             $sum \longleftarrow sum + S[i][k]$ ;
23     Syncthreads ;
24     Insert atomic addition of $sum$ in $C'$;
25     Store result of atomic add at $y[index3]$;
26     return $C'$;
```

the true branch, we insert a loop to perform the partial sums. After the if block, we add another syncthreads to make sure that all the partial results are ready. Finally, the atomicadd instruction is inserted to add the partial sums and stores the results at the new offset *index*3.

5. Experimental Results

5.1 Experimental Protocol

We have used Ocelot v2.1 to build the instrumentation and the optimization passes for PTX codes. CUDA Toolkit v5.5 was used for the nvcc compiler and also to write the CUDA driver API for executing PTX codes. Experiments were done on a machine with Tesla K20 and compute capability 3.0. The benchmarks presented are from Rodinia v2.4 [11, 12] and PolyBench/GPU v1.0 [19]. The sparse matrix-vector multiplication (SpMV) application is taken from SHOC [14]. The spmv_csr_kernel from SHOC has uncoalesced access of the column vector that stores indices of the actual elements in the matrix. This access is affine and we were able to apply the transformation. The execution time is measured by taking the average of 100 executions.

5.2 Dynamic Analysis Results

Type	C/U	Stride	Bandwidth
Load	Coalesced	0	1
Load	Uncoalesced	N	N
Store	Uncoalesced	N	N

Table 1. Sample Output of Dynamic Analysis

To evaluate our dynamic analysis tool, we characterized the global memory coalescing properties of Rodinia and PolyBench/GPU benchmarks. The time for the dynamic analysis depends on the size of the memory traces. The total time for the instrumentation, trace generation and running the analysis of an application is generally in the order of hundreds of milliseconds. The profile sizes are approaximately in the range of ten kilobytes.

We feed one kernel at at time to the analysis tool and produce useful information for each of the static global memory load/store operation. For each operation the tool is designed to report the following: type (Load/Store), Coalesced/Uncoalesced, stride (distance between memory accessed by consecutive thread in the thread space) and the bandwidth of the operation. For example, for the Fan1 kernel of the Gaussian Elimination benchmark, the tool produces output like Table 1. Table 2 depicts a summary of the results of the analysis on Rodinia and Polybench/GPU. For space limitation, we only report the benchmark kernels that has uncoalesced access detected by the tool. Seven Rodinia benchmarks are identified to have uncoalesced accesses. Three out of these seven benchmarks (*MUMmerGPU*, *k-Nearest Neighbor* and *Stream Cluster*) use array of structures where each thread accesses all the members of a structure element. This results in a strided access for the threads in a warp. Using existing APIs [13], the array of structures can be converted to structure of arrays to achieve coalesced accesses. While our static transformation framework could be extended in the future to incorporate such an optimization, it currently does not. Hence, we do not report performance for these applications. *Myocyte*, on the other hand, assigns each thread to compute a full row of a matrix which is very inefficient. *Gaussian elimination* distributes a 2D matrix onto a 2D thread block but assigns the fastest growing dimension to threadIdy, leading into strided access. *Cell* has the same issue for 3D matrices and thread blocks. We optimize these three benchmarks using static transformation (results in the next section).

The PolyBench/GPU suite has nine applications with uncoalesced memory accesses. Among them, the *Gram-schmidt* kernel2 requires re-writing of the whole application to achieve coalesced access - which is out of scope for this paper. *Syrk* and *Syr2k* do not have much scope of improvement. We transform the remaining six applications. We also analyzed SHOC sparse matrix-vector (SpMV) multiplication as an representative for irregular applications. The results are in the following section.

Benchmark	Kernel Execution Time		Data Copy Times		GFLOPS		**Kernel Speedup**	App. Speedup
	Original	Transformed	Host to Device	Device to Host	Original	Transformed		
(R) Myocyte	79.81ms	2.25ms	43.81ms	29.25ms	1.8	63.5	35.5×	2.1×
(R) Gaussian	1.91s	0.43s	0.01s	0.01s	17.91	78.71	4.4×	4.3×
(R) Cell	6.73ms	1.74ms	1.37ms	1.55ms	n/a	n/a	3.9×	2.1×
(P) covariance	23.27s	4.61s	0.011s	0.01s	5.91	29.82	5.1 ×	5.1×
(P) GESUMMV	82.67ms	10.59ms	20.63ms	0.01ms	3.25	25.35	7.8×	3.3×
(P) AtAx	28.71ms	15.28 ms	40.87ms	0.02ms	9.35	17.56	1.9×	1.3×
(P) Correlation	23.27s	4.61s	0.02s	0.01s	5.91	29.82	5.1 ×	5.1×
(P) mvt	23.61ms	10.19ms	40.89ms	0.02ms	11.37	26.33	2.3×	1.3×
(P) BiCG	28.86ms	15.33ms	40.88ms	0.02ms	9.28	17.51	1.9×	1.2×
SpMV	27.49ms	8.01ms	10.44ms	0.01ms	5.22	17.23	3.4×	2.1×

Table 3. Execution times of applications on Tesla K20

Rodinia			
Benchmark	Kernel	Total	Uncoalesced
Gaussian	Fan1	$2N+1$	$2N$
	Fan2	$4N^2+4N$	$3N^2+N$
Kmeans	invert_mapping	$2NK$	NK
MUMmerGPU	printKernel	$5N$	$5N$
Myocyte	solver	$7N^2$	$7N^2$
k-NN	euclid	$5N$	$4N$
Cell	evolve	$2N^3$	$2N^3$
StreamCluster	compute_cost	$6N$	$5N$

Polybench/GPU			
Benchmark	Kernel	Total	Uncolaesced
AtAx	atax_kernel1	N^2+3N	N^2
BiCG	bicg_kernel2	N^2+3N	N^2
Correlation	corr_kernel	$7N^2+N$	$5N^2$
covariance	covar_kernel	$8N^2$	$5N^2$
GESUMMV	gsumv_kernel	$2N^2+8N$	$2N^2$
Gramschmidt	gram_kernel2	$2N+1$	$2N$
mvt	mvt_kernel1	N^2+3N	N^2
SYRK	syrk_kernel	$6N^2$	N^2
SYR2K	syr2k_kernel	$8N^2$	$2N^2$

Table 2. Benchmarks with Uncoalesced Access in Rodinia and Polybench/GPU

5.3 Static Transformation Results

From the dynamic analysis tool, the applications identified with improvement potential are then fed to a static transformation framework for optimization. We report the improvement of execution times for these applications in Table 3. Benchmarks taken from Rodinia are marked with *R* while benchmarks taken from Polybench/GPU are marked with *P*. For each kernel, we also measured the time for copying data between host and device. We compared the performance in GFLOPS. Cell only performs copy operation, therefore GFLOPS is not reported for this benchmark. The effective bandwidth comparison is reported in Figure 3. Note that y-axis is in logarithmic scale for this figure.

We observe significant speed up for all of the applications. The speedup ranges from 2× to 35× on K20. Myocyte in its original form suffers from a high memory latency due to the high amount (refer to Table 2) of uncoalesced accesses. Our transformation successfully re-distributed the workload so that threads now perform the same number of reads and writes in a coalesced fashion, leading to a significant 35× speedup. Similarly, simple geometric transformation of thread dimensions in *Gaussian Elimination* and *Cell* improves their performances by around 4×. Although it seems that Cell had more uncoalesced access, it uses a 3D thread block on O(N^3) data vs Gaussian Elimination using 2D thread block on O(N^2) data. Therefore the speedup is similar. The Polybench/GPU benchmarks and SpMV achieved 2× to 8× speedup after the transformation, which fixed the uncoalesced load operations. As the speed up is related to the amount of uncoalesced access in the original code, GESUMMV has higher speedup compared to AtAx, BiCG, etc due to its 2× more uncoalesced accesses.

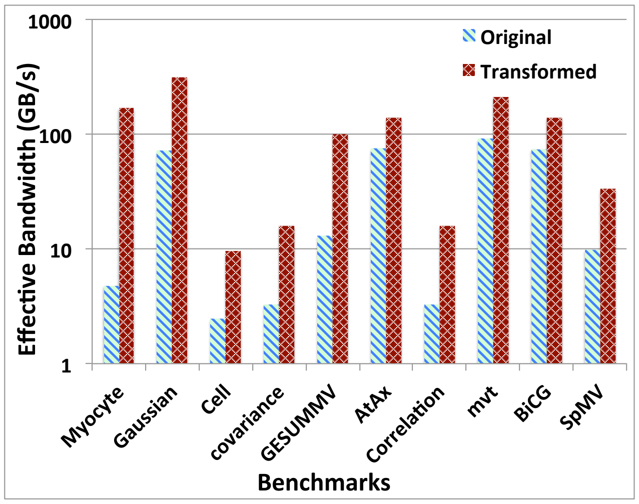

Figure 3: Effective bandwidth on Tesla K20. Y-axis is in logarithmic scale

Figure 3 provides useful insights into the benefits of our transformation algorithms. Five out of the ten benchmarks reached over 100GB/s bandwidth after our optimization, and the high memory bandwidth directly contributes to overall application performance, because the time for fetching data from global memory is significantly reduced by coalesced accesses. Memory bounded kernels benefit more from our algorithms than compute bounded ones. The average speedup is 6× for all of the applications tested.

5.4 Discussions

Dynamic analysis, in general, can be inaccurate in some cases as the analysis might be dependent on input data sets. But the effectiveness of our specific characterizing tool does not depends on choosing appropriate input sets. To the best of our knowledge, we are not aware of any real benchmark on which different input datasets can make the same memory reference changing from being coalesced to uncoalesced, or vice-versa. The input dataset may only change the runtime uncoalesced access count/stride/bandwidth usage metric. But, ignoring corner/artificial cases, it will not change the fact whether a memory reference is coalesced or not. However, the final performance of the transformed code can be affected, of course, by the input dataset. In our test suite, for 6 out of 10 benchmarks the control flow is independent of the input, therefore the same transformation is always required whatever the input, and such transformation is automatically computed and implemented in our framework.

Our dynamic analysis for detecting uncoalesced access operates on arbitrary CUDA/PTX codes. Our transformation scheme based on geometry permutation can also handle any CUDA/PTX program. Therefore, we can analyze all Rodinia benchmarks and apply the geometric permutation on them. However our intra- and inter-thread optimization framework using the polyhedral model is limited to affine CUDA programs, and cannot handle all Rodinia benchmarks.

Coalesced access may increase the register pressure and result into higher amount of spill. But we observe significant speedup by achieving coalesced access (as shown in Table 3). Therefore, the effects of other factors, if any, must have been negligible.

6. Related Work

Many previous works focused on improving the programming productivity by automate transformation tools such as from C to CUDA [5, 6] and OpenMP to CUDA [22–24]. PPCG [36] uses polyhedral analysis and convert legacy affine sequential C codes to CUDA automatically. In contrast, our work can operate on arbitrary input CUDA/PTX codes and takes as input a CUDA program, and our intra-thread optimization focuses exclusively on data coalescing. Par4All [4] transforms C or Fortran code to CUDA or OpenCL code. Par4All uses a polyhedral analysis tool called PIPS but the tool itself is not entirely based on polyhedral analysis. Unlike PPCG, it does not use any shared memory. Optimization techniques such as loop collapsing or thread coarsening are used in [27, 38], they however differ from coalescing-centric approach to find a new thread block geometry. Few works have been proposed for specific algorithms such as [15, 25, 28, 29]. Inspector/executor based strategies [35, 37] have been proposed to support non-affine irregular codes. Another set of work provides directive-based CUDA code optimizations [21, 33] or API to transform CUDA codes [13] but rely on manual annotations from the programmer.

CUPL [3] uses polyhedral methods to detect possible uncoalesced accesses of affine CUDA codes. In contrast, our dynamic analysis method can detect coalesced or uncoalesced access pattern in any affine and irregular PTX codes. To the best of our knowledge, CUPL limits to detecting uncoalesced accesses, and does not automatically transform programs. Therefore CUPL cannot lead to any automatic improvement in performance, in contrast to our approach which automatically transforms programs. GRace [39] and GMProf [40] was developed to detect data races in shared memory, using a similar approach to ours that combines dynamic analysis and static transformation. Previous dynamic analysis techniques ([9, 10]) also focuses on program correctness and aims to detect race conditions and bank conflicts. To the best of our knowledge, we present the first dynamic analysis approach for improving the global memory access pattern on GPU. In addition, our framework targets at PTX code that can be derived from any heterogeneous programming language or directives.

7. Conclusion

In this paper, we have combined dynamic analysis approach with static transformation with an aim to improve locality of global memory data during a thread warp execution. Given a PTX code of a program, we (1) characterize its global memory access operations and separate coalesced and uncoalesced access, (2) study access pattern of the uncoalesced accesses and recommend some improvement strategy if possible and (3) implement transformations of the input PTX (or CUDA) code to improve data coalescing. We have characterized GPU benchmark suites using the dynamic analysis and transformed a number of. Our transformed version ensures coalesced access and improves the kernel computation time by $2\times$ to $35\times$.

Acknowledgment This work was supported in part by the U.S. National Science Foundation through awards 0926127, 1321147 and 1440749.

References

[1] PoCC, the polyhedral compiler collection. *http://pocc.sourceforge.net*.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison Wesley, 1986.

[3] M. Amilkanthwar and S. Balachandran. Cupl: A compile-time uncoalesced memory access pattern locator for cuda. In *ICS*, pages 459–460. ACM, 2013.

[4] M. Amini, O. Goubier, S. Guelton, J. O. Mcmahon, F. xavier Pasquier, G. Pan, and P. Villalon. Par4all: From convex array regions to heterogeneous computing. http://www.par4all.org/.

[5] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A compiler frame-

work for optimization of affine loop nests for gpgpus. In *ICS*, pages 225–234. ACM, 2008.

[6] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *PPOPP*, pages 1–10. ACM, 2008.

[7] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In *CC*, pages 244–263. Springer-Verlag, 2010.

[8] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT*, pages 7–16. IEEE Computer Society, 2004.

[9] M. Boyer, K. Skadron, and W. Weimer. Automated Dynamic Analysis of CUDA Programs. In *Third Workshop on Software Tools for MultiCore Systems*, 2008.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using cuda. *J. Parallel Distrib. Comput.*, 68(10):1370–1380, Oct. 2008.

[11] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54. IEEE, 2009.

[12] S. Che, J. Sheaffer, M. Boyer, L. Szafaryn, L. Wang, and K. Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *IISWC*, pages 1–11. IEEE, 2010.

[13] S. Che, J. W. Sheaffer, and K. Skadron. Dymaxion: optimizing memory access patterns for heterogeneous systems. In *SC*, pages 13:1–13:11. ACM, 2011.

[14] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *GPGPU*, pages 63–74. ACM, 2010.

[15] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC*, pages 1–12. IEEE Press, 2008.

[16] P. Feautrier. Dataflow analysis of scalar and array references. *Intl. J. of Parallel Programming*, 20(1):23–53, Feb. 1991.

[17] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *Intl. J. of Parallel Programming*, 21(6):389–420, Dec. 1992.

[18] Georgia Institute of Technology. GPUOcelot. https://code.google.com/p/gpuocelot.

[19] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *InPar*, pages 1–10. IEEE, 2012.

[20] 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 *GPGPU-6*, pages 24–31. ACM, 2013.

[21] T. Han and T. Abdelrahman. hicuda: High-level gpgpu programming. *Parallel and Distributed Systems, IEEE Transactions on*, 22(1):78–90, Jan 2011.

[22] S. Lee and R. Eigenmann. Openmpc: Extended openmp programming and tuning for gpus. In *SC*, pages 1–11. IEEE Computer Society, 2010.

[23] S. Lee and R. Eigenmann. Openmpc: Extended openmp for efficient programming and tuning on gpus. *Int. J. Computational Science and Engineering*, 7(1):116, 2012.

[24] S. Lee, S.-J. Min, and R. Eigenmann. Openmp to gpgpu: A compiler framework for automatic translation and optimization. *SIGPLAN Not.*, 44(4):101–110, Feb. 2009.

[25] Y. Li, J. Dongarra, and S. Tomov. A note on auto-tuning gemm for gpus. In *ICCS*, pages 884–892. Springer-Verlag, 2009.

[26] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *POPL*, pages 201–214. ACM, 1997.

[27] A. Magni, C. Dubach, and M. F. P. O'Boyle. A large-scale cross-architecture evaluation of thread-coarsening. In *SC*, pages 11:1–11:11. ACM, 2013.

[28] R. Nath, S. Tomov, T. T. Dong, and J. Dongarra. Optimizing symmetric dense matrix-vector multiplication on gpus. In *SC*, pages 6:1–6:10. ACM, 2011.

[29] A. Nukada and S. Matsuoka. Auto-tuning 3-d fft library for cuda gpus. In *SC*, pages 1–10. ACM, 2009.

[30] NVIDIA Corporation. *Parallel Thread Execution ISA*.

[31] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, June 2011.

[32] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache. Loop transformations: Convexity, pruning and optimization. In *POPL*, pages 549–562. ACM, 2011.

[33] S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W.-M. W. Hwu. Languages and compilers for parallel computing. chapter CUDA-Lite: Reducing GPU Programming Complexity, pages 1–15. Springer-Verlag, 2008.

[34] University of Illinois Urbana-Champaign. Clang. http://clang.llvm.org.

[35] A. Venkat, M. Shantharam, M. Hall, and M. M. Strout. Non-affine extensions to polyhedral code generation. In *CGO*, pages 185:185–185:194. ACM, 2014.

[36] 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.

[37] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu. In *PPoPP*, pages 57–68. ACM, 2013.

[38] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A gpgpu compiler for memory optimization and parallelism management. In *PLDI*, pages 86–97. ACM, 2010.

[39] M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal. Grace: A low-overhead mechanism for detecting data races in gpu programs. *SIGPLAN Not.*, 46(8):135–146, Feb. 2011.

[40] M. Zheng, V. Ravi, W. Ma, F. Qin, and G. Agrawal. Gmprof: A low-overhead, fine-grained profiling approach for gpu programs. In *HiPC*, pages 1–10. IEEE, 2012.