GROPHECY: GPU Performance Projection from CPU Code Skeletons

Jiayuan Meng, Vitali A. Morozov, Kalyan Kumaran, Venkatram Vishwanath, Thomas D. Uram
Argonne National Laboratory
{jmeng, morozov, kumaran, venkatv, turam}@anl.gov

Abstract

We propose *GROPHECY*, a GPU performance projection framework that can estimate the performance benefit of GPU acceleration *without* actual GPU programming or hardware. Users need only to skeletonize pieces of CPU code that are targets for GPU acceleration. Code skeletons are automatically transformed in various ways to mimic tuned GPU codes with characteristics resembling *real* implementations. The synthesized characteristics are used by an existing analytical model to project GPU performance. The cost and benefit of GPU development can then be estimated according to the transformed code skeleton that yields the best projected performance. With GROPHECY, users can leap toward GPU acceleration only when the cost-benefit makes sense. The framework is validated using kernel benchmarks and data-parallel codes in legacy scientific applications. The measured performance of manually tuned codes deviates from the projected performance by 17% in geometric mean.

1. Introduction

Graphics processing units (GPUs) increasingly are being used to accelerate scientific computing applications. While GPUs have yielded 10× or even 100× speedups for some applications, studies have shown that such acceleration is not always the case [26]. Application developers are increasingly having to ponder the viability of using GPUs to benefit their science and whether it is indeed worth investing the time and effort to port their code to run on GPUs. Developers have often used GPU manufacturers' quoted peak flop rate to project performance; however, this is often too optimistic. To understand the viability of using GPUs to accelerate performance, one needs to take into account several factors, including an application's intrinsic nature, such as data parallelism, memory access patterns, control flow divergence, and computational intensity.

To the best of our knowledge, currently the only way to evaluate the potential of GPU acceleration is to invest in real hardware and develop actual GPU code. While APIs such as "C for CUDA" [29], Brook+ [2], and OpenCL [18] have been released to ease general purpose GPU programming, learning and writing GPU code remain nontrivial tasks. To tune GPU kernels, developers often have to investigate various code transformations. Additionally, the implementation space can be too large to be explored manually. The whole process can be tedious and error-prone. Although some tools do simplify GPU programming and semi-automate code tuning, these tools require physical hardware for performance evaluation and fail to project performance for unavailable GPUs. Regardless of how GPU code is generated, if the achieved performance is similar to or even worse than CPU performance, significant time and effort would have been wasted already. Needed is a mechanism that can help project the performance of an application on current and future GPUs with minimal effort, thus enabling developers to better evaluate whether they should indeed be investing in porting their application to GPUs.

To this end, we propose a GPU performance projection framework, named *GROPHECY*, that allows developers to project achievable *GPU* performance from skeletonized *CPU* code. No actual GPU programming or hardware is needed to cast projections. GROPHECY automatically explores the GPU implementation space by transforming code skeletons in various ways, The transformed code skeletons, named *code layouts*, depict structures of their corresponding GPU code, which can look very different from the original CPU code. Statistics are gathered from code layouts to synthesize characteristics of real GPU code. These statistics are then used by an analytical model [13] to project performance over a target GPU architecture. The code layout that yields the best projected performance indicates what transformations are needed and how much performance can be gained. Such cost-benefit analysis helps users determine whether GPU acceleration is beneficial *before* actual development is undertaken. We note that the proposed framework does not change data structures or modify algorithms; such implementation alternatives can be explored by using different CPU code skeletons.

We validate GROPHECY using both microbenchmarks and data-parallel codes in legacy high-performance computing applications. The contributions of our paper include the following:

1. Definition of CPU code skeleton that can be used to project performance on GPUs.
2. Automated mechanism to restructure CPU code skeleton and mimic transformations needed to tune GPU code. Investigated transformations include mapping parallel tasks to GPU thread contexts, staging of computations, ordering of computations to improve cache performance, and loop unrolling.
3. Systemic model to characterize the benefits and side effects of GPU code transformations.
4. Ability to project a CPU kernel's performance on GPUs without producing the GPU code and without accessible GPUs. Only hardware specifications and application statistics are needed.
5. Ability to explore future GPU generations and evaluate their performance. Users can vary the GPU hardware specifications and study how code can adapt to the hardware as well as the achieved performance.
6. Pverall workflow for cross-platform performance projection without cross-platform implementations in either software or hardware.

2. Related Work

Existing tools such as PGI compiler [39], C-to-CUDA for affine programs [5], OpenMPC [25], and Mint [10] can produce GPU code from an annotated legacy code. There also exist *metaprogramming* tools such as PyCUDA [20] that autotune implementation parameters (e.g., thread block size) once a programmer devises a GPU code template. Model-driven autotuning frameworks have also been proposed for GPU applications in various domains [8, 28]. However, these tools do not tune a general data-parallel code with transformations such as staging and caching strategies, as described in Section 6. CUDA-lite explores these transformations. However, it relies on annotated GPU code to specify parameters such as thread block sizes and arrays to cache; therefore, a single annotated code does not explore all possible configurations [37]. In all these tools, a physical GPU is still required to run different implementations and evaluate their performance. No statistics are generated to provide insights into performance bottlenecks.

Recently, several GPU performance models have been proposed [3, 13, 21, 41]. These techniques predict the performance of *actual GPU codes* over accessible GPUs by behavioral observation or hardware abstraction. However, GPU performance models alone are not able to transform CPU code structures for optimization over GPUs, let alone projecting GPU performance from CPU code.

Performance models increasingly have been used for application tuning over complex or large scale systems [11, 23, 30]. These models target performance over a cluster or a heterogeneous platform, with a focus on the modeling and optimization of communication and scheduling among nodes. Lee et al. [24] used machine learning techniques to model the relationship between tunable parameters and application performance. Snavely et al. [36] proposed a framework for performance modeling and prediction, in which traces are used to characterize application signatures. All these techniques have to measure or profile application statistics over each type of nodes, assuming code already exists for that type of architecture.

Cross-platform performance predictions have been studied by Yang et al. [40]; they profiled partial execution of an application on different platforms to infer relative full-application performance. Lee et al. [22] estimated application performance over different microarchitectural configurations using regression modeling, which is trained according to an initial set of performance data. Different from these approaches, the problem addressed by this paper requires cross-platform projection for a fundamentally different architecture, *without* implementing code, let alone executing any piece of the application, on that architecture. Recently, Carrington et al. [7] proposed a tool that finds common computation and data access patterns, referred to as *idioms*, and uses modeling to project their performance if ported to FPGA. However, codes to be accelerated on GPUs are usually more complex than idioms; several code transformations have to be explored to project performance.

3. Background

In this section, we introduce the basics of GPU architecture and GPU programming. Terminologies used for GPU hardware units and programming models are based on NVIDIA's GPU architectures and the "C for CUDA" API.

3.1 The GPU Architecture

The architecture of a typical GPU has a set of *streaming multiprocessors* (SMs), each containing several *streaming processors* (SPs) that operate in a SIMD fashion. A group of hardware thread contexts that operate in lockstep is referred to as a *warp*.

The GPU has its own DRAM, referred to as *global memory*, which can be accessed from any SM. The GPU also has implicit, noncoherent caches for particular data accesses such as texture and constant memory. Each SM has its own L1 storage, referred to as *shared memory*. Data in the shared memory can be accessed by multiple hardware thread contexts on the same SM. While the latency to access shared memory is low (almost the same as accessing registers), accessing global memory can take hundreds of cycles. GPUs hide such latency by having multiple warps active on each SM; upon global memory accesses, an SM can switch to another warp and continue execution.

The latency to access global memory can vary according to data access patterns. Depending on the sequence of addresses accessed by threads in a warp, requests from a SIMD memory instruction can be combined into fewer memory transactions. This process is referred to as *coalesced* memory access. Otherwise, several memory transactions are generated, and they are serialized. This process is referred to as *uncoalesced* memory access. A warp cannot continue to execute until all of its memory transactions finish. The coalescing mechanisms vary across different GPU generations.

3.2 The GPU Programming Model

The CUDA programming model [29] eases general-purpose, data-parallel programming for the GPU architecture. To construct a *GPU kernel*, a developer decomposes a parallel `for` loop into a grid of coordinated *thread blocks*, each consisting of a set of coordinated scalar threads; threads with adjacent coordinates are implicitly grouped into a warp. During GPU execution, a thread block is mapped to one SM, and one SM can execute multiple thread blocks. Although threads can take different control paths, branches are generally discouraged. Threads in the same thread block can be synchronized with low overhead. They can also share data through the shared memory. All threads can access global memory. Note that caching global memory data into the shared memory is explicitly controlled by the GPU kernel.

4. The GPU Performance Projection Framework

Given a piece of CPU code and a target GPU architecture, the framework takes three major steps to estimate the optimized implementation and its performance, as illustrated in Figure 1.

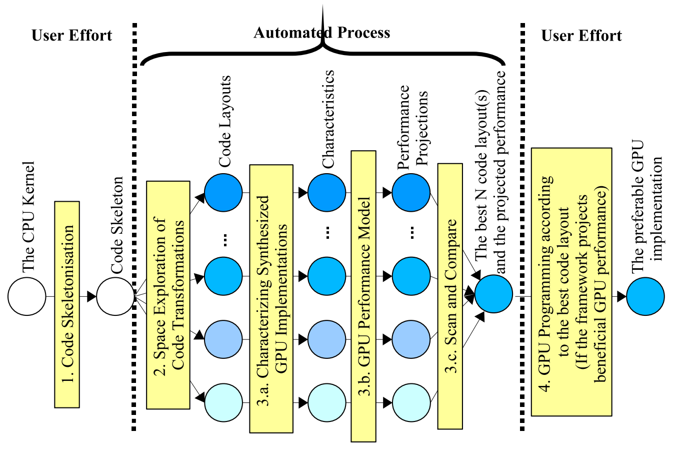

User Effort
Automated Process
User Effort
The CPU Kernel
1. Code Skeletonisation
Code Skeleton
2. Space Exploration of Code Transformations
Code Layouts
3.a. Characterizing Synthesized GPU Implementations
Characteristics
3.b. GPU Performance Model
Performance Projections
3.c. Scan and Compare
The best N code layout(s) and the projected performance
4. GPU Programming according to the best code layout (If the framework projects beneficial GPU performance)
The preferable GPU implementation

Figure 1: An overview of GROPHECY.

The first step is the only step where users are involved. The user abstracts the CPU code's parallelism, computational intensity, and data accesses in a code skeleton. Note that such information is intrinsic within the computation and is not GPU-specific. In other words, developers can extract such general information from legacy code *without* GPU knowledge. The form of the code skeleton is described in Section 5.

In the second step, GROPHECY automatically explores GPU implementations by transforming the code skeleton. The goal is to explore different ways to lay out code on the GPU, both spatially and temporally. Each *code layout* is a structural abstraction of a particular GPU implementation. Section 6 describes this step in more detail.

In the last step, the proposed code layouts are used to characterize their corresponding GPU implementations, *without* these implementations being actually coded. In order to project performance, the synthesized characteristics serve as inputs to an existing GPU performance model proposed by Hong and Kim [13]. The code layout that achieves the best performance is recorded. This step is described in Section 7. The performance model takes into account hardware specifications, which are collected beforehand for each GPU architecture. At this point, users can decide whether it is worthwhile to accelerate a kernel on the GPU.

5. Code Skeletonization

A code skeleton serves as one way to abstract the CPU code structure and a starting point for code transformation. As a pedagogical example, the code skeleton for dense matrix multiplication (denoted with `MatMul`) is shown in Listing 2. The corresponding CPU code is shown in Listing 1 in `C`. The syntax of a code skeleton is not the focus of this paper. It is briefly introduced in the comments of the example code skeletons and is not discussed in further detail.

Listing 1: `MatMul`'s CPU code

```
 1 float A[N][K], B[K][M];
   float C[N][M];
 3 int i, j, k;
   for(i=0; i<N; ++i){
 5  for(j=0; j<M; ++j){
      float sum = 0;
 7    for(k=0; k<K; ++k){
        sum+=A[i][k]*B[k][j];
 9    }
    C[i][j] = sum;
11 }
```

Listing 2: `MatMul`'s code skeleton

```
float A[N][K]
float B[K][M]
float C[N][M]
/* the loop space */
parallel_for(N, M)
: i,j
{
  /* computation w/t
   * instruction count
   */
  comp 1
  /* streaming loop */
  stream k = 0:K {
    /* load */
    ld A[i][k]
    ld B[k][j]
    comp 3
  }
  comp 5
  /* store */
  st C[i][j]
}
```

Listing 3: `MatMul`'s optimized GPU code

```
float A[N][K], B[K][M], C[N][M];
dim3 block(BlkSize, BlkSize);
dim3 grid(N/BlkSize, M/BlkSize);
MatrixMul<<<grid, block>(A, B, C);

__global__ MatrixMul(A, B, C)
{
  __shared__ a[BlkSize][BlkSize];
  __shared__ b[BlkSize][BlkSize];
  int ty = threadIdx.y;
  int tx = threadIdx.x;
  int y = blockIdx.y*blockDim.y+ty;
  int x = blockIdx.x*blockDim.x+tx;
  float sum = 0.f;
  for(int n=0; n<K; n+=BlkSize){
    a[ty][tx]=A[y][n+tx];
    b[ty][tx] = B[n+ty][x];
    __syncthreads();
    for(int k=0; k<BlkSize; ++k){
      sum += a[ty][k]*b[k][tx];
    }
    __syncthreads();
  }
  C[y][x] = sum;
}
```

The following information forms a code skeleton that expresses a computational kernel.

Data parallelism is expressed as a set of parallel, homogeneous tasks repeated over different data elements. Users should express data parallelism in its finest granularity (*i.e.*, down to the innermost parallel `for` loops).

A task corresponds to one iteration of the innermost parallel `for` loop. It is expressed as a sequence of data accesses and computation.

Data accesses are expressed as a set of load and store operations. The accessed array elements are expressed given loop indices, array sizes, and other constants. Indirect data accesses can be expressed as well; GROPHECY will assume indirect accesses are random unless users provide further hints (see Section 9.4 and Listing 6).

Computation instructions are counted by using methods described in Section 7.3. Together with the number of memory instructions, they indicate the computational intensity of the kernel.

Branch instructions are counted to judge the applicability of loop unrolling.

For loops wrap around blocks of computation and data accesses to mark repetition within a task. They can be nested and the nesting does not have to be perfect.

Streaming loops are a special type of `for` loop; they are marked where a sequence of data elements are fetched and processed and can be discarded immediately. It is a common pattern for reduction. Streaming loops can be temporally decomposed into stages for the purpose of caching. Line 7 in Listing 1 is an example of a streaming loop.

Macros that define array sizes and the number of loop iterations. By adjusting the macros, the same code skeleton can be used for workloads at different scales.

Once constructed, the code skeleton can then be transformed to mimic GPU optimizations. Note that the mimicked GPU implementation can differ significantly from the original CPU code. As an example, Listing 3 shows the GPU kernel of `MatMul`, where `for` loops are not only spatially decomposed among threads but also temporally decomposed into stages for the purpose of caching. Both transformations are common and critical in manual GPU optimization.

6. Code Transformations

Given the code skeleton, GROPHECY transforms and lays out code for a target GPU (recall Figure 1, Step 2). This section describes how code layouts are represented (Section 6.1), how the space of possible layouts is searched (Section 6.2), and additional representations and metrics needed to carry out this search (Sections 6.3–6.7).

6.1 Code Layout Parameterization

Code transformation involves the following factors, whose values jointly define a particular code layout.

Thread block sizes, represented as $\mathbb{B} = \{b_1, ..., b_n\}$, where n is the dimensionality of the loop space and b_i is the length of the thread block in the ith dimension; $size(\mathbb{B})$ denotes the number of threads in a thread block. We vary the thread block size given the loop space and the hardware constraint on the number of threads per block.[1]

Staging, or temporarily decomposing streaming loops into sequential stages of iterations. Within one stage, a thread block only needs to cache the portion of data elements used in this stage. Staging can be expressed as two integer vectors. For a code skeleton with n streaming loops, $\mathbb{S} = \{s_1, ..., s_n\}$ contains s_i which defines the *staging size*, or the number of iterations in one stage for the ith streaming loop. Moreover, some consecutive streaming loops actually form a multidimensional streaming loop, whose traversal orders are interchangeable with regard to outer loops and inner loops. Different traversal orders may result in different performances as a result of data locality and caching. Therefore, $\mathbb{O} = \{o_1, ..., o_n\}$ defines the traversal order where o_j is the identifier of the jth streaming loop to be traversed.

Folding, or assigning multiple tasks to one thread. It is represented as $\mathbb{F} = \{f_1, ..., f_n\}$, where n is the dimensionality of the loop space and f_i is the number of indices assigned to a thread along the ith loop. When folding is not applied, GROPHECY assumes each thread computes one task and $f_i = 1$ for all i's. The *folding degree*, F, is defined as the total number of tasks assigned to a thread, or $\prod_{i=1}^{n} f_i$. For the purpose of data reuse and coalescing, folding always assigns neighboring tasks to threads with adjacent thread indices [27]. Once applied, additional loop statements will be added so that a thread can iterate through assigned tasks. These additional loop statements are considered as streaming loops, and staging can be applied.

Caching Strategy. The caching strategy categorizes data accesses into uncached accesses to global memory and cached accesses to shared memory. For shared memory, the caching strategy also describes which array segments are cached. We use *bounded regular sections* (BRS) [12], a derived form of *regular section descriptors* (RSD) [6, 4], to represent data accesses. A data access statement in the code skeleton can be represented as $\mathbb{A}(D, \Theta, I)$. D is the array to be accessed. $\Theta = \{\theta_1, ..., \theta_m\}$, where θ_j is the index to D's jth dimension. Each θ can be a function involving $I = \{I_1, ..., I_n\}$, which are indices of the loops that contain this data access statement. For all data accesses in a code skeleton, a code layout uses $\{\dot{\mathbb{A}}\}$ to specify the set of uncached memory accesses and $\{\bar{\mathbb{A}}\}$ to specify the set of cached memory accesses. The shape of D's region cached in shared memory during each stage of the kth streaming loop is denoted with $ShMem(D_i, k)$; $k = 0$ corresponds to cached data for memory accesses outside any streaming loops. $ShMem(D_i, k)$ is a *footprint* defined in Section 6.3 and can be obtained by Equation 5.

Loop Unrolling. Loop unrolling reduces instructions due to loop overhead and is especially important for computation-bounded workloads. It can be expressed by $\mathbb{L} = \{l_i, ..., l_n\}$, where l_i is the number of iterations to be unrolled for the ith loop. According to our empirical studies of the NVCC compiler [29], GROPHECY applies loop unrolling to any inner-thread, branch-free loops whose number of iterations can be determined

[1]In a code layout, the dimensionality of a modeled thread block is not restricted since a high-dimensional loop space can be flattened and reduced to a lower-dimensional space.

statically.[2] Iterations are batched into groups, each concatenating 16 unrolled iterations. Its effect is reflected in the estimated instruction count.

In summary, a code layout can be uniquely represented by using $\{\mathbb{B}, \mathbb{S}, \mathbb{O}, \mathbb{F}, \{\dot{\mathbb{A}}\}, \{\bar{\mathbb{A}}\}, \{ShMem(D_i)\}, \mathbb{L}\}$. Varying thread block size, folding, and staging can all be regarded as forms of tiling. There has been much research on tiling [16, 34, 35] and loop unrolling [9] in CPU code optimizations. GROPHECY models and projects their effects on computational intensity and data access patterns for GPUs.

6.2 The Search Space

A brute-force search for code layouts would explore all possible combinations of the factors described above. Note that factors determining a code layout have noncyclic dependencies: $\mathbb{B} \rightarrow \mathbb{F} \rightarrow [\mathbb{S}, \mathbb{O}] \rightarrow [\{\dot{\mathbb{A}}\}, \{\bar{\mathbb{A}}\}, \{ShMem(D_i)\}, \mathbb{L}]$, where $X \rightarrow Y$ means "Y depends on X." Therefore, as long as the space exploration follows such dependencies, the probed space and the resulting projection would remain the same; further phase-ordering does not matter [33].

The pseudocode for space exploration of code layouts is described below:

```
for all $\mathbb{B}$ such that $size(\mathbb{B}) \leq Max\_block\_size$:
  for all $\mathbb{F}$ such that $size(\mathbb{B}) \times size(\mathbb{F}) \leq size(Loop\_space)$:
    for all $\mathbb{S}$:
     for all $\mathbb{O}$:
       for all $\{\{\dot{\mathbb{A}}\}, \{\bar{\mathbb{A}}\}, \{ShMem(D\_i)\}\}$:
         maximize $\mathbb{L}$ for any applicable loops
         emit $\{\mathbb{B}, \mathbb{S}, \mathbb{O}, \mathbb{F}, \{\dot{\mathbb{A}}\}, \{\bar{\mathbb{A}}\}, \{ShMem(D\_i)\}, \mathbb{L}\}$
```

Users can provide hints in an input file to narrow down the search space. Hints can be written by specifying domains of each factor in the code layout. GROPHECY will then probe only the space defined by the specified domains. In fact, users can specify a particular code layout to be evaluated.

Without any hints, GROPHECY would autoprune the space to trade-off accuracy for speed. Different $\mathbb{B}$'s and $\mathbb{F}$'s with values that are reasonably far apart are explored. As a result, suboptimal code layouts may be recommended as a tradeoff for reduced projection overhead.

The space can be further pruned by identifying data accesses that might benefit from caching and by determining appropriate staging sizes; both require analysis of data usage.

6.3 Data Usage Representations

The unique set of data elements referenced by a group of loop indices upon *one particular* data access statement, $\mathbb{A}$, is defined as a *pattern*, which is a blocked regular section (BRS), too. We use $\mathbb{P}(\mathbb{A}, \mathbb{T})$ to denote a pattern, where $\mathbb{T}$ is a *tile* referring to the ranges into which loop indices fall. Formally, $\mathbb{T} = \{t_1, ..., t_n\}$, where component is a range defined with a lower bound, an upper bound, and a stride, represented as $t_i : \langle T_i^l, T_i^u, T_i^s \rangle$. The size of a tile can be calculated as $size(\mathbb{T}) = \prod_{i=1}^{n} \lfloor (T_i^u - T_i^l) \div T_i^s \rfloor$. The shape and size of a pattern can be obtained by simply replacing scalar indices of I in $\mathbb{A}(D, \Theta, I)$ with corresponding ranges in $\mathbb{T}$. The tile can represent loop indices in various scopes. In fact, $\mathbb{F}$ defines the rectangular shape of the tile associated with parallel `for` loop indices assigned to a thread, and $\mathbb{B}$ defines the tile shape for thread blocks. We use $\hat{\mathbb{F}}$ to represent the tile associated with a thread plus full ranges of all inner-task loop indices and $\hat{\mathbb{B}}$ to represent the tile associated with a thread block plus $\hat{\mathbb{F}}$.

As an example, for the code skeleton of `MatMul` (Listing 2), the access statement to A is $A[i][k]$. The tile associated with a 16×8 thread block beginning at loop indices of $[i = X, j = Y]$ is $\mathbb{B} = [i : \langle X, (X + 16), 1 \rangle; j : \langle Y, (Y + 8), 1 \rangle]$. The pattern of A associated with this thread block is $\mathbb{P}(A[i][k], \mathbb{B} + [k : \langle 0, K, 1 \rangle]) = A[\langle X, (X + 16), 1 \rangle][\langle 0, K, 1 \rangle]$.

[2]Our experiments show NVCC only applies loop unrolling when the number of iterations is sufficiently small. Nevertheless, a programmer can manually turn on loop unrolling by rewriting a large loop as iterations of smaller loops.

A *footprint*, denoted with $\mathbb{H}(D, \mathbb{T}) = \cup\{\mathbb{P}(\mathbb{A}(D), \mathbb{T}) | \forall \mathbb{A}(D)\}$, is the unique set of data elements in D referenced by *all* patterns within the kernel function (i.e., all data access statements) over a tile $\mathbb{T}$. In the `MatMul` example, because there is only one access to A, the footprint of A over the entire thread block is the same as the access pattern: $\mathbb{H}(A, \hat{\mathbb{B}}) = A[\langle X, (X + 16), 1 \rangle][\langle 0, K, 1 \rangle]$. For calculating the size of tiles, patterns, and footprints, the values of X and Y do not matter and can be assumed to be 0.

When a footprint contains multiple patterns, it might not be straightforward to estimate the number of unique elements in it, because of pattern overlapping. To balance accuracy and time, we first use the `INTERSECT` operator in BRS to determine overlapping patterns. We then use the `UNION` operator in BRS to merge overlapping patterns and eventually leave only non-overlapping patterns in the footprint. The size of a footprint, denoted with $size(\mathbb{H})$, is the total number of data elements in the footprint and can be calculated as the summed size of all non-overlapping patterns in that footprint.

The size of a pattern for an *indirect* access statement can also be estimated. Because indirect indices may point to anywhere in the corresponding dimension, the worst case is assumed; the resulting index range covers the entire dimension. However, users can provide hints about the range of indirect indices.

6.4 Identifying Cacheable Data

We define the *degree of sharing* as the average number of threads in a thread block that access the same data element. Any array with a degree of sharing larger than one becomes a candidate for caching in GPU's shared memory. The set of cacheable arrays are denoted with $\{\bar{D}\}$. Given an array D, its degree of sharing can be calculated as follows.

$$ShrDegree(D) = \frac{size\left(\mathbb{H}(D, \hat{\mathbb{F}})\right) \times size(\mathbb{B})}{size\left(\mathbb{H}(D, \hat{\mathbb{B}})\right)} \tag{1}$$

There may also be data elements that are accessed multiple times within an individual thread. GROPHECY assumes compilers or programmers can identify such patterns and use registers to reuse such elements. For data access statements whose indices can be statically determined to address the same element, only the first load and the last store are marked as memory accesses; all other accesses are treated as computation instructions and are not considered for caching.

6.5 Determining Staging Sizes

An appropriate staging size has to be large enough so that in each stage, all threads can simultaneously load different data elements into the shared memory. Ideally, a multiple of $size(\mathbb{B})$ elements within an array can be cached in each stage. Therefore, for each data reference addressed with the streaming loop index, we calculate the suggested staging size and then evaluate only these staging sizes in space exploration.

$$StageShrDegree(D, k) = \frac{size\left(\mathbb{H}(D, \hat{\mathbb{B}} \wedge [k : \langle 0, 1, 1 \rangle])\right) \times size\left([k : \langle K_l, K_u, K_s \rangle]\right)}{size\left(\mathbb{H}(D, \hat{\mathbb{B}} \wedge [k : \langle K_l, K_u, K_s \rangle])\right)} \tag{2}$$

$$StageSize(D, k) = \frac{size(\mathbb{B}) \times StageShrDegree(D, k)}{size\left(\mathbb{H}(D, \hat{\mathbb{B}} \wedge [k : \langle 0, 1, 1 \rangle])\right)} \tag{3}$$

$$NumStages(k) = \left(\begin{array}{rl} \left\lceil \frac{size([k:\langle K_l, K_u, K_s \rangle])}{StageSize(k)} \right\rceil, & if\ k > 0 \\ 1, & if\ k = 0 \end{array} \right) \tag{4}$$

For an involved array, D, the suggested staging size is obtained by dividing the thread block size by the average number of unique data elements referenced by an individual thread, expressed in Equation 3, where k is the identifier of the streaming loop index with the range $k : \langle K_l, K_u, K_s \rangle$, and the operator "$X \wedge Y$" sets the ranges of indices in tile X with corresponding ranges in Y. The tile $\mathbb{H}(D, \hat{\mathbb{B}} \wedge [k : \langle 0, 1, 1 \rangle])$ corresponds to the entire thread block within one iteration of the streaming loop. Equation 4 calculates the number of stages associated with the streaming loop; $k = 0$ refers to the global scope outside of any streaming loops.

6.6 Estimating Shared Memory Usage

The number of elements in D to be cached during one stage is defined as $ShMem(D, k)$, and it can be calculated using Equation 5. $ShMem(D, 0)$ refers to data cached outside of any streaming loops. Theoretically, the same shared memory space can be reused to cache arrays at different periods of time; currently GROPHECY does not consider this optimization and shared memory space is allocated for every $ShMem(D, k)$. We use $ElemBytes(D)$ to refers to the number of bytes per each element in D, which is given by the code skeleton.

$$ShMem(D,k) = \begin{pmatrix} \mathbb{H}(D, \hat{\mathbb{B}} \wedge [k : \langle 0, StageSize(k), 1 \rangle]), & if\ k > 0 \\ \mathbb{H}(D, \hat{\mathbb{B}}), & if\ k = 0 \end{pmatrix} \quad (5)$$

$$ShMemAlloc(D) = ElemBytes(D) \times \sum_k size\,(ShMem(D,k)) \quad (6)$$

When staging is applied, the modeled code layout is added with a loop to iterate through stages, a set of instructions to cache data into shared memory, and two synchronization instructions per stage to coordinate among threads about the start and end of the caching process.

6.7 Effects of Approximation

Note that BRS's `UNION` operator produces a new pattern in the form of BRS that can best approximate the merged patterns. Although the resulting pattern remains simple, it may cover additional elements, which can lead to overestimation of footprint sizes. This may artificially increase the sharing degree of an array and include unnecessary arrays as caching candidates; the effect is longer space exploration time.

An oversized merged pattern may also lead to larger shared-memory allocation; however, for the purpose of both programmability and performance, GPU programmers usually prefer to explicitly cache a simple, regular array region with a few unused elements, rather than to cache several smaller array fragments without redundancy. The approximation used in the `UNION` operator actually captures such behavior.

7. Characterizing Code Layouts

Given the parameterization of a code layout in Section 6, the next step of GROPHECY is to characterize its performance on some candidate GPU hardware (recall Figure 1, Step 3). This section describes this characterization. The characteristics of a GPU implementation is synthesized by using parameters that describe the corresponding code layout the underlying hardware. Table 1 and Table 5 lists the two sets of parameters, respectively.

Table 2 lists the synthesized characteristics that eventually serve as inputs to the GPU performance model. While several characteristics can be obtained directly from Table 1, the numbers of active thread blocks per SM, computation instructions, and each category of memory instructions are yet to be calculated.

7.1 Active Thread Blocks per SM

The number of concurrent thread blocks on each SM is determined by four factors: the number of thread blocks available on each SM, the maximum active thread blocks imposed by the hardware, the maximum active warps imposed by the hardware, and the number of thread blocks that the shared memory can accommodate given the consumption of each thread block. The limited number of registers imposes an additional constraint; however, it remains future work to model register usage and include that information in the framework. The number of active thread blocks per SM is calculated as follow:

$$SharedMem\_bytes\_per\_block = \sum_i ShMemAlloc(D_i) \quad (7)$$

$$Blks\_per\_ShrM = \left\lfloor \frac{SharedMem\_size}{SharedMem\_bytes\_per\_block} \right\rfloor$$

$$warps\_per\_block = \left\lceil \frac{Thread\_per\_block}{warp\_size} \right\rceil$$

$$Active\_blocks\_per\_SM = \min(\frac{Max\_active\_warps\_per\_SM}{warps\_per\_block}, Blks\_per\_ShrM, Max\_active\_blks\_per\_SM, \frac{Blocks}{Active\_SMs}) \quad (8)$$

7.2 Global Memory Accesses

Since latency in accessing shared memory is similar to that of register accesses, we treat shared memory loads and stores as computation instructions, not memory accesses. The average number of bytes accessed by a SIMD memory instruction is as follows.

$$data\_reqs(D_i) = \sum_k size\,(ShMem(D_i,k)) + \sum_j size\left(\mathbb{P}(\dot{\mathbb{A}}_j(D_i), \hat{\mathbb{B}})\right) \quad (9)$$

$$Avg\_elem\_bytes = \frac{\sum_i (ElemBytes(D_i) \times data\_reqs(D_i))}{\sum_i data\_reqs(D_i)} \quad (10)$$

$$Load\_bytes\_per\_warp = Avg\_elem\_bytes \times warp\_size \quad (11)$$

Instructions that access global memory have two categories. Those that cache data into shared memory are synthesized by GROPHECY and are referred to as *caching memory instructions*. They can be counted according to shared-memory usage. Those that directly use global memory data for computation are referred to as *direct memory instructions*. They can be counted according to access statements specified in code skeletons.

$$caching\_mem\_insts(D) = \sum_k \left(\left\lceil \frac{size(Shmem(D,k))}{Thread\_per\_block} \right\rceil \times NumStages(k) \right) \quad (12)$$

$$direct\_mem\_insts\left(\dot{\mathbb{A}}_j(D)\right) = size\left(\mathbb{P}(\dot{\mathbb{A}}_j(D), \hat{\mathbb{F}})\right) \quad (13)$$

Whether a caching memory instruction is coalesced depends on what data elements are simultaneously accessed by a warp. We assume that adjacent threads access adjacent shared-memory elements. We then generate memory access indices corresponding to the first $warp\_size$ elements of a shared-memory object $ShMem(D, k)$. This sequence of indices is analyzed to determine whether the access is coalesced and how many memory transactions (Mem_trans) are incurred. The algorithm to determine coalescing and the number of resulting memory transactions are inferred from [14, 29].[3] Note that the algorithm varies across GPU generations.

Whether a direct memory instruction is coalesced depends on the data access pattern of a warp. We use $\mathbb{W}$ to refer to the tile whose indices associated with the parallel `for` loop correspond to

[3]In a nutshell, for compute capabilities < 1.3, a SIMD memory access issues two memory transactions if and only if threads access adjacent array elements and the first address is aligned; otherwise every thread incurs one memory transaction. For higher compute capabilities, coalescing occurs for regularly strided array indices. The address span of all memory accesses from a half warp is computed; the number of memory transactions per a half warp is calculated by dividing the address span with the memory transaction size. If addresses are not aligned, there is an additional memory transaction. In the worst case, every thread incurs one memory transaction.

Table 1: Parameters describing a code layout

Parameter	Description	Obtained
$\mathbb{B}$	The shape of the thread block	Enumeration
$\mathbb{F}$	Parallel `for` loop indices assigned to one thread along each dimension	Enumeration
F	No. of tasks assigned to each thread	Definition
$ShMem(D_i, k)$	Elements of D_i cached by a thread block in one stage of the k^{th} streaming loop	Equation 5
$ShMemAlloc(D_i)$	Shared memory allocation used to cache D	Equation 6
$NumStages(k)$	No. of stages for the k^{th} streaming loop	Equation 4
$\{\dot{\mathbb{A}}_i\}$	The set of global memory access statements for an individual thread (not including global memory accesses that cache data into shared memory)	Definition
$ElemBytes(D_i)$	No. of bytes per element in D_i	Code skeleton
$comp_j$	No. of instructions per basic block	Code skeleton

Table 2: Workload characteristics serving as inputs to the GPU performance model

Parameter	Description	Obtained
$Thread\_per\_block$	No. of threads in a thread block	$size(\mathbb{B})$
$Blocks$	No. of thread blocks	$size(Loop\_space) \div (size(\mathbb{B}) \times F)$
$Active\_blocks\_per\_SM$	No. of concurrently running blocks on one SM	Equation 8
$Total\_insts$	Dynamic no. of instructions in one thread	$Comp\_insts + Mem\_insts$
$Comp\_insts$	Dynamic no. of computation instructions in one thread	Section 7.3
Mem_insts	Dynamic no. of *global* memory instructions in one thread	$Uncoal\_Mem\_insts + Coal\_Mem\_insts$
$Uncoal\_Mem\_insts$	No. of uncoalesced memory instructions in one thread	Equation 15
$Coal\_Mem\_insts$	No. of coalesced memory instructions in one thread	Equation 14
$Synch\_insts$	No. of synchronization instructions in one thread	Section 7.3
$Load\_bytes\_per\_warp$	Average no. of bytes accessed by a warp's SIMD memory instruction	Equation 11

$warp\_size$ adjacent indices. All other inner-thread loop indices in $\mathbb{W}$ are fixed as a constant (which corresponds to a particular iteration). The pattern of a warp's global memory access, $\dot{\mathbb{A}}(D)$, can then be calculated as $\mathbb{P}(\dot{\mathbb{A}}(D), \mathbb{W})$. The resulting sequence of indices is analyzed in the same way as caching accesses to determine coalescing.

We use † for coalesced data accesses and $'$ for uncoalesced data accesses. The numbers of coalesced and uncoalesced memory instructions are calculated in Equation 14 and Equation 15, respectively. The average number of memory transactions per uncoalesced instruction is calculated in Equation 16.

$$Coal\_Mem\_insts = \sum_j direct\_mem\_insts(\dot{\mathbb{A}}_j^\dagger) + \sum_i caching\_mem\_insts^\dagger(D_i) \quad (14)$$

$$Uncoal\_Mem\_insts = \sum_j direct\_mem\_insts(\dot{\mathbb{A}}_j') + \sum_i caching\_mem\_insts'(D_i) \quad (15)$$

$$Uncoal\_per\_mw = \frac{\sum_j Mem\_trans(\dot{\mathbb{A}}_j') + \sum_{i,k} Mem\_trans\left(ShMem'(D_i,k)\right)}{Uncoal\_Mem\_insts} \quad (16)$$

7.3 Computation Instructions

Instruction count of a GPU implementation can be a lot different from the corresponding CPU implementation because of differences in ISA, compilers, and code structure. To estimate GPU instruction count from the CPU code, we categorize GPU instructions into two categories.

Functional instructions are those at the core of an algorithm that perform the actual computation. Their quantity is rarely affected by code transformation. Users include the instruction counts in the code skeleton as inputs to GROPHECY.

Peripheral instructions are those that assist functional instructions in data preparation, branching, loop management, and synchronization. They vary significantly according to code transformations. GROPHECY synthesizes and deduces their instruction counts without user intervention.

The CPU code can be viewed as a summary of functional instructions. In order to account for ISA and compiler differences between CPU and GPU, functional instructions have to be counted based on codes compiled for GPU, which can be obtained as PTX assembly produced by NVCC [29]. To accomplish this step, users prefix the target CPU function with "__global__" so that the function is treated by NVCC as a GPU kernel. In addition, users mark the boundary of every basic block in the CPU kernel code with embedded assembly markers like "asm("_Begin_")" and "asm("_End_")". In the produced PTX code, the numbers of instructions between the markers are then recorded in the code skeleton. A script is provided to automate this process. Note that with this method, loop and branch instructions are not counted; they will be synthesized as peripheral instructions.

While the PTX assembly generated in this way has all memory instructions as global memory accesses and uses a single thread to calculate the entire workload, instruction counts for the marked code region in inner loop bodies are similar to those of actual GPU implementation. Note that this approach takes into account CUDA-specific instructions such as "multiply-and-add". Given N basic blocks each having Ins instructions and looping LP times within a task, the number of functional instructions for a given thread with F tasks is as follows.

$$Func\_insts = F \times \sum_{i=1}^{N} (Ins_i \times LP_i) \quad (17)$$

GROPHECY synthesizes several types of peripheral instructions.

Loop and branch management instructions include instructions that increment loop indices, test branch conditions, and the branches themselves. Note that they are not included when counting functional instructions in basic blocks. The per iteration or per branch instruction counts are multiplied by the number of loop iterations or branches, and are then added to the total instruction count. Unrolled loops do not have loop management instructions.

Data preparation instructions are those that move fetched global memory data to corresponding shared-memory addresses during caching. They are added for every caching instruction.

Index calculation instructions are used to calculate indices for global array accesses. Often, they are needed when adjacent threads do not use adjacent indices. They are added for every uncoalesced memory access.

Synchronization instructions are needed after caching data into the shared memory. They are also needed at the end of every stage before the next stage overwrites the cached arrays. They can be calculated as twice the number of stages.

We studied several PTX codes and empirically obtained the numbers of peripheral instructions in each of the above cases, as listed in Table 3. The number of computation instructions is the sum of all functional instructions and all peripheral instructions.

Table 3: Compiler parameters used in GROPHECY

Parameter	Estimated Value
No. of loops to unroll	16
No. of peripheral instructions per conditional branch	3
No. of peripheral instructions per loop iteration	5
No. of peripheral instructions for loading an element into shared memory	2
No. of peripheral instructions to calculate an index for an uncoalesced access	4

7.4 Adopting a GPU Performance Model

We adopt the GPU performance model developed by Hong and Kim [13], which has an average error of 13% according to their experiments. This analytical model approximates the execution of a GPU kernel as computation phases of equal length, with global memory accesses in between. It then estimates the average overlapping between memory accesses and computation to determine whether the execution is computation bounded or memory bounded. The execution time is estimated by using workload characteristics in Table 2 and hardware specifications in Table 5.

While this GPU performance model mostly uses off-the-shelf hardware specifications, it obtains a couple of hardware parameters (namely, DRAM access latency and delays between memory transactions) by measuring microbenchmarks on a physical GPU. Nevertheless, users need not go through this step because such observed hardware parameters can be made public by a third party.

8. Methodology

Two GPUs, Quadro FX5600 and Tesla C1060, are used in our experiments to validate the projection across GPU generations. Their characteristics are listed in Table 5.[4]

The benchmarks used for our evaluation are listed in Table 4. All implementations use single precision. The sizes of matrices in `IspinEx` and `SpinFlap` are according to real input data. GPU implementations are compiled with NVCC over an Intel Xeon E5430 CPU. Host codes are compiled using GCC 4.0.1 with "-O2" optimization. The GPU kernel execution time is measured by surrounding kernel calls with "cudaThreadSynchronize" and counting the number of CPU cycles in between using the "rdtsc" instruction. Time is then converted to microseconds according to the CPU clock rate. The measured execution times are the average of 10 runs.

9. Evaluation

The benchmarks are manually implemented and tuned as GPU kernels. We record such development process and compare it with the process using GROPHECY. For validation, implemented GPU source codes are manually inspected and compared with

[4]Due to architectural similarities in shader cores between C1060 and GTX280, we adopt the values of Mem_LD, $Departure\_del\_uncoal$, and $Departure\_del\_coal$ for C1060 from those reported in [13] for GTX280.

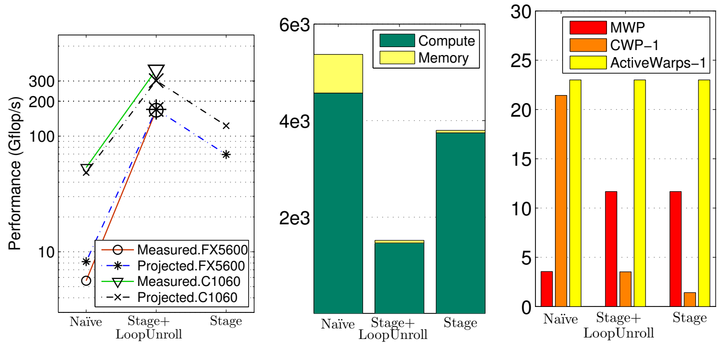
Table 4: Workload properties

Benchmark	Key Properties	Input Size
`MatMul`	dense linear algebra	$A[800][400] \times B[400][800]$
`HotSpot` [15]	stencil computation structured grid	512×512
`IspinEx` [17, 31, 32]	sparse linear algebra	$A[132][132]$(sparse, real numbers)$\times B[132][2048]$ (dense, complex numbers)
`SpinFlip` [17, 31, 32]	irregular data exchange similar to spectral methods	132×2048 (complex numbers)

statistics generated from their corresponding code layouts. The code layout associated with an actual GPU implementation is obtained by specifying the parameters in Table 1 as hints to GROPHECY when generating the code layout. In *all* our experiments, the measurements of shared-memory usage and the number of coalesced and uncoalesced memory instructions are precise. We further validate GROPHECY by evaluating the accuracy of projected performance and the quality of automatically suggested code layouts. The time to transfer data between CPU and GPU remains the same for all code layouts generated from the same code skeleton; therefore, we report only the GPU kernel execution time for comparison purposes.

GROPHECY also uses the underlying analytical model to estimate the bottleneck of a workload. *Memory Warp Parallelism* (MWP) refers to the maximum number of overlapping global memory accesses given the latency delay between memory transactions and the memory bandwidth. *Computation Warp Parallelism* minus one (CWP-1) refers to the demanded number of warps whose computation can together hide the latency of an outgoing global memory access. The number of warps that are active on an SM is denoted with $ActiveWarps$, and $ActiveWarps - 1$ is the number of available warps for latency hiding. The workload is memory bounded if MWP is the smallest, computation bounded if $CWP-1$ is the smallest, and parallelism bounded (i.e., not enough warps) if $N-1$ is the smallest.[5]

9.1 MatMul: Staging and Loop Unrolling



(a) Performance (b) Inst.Count(Fx5600) (c) Bottleneck(FX5600)

Figure 2: Validating projections of `MatMul`. In all cases, the thread block size is 16×16. Enlarged markers for measured data correspond to manually tuned implementations. Enlarged markers for projected data correspond to code layouts suggested by GROPHECY. Staging reduces memory instructions and transforms the GPU kernel from memory bounded ($MWP < CWP - 1$) to computation bounded ($MWP > CWP - 1$). Without loop unrolling, the number of computation instructions almost doubles.

The baseline implementation (denoted with `Naïve`) fetches ev-

[5]The original performance model compares MWP, CWP, and N instead. Deduction shows that comparison of MWP and $CWP - 1$ determines whether an SM can issue another global memory access by the time the next one is encountered. Our experiments also show that such comparison is a better clue in identifying the bottleneck. Because of space limitation, the proof is omitted.

Table 5: Hardware parameters. Parameters marked with "*" are used in modeling code layouts and generating workload statistics. All other parameters are used by the underlying GPU performance model.

Parameter	Description	Obtained	FX5600	C1060
$SharedMem\_size^*$	The size of the shared memory	DeviceQuery	16 KB	16 KB
$Max\_active\_blks\_per\_SM^*$	The maximum No. of thread blocks that can run concurrently on one SM	CUDA Occupancy Calculator	8	8
$Max\_active\_warps\_per\_SM^*$	The maximum No. of warps that can run concurrently on one SM	CUDA Occupancy Calculator	24	32
$warp\_size^*$	No. of threads in a warp	[29]	32	32
$Active\_SMs$	No. of stream processors	DeviceQuery	16	30
$Issue\_cycles$	No. of cycles to execute on instruction	[19]	4	4
$Freq$	Clock frequency	DeviceQuery	1.35GHz	1.3GHz
$Mem\_Bandwidth$	Memory bandwidth	Machine specification	76.8GB/s	104.2GB/s
Mem_LD	DRAM access latency	[13]	420 cycles	450 cycles
$Departure\_del\_uncoal$	Delay between two uncoalesced memory transactions	[13]	10 cycles	40 cycle
$Departure\_del\_coal$	Delay between two coalesced memory transactions	[13]	4 cycles	4 cycles
Peak flop rate	(Not used by GROPHECY)	Machine specification	518.4 GFlops	933.1 GFlops
Compute capability	Better coalescing mechanism if it is ≥ 1.3	DeviceQuery	1.0	1.3

ery column and row from the global memory when computing *each* element. As shown in Listing 3, in order to utilize limited shared-memory space, the manually tuned implementation (denoted with `Stage+LoopUnroll`) caches a block of data from each of the two input matrices into the shared memory, computes the partial dot product, and moves on to the following blocks while aggregating the partial sums. Cached data can be reused by multiple threads in the same thread block. This is a typical example of staging. Given the code skeleton in Listing 2, GROPHECY parameterizes the code layout with a stage size of 16 so that every thread in a 16×16 thread block can load an element in every batch.

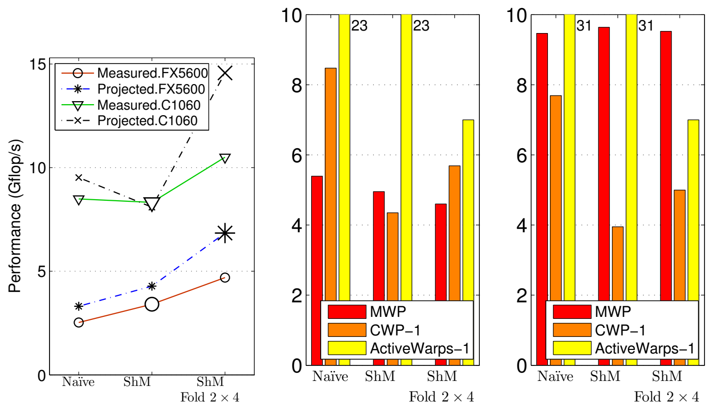
In fact, `Stage+LoopUnroll` is exactly the code layout suggested by GROPHECY after searching the space of 6372 code layouts. It achieves 167 Gflop/s on FX5600 and 375 Gflop/s on C1060. The measured achieved performance on FX5600 and C1060 deviates from the projected performance by 2% and 23%, respectively. Using statistics generated from the code layout, Figure 2b shows that global memory accesses are drastically reduced after staging is applied, which effectively transforms the memory bounded workload to computation bounded (Figure 2c).

Moreover, the NVCC compiler implicitly unrolls the innermost loop, which has significant performance benefit. GROPHECY evaluates its benefit by disabling loop unrolling. Such code layout is denoted with `Stage` in Figure 2. Without loop unrolling, the number of computation instructions doubles, which almost doubles the total execution time of a computation bounded implementation.

9.2 HotSpot: Folding and Coalescing

`HotSpot` is an ordinary differential equation solver used in simulating microarchitecture temperature. Every element is computed by gathering a 3×3 neighborhood of elements (i.e., the stencil) from the input array. The `Naïve` implementation does no caching. Manually improved code, denoted with `ShM`, caches neighborhood data so it can be reused among neighboring threads.[6] As Figure 3a shows, this improves performance on FX5600, but slightly degrades performance on C1060. Such a phenomenon is correctly projected by GROPHECY. Furthermore, GROPHECY searches the space of 3804 code layouts and suggests a folding degree of 2 and 4 in the first and second dimension of the loop space, respectively (i.e., every thread computes eight elements). This code layout is denoted with `ShM+Fold` 2×4. After implementing this suggested code layout, experiments show that the suggested implementation has a speedup of 1.37× and 1.26× over `ShM` on FX5600 and C1060, achieving 5 Gflop/s and 11 Gflop/s, respectively. The measured achieved performance on FX5600 and C1060 deviates from the projected performance by 31% and 27%, respectively.

[6]Some code transformations are not captured by GROPHECY. For example, the performance of `HotSpot` can be further improved by using the pyramid method to replicate computation and reduce data movement. This would require a different code skeleton.



(a) Performance (b) Bottleneck(FX5600) (c) Bottleneck(C1060)

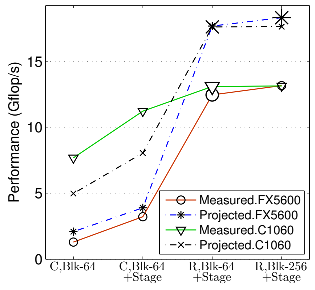
Figure 3: Validating projections of `HotSpot`. In all cases, the thread block size is 16×16. Enlarged markers for measured data correspond to manually tuned implementations. Enlarged markers for projected data correspond to code layouts suggested by GROPHECY. `Naïve` is memory bounded ($MWP < CWP - 1$) on FX5600, but it is already computation bounded ($MWP > CWP - 1$) on C1060 because of better coalescing. Therefore caching has different effects on the two GPU hardware.

`Naïve` has many uncoalesced memory accesses. On FX5600, it leads to a significant number of memory transactions (large $Uncoal\_per\_mw$); therefore, `Naïve` is memory bounded (Figure 3b), and `ShM` reduces global memory accesses to overcome the memory bottleneck. In the case of C1060, the hardware's improved coalescing mechanism already groups requests into fewer memory transactions. As a result, `Naïve` is already computation bounded (Figure 3c). `ShM` actually increases the instruction count for the purpose of caching; therefore, its performance is even slightly worse than `Naïve`.

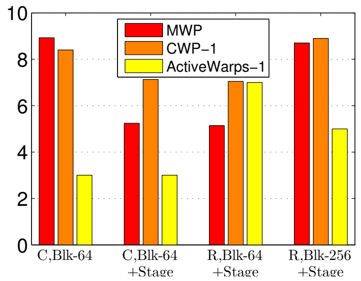
By allowing a thread to process more neighborhood-gathering tasks, folding improves data reuse; the larger amount of data to gather also reduces memory accesses that do not utilize the full SIMD width. Moreover, folding reduces *per task* computation instructions by initializing thread indices and loading arguments once for *all tasks* within a thread. Hence it improves performance on both FX5600 and C1060.

9.3 IspinEx: Sparsity and Code Restructuring

`IspinEx` is a piece of code that lies in the core of GFMC [17, 31, 32], a quantum physics application that performs Monte Carlo calculation for light nuclei. It multiplies a 132×132 sparse matrix of real numbers with a 132×2048 dense matrix of complex numbers. The sparse matrix `A` is compressed and represented with three vectors: `T`, `J`, and `I`. `T[n]` stores the values of nonzero elements. `A`'s first nonzero element in the jth row corresponds to the `T` element with the index of `J[j]`. `I[n]` stores the column number of the nth element in T. In other words, the mth



(a) Single-precision performance



(b) Bottleneck on C1060

Figure 4: Validating projections of `IspinEx`. Enlarged markers for measured data correspond to manually tuned implementations. Enlarged markers for projected data correspond to code layouts suggested by GROPHECY. Although C1060 has more SMs, each SM has an insufficient number of warps to hide latency ($ActiveWarps - 1 < MWP$ and $ActiveWarps - 1 < CWP - 1$) because of limited input size of real data set. Therefore C1060 does not gain much performance compared with FX5600.

nonzero element in the jth row of `A` is $A[j][I[J[j] + m]]$, which corresponds to $T[J[j] + m]$. Listing 4 is the code skeleton of `IspinEx`.

To project the performance of `IspinEx`, To estimated the overall workload size, GROPHECY is "hinted" that the average number of nonzero elements in one row is 14. Because rows in `A` have different numbers of nonzero elements, the numbers of tasks associated with different rows also vary. To balance the workload among threads, we force a thread to process all tasks in columns by setting the folding degree in the first dimension (corresponding to index j) as 132.

Listing 4: Baseline code skeleton of `IspinEx`. Array `B` is in the form of complex numbers

```
 1 /* compute data as
    * complex numbers
 3  */
   #define ELEMS 1848
 5 #define ROWS 132
   #define COLS 2048
 7
   int J[ROWS+1]
 9 int I[ELEMS]
   float T[ELEMS]
11 float B[ROWS][COLS][2]
   float C[ROWS][COLS][2]
13 parallel_for(ROWS, COLS)
   : j, i
15 {
     ld J[j]
17   ld J[j+1]
     begin = J[j]
19   end = J[j+1]
     comp 4
21   /* The No. of nonzero
      * elements depends
23    * on data in array J.
      * Non-constant boundary
25    * disables unrolling.
      * Hint the average loop
27    * size.
      */
29   stream n = begin:end
     (hint:14)
31   {
       ld T[n]
33     ld I[n]
       r = I[n]
35     /* indirect accesses to
        * the complex number
37      */
       /* real part */
39     ld B[r][i][0]
       /* imaginal part */
41     ld B[r][i][1]
       /* sum+=T[n]*B[r][i] */
43     comp 22
     }
45   /* C[j][i] = sum */
     comp 2
47   st C[j][i][0]
     st C[j][i][1]
49 }
```

Listing 5: Modified code skeleton of `IspinEx`. Array `B` is in the form of real numbers

```
/* compute data as
 * complex numbers
 */
#define ELEMS 1848
#define ROWS 132
#define COLS 4096

int J[ROWS+1]
int I[ELEMS]
float T[ELEMS]
float B[ROWS][COLS]
float C[ROWS][COLS]
parallel_for(ROWS, COLS)
: j, i
{
  ld J[j]
  ld J[j+1]
  begin = J[j]
  end = J[j+1]
  comp 4
  /* The No. of nonzero
   * elements depends
   * on data in array J.
   * Non-constant boundary
   * disables unrolling.
   * Hint the average loop
   * size.
   */
  stream n = begin:end
  (hint:14)
  {
    ld T[n]
    ld I[n]
    r = I[n]
    /* indirect accesses */
    ld B[r][i]
    /* sum+=T[n]*B[r][i] */
    comp 11
  }
  /* C[j][i] = sum */
  comp 2
  st C[j][i]
}
```

Despite the fact that indirect accesses make it impossible to statically calculate the exact array access indices, GROPHECY is able to estimate the size of array footprints because there is a one-to-one mapping between indirect indices and the actual indices they point to. Because all threads share *the same* indirect index r when accessing array `B` (Line 39 and 41 in Listing 4), GROPHECY deduces that all threads access the same row. The degree of coalescing can then be inferred solely from the column indices, i, which is not indirect.

Similar to `MatMul`, GROPHECY identifies the opportunity of staging given the streaming loop. Staging enables thread blocks to cache `T`, `J`, and `I`; hence `C.Blk-64+Stage` outperforms the naïve `C.Blk-64` implementation. Note that array `B` is not cached; the indirect access can refer to any row in `B`, and the shared memory is too small to accommodate all rows.

GROPHECY further reveals that `C.Blk-64+Stage` exhibits a significant amount of uncoalesced accesses on FX5600, mostly due to strided accesses caused by the interleaved real and imaginary parts of complex numbers. Therefore, the developer adjusts the CPU implementation by treating the real part and imaginary part of a complex number as independent elements that can be processed in parallel. The user can simply adjust the code skeleton from Listing 4 to Listing 5. Given the updated code skeleton, GROPHECY deduces that uncoalesced instructions per thread can be reduced from 4,050 to 0 and that performance can be further improved by 3.7× on FX5600. We adjusted the actual GPU implementation accordingly and found it gained 3.9× speedup over the previous implementation. This layout is denoted with `R.Blk-64+Stage`. GROPHECY searches a space of 2448 code layouts and suggests the same code layout for C1060. For FX5600, GROPHECY suggests the same layout with a thread block size of 256, denoted with `R.Blk-256+Stage`, which performs almost identically to the manually tuned version of `R.Blk-64+Stage`. Specifically, `R.Blk-64+Stage` achieves 13 Gflop/s on both GPU generations. The measured achieved performance on FX5600 and C1060 deviates from the projected performance by 28% and 25%, respectively.

GROPHECY also projects that the suggested code layout will not benefit much from upgrading to C1060. As Figure 4b shows, the number of active warps is usually the performance bottleneck, hence it cannot spawn enough warps to fully exploit GPU's latency hiding capability. Note that the input size is obtained from real data sets.

9.4 SpinFlap: Indirect Accesses and Thread Block Sizes

`SpinFlap` is a piece of code that belongs to the same GFMC application as `IspinEx`. During its computation, distant columns of the input matrix are grouped into units of four. In each group, the four complex numbers in the same row are processed together. However, which columns to be grouped together is determined by values in another matrix. The code skeleton is illustrated in Listing 6.

Listing 6: Code skeleton of `SpinFlap`

```
#define ROWS 132
#define COLS 2048
float A[ROWS][COLS][2]
float B[ROWS][COLS][2]
float C[ROWS][COLS][2]
/* M: index array for indirect accesses.
 *    A sample data array is provided as a hint to better assess coalescing.
 */
int M[COLS/4][4] : hints<sample="./M.txt">
parallel_for(ROWS, COLS/4) : j, i
{
  for n = 0:4
  {
    ld M[i][n]
    /* load the complex number in A */
    ld A[j][M[i][n]][0]
    ld A[j][M[i][n]][1]
    /* load the complex number in B */
    ld B[j][M[i][n]][0]
    ld B[j][M[i][n]][1]
    comp 56
  }
  comp 228
  /* produce the complex numbers */
  for n = 0:4
  {
    /* store the computed complex number */
    st C[j][M[i][n]][0]
    st C[j][M[i][n]][1]
  }
}
```

Except indirect indices in `M`, no data is reused by multiple threads. Compared with the `Naïve` implementation without caching, code layouts denoted with `ShM` only gain moderate performance after caching elements in `M`. The performance therefore depends largely on the degree of coalescing, and it varies according to values of indirect indices.[7] To better assess the coalescing degree, we group every *warp_size* elements in the same column of `M` as indices of a sample SIMD memory access to estimate coalescing. The average number of memory transactions of all samples are used as the coalescing degree, or *Uncoal_per_wm*, for the indirect accesses. The value turns out to be 32 for FX5600 and 5.7 for C1060.

With this hint about the degree of coalescing, the performance of `SpinFlap` can be projected in the same way as other examples and results are shown in Figure 5. The manually tuned implementation for C1060 has a thread block size of 1×256. GROPHECY searches a space of 16113 code layouts and suggests the same code layout for C1060. For FX5600, GROPHECY suggests a code layout with a thread block size of 1×32 and a folding degree of 44×1. The actual implementation of the suggested code layout performs 5% worse than the manually tuned implementation with a thread block size of 12×16 and no folding. The best achieved performance on FX5600 and C1060 is 16 Gflop/s and 50 Gflop/s, respectively. The measured achieved performance on FX5600 and C1060 deviates from the projected performance by 30% and 10%, respectively.

Analysis shows that `SpinFlap` is memory bound. However, it does not necessarily mean that the performance on FX5600 and C1060 is proportional to their memory bandwidth. In this case, it is actually the microarchitectural mechanisms of coalescing that plays an important role. This phenomenon is captured by GROPHECY.

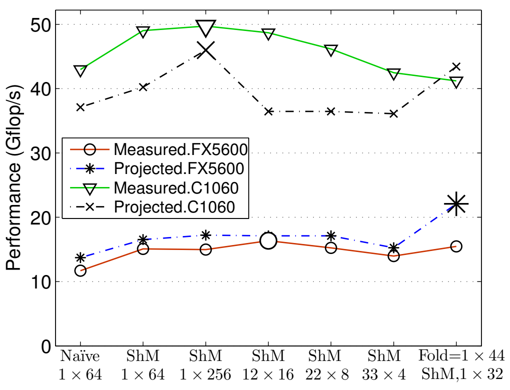


Figure 5: Validating projections of `SpinFlap` in single precision. Enlarged markers for measured data correspond to manually tuned implementations. Enlarged markers for projected data correspond to code layouts suggested by GROPHECY. GROPHECY can project the performance for workloads with indirect accesses as well.

10. Limitations

Note that modifying algorithms, restructuring data, and automatic parallelization are beyond the scope of GROPHECY. Nevertheless, the framework could be used as a guide while designing a parallel implementation of an existing serial code.

10.1 Challenges in Code Skeletonisation

For some legacy codes, users still have to develop a parallel implementation first before generating code skeletons. Although code skeletonisation reduces programming effort compared to rewriting code in CUDA, it may still take significant time if the users are not familiar with the CPU code or if the code is complex. We hope this step can be made easier in the future using compiler analysis and code annotations.

Currently, the code skeleton works best for kernels with a representative control path. Explicitly expressing multiple control paths would require user-provided hints that describes the probability of falling into each path. Details will be studied in our future work. Data-dependent control flow may also challenge code skeletonisation; hints have to be provided to better estimate workload. Line 21 of Listing 4 shows an example of using hints to specify the average size of a data-dependent loop space. Moreover, the ability to use code skeletons to express and model pointer-based irregular data accesses also needs further investigation.

[7]Transposing the matrix and parallelizing the j loop can produce more coalesced accesses, but overall it is not beneficial considering that this would affect other kernels in the same application (*e.g.*, `IspinEx`).

10.2 Constraints in Transformation and Modeling

Currently, GROPHECY is not able to project performance for GPU implementations that utilize texture memory and constant memory, as well as implicit cache hierarchies, which are present in NVIDIA's Fermi architecture [1]. In addition, GROPHECY does not model instruction level parallelism, which may be an important factor in future GPU generations [38]. Finally, so far GROPHECY is validated only with single-precision kernels; a performance model that accounts for longer latency in double-precision instructions is needed for projecting double-precision code.

Data transfer time between CPU and GPU is another important factor for the evaluation of GPU acceleration. However, it depends more on the PCIe bus than on the GPU itself. We have modeled the PCIe's achievable throughput for various message sizes and validated this experimentally on various GPU-integrated systems. Such modeling will be integrated into GROPHECY in our future work.

10.3 Sources of Inaccuracy

Inaccuracy in the projected performance can result from the synthesized characteristics and the GPU performance model. Sources of inaccuracy in the synthesized characteristics include the following: (1) for irregular data accesses, the estimated size of data patterns, footprints, and coalescing degree can deviate from actual values; (2) the number of functional instructions is estimated from NVCC-compiled CPU source, which can be different than the optimized GPU code; (3) the number of peripheral instructions is empirically obtained and can vary across workloads; and (4) register usage is not considered, but it can affect the number of active thread blocks per SM.

Sources of inaccuracy intrinsic in the GPU performance model mainly include the following: (1) the computation and memory access intensity is assumed to be uniform during execution of a workload; (2) the DRAM memory scheduler schedules memory requests equally for all warps; (3) computation instructions have the same latency; and (4) bank conflicts are not considered. These factors are discussed in more detail elsewhere [13].

11. Conclusion

We propose a GPU performance projection framework, named GROPHECY, for fast evaluation of GPU acceleration *without* actual GPU hardware or GPU programming. GROPHECY offers several benefits. (1) the potential GPU performance of a CPU code piece can be projected, and code pieces unsuitable for GPU acceleration can be identified without GPU development; (2) structures of favorable GPU implementations are suggested as a clue for actual GPU development to reduce optimization effort; and (3) performance trends over different GPU architectures or even future GPU generations can be projected, so that users can decide whether or not to upgrade to a different GPU hardware.

In our experiments, we compare the projected optimized GPU performance with that of manually tuned code. The measured performance of manually tuned codes deviates from the projected performance by 17% in geometric mean, with a maximum of 31%. With GROPHECY, the process of evaluating potentials of GPU acceleration can be reduced significantly. GROPHECY also suggests code layouts whose actual performance is no worse than 95% of that yielded by manually tuned implementations. In some cases, the suggested implementation even has a speedup up to 1.37× compared with manually tuned code.

Acknowledgments We thank Steven Pieper, who introduced GFMC kernels, and Sunpyo Hong, who provided advice on the GPU performance model. We also thank Richard Vuduc and the anonymous reviewers for their helpful comments. Gail Pieper's careful editing of the manuscript is also gratefully acknowledged. This work was suppored by the U.S. Department of Energy under contract DE-AC02-06CH11357. The research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

12. References

[1] NVIDIA's next generation CUDA compute architecture: Fermi. *NVIDIA Corporation*, 2009.

[2] Advanced Micro Devices, Inc. AMD Brook+. http://ati.amd.com/technology/streamcomputing/AMD-Brookplus.pdf.

[3] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. Hwu. An adaptive performance modeling tool for GPU architectures. In *PPoPP*, 2010.

[4] V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *PLDI*, 1989.

[5] M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *CC*, 2010.

[6] C. D. Callahan. *A global approach to detection of parallelism.* PhD thesis, 1987.

[7] L. Carrington, M. M. Tikir, C. Olschanowsky, M. Laurenzano, J. Peraza, A. Snavely, and S. Poole. An idiom-finding tool for increasing productivity of accelerators. In *ICS*, 2011.

[8] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *PPoPP*, 2010.

[9] J. W. Davidson and S. Jinturkar. Improving instruction-level parallelism by loop unrolling and dynamic memory disambiguation. In *MICRO 28*, 1995.

[10] D.Unat, X.Cai, and S. Baden. Mint: Realizing CUDA performance in 3D Stencil Methods with Annotated C. In *ICS*, 2011.

[11] H. Gahvari, A. H. Baker, M. Schulz, U. M. Yang, K. E. Jordan, and W. Gropp. Modeling the performance of an algebraic multigrid cycle on HPC platforms. In *ICS*, 2011.

[12] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Trans. Parallel Distrib. Syst.*, 2, 1991.

[13] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ISCA*, 2009.

[14] S. Hong and H. Kim. Memory-level and thread-level parallelism aware GPU architecture performance analytical model. In *GIT ECE Technical Report TR-2009-003*, 2009.

[15] W. Huang, M. R. Stan, K. Skadron, S. Ghosh, K. Sankaranarayanan, and S. Velusamy. Compact thermal modeling for temperature-aware design. In *DAC'04*, 2004.

[16] W. Jalby and U. Meier. Optimizing matrix operations on a parallel multiprocessor with a hierarchical memory system. 1986.

[17] M. H. Kalos, M. A. Lee, P. A. Whitlock, and G. V. Chester. Modern potentials and the properties of condensed ^{4}He. In *Phys. Rev. C 66, 044310-1:14*, 1981.

[18] Khronos Group Std. The OpenCL Specification, Version 1.0. http://www.khronos.org/registry/cl/specs/opencl-1.0.33.pdf,
2009.

[19] David Kirk and Wen mei Hwu. *Programming Massively Parallel Processors: A Hands-on Approach.* Morgan Kaufmann, 1 edition, February 2010.

[20] A. Klckner, N. Pinto, Y. Lee, B. C. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA: GPU run-time code generation for high-performance computing. *CoRR*, 2009.

[21] K. Kothapalli, R. Mukherjee, M. S. Rehman, S. Patidar, P. J. Narayanan, and K. Srinathan. A performance prediction model for the CUDA GPGPU platform. In *HiPC*, 2009.

[22] B. C. Lee and D. M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *ASPLOS-XII*, 2006.

[23] B. C. Lee, J. Collins, H. Wang, and D. Brooks. CPR: Composable performance regression for scalable multiprocessor models. In *MICRO*, 2008.

[24] Benjamin C. Lee, David M. Brooks, Bronis R. de Supinski, Martin Schulz, Karan Singh, and Sally A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *PPoPP*, 2007.

[25] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP programming and tuning for GPUs. In *SC*, 2010.

[26] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ISCA*, 2010.

[27] J. Meng, J. W. Sheaffer, and K. Skadron. Exploiting inter-thread temporal locality for chip multithreading. In *IPDPS*, page 117, 2010.

[28] J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *ICS*, 2009.

[29] NVIDIA Corporation. NVIDIA CUDA compute unified device architecture programming guide. http://developer.download.nvidia.com/compute/cuda/08/NVIDIA_CUDA_Programming_Guide_0.8.pdf, 2007.

[30] J. A. Pienaar, A. Raghunathan, and S. Chakradhar. MDR: performance model driven runtime for heterogeneous parallel platforms. In *ICS*, 2011.

[31] S. C. Pieper, K. Varga, and R. B. Wiringa. Quantum Monte Carlo calculations of A=9,10 nuclei. In *Phys. Rev. C 66, 044310-1:14*, 2002.

[32] S. C. Pieper and R. B. Wiringa. Quantum Monte Carlo calculations of light nuclei. In *Annu. Rev. Nucl. Part. Sci. 51, 53*, 2001.

[33] M. Queva. Phase-ordering in optimizing compilers. Master's thesis, 2007.

[34] J. Ramanujam. Tiling of iteration spaces for multicomputers. In *ICPP*, pages 179–186, 1990.

[35] J. Ramanujam. Tiling multidimensional iteration spaces for nonshared memory machines. In *SC*, pages 111–120, 1991.

[36] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for performance modeling and prediction. In *SC*, 2002.

[37] S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W. m. W. Hwu. CUDA-Lite: Reducing GPU programming complexity. In *LCPC*, 2008.

[38] V. Volkov. Better performance at lower occupancy. Presentation at NVIDIA GTC, 2010.

[39] M. Wolfe. Implementing the PGI accelerator model. In *GPGPU*, 2010.

[40] L. T. Yang, X. Ma, and F. Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *SC*, 2005.

[41] Y. Zhang and J. D. Owens. A quantitative performance analysis model for GPU architectures. In *HPCA*, 2011.