Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance

Rachata Ausavarungnirun Saugata Ghose Onur Kayıran†‡ Gabriel H. Loh†
Chita R. Das‡ Mahmut T. Kandemir‡ Onur Mutlu

Carnegie Mellon University
{*rachata, ghose, onur*}*@cmu.edu*

†Advanced Micro Devices, Inc.
{*onur.kayiran, gabriel.loh*}*@amd.com*

‡Pennsylvania State University
{*das, kandemir*}*@cse.psu.edu*

***Abstract*—In a GPU, all threads within a warp execute the same instruction in lockstep. For a memory instruction, this can lead to *memory divergence*: the memory requests for some threads are serviced early, while the remaining requests incur long latencies. This divergence stalls the warp, as it cannot execute the next instruction until *all* requests from the current instruction complete.**

In this work, we make three new observations. First, GPGPU warps exhibit heterogeneous memory divergence behavior at the shared cache: some warps have most of their requests hit in the cache (high cache utility), while other warps see most of their request miss (low cache utility). Second, a warp retains the same divergence behavior for long periods of execution. Third, due to high memory level parallelism, requests going to the shared cache can incur queuing delays as large as hundreds of cycles, exacerbating the effects of memory divergence.

We propose a set of techniques, collectively called *Memory Divergence Correction* (MeDiC), that reduce the negative performance impact of memory divergence and cache queuing. MeDiC uses warp divergence characterization to guide three components: (1) a cache bypassing mechanism that exploits the latency tolerance of low cache utility warps to both alleviate queuing delay and increase the hit rate for high cache utility warps, (2) a cache insertion policy that prevents data from high cache utility warps from being prematurely evicted, and (3) a memory controller that prioritizes the few requests received from high cache utility warps to minimize stall time. We compare MeDiC to four cache management techniques, and find that it delivers an average speedup of 21.8%, and 20.1% higher energy efficiency, over a state-of-the-art GPU cache management mechanism across 15 different GPGPU applications.

1. Introduction

Graphics Processing Units (GPUs) have enormous parallel processing power to leverage thread-level parallelism. GPU applications can be broken down into thousands of threads, allowing GPUs to use *fine-grained multithreading* [63, 68] to prevent GPU cores from stalling due to dependencies and long memory latencies. Ideally, there should always be available threads for GPU cores to continue execution, preventing stalls within the core. GPUs also take advantage of the *SIMD* (Single Instruction, Multiple Data) execution model [11]. The thousands of threads within a GPU application are clustered into *work groups* (or *thread blocks*), with each thread block consisting of multiple smaller bundles of threads that are run concurrently. Each such thread bundle is called a *wavefront* [1] or *warp* [40]. In each cycle, each GPU core executes a single warp. Each thread in a warp executes the same instruction (i.e., is at the same program counter). Combining SIMD execution with fine-grained multithreading allows a GPU to complete several hundreds of operations every cycle in the ideal case.

In the past, GPUs strictly executed graphics applications, which naturally exhibit large amounts of concurrency. In recent years, with tools such as CUDA [51] and OpenCL [29], programmers have been able to adapt non-graphics applications to GPUs, writing these applications to have thousands of threads that can be run on a SIMD computation engine. Such adapted non-graphics programs are known as general-purpose GPU (GPGPU) applications. Prior work has demonstrated that many scientific and data analysis applications can be executed significantly faster when programmed to run on GPUs [4, 6, 17, 64].

While many GPGPU applications can tolerate a significant amount of memory latency due to their parallelism and the use of fine-grained multithreading, many previous works (e.g., [23, 24, 47, 72]) observe that GPU cores still stall for a significant fraction of time when running many other GPGPU applications. One significant source of these stalls is *memory divergence*, where the threads of a warp reach a memory instruction, and some of the threads' memory requests take longer to service than the requests from other threads [5, 44, 47]. Since all threads within a warp operate in lockstep due to the SIMD execution model, the warp cannot proceed to the next instruction until the *slowest* request within the warp completes, and *all* threads are ready to continue execution. Figures 1a and 1b show examples of memory divergence within a warp, which we will explain in more detail soon.

In this work, we make three new key observations about the memory divergence behavior of GPGPU warps:

Observation 1: There is *heterogeneity across warps* in the degree of memory divergence experienced by each warp at

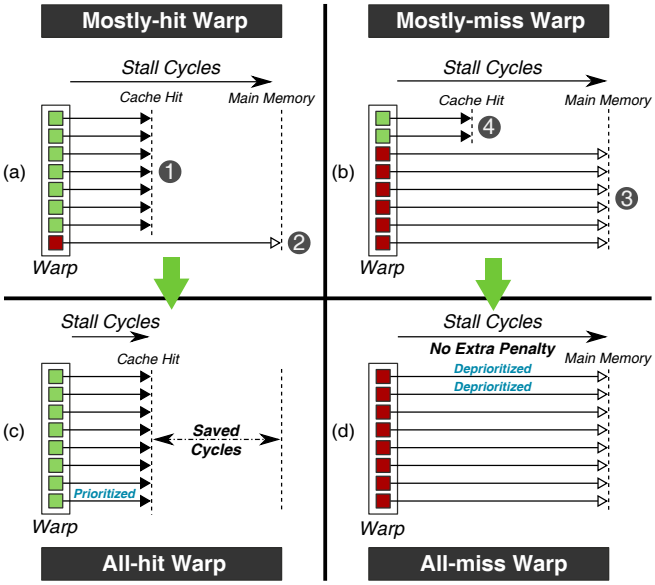

Figure 1: Memory divergence within a warp. (a) and (b) show the heterogeneity between *mostly-hit* and *mostly-miss* warps, respectively. (c) and (d) show the change in stall time from converting *mostly-hit warps into all-hit warps*, and *mostly-miss warps into all-miss warps*, respectively.

the shared L2 cache (i.e., the percentage of threads within a warp that miss in the cache varies widely). Figure 1 shows examples of two different *types of warps*, with eight threads each, that exhibit different degrees of memory divergence:

- Figure 1a shows a *mostly-hit warp*, where most of the warp's memory accesses hit in the cache (❶). However, a single access misses in the cache and must go to main memory (❷). As a result, the *entire warp* is stalled until the much longer cache miss completes.
- Figure 1b shows a *mostly-miss warp*, where most of the warp's memory requests miss in the cache (❸), resulting in many accesses to main memory. Even though some requests are cache hits (❹), these do not benefit the execution time of the warp.

Observation 2: *A warp tends to retain its memory divergence behavior* (e.g., whether or not it is mostly-hit or mostly-miss) *for long periods of execution*, and is thus predictable. As we show in Section 4, this predictability enables us to perform history-based warp divergence characterization.

Observation 3: Due to the amount of thread parallelism within a GPU, *a large number of memory requests can arrive at the L2 cache in a small window of execution time, leading to significant queuing delays*. Prior work observes high access latencies for the shared L2 cache within a GPU [61, 62, 73], but does not identify *why* these latencies are so high. We show that when a large number of requests arrive at the L2 cache, both the limited number of read/write ports and backpressure from cache bank conflicts force many of these requests to queue up for long periods of time. We observe that this queuing latency can sometimes add *hundreds* of cycles to the cache access latency, and that non-uniform queuing across the different cache banks exacerbates memory divergence.

Based on these three observations, we aim to devise a mechanism that has two major goals: (1) convert mostly-hit warps into *all-hit warps* (warps where *all* requests hit in the cache, as shown in Figure 1c), and (2) convert mostly-miss warps into *all-miss warps* (warps where *none* of the requests hit in the cache, as shown in Figure 1d). As we can see in Figure 1a, the stall time due to memory divergence for the mostly-hit warp can be eliminated by converting only the single cache miss (❷) into a hit. Doing so requires additional cache space. If we convert the two cache hits of the mostly-miss warp (Figure 1b, ❹) into cache misses, we can cede the cache space previously used by these hits to the mostly-hit warp, thus converting the mostly-hit warp into an all-hit warp. Though the mostly-miss warp is now an all-miss warp (Figure 1d), it incurs no extra stall penalty, as the warp was already waiting on the other six cache misses to complete. Additionally, now that it is an all-miss warp, we predict that its future memory requests will also not be in the L2 cache, so we can simply have these requests *bypass the cache*. In doing so, the requests from the all-miss warp can completely avoid unnecessary L2 access and queuing delays. This decreases the total number of requests going to the L2 cache, thus reducing the queuing latencies for requests from mostly-hit and all-hit warps, as there is less contention.

We introduce *Memory Divergence Correction* (*MeDiC*), a GPU-specific mechanism that exploits *memory divergence heterogeneity* across warps at the shared cache and at main memory to improve the overall performance of GPGPU applications. MeDiC consists of three different components, which work together to achieve our goals of converting mostly-hit warps into all-hit warps and mostly-miss warps into all-miss warps: (1) a warp-type-aware *cache bypassing mechanism*, which prevents requests from mostly-miss and all-miss warps from accessing the shared L2 cache (Section 4.2); (2) a warp-type-aware *cache insertion policy*, which prioritizes requests from mostly-hit and all-hit warps to ensure that they all become cache hits (Section 4.3); and (3) a warp-type-aware *memory scheduling mechanism*, which prioritizes requests from mostly-hit warps that were not successfully converted to all-hit warps, in order to minimize the stall time due to divergence (Section 4.4). These three components are all driven by an online mechanism that can identify the expected memory divergence behavior of each warp (Section 4.1).

This paper makes the following contributions:

- We observe that the different warps within a GPGPU application exhibit heterogeneity in their memory divergence behavior at the shared L2 cache, and that some warps do not benefit from the few cache hits that they have. This memory divergence behavior tends to remain consistent throughout long periods of execution

for a warp, allowing for fast, online warp divergence characterization and prediction.
- We identify a new performance bottleneck in GPGPU application execution that can contribute significantly to memory divergence: due to the very large number of memory requests issued by warps in GPGPU applications that contend at the shared L2 cache, many of these requests experience *high cache queuing latencies*.
- Based on our observations, we propose *Memory Divergence Correction*, a new mechanism that exploits the stable memory divergence behavior of warps to (1) improve the effectiveness of the cache by favoring warps that take the most advantage of the cache, (2) address the cache queuing problem, and (3) improve the effectiveness of the memory scheduler by favoring warps that benefit most from prioritization. We compare MeDiC to four different cache management mechanisms, and show that it improves performance by 21.8% and energy efficiency by 20.1% across a wide variety of GPGPU workloads compared to a a state-of-the-art GPU cache management mechanism [39].

2. Background

We first provide background on the architecture of a modern GPU, and then we discuss the bottlenecks that highly-multithreaded applications can face when executed on a GPU. These applications can be compiled using OpenCL [29] or CUDA [51], either of which converts a general purpose application into a GPGPU program that can execute on a GPU.

2.1. Baseline GPU Architecture

A typical GPU consists of several *shader cores* (sometimes called *streaming multiprocessors*, or SMs). In this work, we set the number of shader cores to 15, with 32 threads per warp in each core, corresponding to the NVIDIA GTX480 GPU based on the Fermi architecture [49]. The GPU we evaluate can issue up to 480 concurrent memory accesses per cycle [70]. Each core has its own private L1 data, texture, and constant caches, as well as a scratchpad memory [40, 49, 50]. In addition, the GPU also has several shared L2 cache slices and memory controllers. A *memory partition unit* combines a single L2 cache slice (which is banked) with a designated memory controller that connects to off-chip main memory. Figure 2 shows a simplified view of how the cores (or SMs), caches, and memory partitions are organized in our baseline GPU.

2.2. Bottlenecks in GPGPU Applications

Several previous works have analyzed the benefits and limitations of using a GPU for general purpose workloads (other than graphics purposes), including characterizing the impact of microarchitectural changes on applications [3] or developing performance models that break down performance bottlenecks in GPGPU applications [14, 18, 35, 41, 42, 59]. All of these works show benefits from using

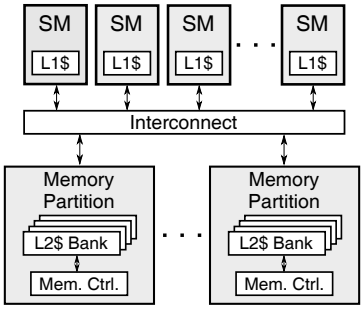

Figure 2: Overview of the baseline GPU architecture.

a throughput-oriented GPU. However, a significant number of applications are unable to fully utilize all of the available parallelism within the GPU, leading to periods of execution where no warps are available for execution [72].

When there are no available warps, the GPU cores stall, and the application stops making progress until a warp becomes available. Prior work has investigated two problems that can delay some warps from becoming available for execution: (1) *branch divergence*, which occurs when a branch in the same SIMD instruction resolves into multiple different paths [3, 12, 16, 47, 74], and (2) *memory divergence*, which occurs when the simultaneous memory requests from a single warp spend different amounts of time retrieving their associated data from memory [5, 44, 47]. In this work, we focus on the memory divergence problem; prior work on branch divergence is complementary to our work.

3. Motivation and Key Observations

We make three new key observations about memory divergence (at the shared L2 cache). First, we observe that the degree of memory divergence can differ across warps. This inter-warp heterogeneity affects how well each warp takes advantage of the shared cache. Second, we observe that a warp's memory divergence behavior tends to remain stable for long periods of execution, making it predictable. Third, we observe that requests to the shared cache experience long queuing delays due to the large amount of parallelism in GPGPU programs, which exacerbates the memory divergence problem and slows down GPU execution. Next, we describe each of these observations in detail and motivate our solutions.

3.1. Exploiting Heterogeneity Across Warps

We observe that different warps have different amounts of sensitivity to memory latency and cache utilization. We study the cache utilization of a warp by determining its *hit ratio*, the percentage of memory requests that hit in the cache when the warp issues a single memory instruction. As Figure 3 shows, the warps from each of our three representative GPGPU applications are distributed across all possible ranges of *hit ratio*, exhibiting significant heterogeneity. To better characterize warp behavior, we break the warps down into the five types shown in Figure 4 based on their hit ratios: *all-hit*, *mostly-hit*, *balanced*, *mostly-miss*, and *all-miss*.

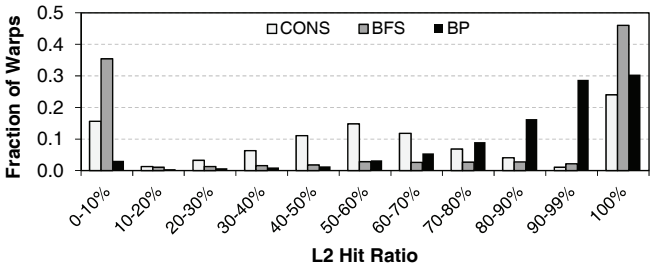

Figure 3: L2 cache hit ratio of different warps in three representative GPGPU applications (see Section 5 for methods).

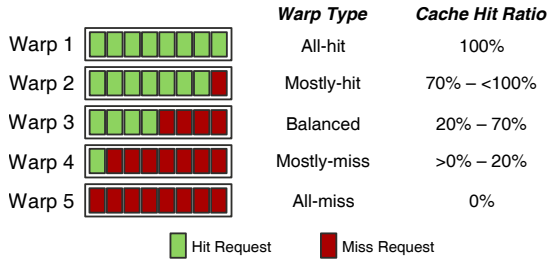

Figure 4: Warp type categorization based on the shared cache hit ratios. Hit ratio values are empirically chosen.

This inter-warp heterogeneity in cache utilization provides new opportunities for performance improvement. We illustrate two such opportunities by walking through a simplified example, shown in Figure 5. Here, we have two warps, *A* and *B*, where *A* is a mostly-miss warp (with three of its four memory requests being L2 cache misses) and *B* is a mostly-hit warp with only a single L2 cache miss (request *B0*). Let us assume that warp *A* is scheduled first.

As we can see in Figure 5a, the mostly-miss warp *A* does not benefit at all from the cache: even though one of its requests (*A3*) hits in the cache, warp *A* cannot continue executing until *all* of its memory requests are serviced. As the figure shows, using the cache to speed up only request *A3* has no material impact on warp *A*'s stall time. In addition, while requests *A1* and *A2* do not hit in the cache, they still incur a queuing latency at the cache while they wait to be looked up in the cache tag array.

On the other hand, the mostly-hit warp *B* can be penalized significantly. First, since warp *B* is scheduled after the mostly-miss warp *A*, all four of warp *B*'s requests incur a large L2 queuing delay, *even though the cache was not useful to speed up warp* A. On top of this unproductive delay, since request *B0* misses in the cache, it holds up execution of the entire warp while it gets serviced by main memory. The overall effect is that despite having *many more* cache hits (and thus much better cache utility) than warp *A*, warp *B* ends up stalling for as long as or even longer than the mostly-miss warp *A* stalled for.

To remedy this problem, we set two goals (Figure 5b):

1) Convert the mostly-hit warp B *into an all-hit warp.* By converting *B0* into a hit, warp *B* no longer has to stall on any memory misses, which enables the warp to become ready to execute much earlier. This requires a little additional space in the cache to store the data for *B0*.

2) Convert the mostly-miss warp A *into an all-miss warp.* Since a single cache hit is of no effect to warp *A*'s execution, we convert *A0* into a cache miss. This frees up the cache space *A0* was using, and thus creates cache space for storing *B0*. In addition, warp *A*'s requests can now skip accessing the cache and go straight to main memory, which has two benefits: *A0–A2* complete faster because they no longer experience the cache queuing delay that they incurred in Figure 5a, and *B0–B3* also complete faster because they must queue behind a smaller number of cache requests. Thus, bypassing the cache for warp *A*'s request allows *both* warps to stall for less time, improving GPU core utilization.

To realize these benefits, we propose to (1) develop a mechanism that can identify mostly-hit and mostly-miss warps; (2) design a mechanism that allows mostly-miss warps to yield their ineffective cache space to mostly-hit warps, similar to how the mostly-miss warp *A* in Figure 5a turns into an all-miss warp in Figure 5b, so that warps such as the mostly-hit warp *B* can become all-hit warps; (3) design a mechanism that bypasses the cache for requests from mostly-miss and all-miss warps such as warp *A*, to decrease warp stall time and reduce lengthy cache queuing latencies; and (4) prioritize requests from mostly-hit warps

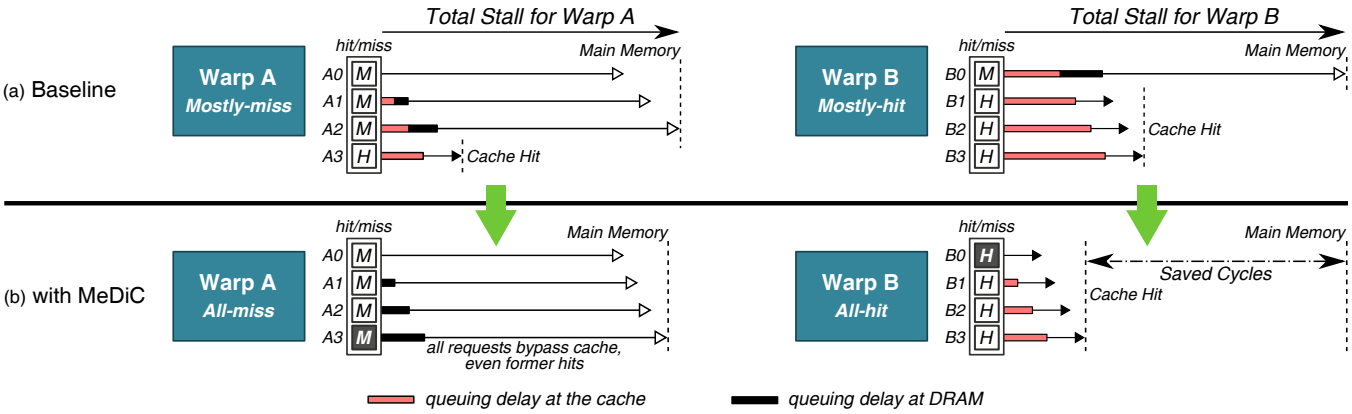

Figure 5: (a) Existing inter-warp heterogeneity, (b) exploiting the heterogeneity with MeDiC to improve performance.

across the memory hierarchy, at both the shared L2 cache and at the memory controller, to minimize their stall time as much as possible, similar to how the mostly-hit warp *B* in Figure 5a turns into an all-hit warp in Figure 5b.

A key challenge is how to group warps into different warp types. In this work, we observe that warps tend to exhibit stable cache hit behavior over long periods of execution. A warp consists of several threads that repeatedly loop over the same instruction sequences. This results in similar hit/miss behavior at the cache level across different instances of the same warp. As a result, a warp measured to have a particular hit ratio is likely to maintain a similar hit ratio throughout a lengthy phase of execution. We observe that most CUDA applications exhibit this trend.

Figure 6 shows the hit ratio over a duration of one million cycles, for six randomly selected warps from our CUDA applications. We also plot horizontal lines to illustrate the hit ratio cutoffs that we set in Figure 4 for our mostly-hit ($\geq$70%) and mostly-miss ($\leq$20%) warp types. Warps 1, 3, and 6 spend the majority of their time with high hit ratios, and are classified as mostly-hit warps. Warps 1 and 3 do, however, exhibit some long-term (i.e., 100k+ cycles) shifts to the balanced warp type. Warps 2 and 5 spend a long time as mostly-miss warps, though they both experience a single long-term shift into balanced warp behavior. As we can see, *warps tend to remain in the same warp type* at least for hundreds of thousands of cycles.

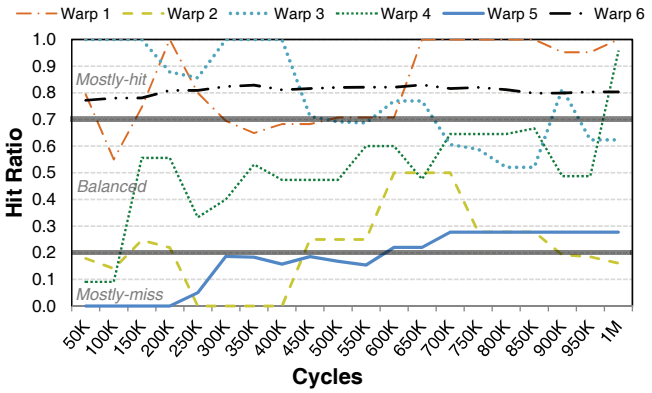

Figure 6: Hit ratio of randomly selected warps over time.

As a result of this relatively stable behavior, our mechanism, MeDiC (described in detail in Section 4), samples the hit ratio of each warp and uses this data for warp characterization. To account for the long-term hit ratio shifts, MeDiC resamples the hit ratio every 100k cycles.

3.2. Reducing the Effects of L2 Queuing Latency

Unlike CPU applications, GPGPU applications can issue as many as hundreds of memory instructions per cycle. All of these memory requests can arrive concurrently at the L2 cache, which is the first shared level of the memory hierarchy, creating a bottleneck. Previous works [3,61,62,73] point out that the latency for accessing the L2 cache can take hundreds of cycles, even though the nominal cache lookup latency is significantly lower (only tens of cycles). While they identify this disparity, these earlier efforts do not identify or analyze the source of these long delays.

We make a new observation that identifies an important source of the long L2 cache access delays in GPGPU systems. L2 bank conflicts can cause queuing delay, which can differ from one bank to another and lead to the disparity of cache access latencies across different banks. As Figure 7a shows, even if every cache access within a warp hits in the L2 cache, each access can incur a different cache latency due to non-uniform queuing, and the warp has to stall until the *slowest* cache access retrieves its data (i.e., memory divergence can occur). For each set of simultaneous requests issued by an all-hit warp, we define its *inter-bank divergence penalty* to be the difference between the fastest cache hit and the slowest cache hit, as depicted in Figure 7a.

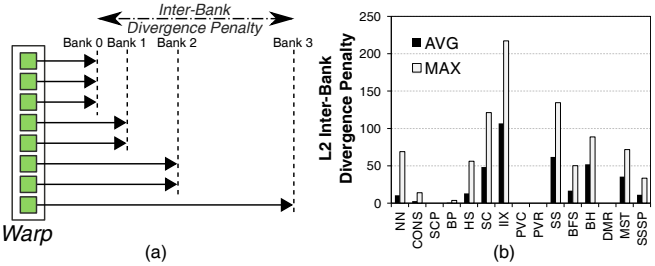

Figure 7: Effect of bank queuing latency divergence in the L2 cache: (a) example of the impact on stall time of skewed queuing latencies, (b) inter-bank divergence penalty due to skewed queuing for all-hit warps, in cycles.

In order to confirm this behavior, we modify GPGPU-Sim [3] to accurately model L2 bank conflicts and queuing delays (see Section 5 for details). We then measure the average and maximum inter-bank divergence penalty observed *only for all-hit warps* in our different CUDA applications, shown in Figure 7b. We find that *on average*, an all-hit warp has to stall for an additional 24.0 cycles because some of its requests go to cache banks with high access contention.

To quantify the magnitude of queue contention, we analyze the queuing delays for a two-bank L2 cache where the tag lookup latency is set to one cycle. We find that even with such a small cache lookup latency, a significant number of requests experience tens, if not hundreds, of cycles of queuing delay. Figure 8 shows the distribution of these delays for BFS [4], across all of its individual L2 cache requests. BFS contains one compute-intensive kernel and two memory-intensive kernels. We observe that requests generated by the compute-intensive kernel do not incur high queuing latencies, while requests from the memory-intensive kernels suffer from significant queuing delays. On average, across all three kernels, cache requests spend 34.8 cycles in the queue waiting to be serviced, which is quite high considering the idealized one-cycle cache lookup latency.

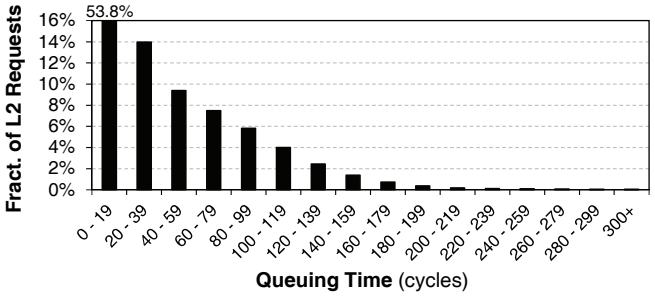

Figure 8: Distribution of per-request queuing latencies for L2 cache requests from BFS.

One naive solution to the L2 cache queuing problem is to increase the number of banks, without reducing the number of physical ports per bank and without increasing the size of the shared cache. However, as shown in Figure 9, the average performance improvement from doubling the number of banks to 24 (i.e., 4 banks per memory partition) is less than 4%, while the improvement from quadrupling the banks is less than 6%. There are two key reasons for this minimal performance gain. First, while more cache banks can help to distribute the queued requests, these extra banks do not change the memory divergence behavior of the warp (i.e., the warp hit ratios remain unchanged). Second, non-uniform bank access patterns still remain, causing cache requests to queue up unevenly at a few banks.[1]

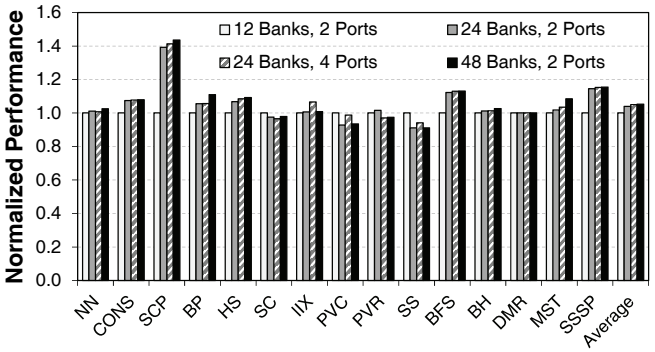

Figure 9: Performance of GPGPU applications with different number of banks and ports per bank, normalized to a 12-bank cache with 2 ports per bank.

3.3. Our Goal

Our goal in this paper is to improve cache utilization and reduce cache queuing latency by taking advantage of heterogeneity between different types of warps. To this end, we create a mechanism that (1) tries to eliminate mostly-hit and mostly-miss warps by converting as many of them as possible to all-hit and all-miss warps, respectively; (2) reduces the queuing delay at the L2 cache by bypassing requests from mostly-miss and all-miss warps, such that each L2 cache hit experiences a much lower overall L2 cache latency; and (3) prioritizes mostly-hit warps in the memory scheduler to minimize the amount of time they stall due to a cache miss.

[1]Similar problems have been observed for bank conflicts in main memory [32, 54].

4. MeDiC: Memory Divergence Correction

In this section, we introduce Memory Divergence Correction (MeDiC), a set of techniques that take advantage of the memory divergence heterogeneity across warps, as discussed in Section 3. These techniques work independently of each other, but act synergistically to provide a substantial performance improvement. In Section 4.1, we propose a mechanism that identifies and groups warps into different warp types based on their degree of memory divergence, as shown in Figure 4.

As depicted in Figure 10, MeDiC uses ❶ warp type identification to drive three different components: ❷ a *warp-type-aware cache bypass mechanism* (Section 4.2), which bypasses requests from all-miss and mostly-miss warps to reduce the L2 queuing delay; ❸ a *warp-type-aware cache insertion policy* (Section 4.3), which works to keep cache lines from mostly-hit warps while demoting lines from mostly-miss warps; and ❹ a *warp-type-aware memory scheduler* (Section 4.4), which prioritizes DRAM requests from mostly-hit warps as they are highly latency sensitive. We analyze the hardware cost of MeDiC in Section 6.5.

4.1. Warp Type Identification

In order to take advantage of the memory divergence heterogeneity across warps, we must first add hardware that can identify the divergence behavior of each warp. The key idea is to periodically sample the hit ratio of a warp, and to classify the warp's divergence behavior as one of the five types in Figure 4 based on the observed hit ratio (see Section 3.1). This information can then be used to drive the warp-type-aware components of MeDiC. In general, warps tend to retain the same memory divergence behavior for long periods of execution. However, as we observed in Section 3.1, there can be some long-term shifts in warp divergence behavior, requiring periodic resampling of the hit ratio to potentially adjust the warp type.

Warp type identification through hit ratio sampling requires hardware within the cache to periodically count the number of hits and misses each warp incurs. We append two counters to the metadata stored for each warp, which represent the total number of cache hits and cache accesses for the warp. We reset these counters periodically, and set the bypass logic to operate in a profiling phase for each warp after this reset.[2] During profiling, which lasts for the first 30 cache accesses of each warp, the bypass logic (which we explain in Section 4.2) does not make any cache bypassing decisions, to allow the counters to accurately characterize the current memory divergence behavior of the warp. At the end of profiling, the warp type is determined and stored in the metadata.

[2]In this work, we reset the hit ratio every 100k cycles for each warp.

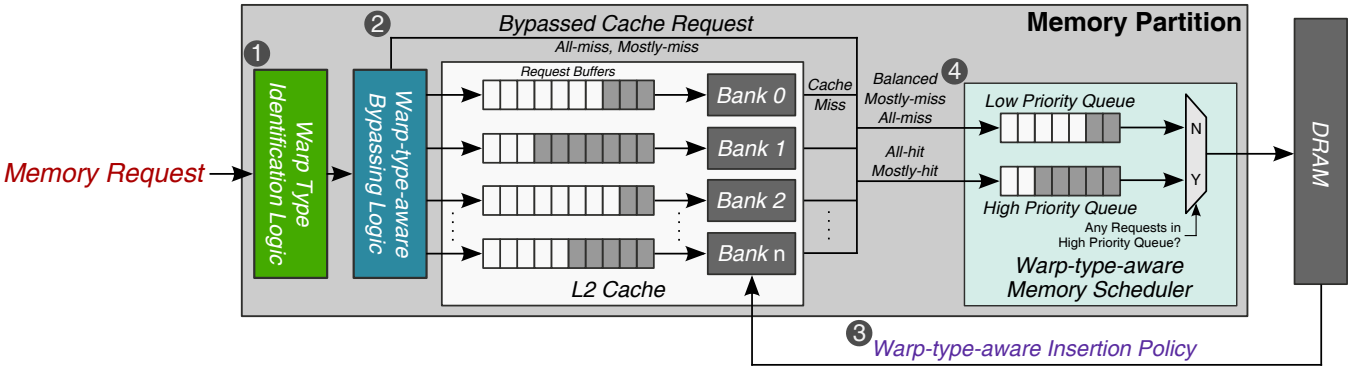

Figure 10: Overview of MeDiC: ❶ warp type identification logic, ❷ warp-type-aware cache bypassing, ❸ warp-type-aware cache insertion policy, ❹ warp-type-aware memory scheduler.

4.2. Warp-type-aware Shared Cache Bypassing

Once the warp type is known and a warp generates a request to the L2 cache, our mechanism first decides whether to bypass the cache based on the warp type. The key idea behind *warp-type-aware cache bypassing*, as discussed in Section 3.1, is to convert mostly-miss warps into all-miss warps, as they do not benefit greatly from the few cache hits that they get. By bypassing these requests, we achieve three benefits: (1) bypassed requests can avoid L2 queuing latencies entirely, (2) other requests that do hit in the L2 cache experience shorter queuing delays due to the reduced contention, and (3) space is created in the L2 cache for mostly-hit warps.

The cache bypassing logic must make a simple decision: if an incoming memory request was generated by a mostly-miss or all-miss warp, the request is bypassed directly to DRAM. This is determined by reading the warp type stored in the warp metadata from the warp type identification mechanism. A simple 2-bit demultiplexer can be used to determine whether a request is sent to the L2 bank arbiter, or directly to the DRAM request queue.

Dynamically Tuning the Cache Bypassing Rate. While cache bypassing alleviates queuing pressure at the L2 cache banks, it can have a negative impact on other portions of the memory partition. For example, bypassed requests that were originally cache hits now consume extra off-chip memory bandwidth, and can increase queuing delays at the DRAM queue. If we lower the number of bypassed requests (i.e., reduce the number of warps classified as mostly-miss), we can reduce DRAM utilization. After examining a random selection of kernels from three applications (BFS, BP, and CONS), we find that the ideal number of warps classified as mostly-miss differs for each kernel. Therefore, we add a mechanism that *dynamically* tunes the hit ratio boundary between mostly-miss warps and balanced warps (nominally set at 20%; see Figure 4). If the cache miss rate increases significantly, the hit ratio boundary is lowered.[3]

[3]In our evaluation, we reduce the threshold value between mostly-miss warps and balanced warps by 5% for every 5% increase in cache miss rate.

Cache Write Policy. Recent GPUs support multiple options for the L2 cache write policy [49]. In this work, we assume that the L2 cache is write-through [61], so our bypassing logic can always assume that DRAM contains an up-to-date copy of the data. For write-back caches, previously-proposed mechanisms [15,43,60] can be used in conjunction with our bypassing technique to ensure that bypassed requests get the correct data. For correctness, fences and atomic instructions from bypassed warps still access the L2 for cache lookup, but are not allowed to store data in the cache.

4.3. Warp-type-aware Cache Insertion Policy

Our cache bypassing mechanism frees up space within the L2 cache, which we want to use for the cache misses from mostly-hit warps (to convert these memory requests into cache hits). However, even with the new bypassing mechanism, other warps (e.g., balanced, mostly-miss) still insert some data into the cache. In order to aid the conversion of mostly-hit warps into all-hit warps, we develop a *warp-type-aware cache insertion policy*, whose key idea is to ensure that for a given cache set, data from mostly-miss warps are evicted first, while data from mostly-hit warps and all-hit warps are evicted last.

To ensure that a cache block from a mostly-hit warp stays in the cache for as long as possible, we insert the block closer to the MRU position. A cache block requested by a mostly-miss warp is inserted closer to the LRU position, making it more likely to be evicted. To track the status of these cache blocks, we add two bits of metadata to each cache block, indicating the warp type.[4] These bits are then appended to the replacement policy bits. As a result, a cache block from a mostly-miss warp is more likely to get evicted than a block from a balanced warp. Similarly, a cache block from a balanced warp is more likely to be evicted than a block from a mostly-hit or all-hit warp.

[4]Note that cache blocks from the all-miss category share the same 2-bit value as the mostly-miss category because they always get bypassed (see Section 4.2).

4.4. Warp-type-aware Memory Scheduler

Our cache bypassing mechanism and cache insertion policy work to increase the likelihood that *all* requests from a mostly-hit warp become cache hits, converting the warp into an all-hit warp. However, due to cache conflicts, or due to poor locality, there may still be cases when a mostly-hit warp cannot be fully converted into an all-hit warp, and is therefore unable to avoid stalling due to memory divergence as at least one of its requests has to go to DRAM. In such a case, we want to minimize the amount of time that this warp stalls. To this end, we propose a *warp-type-aware memory scheduler* that prioritizes the occasional DRAM request from a mostly-hit warp.

The design of our memory scheduler is very simple. Each memory request is tagged with a single bit, which is set if the memory request comes from a mostly-hit warp (or an all-hit warp, in case the warp was mischaracterized). We modify the request queue at the memory controller to contain two different queues (❹ in Figure 10), where a *high-priority queue* contains all requests that have their mostly-hit bit set to one. The *low-priority queue* contains all other requests, whose mostly-hit bits are set to zero. Each queue uses FR-FCFS [55, 79] as the scheduling policy; however, the scheduler always selects requests from the high priority queue over requests in the low priority queue.[5]

5. Methodology

We model our mechanism using GPGPU-Sim 3.2.1 [3]. Table I shows the configuration of the GPU. We modified GPGPU-Sim to accurately model cache bank conflicts, and added the cache bypassing, cache insertion, and memory scheduling mechanisms needed to support MeDiC. We use GPUWattch [36] to evaluate power consumption.

Modeling L2 Bank Conflicts. In order to analyze the detailed caching behavior of applications in modern GPGPU architectures, we modified GPGPU-Sim to accurately model banked caches.[6] Within each memory partition, we divide the shared L2 cache into two banks. When a memory request misses in the L1 cache, it is sent to the memory partition through the shared interconnect. However, it can only be sent if there is a free port available at the memory partition (we dual-port each memory partition). Once a request arrives at the port, a unified bank arbiter dispatches the request to the request queue for the appropriate cache bank (which is determined statically using some of the memory address bits). If the bank request queue is full, the request remains at the incoming port until the queue is freed up. Traveling through the port and arbiter consumes an extra cycle per request. In order to prevent a bias towards any one port or any one cache bank, the simulator rotates which port and which bank are first examined every cycle.

When a request misses in the L2 cache, it is sent to the DRAM request queue, which is shared across all L2 banks as previously implemented in GPGPU-Sim. When a request returns from DRAM, it is inserted into one of the per-bank DRAM-to-L2 queues. Requests returning from the L2 cache to the L1 cache go through a unified memory-partition-to-interconnect queue (where round-robin priority is used to insert requests from different banks into the queue).

GPGPU Applications. We evaluate our system across multiple GPGPU applications from the CUDA SDK [48], Rodinia [6], MARS [17], and Lonestar [4] benchmark suites.[7] These applications are listed in Table II, along with the breakdown of warp characterization. The dominant warp type for each application is marked in *bold* (AH: all-hit, MH: mostly-hit, BL: balanced, MM: mostly-miss, AM: all-miss; see Figure 4). We simulate 500 million instructions for each kernel of our application, though some kernels complete before reaching this instruction count.

Comparisons. In addition to the baseline results, we compare each individual component of MeDiC with state-of-the-art policies. We compare our bypassing mechanism with three different cache management policies. First, we compare to PCAL [39], a token-based cache management mechanism. PCAL limits the number of threads that get to access the cache by using tokens. If a cache request is a miss, it causes a replacement only if the warp has a token. PCAL, as modeled in this work, first grants tokens to the warp that recently used the cache, then grants any remaining tokens to warps that access the cache in order of their arrival. Unlike the original proposal [39], which applies PCAL to the L1 caches, we apply PCAL to the shared L2 cache. We sweep the number of tokens per epoch and use the configuration that gives the best average performance. Second, we compare MeDiC against a random bypassing policy (**Rand**), where a percentage of randomly-chosen warps bypass the cache every 100k cycles. For every workload, we statically configure the percentage of warps that bypass the cache such that Rand yields the best performance. This comparison point is designed to show the value of warp type information in bypassing decisions. Third, we compare to a program counter (PC) based bypassing policy (**PC-Byp**). This mechanism bypasses requests from *static instructions* that mostly miss (as opposed to requests from mostly-miss *warps*). This comparison point is designed to distinguish the value of tracking hit ratios at the warp level instead of at the instruction level.

We compare our memory scheduling mechanism with the baseline first-ready, first-come first-serve (FR-FCFS) memory scheduler [55, 79], which is reported to provide good performance on GPU and GPGPU workloads [2,5,77]. We compare our cache insertion with the Evicted-Address Filter [58], a state-of-the-art CPU cache insertion policy.

[5] Using two queues ensures that high-priority requests are not blocked by low-priority requests even when the low-priority queue is full. Two-queue priority also uses simpler logic design than comparator-based priority [65, 66].

[6] We validate that the performance values reported for our applications before and after our modifications to GPGPU-Sim are equivalent.

[7] We use default tuning parameters for all applications.

System Overview	15 cores, 6 memory partitions
Shader Core Config.	1400 MHz, 9-stage pipeline, GTO scheduler [56]
Private L1 Cache	16KB, 4-way associative, LRU, L1 misses are coalesced before accessing L2, 1 cycle latency
Shared L2 Cache	768KB total, 16-way associative, LRU, 2 cache banks/2 interconnect ports per memory partition, 10 cycle latency
DRAM	GDDR5 1674 MHz, 6 channels (one channel per memory partition), FR-FCFS scheduler [55, 79], 8 banks per rank, burst length 8

Table I: Configuration of the simulated system.

#	Application	AH	MH	BL	MM	AM	#	Application	AH	MH	BL	MM	AM
1	Nearest Neighbor (NN) [48]	19%	**79%**	1%	0.9%	0.1%	10	Similarity Score (SS) [17]	**67%**	1%	11%	1%	20%
2	Convolution Separable (CONS) [48]	9%	1%	**82%**	1%	7%	11	Breadth-First Search (BFS) [4]	**40%**	1%	20%	13%	26%
3	Scalar Product (SCP) [48]	0.1%	0.1%	0.1%	0.7%	**99%**	12	Barnes-Hut N-body Simulation (BH) [4]	**84%**	0%	0%	1%	15%
4	Back Propagation (BP) [6]	10%	27%	**48%**	6%	9%							
5	Hotspot (HS) [6]	1%	29%	**69%**	0.5%	0.5%	13	Delaunay Mesh Refinement (DMR) [4]	**81%**	3%	3%	1%	12%
6	Streamcluster (SC) [6]	6%	0.2%	0.5%	0.3%	**93%**							
7	Inverted Index (IIX) [17]	**71%**	5%	8%	1%	15%	14	Minimum Spanning Tree (MST) [4]	**53%**	12%	18%	2%	15%
8	Page View Count (PVC) [17]	4%	1%	**42%**	20%	33%							
9	Page View Rank (PVR) [17]	18%	3%	28%	4%	**47%**	15	Survey Propagation (SP) [4]	**41%**	1%	20%	14%	24%

Table II: Evaluated GPGPU applications and the characteristics of their warps.

Evaluation Metrics: We report performance results using the harmonic average of the IPC speedup (over the baseline GPU) of each kernel of each application.[8] Harmonic speedup was shown to reflect the average normalized execution time in multiprogrammed workloads [10]. We calculate energy efficiency for each workload by dividing the IPC by the energy consumed.

6. Evaluation

6.1. Performance Improvement of MeDiC

Figure 11 shows the performance of MeDiC compared to the various state-of-the-art mechanisms (EAF [58], PCAL [39], Rand, PC-Byp) from Section 5,[9] as well as the performance of each individual component in MeDiC.

Baseline shows the performance of the unmodified GPU using FR-FCFS as the memory scheduler [55, 79]. **EAF** shows the performance of the Evicted-Address Filter [58]. **WIP** shows the performance of our warp-type-aware insertion policy by itself. **WMS** shows the performance of our warp-type-aware memory scheduling policy by itself. **PCAL** shows the performance of the PCAL bypassing mechanism proposed by Li et al. [39]. **Rand** shows the performance of a cache bypassing mechanism that performs bypassing decisions randomly on a fixed percentage of warps. **PC-Byp** shows the performance of the bypassing mechanism that uses the PC as the criterion for bypassing instead of the warp-type. **WByp** shows the performance of our warp-type-aware bypassing policy by itself.

From these results, we draw the following conclusions:

- Each component of MeDiC individually provides significant performance improvement: WIP (32.5%), WMS (30.2%), and WByp (33.6%). MeDiC, which combines all three mechanisms, provides a 41.5% performance improvement over Baseline, on average. MeDiC matches or outperforms its individual components for all benchmarks except BP, where MeDiC has a higher L2 miss rate and lower row buffer locality than WMS and WByp.
- WIP outperforms EAF [58] by 12.2%. We observe that the key benefit of WIP is that cache blocks from mostly-miss warps are much more likely to be evicted. In addition, WIP reduces the cache miss rate of several applications (see Section 6.3).
- WMS provides significant performance gains (30.2%) over Baseline, because the memory scheduler prioritizes requests from warps that have a high hit ratio, allowing these warps to become active much sooner than they do in Baseline.
- WByp provides an average 33.6% performance improvement over Baseline, because it is effective at reducing the L2 queuing latency. We show the change in queuing latency and provide a more detailed analysis in Section 6.3.
- Compared to PCAL [39], WByp provides 12.8% better performance, and full MeDiC provides 21.8% better performance. We observe that while PCAL reduces the amount of cache thrashing, the reduction in thrashing does not directly translate into better performance. We observe that warps in the mostly-miss category sometimes have high reuse, and acquire tokens to access the cache. This causes less cache space to become available for mostly-hit warps, limiting how many of these warps become all-hit. However, when high-reuse warps that possess tokens are mainly in the mostly-hit category (PVC, PVR, SS, and BH), we find that PCAL performs better than WByp.

[8] We confirm that for each application, all kernels have similar speedup values, and that aside from SS and PVC, there are no outliers (i.e., no kernel has a much higher speedup than the other kernels). To verify that harmonic speedup is not swayed greatly by these few outliers, we recompute it for SS and PVC *without* these outliers, and find that the outlier-free speedup is within 1% of the harmonic speedup we report in the paper.

[9] We tune the configuration of each of these previously-proposed mechanisms such that those mechanisms achieve the highest performance results.

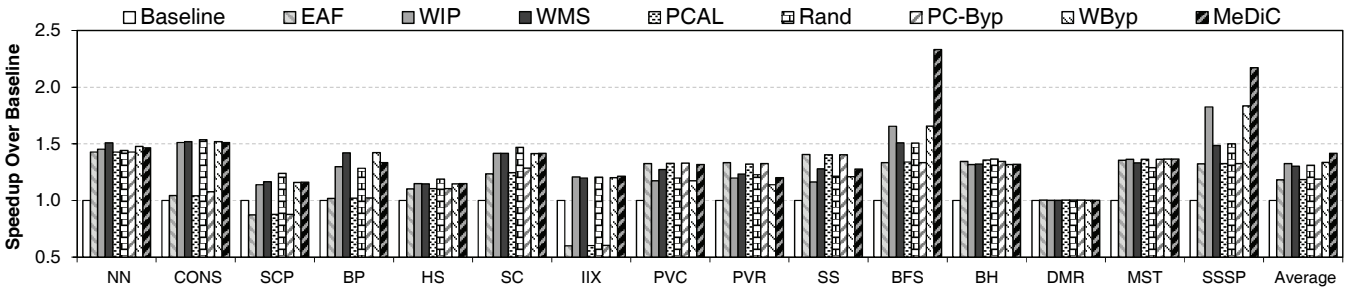

Figure 11: Performance of MeDiC.

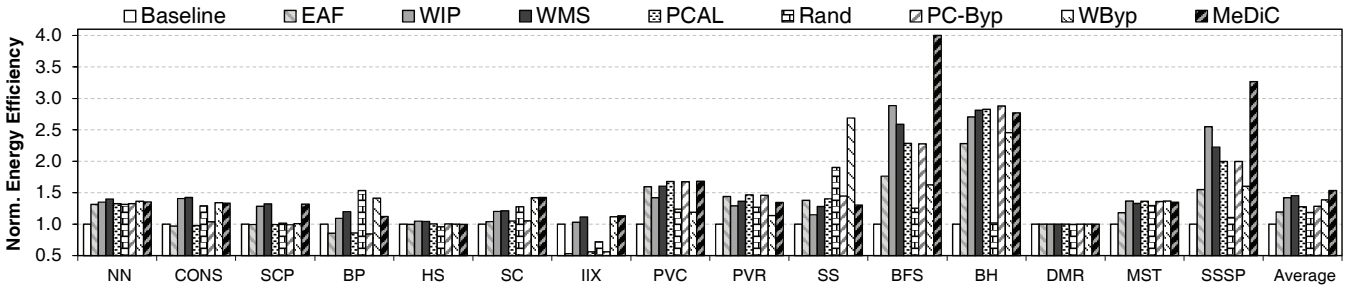

Figure 12: Energy efficiency of MeDiC.

- Compared to Rand,[10] MeDiC performs 6.8% better, because MeDiC is able to make bypassing decisions that do not increase the miss rate significantly. This leads to lower off-chip bandwidth usage under MeDiC than under Rand. Rand increases the cache miss rate by 10% for the kernels of several applications (BP, PVC, PVR, BFS, and MST). We observe that in many cases, MeDiC improves the performance of applications that tend to generate a large number of memory requests, and thus experience substantial queuing latencies. We further analyze the effect of MeDiC on queuing delay in Section 6.3.
- Compared to PC-Byp, MeDiC performs 12.4% better. We observe that the overhead of tracking the PC becomes significant, and that thrashing occurs as two PCs can hash to the same index, leading to inaccuracies in the bypassing decisions.

We conclude that each component of MeDiC, and the full MeDiC framework, are effective. Note that each component of MeDiC addresses the same problem (i.e., memory divergence of threads within a warp) using different techniques on different parts of the memory hierarchy. For the majority of workloads, one optimization is enough. However, we see that for certain high-intensity workloads (BFS and SSSP), the congestion is so high that we need to attack divergence on multiple fronts. Thus, MeDiC provides better average performance than all of its individual components, especially for such memory-intensive workloads.

[10]Note that our evaluation uses an ideal random bypassing mechanism, where we manually select the best individual percentage of requests to bypass the cache for each workload. As a result, the performance shown for Rand is better than can be practically realized.

6.2. Energy Efficiency of MeDiC

MeDiC provides significant GPU energy efficiency improvements, as shown in Figure 12. All three components of MeDiC, as well as the full MeDiC framework, are more energy efficient than all of the other works we compare against. MeDiC is 53.5% more energy efficient than Baseline. WIP itself is 19.3% more energy efficient than EAF. WMS is 45.2% more energy efficient than Baseline, which uses an FR-FCFS memory scheduler [55, 79]. WByp and MeDiC are more energy efficient than all of the other evaluated bypassing mechanisms, with 8.3% and 20.1% more efficiency than PCAL [39], respectively.

For all of our applications, the energy efficiency of MeDiC is better than or equal to Baseline, because even though our bypassing logic sometimes increases energy consumption by sending more memory requests to DRAM, the resulting performance improvement outweighs this additional energy. We also observe that our insertion policy reduces the L2 cache miss rate, allowing MeDiC to be even more energy efficient by not wasting energy on cache lookups for requests of all-miss warps.

6.3. Analysis of Benefits

Impact of MeDiC on Cache Miss Rate. One possible downside of cache bypassing is that the bypassed requests can introduce extra cache misses. Figure 13 shows the cache miss rate for Baseline, Rand, WIP, and MeDiC.

Unlike Rand, MeDiC does not increase the cache miss rate over Baseline for most of our applications. The key factor behind this is WIP, the insertion policy in MeDiC. We observe that WIP on its own provides significant cache miss rate reductions for several workloads (SCP, PVC, PVR, SS, and DMR). For the two workloads (BP and BFS) where

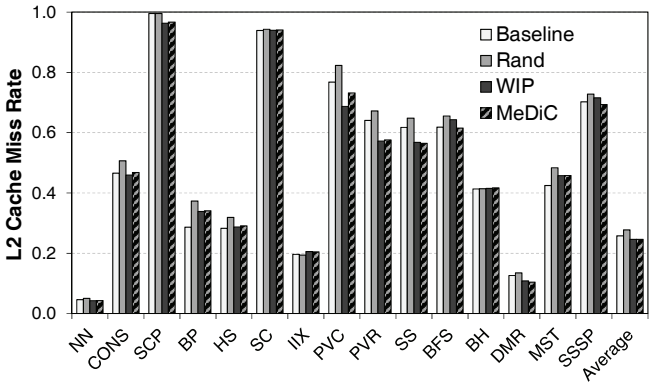

Figure 13: L2 Cache miss rate of MeDiC.

WIP increases the miss rate (5% for BP, and 2.5% for BFS), the bypassing mechanism in MeDiC is able to contain the negative effects of WIP by dynamically tuning how aggressively bypassing is performed based on the change in cache miss rate (see Section 4.2). We conclude that MeDiC does not hurt the overall L2 cache miss rate.

Impact of MeDiC on Queuing Latency. Figure 14 shows the average L2 cache queuing latency for WByp and MeDiC, compared to Baseline queuing latency. For most workloads, WByp reduces the queuing latency significantly (up to 8.7x in the case of PVR). This reduction results in significant performance gains for both WByp and MeDiC.

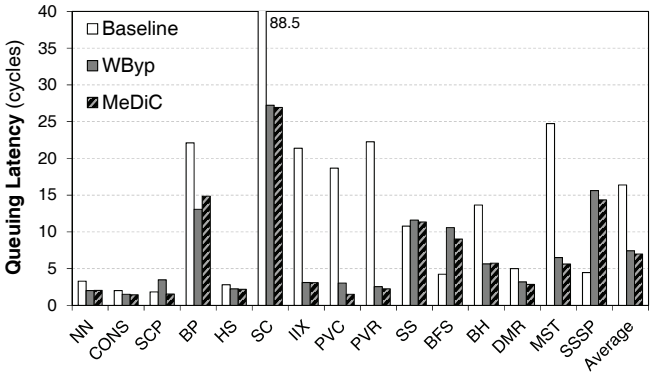

Figure 14: L2 queuing latency for warp-type-aware bypassing and MeDiC, compared to Baseline L2 queuing latency.

There are two applications where the queuing latency increases significantly: BFS and SSSP. We observe that when cache bypassing is applied, the GPU cores retire instructions at a much faster rate (2.33x for BFS, and 2.17x for SSSP). This increases the pressure at each shared resource, including a sharp increase in the rate of cache requests arriving at the L2 cache. This additional backpressure results in higher L2 cache queuing latencies for both applications.

When all three mechanisms in MeDiC (bypassing, cache insertion, and memory scheduling) are combined, we observe that the queuing latency reduces even further. This additional reduction occurs because the cache insertion mechanism in MeDiC reduces the cache miss rate. We conclude that in general, MeDiC significantly alleviates the L2 queuing bottleneck.

Impact of MeDiC on Row Buffer Locality. Another possible downside of cache bypassing is that it may increase the number of requests serviced by DRAM, which in turn can affect DRAM row buffer locality. Figure 15 shows the row buffer hit rate for WMS and MeDiC, compared to the Baseline hit rate.

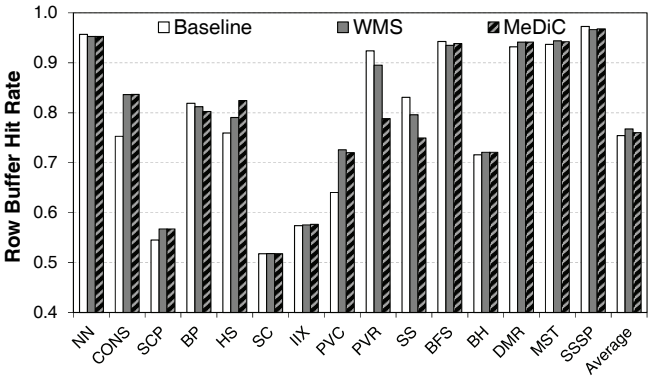

Figure 15: Row buffer hit rate of warp-type-aware memory scheduling and MeDiC, compared to Baseline.

Compared to Baseline, WMS has a negative effect on the row buffer locality of six applications (NN, BP, PVR, SS, BFS, and SSSP), and a positive effect on seven applications (CONS, SCP, HS, PVC, BH, DMR, and MST). We observe that even though the row buffer locality of some applications decreases, the overall performance improves, as the memory scheduler prioritizes requests from warps that are more sensitive to long memory latencies. Additionally, prioritizing requests from warps that send a small number of memory requests (mostly-hit warps) over warps that send a large number of memory requests (mostly-miss warps) allows more time for mostly-miss warps to batch requests together, improving their row buffer locality. Prior work on GPU memory scheduling [2] has observed similar behavior, where batching requests together allows GPU requests to benefit more from row buffer locality.

6.4. Identifying Reuse in GPGPU Applications

While WByp bypasses warps that have low cache utility, it is possible that some cache blocks fetched by these bypassed warps get accessed frequently. Such a frequently-accessed cache block may be needed later by a mostly-hit warp, and thus leads to an extra cache miss (as the block bypasses the cache). To remedy this, we add a mechanism to MeDiC that ensures all high-reuse cache blocks still get to access the cache. The key idea, building upon the state-of-the-art mechanism for block-level reuse [58], is to use a Bloom filter to track the high-reuse cache blocks, and to use this filter to override bypassing decisions. We call this combined design **MeDiC-reuse**.

Figure 16 shows that MeDiC-reuse suffers 16.1% performance degradation over MeDiC. There are two reasons behind this degradation. First, we observe that MeDiC likely implicitly captures blocks with high reuse, as these blocks tend to belong to all-hit and mostly-hit warps. Second, we observe that several GPGPU applications contain access patterns that cause severe false positive aliasing within the Bloom filter used to implement EAF and MeDiC-reuse. This leads to some low reuse cache accesses from mostly-miss and all-miss warps taking up cache space unnecessarily, resulting in cache thrashing. We conclude that MeDiC likely implicitly captures the high reuse cache blocks that are relevant to improving memory divergence (and thus performance). However, there may still be room for other mechanisms that make the best of block-level cache reuse and warp-level heterogeneity in making caching decisions.

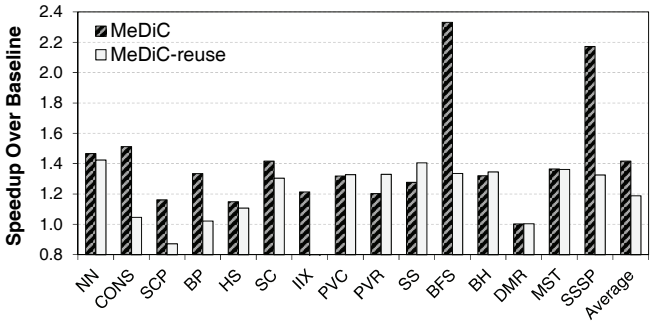

Figure 16: Performance of MeDiC with Bloom filter based reuse detection mechanism from the EAF cache [58].

6.5. Hardware Cost

MeDiC requires additional metadata storage in two locations. First, each warp needs to maintain its own hit ratio. This can be done by adding 22 bits to the metadata of each warp: two 10-bit counters to track the number of L2 cache hits and the number of L2 cache accesses, and 2 bits to store the warp type.[11] To efficiently account for overflow, the two counters that track L2 hits and L2 accesses are shifted right when the most significant bit of the latter counter is set. Additionally, the metadata for each cache line contains two bits, in order to annotate the warp type for the cache insertion policy. The total storage needed in the cache is $2 \times$ *NumCacheLines* bits. In all, MeDiC comes at a cost of 5.1 kB, or less than 1% of the L2 cache size.

To evaluate the trade-off of storage overhead, we evaluate a GPU where this overhead is converted into additional L2 cache space for the baseline GPU. We conservatively increase the L2 capacity by 5%, and find that this additional cache capacity does not improve the performance of any of our workloads by more than 1%. As we discuss in the paper, contention due to warp interference and divergence, and not due to cache capacity, is the root cause behind the performance bottlenecks that MeDiC alleviates. We conclude that MeDiC can deliver significant performance improvements with very low overhead.

[11] We combine the mostly-miss and all-miss categories into a single warp type value, because we perform the same actions on both types of warps.

7. Related Work

To our knowledge, this is the first work that identifies inter-warp memory divergence heterogeneity and exploits it to achieve better system performance in GPGPU applications. Our new mechanism consists of warp-type-aware components for cache bypassing, cache insertion, and memory scheduling. We have already provided extensive quantitative and qualitative comparisons to state-of-the-art mechanisms in GPU cache bypassing [39], cache insertion [58], and memory scheduling [55,79]. In this section, we discuss other related work in these areas.

Hardware-based Cache Bypassing. PCAL is a bypassing mechanism that addresses the cache thrashing problem by throttling the number of threads that time-share the cache at any given time [39] (see Section 5). The key idea of PCAL is to limit the number of threads that get to access the cache. Concurrent work by Li et al. [38] proposes a cache bypassing mechanism that allows only threads with high reuse to utilize the cache. The key idea is to use locality filtering based on the reuse characteristics of GPGPU applications, with only high reuse threads having access to the cache. Xie et al. [76] propose a bypassing mechanism at the thread block level. In their mechanism, the compiler statically marks whether thread blocks prefer caching or bypassing. At runtime, the mechanism dynamically selects a subset of thread blocks to use the cache, to increase cache utilization.

Chen et al. [7,8] propose a combined warp throttling and bypassing mechanism for the L1 cache based on the cache-conscious warp scheduler [56]. The key idea is to bypass the cache when resource contention is detected. This is done by embedding history information into the L2 tag arrays. The L1 cache uses this information to perform bypassing decisions, and only warps with high reuse are allowed to access the L1 cache. Jia et al. propose an L1 bypassing mechanism [22], whose key idea is to bypass requests when there is an associativity stall.

MeDiC differs from these prior cache bypassing works because it uses warp memory divergence heterogeneity for bypassing decisions. We also show in Section 6 that our mechanism implicitly takes reuse information into account.

Software-based Cache Bypassing. Concurrent work by Li et al. [37] proposes a compiler-based technique that performs cache bypassing using a method similar to PCAL [39]. Xie et al. [75] propose a mechanism that allows the compiler to perform cache bypassing for global load instructions. Both of these mechanisms are different from MeDiC in that MeDiC applies bypassing to *all* loads and stores that utilize the shared cache, without requiring additional characterization at the compiler level. Mekkat et al. [43] propose a bypassing mechanism for when a CPU and a GPU share the last level

cache. Their key idea is to bypass GPU cache accesses when CPU applications are cache sensitive, which is not applicable to GPU-only execution.

CPU Cache Bypassing. In addition to GPU cache bypassing, there is prior work that proposes cache bypassing as a method of improving system performance for CPUs [9,13,25,26,52,69]. As they do not operate on SIMD systems, these mechanisms do not (need to) account for memory divergence heterogeneity when performing bypassing decisions.

Cache Insertion and Replacement Policies. Many works have proposed different insertion policies for CPU systems (e.g., [19, 20, 53, 58]). We compare our insertion policy against the Evicted-Address Filter (EAF) [58] in Section 6, and show that our policy, which takes advantage of inter-warp divergence heterogeneity, outperforms EAF. Dynamic Insertion Policy (DIP) [19] and Dynamic Re-Reference Interval Prediction (DRRIP) [20] are insertion policies that account for cache thrashing. The downside of these two policies is that they are unable to distinguish between high-reuse and low-reuse blocks in the same thread [58]. The Bi-modal Insertion Policy [53] dynamically characterizes the cache blocks being inserted. None of these works take warp type characteristics or memory divergence behavior into account.

Memory Scheduling. There are several memory scheduler designs that target systems with CPUs [30, 31, 45, 46, 55, 65, 66, 67], GPUs [5, 77], and heterogeneous compute elements [2, 21, 71]. Memory schedulers for CPUs and heterogeneous systems generally aim to reduce interference across different applications [2,30,31,45,46,55,65,66,67,71].

Chatterjee et al. propose a GPU memory scheduler that allows requests from the same warp to be grouped together, in order to reduce the memory divergence across different memory requests within the same warp [5]. Our memory scheduling mechanism is orthogonal, because we aim to reduce the interference that mostly-hit warps, which are sensitive to high memory latency, experience due to mostly-miss warps. It is possible to combine these two scheduling algorithms, by batching requests (the key mechanism from Chatterjee et al. [5]) for both the high and low priority queues (the key mechanism of our memory scheduler).

Other Ways to Handle Memory Divergence. In addition to cache bypassing, cache insertion policy, and memory scheduling, other works have proposed different methods of decreasing memory divergence [27, 28, 33, 34, 44, 47, 56, 57, 78]. These methods range from thread throttling [27, 28, 33, 56] to warp scheduling [34, 44, 47, 56, 57, 78]. While these methods share our goal of reducing memory divergence, none of them exploit inter-warp heterogeneity and, as a result, are orthogonal or complementary to our proposal. Our work also makes new observations about memory divergence not covered by these works.

8. Conclusion

Warps from GPGPU applications exhibit heterogeneity in their memory divergence behavior at the shared L2 cache within the GPU. We find that (1) some warps benefit significantly from the cache, while others make poor use of it; (2) such divergence behavior for a warp tends to remain stable for long periods of the warp's execution; and (3) the impact of memory divergence can be amplified by the high queuing latencies at the L2 cache.

We propose *Memory Divergence Correction* (MeDiC), whose key idea is to identify memory divergence heterogeneity in hardware and use this information to drive cache management and memory scheduling, by prioritizing warps that take the greatest advantage of the shared cache. To achieve this, MeDiC consists of three *warp-type-aware* components for (1) cache bypassing, (2) cache insertion, and (3) memory scheduling. MeDiC delivers significant performance and energy improvements over multiple previously proposed policies, and over a state-of-the-art GPU cache management technique. We conclude that exploiting inter-warp heterogeneity is effective, and hope future works explore other ways of improving systems based on this key observation.

Acknowledgments

We thank the anonymous reviewers and SAFARI group members for their feedback. Special thanks to Mattan Erez for his valuable feedback. We acknowledge the support of our industrial partners: Facebook, Google, IBM, Intel, Microsoft, NVIDIA, Qualcomm, VMware, and Samsung. This research was partially supported by the NSF (grants 0953246, 1065112, 1205618, 1212962, 1213052, 1302225, 1302557, 1317560, 1320478, 1320531, 1409095, 1409723, 1423172, 1439021, and 1439057), the Intel Science and Technology Center for Cloud Computing (ISTC-CC), and the Semiconductor Research Corporation (SRC). Rachata Ausavarungnirun is partially supported by the Royal Thai Government scholarship.

References

[1] Advanced Micro Devices, Inc., "AMD Graphics Cores Next (GCN) Architecture," http://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf, 2012.

[2] R. Ausavarungnirun *et al.*, "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems," in *ISCA*, 2012.

[3] A. Bakhoda *et al.*, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *ISPASS*, 2009.

[4] M. Burtscher, R. Nasre, and K. Pingali, "A Quantitative Study of Irregular Programs on GPUs," in *IISWC*, 2012.

[5] N. Chatterjee *et al.*, "Managing DRAM Latency Divergence in Irregular GPGPU Applications," in *SC*, 2014.

[6] S. Che *et al.*, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IISWC*, 2009.

[7] X. Chen *et al.*, "Adaptive Cache Management for Energy-Efficient GPU Computing," in *MICRO*, 2014.

[8] X. Chen *et al.*, "Adaptive Cache Bypass and Insertion for Many-Core Accelerators," in *MES*, 2014.

[9] N. Duong *et al.*, "Improving Cache Management Policies Using Dynamic Reuse Distances," in *MICRO*, 2012.

[10] S. Eyerman and L. Eeckhout, "System-Level Performance Metrics for Multiprogram Workloads," *IEEE Micro*, vol. 28, no. 3, 2008.

[11] M. Flynn, "Very High-Speed Computing Systems," *Proc. of the IEEE*, vol. 54, no. 2, 1966.

[12] W. Fung *et al.*, "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," in *MICRO*, 2007.

[13] J. Gaur, M. Chaudhuri, and S. Subramoney, "Bypass and Insertion Algorithms for Exclusive Last-Level Caches," in *ISCA*, 2011.

[14] N. Govindaraju *et al.*, "A Memory Model for Scientific Algorithms on Graphics Processors," in *SC*, 2006.

[15] S. Gupta, H. Gao, and H. Zhou, "Adaptive Cache Bypassing for Inclusive Last Level Caches," in *IPDPS*, 2013.

[16] T. D. Han and T. S. Abdelrahman, "Reducing Branch Divergence in GPU Programs," in *GPGPU*, 2011.

[17] B. He *et al.*, "Mars: A MapReduce Framework on Graphics Processors," in *PACT*, 2008.

[18] S. Hong and H. Kim, "An Analytical Model for a GPU Architecture with Memory-Level and Thread-Level Parallelism Awareness," in *ISCA*, 2009.

[19] A. Jaleel *et al.*, "Adaptive Insertion Policies for Managing Shared Caches," in *PACT*, 2008.

[20] A. Jaleel *et al.*, "High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP)," in *ISCA*, 2010.

[21] M. K. Jeong *et al.*, "A QoS-Aware Memory Controller for Dynamically Balancing GPU and CPU Bandwidth Use in an MPSoC," in *DAC*, 2012.

[22] W. Jia, K. A. Shaw, and M. Martonosi, "MRPB: Memory Request Prioritization for Massively Parallel Processors," in *HPCA*, 2014.

[23] A. Jog *et al.*, "Orchestrated Scheduling and Prefetching for GPGPUs," in *ISCA*, 2013.

[24] A. Jog *et al.*, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance," in *ASPLOS*, 2013.

[25] T. L. Johnson, M. C. Merten, and W. W. Hwu, "Run-Time Spatial Locality Detection and Optimization," in *MICRO*, 1997.

[26] T. L. Johnson and W. W. Hwu, "Run-Time Adaptive Cache Hierarchy Management via Reference Analysis," in *ISCA*, 1997.

[27] O. Kayıran *et al.*, "Neither More Nor Less: Optimizing Thread-Level Parallelism for GPGPUs," in *PACT*, 2013.

[28] O. Kayıran *et al.*, "Managing GPU Concurrency in Heterogeneous Architectures," in *MICRO*, 2014.

[29] Khronos OpenCL Working Group, "The OpenCL Specification," http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf, 2008.

[30] Y. Kim *et al.*, "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," in *HPCA*, 2010.

[31] Y. Kim *et al.*, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *MICRO*, 2010.

[32] Y. Kim *et al.*, "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," in *ISCA*, 2012.

[33] H.-K. Kuo, B. C. Lai, and J.-Y. Jou, "Reducing Contention in Shared Last-Level Cache for Throughput Processors," *ACM TODAES*, vol. 20, no. 1, 2014.

[34] S.-Y. Lee and C.-J. Wu, "CAWS: Criticality-Aware Warp Scheduling for GPGPU Workloads," in *PACT*, 2014.

[35] S.-Y. Lee and C.-J. Wu, "Characterizing GPU Latency Hiding Ability," in *ISPASS*, 2014.

[36] J. Leng *et al.*, "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *ISCA*, 2013.

[37] A. Li *et al.*, "Adaptive and Transparent Cache Bypassing for GPUs," in *SC*, 2015.

[38] C. Li *et al.*, "Locality-Driven Dynamic GPU Cache Bypassing," in *ICS*, 2015.

[39] D. Li *et al.*, "Priority-Based Cache Allocation in Throughput Processors," in *HPCA*, 2015.

[40] E. Lindholm *et al.*, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, vol. 28, no. 2, 2008.

[41] W. Liu, W. Muller-Wittig, and B. Schmidt, "Performance Predictions for General-Purpose Computation on GPUs," in *ICPP*, 2007.

[42] L. Ma and R. Chamberlain, "A Performance Model for Memory Bandwidth Constrained Applications on Graphics Engines," in *ASAP*, 2012.

[43] V. Mekkat *et al.*, "Managing Shared Last-Level Cache in a Heterogeneous Multicore Processor," in *PACT*, 2013.

[44] J. Meng, D. Tarjan, and K. Skadron, "Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance," in *ISCA*, 2010.

[45] O. Mutlu and T. Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *MICRO*, 2007.

[46] O. Mutlu and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems," in *ISCA*, 2008.

[47] V. Narasiman *et al.*, "Improving GPU Performance via Large Warps and Two-Level Warp Scheduling," in *MICRO*, 2011.

[48] NVIDIA Corp., "CUDA C/C++ SDK Code Samples," http://developer.nvidia.com/cuda-cc-sdk-code-samples, 2011.

[49] NVIDIA Corp., "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf, 2011.

[50] NVIDIA Corp., "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2012.

[51] NVIDIA Corp., "CUDA C Programming Guide," http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2015.

[52] T. Piquet, O. Rochecouste, and A. Seznec, "Exploiting Single-Usage for Effective Memory Management," in *ACSAC*, 2007.

[53] M. K. Qureshi *et al.*, "Adaptive Insertion Policies for High Performance Caching," in *ISCA*, 2007.

[54] B. R. Rau, "Pseudo-randomly Interleaved Memory," in *ISCA*, 1991.

[55] S. Rixner *et al.*, "Memory Access Scheduling," in *ISCA*, 2000.

[56] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *MICRO*, 2012.

[57] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-Aware Warp Scheduling," in *MICRO*, 2013.

[58] V. Seshadri *et al.*, "The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing," in *PACT*, 2012.

[59] J. Sim *et al.*, "A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications," in *PPoPP*, 2012.

[60] J. Sim *et al.*, "A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch," in *MICRO*, 2012.

[61] I. Singh *et al.*, "Cache Coherence for GPU Architectures," in *HPCA*, 2013.

[62] SiSoftware, "Benchmarks : Measuring GP (GPU/APU) Cache and Memory Latencies," http://www.sisoftware.net, 2014.

[63] B. J. Smith, "A Pipelined, Shared Resource MIMD Computer," in *ICPP*, 1978.

[64] J. A. Stratton *et al.*, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," Univ. of Illinois at Urbana-Champaign, Tech. Rep. IMPACT-12-01, March 2012.

[65] L. Subramanian *et al.*, "The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost," in *ICCD*, 2014.

[66] L. Subramanian *et al.*, "The Blacklisting Memory Scheduler: Balancing Performance, Fairness and Complexity," *arXiv CoRR*, 2015.

[67] L. Subramanian *et al.*, "MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems," in *HPCA*, 2013.

[68] J. E. Thornton, "Parallel Operation in the Control Data 6600," *AFIPS FJCC*, 1964.

[69] G. Tyson *et al.*, "A Modified Approach to Data Cache Management," in *MICRO*, 1995.

[70] Univ. of British Columbia. GPGPU-Sim GTX 480 Configuration. http://dev.ece.ubc.ca/projects/gpgpu-sim/browser/v3.x/configs/GTX480.

[71] H. Usui *et al.*, "SQUASH: Simple QoS-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators," *arXiv CoRR*, 2015.

[72] N. Vijaykumar *et al.*, "A Case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps," in *ISCA*, 2015.

[73] H. Wong *et al.*, "Demystifying GPU Microarchitecture Through Microbenchmarking," in *ISPASS*, 2010.

[74] P. Xiang, Y. Yang, and H. Zhou, "Warp-Level Divergence in GPUs: Characterization, Impact, and Mitigation," in *HPCA*, 2014.

[75] X. Xie *et al.*, "An Efficient Compiler Framework for Cache Bypassing on GPUs," in *ICCAD*, 2013.

[76] X. Xie *et al.*, "Coordinated Static and Dynamic Cache Bypassing for GPUs," in *HPCA*, 2015.

[77] G. Yuan, A. Bakhoda, and T. Aamodt, "Complexity Effective Memory Access Scheduling for Many-Core Accelerator Architectures," in *MICRO*, 2009.

[78] Z. Zheng, Z. Wang, and M. Lipasti, "Adaptive Cache and Concurrency Allocation on GPGPUs," *IEEE CAL*, 2014.

[79] W. K. Zuravleff and T. Robinson, "Controller for a Synchronous DRAM That Maximizes Throughput by Allowing Memory Requests and Commands to Be Issued Out of Order," Patent US 5,630,096, 1997.