Characterizing and Improving the Use of Demand-Fetched Caches in GPUs

Wenhao Jia
Princeton University
wjia@princeton.edu

Kelly A. Shaw
University of Richmond
kshaw@richmond.edu

Margaret Martonosi
Princeton University
mrm@princeton.edu

ABSTRACT

Initially introduced as special-purpose accelerators for games and graphics code, graphics processing units (GPUs) have emerged as widely-used high-performance parallel computing platforms. GPUs traditionally provided only software-managed local memories (or scratchpads) instead of demand-fetched caches. Increasingly, however, GPUs are being used in broader application domains where memory access patterns are both harder to analyze and harder to manage in software-controlled caches. In response, GPU vendors have included sizable demand-fetched caches in recent chip designs. Nonetheless, several problems remain. First, since these hardware caches are quite new and highly-configurable, it can be difficult to know when and how to use them; they sometimes degrade performance instead of improving it. Second, since GPU programming is quite distinct from general-purpose programming, application programmers do not yet have solid intuition about which memory reference patterns are amenable to demand-fetched caches.

In response, this paper characterizes application performance on GPUs with caches and provides a taxonomy for reasoning about different types of access patterns and locality. Based on this taxonomy, we present an algorithm which can be automated and applied at compile-time to identify an application's memory access patterns and to use that information to intelligently configure cache usage to improve application performance. Experiments on real GPU systems show that our algorithm reliably predicts when GPU caches will help or hurt performance. Compared to always passively turning caches on, our method can increase the average benefit of caches from 5.8% to 18.0% for applications that have significant performance sensitivity to caching.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures

ICS'12, June 25–29, 2012, San Servolo Island, Venice, Italy.
Copyright 2012 ACM 978-1-4503-1316-2/12/06 ...$10.00.

General Terms

Performance, Measurement, Algorithms

Keywords

GPU cache, CUDA, GPGPU, compiler optimization

1. INTRODUCTION

Graphics processing units (GPUs) were introduced to efficiently handle computer graphics workloads in specialized applications such as image rendering and games. In most early GPUs, software-managed local memories (or scratchpads) instead of traditional demand-fetched caches were provided for two reasons. First, traditional GPU workloads involved large amounts of streaming and were deemed difficult to cache [7]. Second, graphics/game programmers were willing to meticulously orchestrate memory access behavior and hand-tune use of software-managed caches [6].

More recently, GPU software has broadened and often includes memory access patterns that are both hard to analyze and hard to manage in software-controlled caches. In response, GPU vendors have included demand-fetched caches in their recent chip designs. For instance, the NVIDIA Fermi models introduced a relatively large (up to 48 KB per core) and configurable L1 cache not previously seen on GPUs [12]. Likewise, AMD's Fusion GPUs also offer a 16 KB L1 cache per core [5]. Both vendors' recent GPUs have sizable, globally coherent L2 caches.

Although the introduction of these seemingly intuitive demand-fetched caches is appealing, several problems remain. First, since these hardware caches are quite new and highly configurable, reasoning about when they will improve or harm application performance is still difficult. Second, since GPU programming is quite distinct from general-purpose programming, application programmers do not yet have widespread intuition about which memory reference patterns in their code are most amenable to demand-fetched caches and which should be optimized in other ways.

Defying conventional wisdom about caches built up over years of their use on general-purpose CPUs, our experiments show that caches are not always helpful to GPU applications, and in many cases they may even hurt performance. For a suite of 12 general-purpose GPU (GPGPU) programs on an NVIDIA Tesla C2070 GPU with L1 caches turned on and off, we find that only 3 of them see substantial performance improvements from caching. Meanwhile, another 3 of them actually see non-trivial performance degradations. This lack of performance predictability is so well-known that it is reflected in GPU vendor directions to application programmers—NVIDIA's programming manual specifically suggests that software developers experiment with cached and uncached versions of their code to see which one works better [14]. Such ad hoc approaches are clearly unappealing because they are time-consuming and error-prone, and they do not work well for software that must run well across many data set sizes and GPUs.

This paper provides methods for efficiently utilizing the newly provided demand-fetched GPU caches despite their seemingly difficult-to-predict payoff. In particular, we start by characterizing GPU application performance on a real GPU system with L1 caches turned on and off. Our results show the degree to which L1 caches may either improve or hurt program performance. Then, based on observations about GPU program access patterns, we provide a taxonomy of GPU memory access locality that can help programmers reason about when caches are likely to be helpful. We also note that since GPUs typically have extensive latency-tolerance mechanisms using multi-threading, it is memory bandwidth rather than latency that often determines cache utility. From our measurements and taxonomy, we develop methods that enable automated compile-time optimizations to determine when to use (or *not* to use) L1 caches in GPUs. Overall, this paper makes the following contributions.

First, we present a systematic taxonomy of GPU memory access patterns. Inspired by the "three C's" of general-purpose CPU caches, the taxonomy helps GPU compiler writers and application developers build intuition about the best ways to improve their application's memory access performance on cache-enabled GPUs.

Noting that memory bandwidth, more so than latency, is an important determinant of cache utility, our work demonstrates compile-time methods to analyze GPU programs and calculate "estimated memory traffic". Our estimates serve as indicators of when caches are likely to be beneficial.

Third, we propose a compile-time algorithm which automatically uses the traffic estimates to predict how caches will affect applications' performance and controls caching accordingly. Our method increases the benefit of using L1 caches from 5.8% to 18.0% for applications that experience significant changes in performance when caches are used.

The remainder of the paper is structured as follows. Section 2 gives background information on GPU hardware and software. Section 3 details our measurement methodology. In Section 4, we present real-system GPU cache characterizations and use those results to develop a GPU memory access locality taxonomy. Based on this taxonomy, Section 5 develops an automatable compile-time algorithm for determining whether/when to employ L1 caching on each global memory load of a GPU program. Section 6 gives experimental results of applying this algorithm. Section 7 discusses related work, and Section 8 concludes the paper.

2. BACKGROUND

This section gives background details on GPU architectures. For consistency, we primarily use NVIDIA and CUDA terminology in this paper. Our techniques are, however, applicable to a broader range of GPUs from different vendors.

2.1 GPU Hardware Characteristics

A modern GPU is a many-core processor optimized for throughput computing. Each core, or Streaming Multiprocessor (SM) in NVIDIA terminology, is capable of executing a large number of threads in parallel. SMs have internal memory units, including registers private to each thread and software-managed local memories (shared memory) accessed by all threads running on the same SM. Figure 1 shows a typical GPU memory hierarchy as well as possible data flow paths among these components. A GPU program consists of many thread blocks. A thread block is always scheduled to run on a single SM so that threads in that block can cooperate through the use of shared memory in that SM. When threads access data in the large off-chip DRAM called global memory, their accesses go through a two-level cache hierarchy. Each L1 cache is private to its SM. L1 caches in different SMs are *not coherent* with each other. The L2 cache is shared by and coherent across all SMs on the chip.

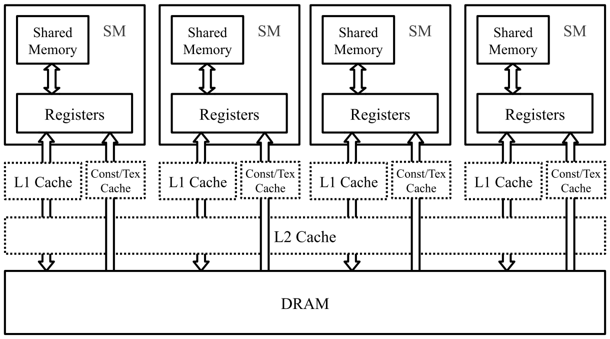

Figure 1: A typical GPU memory hierarchy. Arrows indicate data-flow paths.

Current NVIDIA GPUs have configurable L1 caches whose size can be either 16 KB or 48 KB. The L1 cache in each SM shares the same total 64 KB of memory cells with shared memory, giving users dynamically configurable choices regarding how much storage to devote to the cache versus the shared memory. AMD GPU L1 caches have a fixed size of 16 KB. The NVIDIA Tesla GPU we use has a non-configurable 768 KB L2 cache, while L2 caches on AMD GPUs have a size of 64 KB per core. Finally, a GPU has constant and texture caches; these are separate from the two-level demand-fetched caches and are only accessed through special constant and texture instructions.

Because L1 caches are not coherent across SMs, current NVIDIA GPUs treat every global memory store as a write-through transaction (write through L1 caches to the L2 cache) followed by an invalidate transaction (invalidate the copy in the L1 cache). In other words, global memory stores ignore L1 caches and this behavior is not configurable at the hardware level. For that reason, in this paper we only study global memory loads—not stores—in GPU programs.

2.2 GPU Software Characteristics

Each GPU program has one or more kernel functions that are launched and executed on the GPU. Each kernel divides its work into identically sized thread blocks. The size of a thread block is the number of software threads in that block. From a programmer's perspective, each instruction in the kernel is executed by all threads in the same thread block concurrently. However, on the real hardware, because of the limited number of SIMD units, software threads are actually executed in groups of threads called warps. A warp has 32 threads on current NVIDIA GPUs, and the SIMD units execute one warp at a time, switching to other warps when long-latency instructions are encountered. Because threads within the same warp execute instructions concurrently on the hardware, their global memory accesses are "coalesced" into the smallest number of unique memory requests. Memory accesses from different warps do not get coalesced, so they may result in redundant memory traffic if the memory fetches are not cached. Finally, each thread executes exactly the same binary instructions as other threads in the program, so it has to use thread ID and thread block ID variables to find out its own identity and operate on its individual data accordingly. Thread ID and block ID variables can be one- or multi-dimensional, but they must take consecutive values in their respective ranges.

GPU programmers are usually encouraged to declare and use shared memory variables as much as possible, because the on-chip shared memories have much shorter latency and much higher bandwidth than the off-chip global memory. Small, repeatedly-used data are good candidates to be declared as shared memory objects. However, because shared memories have limited capacity and an SM cannot access another SM's shared memory, many programs cannot use shared memories due to the algorithmic characteristics of these programs. In these situations, the global memory is always a bigger, slower fallback choice.

2.3 An Example GPU Program: BFS

As an example, Figure 2 shows the CUDA version of the breadth-first search (BFS) application taken from the Rodinia benchmark suite [3]. We revisit BFS later in the paper to explain our work. This program does breadth-first graph traversal to search for a specified item. More details about this program can be found in [9], but here we give a brief summary of its execution flow.

When the program runs, two kernels are called repeatedly in a cyclic fashion. Both kernels have a thread block size of 512 threads. The first kernel expands the search frontier to the next node level of the graph, and the second kernel does the actual visit (`visit()`) and then sets up conditions for the next run of the first kernel. The program terminates when a certain number of calls are performed, corresponding to the total number of node levels in the graph. The arrays `now[]`, `visited[]`, and `next[]` are used for tracking node visit status, and they are stored in global memory. The other global memory array `children[]` stores the child node IDs of each node consecutively. In this example, each node of the graph has a fixed number of 8 children.

```
#define TRUE 1
#define FALSE 0
kernel( int now[], int visited[], int next[],
        int children[] ) {
  int nodeID = blockIdx.x * 512 + threadIdx.x;
  if ( now[nodeID] == TRUE ) {
    now[nodeID] = FALSE;
    for ( int i = 0; i < 8; i++ ) {
      int childID = children[nodeID * 8 + i];
      if ( visited[childID] == FALSE )
        next[childID] = TRUE;
    }
  }
}
kernel2( int next[], int now[] ) {
  int nodeID = blockIdx.x * 512 + threadIdx.x;
  if ( next[nodeID] == TRUE ) {
    next[nodeID] = FALSE;
    now[nodeID] = TRUE;
    visit(nodeID);
    visited[nodeID] = TRUE;
  }
}
```

Figure 2: An example breadth-first search GPU program. The global load instructions are underlined.

Parameters	Values
CPU model	Intel Core 2 Duo E8500
System DRAM capacity	4 GB
GPU model	NVIDIA Tesla C2070
SM count	14
SM SIMD width	32
SM clock rate	1.15 GHz
L1 cache size	16 KB or 48 KB
Shared memory size	48 KB or 16 KB
L2 cache size	768 KB
GPU DRAM capacity	6 GB
GPU DRAM clock rate	1.5 GHz
GPU DRAM bandwidth	144 GB/sec
Single-/Double-precision floating point performance	1030/515 GFLOPS
CUDA version	4.0

Table 1: Our CPU/GPU measurement platform.

2.4 Why Study GPU Caches?

Since GPU caches have considerable configurability, multiple decisions must be made to determine how to use them. For example, NVIDIA programmers can select whether or not to turn on caching depending on overall data access patterns. Additionally, they can select the size of the L1 cache when it is used. This configurability gives programmers flexibility but also introduces performance uncertainties.

GPU programs usually have hundreds of threads running concurrently on the same processor core. The cache capacity per thread is very small, despite the relatively large overall cache size. This makes it difficult to fit the entire working set into caches. Such capacity constraints challenge the conventional wisdom gained from CPU caches. To get the most benefit from limited per-thread cache capacity, programmers must determine which portions of the working set will benefit from caching and only cache these portions (Section 3.1).

Because experimentally choosing the desired cache configuration and deciding which instructions should load data into the cache has many drawbacks, our work seeks to build better programmer intuition as well as to help automate cache configuration and data caching decisions.

3. METHODOLOGY

3.1 Real-System Measurement Infrastructure

All of our experiments are performed using a real running system, with an NVIDIA Tesla C2070 GPU and an Intel Core 2 Duo E8500 CPU. Table 1 lists the details.

The L1 cache in this GPU is controllable in several ways. First, it shares storage cells with shared memory, and we can configure the partitioning to select the size of each unit. Specifically, one can have a 16 KB L1 cache with a 48 KB shared memory, or one can have a 48 KB shared memory with a 16 KB L1 cache. The second way to control the L1 cache is to turn it on or off for an entire program. We refer to this as "cache all" or "cache none" respectively. This is done

Kernels	Thread block sizes	Shared memory use per block
backprop-1	16 × 16	0
backprop-2	16 × 16	1088 B
bfs-1	512	0
bfs-2	512	0
cfd-1	192	0
cfd-2	192	0
cfd-3	192	0
cfd-4	192	0
heartwall	512	11872 B
hotspot	16 × 16	3072 B
kmeans-1	256	0
kmeans-2	256	0
leukocyte-1	320	14568 B
leukocyte-2	175	0
leukocyte-3	176	0
lud-1	32	3072 B
lud-2	16	1024 B
lud-3	16 × 16	2048 B
nw-1	16	2180 B
nw-2	16	2180 B
particlefilter	128	0
srad-1	512	4096 B
srad-2	512	0
srad-3	512	0
srad-4	512	0
srad-5	16 × 16	6144 B
srad-6	16 × 16	6144 B
streamcluster	512	1024 B

Table 2: Rodinia programs and kernel attributes.

by passing the compiler flags `-dlcm=ca` or `-dlcm=cg` to the Parallel Thread Execution (PTX) assembler. (PTX is the CUDA ISA [13].) Finally, one can also select whether each individual global load instruction's memory access should go through the L1 cache or not, by using the global memory access modifiers available in PTX 2.x.

3.2 Applications

Our experiments use the CUDA version of the applications in the Rodinia 1.0 benchmark suite [3]. Rodinia contains a diverse set of real-world GPU programs. Each program in the suite has one or more GPU kernels, and our methods analyze each kernel independently. (Cross-kernel optimizations are rare in GPUs and could be a topic for future work.) Table 2 lists the programs and some per-kernel characteristics; different kernels in an application are distinguished by numbering. All measured numbers are hardware performance counter values collected with NVIDIA Visual Profiler. All reported kernel runtimes are the time between kernel launch and finish, excluding CPU work and CPU-GPU communication time, which are not the focus of this study. Because programs have nearly identical execution times from one run to another (less than 1% variance), we present results from a single program run. Within a single run, a particular kernel may be invoked several times; when this occurs, we present the runtime as total time accumulated over all launches of this kernel in the entire run.

4. CHARACTERIZING GPU CACHE USE

This section experimentally characterizes when L1 caches help and hurt performance of a suite of GPGPU kernels. From this, we determine cache prediction metrics, and we construct a model that helps programmers understand how different types of memory accesses interact with GPU caches.

4.1 L1 Cache Performance Impact

To measure the impact of L1 caches on GPU application performance, we run the entire Rodinia benchmark suite on our platform with the L1 caches turned on and off. The suite's applications use the GPU memory system in widely varying ways. Some of the kernels load data only into the texture and constant caches; on the NVIDIA hardware, these data never enter the L1 demand-fetched cache. Other kernels use explicit software management to pull data from global memory into the software-managed shared memory. While such data pass through the L1 cache en route to shared memory, they are only accessed from shared memory from that point on. For a third group of kernels, accesses are made directly to global memory without making use of shared memory. In such cases, data are implicitly demand-fetched into the L1 cache through the global memory accesses; caching of these data is similar to how data are managed by CPU caches. It is the L1 hits to these data that offer the largest potential payoff in performance.

Figure 3 presents the speedup obtained when 16KB L1 caches are used relative to when they are not used. The bars in the graph are grouped to indicate how each kernel uses the memory system based on the three usage categories from the preceding paragraph. Values smaller than 1 indicate that an L1 cache offers speedup over the no-cache case.

Among the three groups of kernels, the performance of the first group of kernels is expected to be unaffected by whether L1 caches are present or not, because their memory accesses do not go through L1 caches. This expectation is confirmed by our results: almost no performance variation is seen.

The second group of kernels shows very small responses to the use of L1 caches. This is because the initialization phases of these kernels, during which accesses go through L1 caches, are usually much shorter than the computation phases of these kernels, during which accesses focus on the shared-memory and no longer go through L1 caches. Notably, the `srad` kernels show up to 10% performance variations caused by the use of L1 caches. This is because these kernels have fairly short computation phases, and even the initialization phases affect the total runtime.

The third group of kernels (solid bars in Figure 3) are of the most interest to us, because they should have the greatest potential benefit from caching—they all use global memory (rather than shared memory) to hold their working sets, and so they continue to have global memory accesses even during the main computation phases. Because these global memory accesses can use the L1 cache, these are the cases where a demand-fetched L1 cache can be most helpful. However, instead of seeing uniform benefits from caching, we see much more variable performance. In particular, when the cache is turned on, `backprop-1` and `particlefilter` show significant speedup while `bfs-1`, `cfd-1`, and `kmeans` show noticeable slowdown.

To investigate such varying cache effects, Figure 4 plots the L1 cache hit rates of these kernels. There is no clear relationship between performance changes and L1 hit rates. Two kernels with large performance variations (`kmeans-2` at 0.59× and `particlefilter` at 1.9×) do show the smallest (0.2%) and the largest (99%) L1 hit rates respectively. For the other programs, however, the hit rate cannot be directly related to performance. For example, `backprop-1` and `bfs-1` show opposing performance trends, though they have similar cache hit rates. In fact, while it might be appealing to

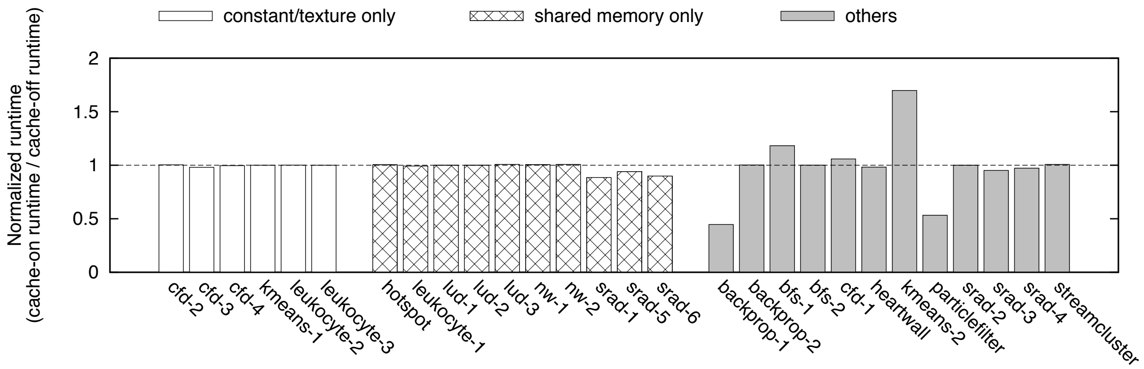

Figure 3: 16 KB L1 caches have little or unpredictable performance impact on the Rodinia benchmark suite. Lower bars indicate better performance (i.e. shorter runtime) when the L1 caches are turned on.

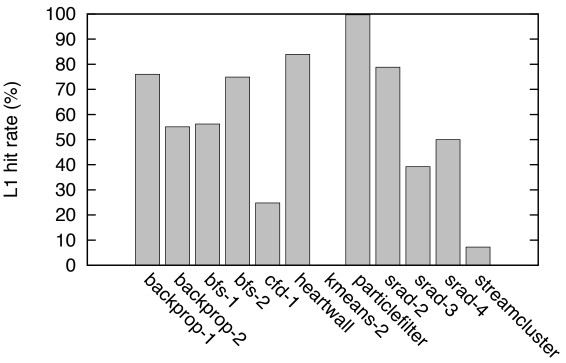

Figure 4: The L1 hit rate is not a good predictor of program performance variations.

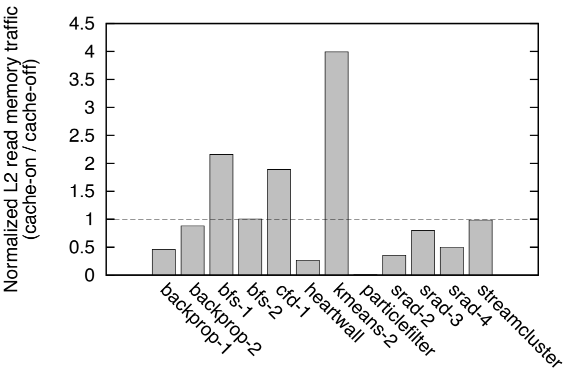

Figure 5: The amount of L2 read memory traffic better indicates cache impact on program runtime.

devise a caching policy based on a hit rate threshold—turn on caching if profiling shows a hit rate above the threshold and turn off caching otherwise—there is no reasonable threshold to choose for these results.

To find a better indicator of caching's performance influence, we look more closely at `bfs-1`. In fact, the performance slowdown stems from the 128 B L1 cache line size, which is quite long relative to the amount of spatial access locality in the kernel. The default input graph supplied with the benchmark causes threads in `bfs-1` to issue random, scattered 4 B accesses to main memory. When the L1 cache is off, each 4 B access is turned into a 32 B memory request and sent to the L2 cache and DRAM. However, when the L1 cache is on, each thread has to fetch an entire 128 B cache line for the same 4 B access. This quadruples the required L2-to-L1 traffic, and results in much higher DRAM-to-L2 traffic as well. However, much of this increased memory traffic goes unused because the graph's access locality is low. Since `bfs-1` is a memory bandwidth bound application, extra wasted memory traffic leads to a 18% performance slowdown in Figure 3. The observation made using `bfs-1` inspires the use of memory traffic as an estimator of caching's benefit.

Figure 5 shows the relative reduction in memory traffic from L2 to L1 when the L1 cache is turned on, compared to when the L1 cache is off, for the third group of kernels. This is obtained by multiplying the observed number of L2 read requests and the request size (32 B or 128 B). The graph shows that the amount of L2 read memory traffic is better correlated (than hit rate) with the L1 cache's performance impact. In kernels where L1 caches harm performance (`bfs-1`, `cfd-1`, and `kmeans-2`), the amount of memory traffic increases. The opposite is true for kernels that benefit from L1 caches (`backprop-1` and `particlefilter`). The rest of the programs are not memory bandwidth bound, and so their memory traffic changes are inconsequential.

Across the test suite, it is generally true that increased L2-to-L1 memory traffic results in equal or worse performance and reduced traffic results in equal or better performance. Using memory traffic as a metric for driving our caching policy algorithm is intuitive in several ways. First, if computation is identical whether caching is turned on or off, the difference in memory traffic can be expected to determine runtime changes. Second, at compile-time, aggregate traffic estimates for GPU programming models are easier to collect than cache hit rate estimates. Finally, while we are the first to apply traffic estimation to caching decisions, they have seen prior use for other GPU optimizations [1].

4.2 A Taxonomy of Memory Access Locality

CPU programmers have long used the three C's (compulsory, conflict, and capacity) taxonomy for cache misses to help them understand and reduce the cache miss rates of their programs [10]. Since L1 cache hit rates are not particularly predictive of a GPU cache's potential benefit, a new model is needed to help GPU programmers understand how global memory accesses interact with the cache and how they can modify their program to improve performance.

Our characterization experiments show that memory traffic can be indicative of how the L1 cache impacts performance. The new model therefore needs to translate how different memory access patterns result in different amounts of memory traffic and how those levels of traffic change with the use of caches. Because of the complex execution model used to program and execute work on GPUs (e.g. threads, warps, and blocks), this model must also relate memory traffic to how computation is scheduled.

Instead of using different types of cache misses, our taxonomy for GPUs introduces three types of locality.

- *Within-warp data locality* applies to a single load instruction being executed by threads within the same warp. If these threads access data mapped to the same cache line, they are said to have within-warp locality.
- *Within-block (cross-warp) data locality* applies to a single load instruction being executed by threads within the same thread block but in *different* warps. If these threads access data mapped to the same cache line, they are said to have within-block locality.
- *Cross-instruction data reuse* applies to more than one instruction (including the same instruction being executed more than once) being executed by the same or different threads within the same thread block. If these threads access data mapped to the same cache line, they are said to have cross-instruction data reuse.

To understand each type of locality, recall how threads are organized into warps. Threads in the same block have consecutive IDs. Warps are formed as groups of 32 threads with consecutive IDs. For example, if `tid` is the thread ID variable, the threads in the first warp of a block will have `tid` from 0 to 31, the second warp will be 32 to 63, etc.

Within-warp locality is frequently due to threads in a warp (32 threads per warp) sharing contiguous elements of an array (e.g. `array[tid]`), resulting in coalesced accesses to the same cache line. *Within-block locality* results from threads having IDs with a distance greater than 32 accessing the same cache line (e.g. `array[tid % 32 * 32]`. The critical difference between within-warp locality and within-block locality is that, without an L1 cache, two identical requests from the same warp (bearing within-warp locality) generate only one L2 request while two identical requests from the same block but different warps (bearing within-block locality) generate two L2 requests. Turning caches on may exploit within-block locality by reducing the number of L2 requests from two to one, but it cannot exploit within-warp locality because the number of L2 requests is already at its minimum due to coalescing. Hence within-warp locality does not benefit from caches while within-block locality does.

Cross-instruction reuse always exists between multiple instructions executing at different moments in time, similar to temporal locality in CPUs. However, due to the small per-thread cache capacity, instructions with cross-instruction reuse have a much smaller chance of retaining their data in the cache. Consequently, it is difficult to exploit cross-instruction reuse through caching on GPUs.

Finally, if an instruction executed by a whole thread block exhibits none of the locality above, it is said to have *non-locality.* Instructions with non-locality should not use the cache, because turning off the cache for them results in smaller (32 B) request sizes than turning on the cache would (128 B). The random accesses in BFS are examples of such non-locality; this program benefits from the lower overall bandwidth of non-cached operation.

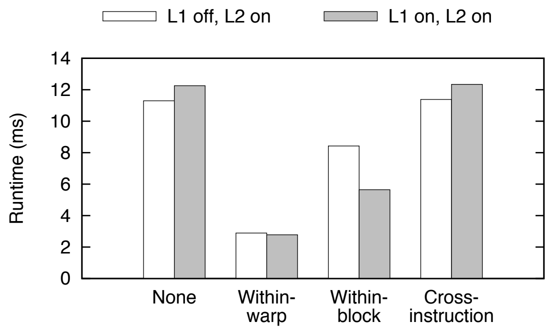

Figure 6: For BFS, within-warp locality improves program performance regardless of whether the L1 cache is turned on. Caches are helpful to within-block locality. The L1 cache is too small for cross-instruction reuse.

4.3 Taxonomy Example: BFS

Below we use the BFS example (Section 2.3) to quantitatively demonstrate the caching benefits for each type of locality. Experimental results confirm our analysis of locality and motivate our caching strategy in Section 5.4.

We use different data layouts in a special BFS input set to vary the types of memory access locality and test their amenability to caching. The special input graph is a carefully designed directed octree. Its graph structure is manipulated to contain *one and only one* type of locality at a time. For example, to produce cross-instruction reuse, a thread is directed by `children[]` to access 8 adjacent elements in `visited[]` over 8 iterations of a loop. However, different threads in the same thread block (including those in the same warp) access random 8-element groups to make sure no within-warp or within-block locality is present. Likewise, graphs with *only* within-warp or *only* within-block locality can be constructed, while the degree of locality is kept at 8 to ensure *fair* comparisons across different types of locality. For within-warp locality, this means every 8 threads in the same warp access adjacent elements in `visited[]`; for within-block locality, this means 8 threads from 8 distinct blocks access adjacent elements in `visited[]`. Finally, a randomly structured tree is used as the non-locality input.

As shown in Figure 6, when BFS runs on an input with no locality, the use of an L1 cache slightly hurts its performance (8.5% slowdown) due to the long cache line size resulting in an almost 2× increase in L2 traffic. The same program running on an input having within-warp locality sees greatly improved performance—3.9× in the cache-off case caused by a 3.6× traffic reduction, compared to the non-locality input. Turning the L1 cache on, however, has a negligible effect on performance, since within-warp locality uses coalescing instead of caching to reduce memory traffic. For the third input (within-block locality), the L1 cache successfully exploits the locality (2.2× runtime reduction due to 3.1× traffic reduction, relative to non-locality case). Even when the L1 cache is off, the always-on L2 cache can still exploit the locality to a lesser degree (1.3× runtime reduction). Lastly, caches cannot exploit the cross-instruction reuse even if it exists in the input—a thread's cache lines are evicted from the cache before they get reused. These experiments reinforce our analysis of locality and show, in a real application, the relative performance benefits of caching for each type of locality.

5. A COMPILE-TIME ALGORITHM FOR IMPROVING GPU CACHE USE

This section presents an algorithm that performs compile-time analysis of GPU memory access patterns and accurately predicts whether caching would be useful to each load instruction in a program. Even though here we apply our algorithm to the kernels manually, it can be easily carried out automatically by a compiler. The algorithm is designed using the following abstract GPU execution model.

First, we assume all threads in a thread block execute an instruction simultaneously, before they move on to the next instruction together. Even though in real hardware, threads execute instructions in units of warps instead of blocks, and a warp may execute a few instructions consecutively before it stalls and gets swapped out by another warp, the simpler model without warps is intuitive and accurate enough for our cache analysis and optimization.

Second, our model considers each SIMD load instruction in isolation from other instructions, as if the cache is flushed before and after each instruction is executed by all threads in a block. (This assumption has no effect on analyzing within-warp locality, within-block locality, and scattered accesses, because they all exist within the scope of a single instruction.) Thus, our algorithm does not directly aim to exploit cross-instruction data reuse, but the final step may optionally use some heuristics to try to cache for it (Section 5.4).

Finally, minor effects such as cache line mapping conflicts are ignored; our results show this is generally acceptable.

5.1 Algorithm Overview

For a given GPU program kernel, our algorithm first discovers the memory access patterns of each individual SIMD load instruction. It achieves this by representing the data address of each load instruction as a function of the thread ID variables. Based on that information, the algorithm then estimates how much memory content will be transferred when the instruction is executed by an entire thread block, with the cache on and off. Based on these traffic estimates, the algorithm decides whether the data of each load instruction should be cached or not.

The algorithm has 3 steps: analyze access patterns, estimate memory traffic, and determine which instructions should use the cache. They can be applied to both high-level (such as the pseudocode below) and low-level (such as PTX code in our experiments) GPU program representations.

5.2 Step 1: Analyze Access Patterns

Our access pattern analysis takes advantage of the fact that GPU threads rely on their thread IDs and block IDs to identify and load their own data. Consequently, when a thread executes a load instruction, the data address that instruction loads is usually a function of its thread ID, its block ID, and some other values that are the same for all threads in the entire thread block (such as kernel input arguments). Thus, one can analyze a particular single thread's behavior and extrapolate the overall kernel behavior.

Values passed by heap (e.g. input data stored in global memory) may cause uncertainties. However, these values have to be explicitly loaded, and hence they are identifiable and can be marked as unanalyzable. We find that these values often lead to either scattered accesses (such as looking up children nodes in `bfs`) or instances of cross-instruction reuse (such as iterating over elements of multi-dimensional input points in `k-means`), and our algorithm does not cache them (Section 5.4). `cfd-1` is the only exception in which unanalyzable accesses have caused problems (Section 6).

To simplify the analysis, we assume the block ID variable is 0, meaning we use the first thread block's access patterns to estimate access patterns of all other thread blocks. Future work can look into using statistical sampling to estimate the average thread block behavior.

The analysis step is shown in Algorithm 1. For brevity, it uses a one-dimensional thread block to explain the execution flow. Multi-dimensional thread blocks can easily be handled by using multiple thread ID variables. In the pseudocode, there are three special notations. τ is the thread ID variable itself, and $f(\tau)$ is a computable function of τ which may be a combination of τ, constants and basic arithmetic operations (`+`, `-`, $\times$, `/`, `%` and their combinations in our experiments). The symbol ϕ represents any unprocessed variable. The symbol ∞ represents an unknown value. Similar to the "Not a Number (`NaN`)" data value, an instruction with at least one ∞ as its input operand produces a result of ∞. In this algorithm, an ∞ is an unanalyzable variable which blocks the compiler from analyzing values produced from that ∞. These unknown values are usually caused by loading content from the heap.

Algorithm 1 Analyze the memory access patterns of global load instructions in a kernel

```
initialize all variables in the kernel to $\phi$
delete all loop back edges and set loop induction variables to 0
set kernel input arguments to user-supplied sample values
set results of all memory load instructions to $\infty$
set thread ID variables to $\tau$
repeat
  for all instructions in the program do
    if all source operands of an instruction have known values:
    constant, $f(\tau)$, or $\infty$ then
      define the result of this instruction by executing the
      arithmetic operation represented by this instruction
      propagate this definition to all of its uses in instructions
      dominated by this instruction
    end if
  end for
until no new propagation or definition happens
mark all $\phi$ variable as $\infty$
output the data addresses of all load instructions
```

This algorithm is iterative like common constant folding and propagation compiler optimizations. It sets "seeds" (loop induction variables, kernel input arguments, thread ID variables and memory load results) to known values in the initialization phase and then uses data-flow analysis to fold and propagate these known values. The deletion of loop back edges in Algorithm 1 lets us analyze inside loops at the cost of using the first iteration to estimate the overall loop behavior. This is generally not a problem because loops usually correspond to cross-instruction reuse which we do not aim to analyze. When the algorithm has converged and terminated, it outputs data addresses of computable load instructions and marks the rest as unknown (∞).

5.3 Step 2: Estimate Memory Traffic

Based on Step 1's analysis output, Step 2 estimates the amount of memory traffic resulting from these memory access patterns for both cache-on and cache-off cases.

Algorithm 2 shows the process of estimating the cache-on memory traffic. It makes use of the fact that when a sufficiently large cache is on, no content would be loaded repeatedly. So the memory traffic is equal to the number of unique bytes the instruction needs.

Algorithm 2 Estimate the amount of memory traffic generated by threads in one block when cache is on

```
set M to 128, the memory request size when cache is on
for all global load instructions in the kernel do
    let N = the number of threads in each block
    declare tags[N] which stores the upper bits of the memory
    addresses threads load when they execute this load
    for i = 0; i < N; i++ do
        use the data address expression to compute which memory
        address thread i is loading and assign that address to addr
        tags[i] = addr / M
    end for
    count the total number of unique values in tags[] and store
    it in U, counting each $\infty$ as one unique value
    U is the total number of unique memory requests of this load
    output U $\times$ M, the total amount of memory traffic generated
    by all threads executing this load instruction when the cache
    is turned on for it
end for
```

Algorithm 3 estimates the cache-off memory traffic. It first calculates the amount of traffic generated by each warp, and then it simply adds these amounts together because there is no caching to prevent loading redundant data.

Algorithm 3 Estimate the amount of memory traffic generated by threads in one block when cache is off

```
set M to 32, the memory request size when cache is off
set B to the thread block size
for all global load instructions in the kernel do
    set W to B / 32, the number of warps in the block
    set nreqs, the total number of memory requests sent by all
    warps in the block, to 0
    for all W warps in this thread block do
        declare tags[32] which is used to store the upper bits of
        the memory addresses threads of this warp load when they
        execute this load instruction
        for i = 0; i < 32; i++ do
            use the data address expression to compute which mem-
            ory address thread i is loading and assign it to addr
            tags[i] = addr / M
        end for
        count the total number of unique values in tags[] and
        store it in n, counting each $\infty$ as one unique value
        nreqs = nreqs + n
    end for
    output nreqs $\times$ M, the total amount of memory traffic gen-
    erated by all threads executing this load instruction when
    the cache is turned off for it
end for
```

As an example, Table 3 lists the access patterns, locality types, and estimated cache-on and cache-off traffic for a few typical instructions found in the Rodinia benchmarks.

5.4 Step 3: Determine Cache Use

The final step of the algorithm uses the following strategy to decide whether each instruction should use the cache or not, depending on the estimated memory traffic and access patterns of this instruction.

1. *If the cache-on memory traffic is equal to the cache-off memory traffic of this instruction:* This occurs because this instruction only has within-warp locality and has no within-block locality. If the cache-on memory traffic is larger than cache capacity, we do not let this instruction use caches. If the cache-on memory traffic is smaller than cache capacity, we have two choices. The conservative strategy does not turn on caches, reasoning that within-warp locality does not need caching and caching may accidentally evict useful cross-instruction reuse of other instructions. The aggressive strategy turns on caches, hoping to exploit cross-instruction reuse which may exist between this and some other instructions. Section 6 evaluates both strategies.

2. *If the cache-on memory traffic is smaller than the cache-off memory traffic:* This occurs because this instruction has some amount of within-block locality. Note that it may have an arbitrary amount of within-warp locality at the same time. If the cache-on traffic is smaller than cache capacity, we turn on caching for this instruction; otherwise we turn off caching to prevent cache thrashing.

3. *If the cache-on memory traffic is larger than the cache-off memory traffic:* We have successfully identified a scattered access. We turn caching off for this instruction.

4. *If an instruction has unknown (∞) memory access data addresses:* We assume that it is likely to issue scattered or cross-instruction reuse accesses (Section 5.2) and we do not use caching for this instruction.

Our algorithm does not try to identify cross-instruction data reuse, though it does have an optional heuristic which aggressively tries to cache for cross-instruction reuse existing across instructions having within-warp locality. A complete analysis of cross-instruction reuse must account for how each instruction affects other instructions in relation to their relative positions in the program's control flow graph and their respective required cache sizes. This is beyond the scope of this paper but is certainly of value for future work.

Table 3 lists the cache-on and cache-off traffic for each instruction. Their caching decisions can be easily computed based on the rules above.

6. EXPERIMENTAL RESULTS

In Figure 3, 5 kernels show unpredictable performance influence from caches and need an intelligent cache management algorithm. We apply our analysis algorithms to these kernels and the results are shown in Figure 7. Both 16 KB and 48 KB L1 cache sizes are tested. Results *less than 1* indicate performance improvements from caching.

Compared to not having caches (white bars in Figure 7), simply keeping caches on for all kernels (cross-hatched bars) results in only 5.8% performance improvement on average, due to the harmful effects caches have on some kernels. Our conservative strategy (gray bars), which keeps within-block locality cached and scattered accesses uncached, is able to identify and modify these harmed kernels. Consequently, the average benefit of caching increases to 16.9%—a 2.9× boost. Further caching within-warp locality with the aggressive approach (black bars) offers an 18.0% speedup. This is consistent with our analysis that within-warp locality does not benefit much from caching, even though `backprop-1` and `bfs-1` still show clear improvements (5.5% on average).

As a matter of fact, for most kernels, our method can reliably achieve performance close to the better result of the two fixed choices—cache-all and cache-none—without the need for pre-run profiling. In addition, our method actually outperforms both choices on `bfs-1`, because `bfs-1` has a mix

Kernels	Instructions	Access patterns	Types of locality	Warp count	Cache-on traffic	Cache-off traffic
`bfs-1`	`now[nodeID]`	τ	within-warp	16	2 KB	2 KB
	`children[nodeID * 4]`	4τ	within-warp	16	8 KB	8 KB
	`visited[childID]`	∞	none	16	64 KB	16 KB
`backprop-1`	`delta[index_x]`	τ % 16	within-warp, within-block	8	128 B	512 B
	`ly[index_y]`	$\tau/16$	within-warp, within-block	8	128 B	256 B
`kmeans-2`	`input[point_id * nfeatures]`	34τ	none	8	32 KB	8 KB

Table 3: Example instructions and their locality. τ is the thread ID variable. ∞ is an unknown access. The warp count is the number of warps in a thread block. All array elements are 4-byte types (int or float).

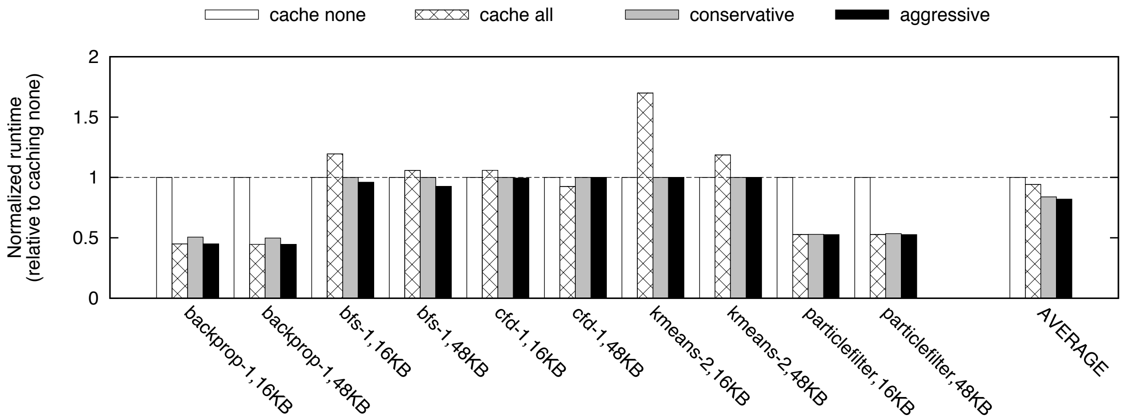

Figure 7: Normalized runtime of different caching approaches. Results less than 1 indicate caching is helpful. Caching all data improves performance by 5.8%. Our analysis algorithms improve performance much more: 16.9% or 18.0%. Both 16 KB and 48 KB L1 caches are tested.

of load instructions that need caching and load instructions that do not need caching. A kernel-wide decision for caching or not always helps one type of instruction but hurts the other, losing to the instruction-level decisions we use here.

Finally, `cfd-1` with a 48 KB L1 cache is the only case in which our method performs worse than the better result of the two fixed choices. This is because this kernel has some scattered accesses, along with some other cross-instruction reuse which can only be cached by a sufficiently large cache. With the 16 KB cache, the cache-all choice has worse performance caused by the scattered accesses, while the cross-instruction reuse cannot be cached by this smaller cache. With the 48 KB cache, the cache-all choice caches enough cross-instruction reuse that it actually overcomes the harm from scattered accesses and the overall performance comes out ahead. Our method is unable to analyze both access types and chooses not to use caches for unknown loads, and so its performance is the same as the cache-none choice with both cache sizes. Overall, we conclude these results by noting the degree to which our method can offer significant performance improvements without the need for profiling.

7. RELATED WORK

As GPUs have gained wider use, there has been research aiming to characterize and improve their memory hierarchies. Brook [2] and Merrimac [4] represent two early efforts at utilizing GPUs for general purpose computing. They both derive their methods based on GPUs which rely exclusively on software-managed on-chip storage. Hong and Kim build an analytical GPU architecture model with memory-level parallelism awareness, but their model does not consider demand-fetched caches [11]. Wong et al. use microbenchmarking techniques to discover and describe many undocumented features of NVIDIA GPUs, including memory structures and cache parameters [19].

A large body of work studies how to transform data layout to improve GPU memory system efficiency. Various compile-time and runtime algorithms are proposed to improve GPU program memory access patterns, aiming at different parts of the memory system including reducing irregular memory accesses [21], coalescing loop nest accesses [1], improving coalescing [20], and increasing memory controller parallelism [16]. However, none of this work targets caches. In addition, these studies usually change the programs themselves, while our work attempts to analyze memory behaviors of given programs. One study [17] provides a limited observation of GPU cache impact on a handful of simple kernels. In contrast, our work provides a more systematic characterization of GPU cache effectiveness and uses that to develop an algorithm for automating the choice of how and when to use demand-fetched caches.

Software-managed on-chip storage such as shared memory on NVIDIA GPUs has been present on a number of modern parallel processors [15]. In embedded systems, where it is called scratchpad memory, this type of storage is even more commonplace than on GPUs and has been carefully studied [18]. On GPUs, some work studies high-level programming models which partially automate the use of shared memory [8]. However, none of this work includes caches in their systems. A systematic study of comparisons and tradeoffs between GPU shared memory and caches can guide future GPU designs. Our work on characterizing the use of GPU caches provides groundwork for achieving this goal.

8. CONCLUSIONS

Though L1 caches are being included in current GPUs, little work has been done to study their effectiveness. In this paper, we evaluate the performance impact of using L1 caches on the Rodinia benchmark programs, and we provide an automatable algorithm for exploiting caches to reliably improve performance. L1 caches improve the performance of some programs while hurting or not affecting others. Our analysis shows that instead of cache hit rate, the metric commonly used in CPUs to describe cache effectiveness, memory traffic is a more indicative predictor of GPU cache benefits.

To help GPU programmers understand cache impacts, we develop a GPU-oriented locality taxonomy reminiscent of the three C's model for CPU caches. The taxonomy's three types of memory access locality (within-warp, within-block, and cross-instruction reuse) help programmers reason about memory accesses in GPU execution models. Within-block locality has the greatest potential for benefiting from caching based on memory traffic reduction. Cross-instruction reuse has the possibility of benefiting from caching depending on the working set size relative to cache capacity. Within-warp locality does not benefit from caching.

We present a compile-time algorithm which analyzes load instructions in a GPU kernel and determines whether each load should use the cache. For kernels that previously experienced unpredictable performance variations from using L1 caches in an all-or-nothing fashion, our algorithm offers significant performance improvements. Compared to a 5.8% caching benefit gained by turning caches on all the time, our algorithm improves performance by 16.8% (conservative) or 18% (aggressive) by more carefully analyzing data transfers and using the per-load caching control provided in real GPUs. Furthermore, our approach can be implemented at compile-time and requires no dynamic execution profiling.

9. ACKNOWLEDGMENTS

We thank Carole-Jean Wu and the anonymous reviewers for their comments on this work. This work was supported in part by the National Science Foundation under Grant No. CCF-0916971. The authors also acknowledge the support of the Gigascale Systems Research Center, one of six centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity. Finally, we acknowledge equipment donations from NVIDIA.

10. REFERENCES

[1] M. M. Baskaran et al. A compiler framework for optimization of affine loop nests for GPGPUs. In *Proc. 22nd ACM Intl. Conf. on Supercomputing*, 2008.

[2] I. Buck et al. Brook for GPUs: Stream computing on graphics hardware. In *31st Intl. Conf. on Computer Graphics and Interactive Techniques*, 2004.

[3] S. Che et al. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. IEEE Int. Symp. Workload Characterization*, 2009.

[4] W. J. Dally et al. Merrimac: Supercomputing with streams. In *Proc. 2003 ACM/IEEE Conf. Supercomputing*, 2003.

[5] E. Demers. Evolution of AMD graphics, 2011. Presented at AMD Fusion Developer Summit.

[6] K. Fatahalian and M. Houston. A closer look at GPUs. *Communications of the ACM*, 51(10):50–57, October 2008.

[7] M. Gebhart et al. Energy-efficient mechanisms for managing thread context in throughput processors. In *Proc. 38th Ann. Int. Symp. Computer Architecture*, 2011.

[8] T. D. Han and T. S. Abdelrahman. hiCUDA: High-level GPGPU programming. *IEEE Trans. on Parallel and Distributed Systems*, 2011.

[9] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *Proc. 14th Intl. Conf. High Performance Computing*, 2007.

[10] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989.

[11] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proc. 36th Ann. Int. Symp. Computer Architecture*, 2009.

[12] NVIDIA Corp. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, 2009.

[13] NVIDIA Corp. *PTX: Parallel Thread Execution ISA Version 2.3*, March 2011.

[14] NVIDIA Corp. *Tuning CUDA Applications for Fermi*, May 2011. Page 3.

[15] S. Seo et al. Design and implementation of software-managed caches for multicores with local memory. In *IEEE 15th Intl. Symp. on High Performance Computer Architecture*, 2009.

[16] I.-J. Sung, J. A. Stratton, and W.-M. W. Hwu. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In *Proc. 19th Int. Conf. on Parallel Architectural and Compilation Techniques*, 2010.

[17] Y. Torres and A. Gonzales-Escribano. Understanding the impact of CUDA tuning techniques for Fermi. In *2011 Intl. Conf. on High Performance Computing and Simulation*, 2011.

[18] S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. on Embedded Computing Systems*, 2006.

[19] H. Wong et al. Demystifying GPU microarchitecture through microbenchmarking. In *IEEE Intl. Symp. on Performance Analysis of Systems Software*, 2010.

[20] Y. Yang et al. A GPGPU compiler for memory optimization and parallelism management. In *Proc. 2010 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2010.

[21] E. Z. Zhang et al. Streamlining GPU applications on the fly—thread divergence elimination through runtime thread-data remapping. In *Proc. 24th ACM Intl. Conf. on Supercomputing*, 2010.