WarpPool: Sharing Requests with Inter-Warp Coalescing for Throughput Processors

John Kloosterman, Jonathan Beaumont, Mick Wollman, Ankit Sethia,
Ron Dreslinski, Trevor Mudge, Scott Mahlke

Computer Engineering Lab

University of Michigan, Ann Arbor, MI

{jklooste, jbbeau, mwollman, asethia, rdreslinki, tnm, mahlke}@umich.edu

ABSTRACT

Although graphics processing units (GPUs) are capable of high compute throughput, their memory systems need to supply the arithmetic pipelines with data at a sufficient rate to avoid stalls. For benchmarks that have divergent access patterns or cause the L1 cache to run out of resources, the link between the GPU's load/store unit and the L1 cache becomes a bottleneck in the memory system, leading to low utilization of compute resources. While current GPU memory systems are able to coalesce requests between threads in the same warp, we identify a form of spatial locality between threads in multiple warps. We use this locality, which is overlooked in current systems, to merge requests being sent to the L1 cache. This relieves the bottleneck between the load/store unit and the cache, and provides an opportunity to prioritize requests to minimize cache thrashing. Our implementation, *WarpPool*, yields a 38% speedup on memory throughput-limited kernels by increasing the throughput to the L1 by 8% and the reducing the number of L1 misses by 23%. We also demonstrate that *WarpPool* can improve GPU programmability by achieving high performance without the need to optimize workloads' memory access patterns. A Verilog implementation including place-and-route shows *WarpPool* requires 1.0% added GPU area and 0.8% added power.

Categories and Subject Descriptors

C.1.4 [Computer Systems Organization]: Architectures—*Parallel architectures*

General Terms

Design, Performance

Keywords

GPGPU, memory coalescing, memory divergence

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

MICRO-48 December 05 - 09, 2015, Waikiki, HI, USA

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 978-1-4503-4034-2/15/12\$15.00

DOI: <http://dx.doi.org/10.1145/2830772.2830830>

1. INTRODUCTION

GPUs are throughput processors designed to hide memory latency using multithreading. During the time some threads on the GPU are waiting for long-latency memory operations, others can be scheduled to do computation. However, in order for this strategy to keep the utilization of the arithmetic units high, data needs to come from the memory system fast enough to maintain a set of threads that are ready to do computation. Therefore, keeping the arithmetic units utilized depends on the throughput of the memory system matching the throughput of the compute pipelines.

However, previous studies have shown that for many benchmarks, the throughput of the global memory system is not adequate to keep the GPU from stalling. This can be due to saturated DRAM bandwidth [26], limited L1 cache resources such as MSHRs and cache sets [9], or small per-thread cache capacity [23]. To achieve high throughput despite bandwidth limitations in the global memory system, the GPU has a memory hierarchy that merges requests to the same cache lines to reduce traffic.

On a GPU, threads in a 32-wide warp execute in lock-step, but the threads in one load can generate accesses to many different cache lines. A *memory coalescer* is the first unit in the memory hierarchy, responsible for combining memory accesses to the same cache line made by the 32 threads in a warp. This unit is effective because spatial locality is expressed in a GPU through nearby threads accessing the same cache lines [8]. Combining requests early in the memory pipeline is better for performance and energy efficiency than sending duplicate requests to the higher-level caches and DRAM.

However, the coalescer can become the bottleneck in memory system throughput, which happens under memory divergence, where the threads in a warp request more than one cache line in a load or store instruction. Because the L1 can only service one request per cycle, up to 32 requests must be serialized over 32 cycles instead of being serviced simultaneously. Another throughput problem is caused by limited cache resources, where the memory system stalls when the cache cannot allocate a resource like an MSHR to issue another outstanding miss. In both of these cases, the GPU becomes underutilized because the memory system cannot supply data fast enough to keep the arith-

metic units busy.

The current memory coalescer is limited to merging requests between threads in the same warp. However, we show that spatial locality is not limited to threads in the same warp, so allowing the coalescer to merge requests from threads in multiple warps would allow for a greater reduction in requests. If the coalescer were able to merge requests across warps using this inter-warp spatial locality before they reach the L1 cache, it would increase the effective bandwidth of the cache by relieving the one access per cycle bottleneck. During times when L1 resources are at a premium, it would enable resources to service requests from as many warps as possible. As well, having requests from multiple warps in scope allows the coalescer to act as a gatekeeper to the L1 cache and reduce thrashing.

We propose a novel memory coalescer, *WarpPool*, which is able to find inter-warp spatial locality. It increases the effective throughput to the L1, uses cache resources more efficiently, and reduces cache thrashing by prioritizing some warps' access to the cache. After a first level of coalescing to find intra-warp spatial locality, requests are inserted into inter-warp coalescing queues that merge requests from multiple warps. Doing both intra-warp and inter-warp coalescing reduces the number of requests that need to be made to the L1 cache. Because the requests exiting the coalescer now fetch data for more than one load, more than one load's requests can enter the coalescer per cycle, which will keep throughput high under memory divergence. When cache resources are scarce, requests will build up in the inter-warp queues, which will increase the amount of inter-warp coalescing and enable requests using cache resources to service multiple loads. Furthermore, requests from the inter-warp queues can be selected to exit to the cache in an order that enhances temporal locality in the cache.

In this paper, we make the following contributions:

1. We characterize a class of inter-warp spatial locality that current coalescers are unable to capture. We show that using this locality to merge requests would remove the bottleneck in a class of workloads limited by memory system throughput.
2. We propose *WarpPool*, an inter-warp memory coalescer that is able to merge requests between warps to convert this locality into increased bandwidth to the L1 cache and more efficient utilization of cache resources. It is also able to prioritize warps' access to the L1 cache, which reduces cache thrashing.
3. We implement *WarpPool* in GPGPU-sim [1] and achieve a 38% geometric mean speedup across a set of memory throughput-limited kernels. *WarpPool* increases the throughput to the L1 cache by 8% and reduces the number of L1 misses by 23%.
4. We evaluate a case study demonstrating that *WarpPool* improves GPU programmability by achieving a 2.0 $\times$ speedup on straightforward code for which manual optimizations give a 2.6 $\times$ speedup.

Figure 1: Diagram of GTX 480 memory system. Each SM has a load/store unit with a memory coalescer which sends requests to a private L1 cache. Requests to shared L2 caches and DRAM partitions are sent over an interconnect.

Figure 2: Memory request reduction for the spmv benchmark, showing the number of requests remaining at each level of the memory hierarchy as the coalescer and caches convert locality into fewer requests.

2. BACKGROUND AND MOTIVATION

2.1 Background

GPUs are made up of multiple streaming multiprocessors (SMs), in Nvidia terminology. Inside each SM, warp schedulers select threads with ready operands and issue them to functional units. Threads are scheduled as a group of 32, called a warp. Threads in a warp execute the same instructions in lockstep, but can supply different values as inputs to those instructions.

The load/store unit (LSU) is the functional unit responsible for loads, stores, and memory barrier instructions. Like the other functional units, it is scheduled a warp of 32 threads at a time. There are multiple memory spaces in the GPU, and some, like the *shared* scratchpad memory, have enough throughput to service a different request from each of the 32 threads in a warp each cycle. However, all the data sent to the GPU for computation needs to be loaded from global memory, which can only process one request per cycle, and the final results of the GPU's computations also need to be stored in global memory for transfer back to the CPU. Global memory is backed by DRAM and implemented using a cache hierarchy similar to a CPU memory system, as shown in Figure 1.

The memory coalescer conserves bandwidth by merging requests to the same cache line made by threads in a warp, taking advantage of spatial locality between the threads in a warp to reduce the number of requests.

Figure 3: Memory throughput-limited workloads overwhelm the coalescing system either through generating many requests with memory divergence or by causing cache resource shortages by saturating memory bandwidth. The kernel number is its sequence in the kernel execution order in the benchmark.

Figure 4: A portion of the execution of the GEMM benchmark. The solid line is the number of memory instructions waiting to be scheduled. The background is grey when the L1 cache resources are full. The bars at the bottom show when arithmetic is scheduled. For the cache resource-bound benchmarks, there is little overlap of computation with cache resource stalls.

When the warp scheduler sends a warp to the LSU, a load or store instruction contains 32 addresses, one for each lane in the warp. The memory coalescer determines which of the 32 addresses point to the same cache line and merges requests to the same line together. The coalescer is effective at reducing requests because spatial locality is often expressed in a GPU as nearby threads requesting nearby data.

Because there is high demand on the global memory system for limited bandwidth, each stage of the memory system is designed to conserve bandwidth by using locality to merge requests. Figure 2 shows the units in the GPU memory pipeline along with the percentage of requests that make it through each stage for the *spmv* benchmark, representative of a set of memory divergent benchmarks detailed in Section 2.2. The memory coalescer reduces the number of requests by over 40%, only 17% of requests get past the L1 to the L2, and 10% of requests reach DRAM. Since the bandwidth decreases as requests go further down the pipeline and the energy cost to make an access increases at each stage, it is advantageous to merge requests as early as possible.

Since GPU cache lines are 128 bytes, designed so that each of 32 threads in a warp can request a 4-byte word, all 32 requests in many loads and stores map to a single cache line. However, each request may map to a num-

ber of different cache lines, called *memory divergence*. Under memory divergence, the coalescer is unable to reduce the number of requests, and must make up to 32 serialized requests to the L1 cache in the worst case. The L1 is only able to service one request per cycle, so a divergent load or store takes one cycle per distinct cache line to complete.

2.2 Oversubscription of L1 Bandwidth

Although the interface between the LSU and the L1 cache is the fastest link in the global memory system, it is the link that must accept the largest number of requests relative to its output throughput (32 to 1). The memory coalescer is responsible for matching the large throughput in to the small throughput out. However, under two common scenarios it is not able to do so.

Figure 3 shows the average number of cache lines per memory operation and the average number of waiting memory operations for kernels from the Parboil [27], Rodinia [3], and PolyBench [6] benchmark suites. These kernels were selected because they had more memory instructions ready to issue than the LSU could process for over 90% of their execution time. These instructions could be executed by the LSU if the memory system had higher throughput, so these are the workloads for which improving the throughput of the memory system has the potential to improve performance.

The first category of workloads are **memory divergent**, where the intra-warp coalescer requires an average of 13 cycles to send a warp’s load or store instruction to the L1 because each thread requested a different cache line. This can be as high as 32 for *ATAX_1*. The L1 cache needs to service up to 32 times as many cache line requests for these workloads per load or store instruction, creating a bandwidth bottleneck at the L1 cache. The baseline memory coalescer is unable to merge many requests for these workloads, so it is unable to handle the bandwidth demand. A better coalescer would be able to reduce the effective level of divergence by finding locality between multiple divergent loads before sending them to the L1 cache.

The second category are **bandwidth-limited**. These workloads have low memory divergence, but due to high miss rate or high memory intensity saturate DRAM bandwidth and cause the L1 cache to run out of resources like MSHRs. Figure 4 shows how the execution of the GEMM benchmark, with 8.9 average waiting mem-

Figure 5: Many memory throughput-limited kernels show a large degree of inter-warp spatial locality. Each cell corresponds to a kernel, and inside each cell the window size gets larger from left (baseline window of only intra-warp coalescing) to right (window of 128 requests after intra-warp coalescing). As the window size increases the number of requests that must be sent to the cache decreases.

ory operations, follows a cyclical pattern. Most of the time, the cache has no resources to accept a new request, which causes ready memory instructions to back up and no computation to be done. When data comes back from the higher levels of the memory system, arithmetic begins to execute and new memory instructions are issued until the cache stalls again. A better coalescer would be able to make better use of limited cache resources by continuing to accept requests while the L1 cache’s resources are full, using the time the cache is stalled to reduce the number of requests, and then making better use of the time the cache is accepting new requests by having each request service multiple load or store instructions. This way, more arithmetic can be done per request the L1 is able to service.

The benefits of a coalescer that is able to merge more requests propagate down to the rest of the memory system, where performing the merging is more expensive. The more requests that can be removed early in the pipeline, the fewer requests the later stages in the pipeline need to service.

2.3 Increasing Coalescing Window Size

The baseline coalescer can only merge requests between threads in one warp. To find more opportunities, the coalescer needs to look between requests made by different warps across multiple load instructions, increasing its window from the threads in one warp to requests from multiple warps.

Figure 5 shows the relative reduction in memory requests made by the memory throughput-limited kernels that could be achieved by increasing the coalescer’s window size to multiple warps, analyzed using a trace of global memory requests made to the L1. The window size increases from 0, equivalent to doing only intra-warp coalescing, to a window of 128 cache line requests made to the L1. The kernels without inter-warp locality have been split into a new category of **cache-limited** kernels that exhibit intra-warp temporal locality.

The **memory divergent inter-warp** workloads show inter-warp locality with larger window sizes. Because divergence creates many requests, locality begins to show up only at larger window sizes: if each load generates 32 requests, the window size of 128 is a window of 4 load instructions. Patterns like indirect accesses (**spmv**) and large memory strides (**SYRK**) create divergence and inter-warp locality. For these workloads, the spatial lo-

cality can only be found by having a larger window size than one warp, because the only spatial locality is between warps.

The **bandwidth-limited inter-warp** workloads exhibit a high degree of inter-warp locality. The locality is caused by the memory access patterns in these workloads, such as accessing a matrix column-wise and row-wise, but contiguously inside a warp (**GEMM**) or repetition of the same accesses in an inner loop across all threads (**streamcluster**). In these workloads, there is spatial locality both within warps and between warps. A coalescer with a larger window size will be able to find more coalescing opportunities in them.

A third category, **cache-sensitive** workloads, have low inter-warp spatial locality but exhibit temporal locality inside warps. Much previous work has focused on this category of workloads, by limiting access to the cache through scheduling [23] or bypassing [9]. A coalescer with a larger window size will not be able to find more opportunity to merge requests, but it will have more scope to prioritize requests. It will be able to choose a request from among multiple warps to send to the cache, using that ability to prioritize certain warps’ access to the cache.

Therefore, by increasing the window size in which the coalescer can merge requests together to include requests made by different warps, the coalescer is able to increase effective bandwidth to the L1 cache and stop this link in the memory system from becoming the bottleneck for the throughput of both the memory system and the entire GPU. A coalescer able to merge across warps will be able find spatial locality that is out of the scope of the current intra-warp coalescer, and has the opportunity to help even workloads without spatial locality.

To turn better coalescing into speedup, the improved coalescer will need to address the reasons why L1 bandwidth limits performance for each category of workload. For **memory divergent inter-warp** workloads, the coalescer will need to serialize divergent memory operations from more than one warp in parallel. For the **bandwidth-limited inter-warp** workloads, the coalescer will need to buffer requests received when cache resources are full. For the **cache-limited** workloads, it will need to leverage the coalescing window to schedule requests in a way that reduces cache thrashing.

Figure 6: Diagram of the *WarpPool* system.

3. WARPPool DESIGN

3.1 Overview

The *WarpPool* system creates a window in which requests from multiple warps can be coalesced, in order to capture inter-warp spatial locality. Requests are inserted into this window after intra-warp coalescing, and requests removed from the window are sent to the L1 cache. In order for the inter-warp coalescing window to yield speedup, it needs to be supported by an intra-warp coalescing pipeline in the front end that can insert requests into the window at the same rate as they drain out. On the other end of the pipeline, selecting requests from the window to send to the cache needs to be done in a way that preserves intra-warp temporal locality, since reordering memory requests can easily cause cache thrashing.

A high-throughput intra-warp coalescing pipeline, a window to capture inter-warp spatial locality, and a selection policy that preserves intra-warp locality make up the substance of the *WarpPool* system, shown in Figure 6. Instruction queues ① hold load and store instructions issued by the scheduler, to prioritize access to the coalescer and cache. These issue into the intra-warp coalescers ②, which merge requests to the same cache line inside a warp. Inter-warp coalescing queues ③ combine requests between warps to find new inter-warp coalescing opportunities. Then, the request selector ④ determines which requests exit the inter-warp queues in a way that maximizes coalesces while maintaining intra-warp spatial locality.

In the first stage, *WarpPool* queues load and store instructions. Loads and stores are inserted into a queue ① based on which warp they were issued from, which allows *WarpPool* to prioritize some warps’ access to the intra-warp coalescers. In the next stage, *WarpPool* uses multiple intra-warp coalescers ② to capture intra-warp spatial locality. These coalescers are identical to the baseline intra-warp coalescer, but there are more of them so that multiple divergent loads and stores can be serialized in parallel.

After intra-warp coalescing, *WarpPool* captures spatial locality between threads from different warps using inter-warp coalescing queues ③. Requests are mapped to a coalescing queue based on their address, similar to the way that requests are mapped to cache sets. Requests for the same cache line from different load in-

structions are matched against each other and merged into one request to be sent to the cache.

Requests stay in the queues until sent by a selector to the L1 cache ④. The order that requests are sent to the cache is crucial for maintaining the intra-warp temporal locality exhibited by many benchmarks. *WarpPool* leverages having coalescing window containing many requests to schedule requests in a way that maximizes the number of coalesces and prioritizes access to the L1.

After loads return from the L1, the data from the cache line needs to be written to the registers of the threads that requested the line. *WarpPool* maintains metadata about the mapping of words in the cache lines to threads in a load (stage ⑤ in Figure 6), which allows a request made on behalf of multiple loads to be de-coalesced and written back to the correct registers. By taking advantage of common mapping patterns, this metadata can be kept to a manageable size, as explained in Section 3.6. *WarpPool* uses the crossbar already present in the GPU load-store unit to move the data from the cache line to the threads for writeback. Although *WarpPool* adds more stages to the GPU’s memory pipeline¹, there are enough warps to hide this added latency with multithreading.

In the following sections, each part of the *WarpPool* system is described in greater detail. This is followed by a discussion of how metadata mapping data to threads is stored, how stores are handled by the system, and how memory consistency is maintained even as loads and stores are reordered in the coalescing queues.

3.2 Instruction Queues

Queues at the front of the pipeline allow *WarpPool* to prioritize access to the coalescing resources, which improves cache locality. These queues hold load and store instructions before address generation, so as to avoid storing 128 bytes of addresses. Loads and stores are mapped to one of these queues based on which warp they were scheduled from, with lower warp IDs mapped to queues with higher priority. In the configuration evaluated in Section 4.2, there are 16 of these queues with 3 warps mapped to each queue. The queues are needed over and above the scheduler for priority because in cases where the LSU has been stalled for several cycles then becomes available again, the GTO scheduler will

¹The implementation in Section 3.9 has 6 pipeline stages.

schedule from its current warp rather than the oldest warp. Using these queues, *WarpPool* has more control of which warp can issue memory instructions, allowing it to prioritize warps to improve temporal locality. We use a fixed priority order for the queues, which was proved effective by Jia et. al [9].

3.3 Intra-Warp Coalescers

The intra-warp coalescers merge requests in the same warp to the same cache lines. Intra-warp coalescing is the bottleneck for the memory divergent benchmarks, because it takes multiple cycles for each request to exit the coalescer. To relieve this bottleneck, *WarpPool* has multiple intra-warp coalescers. Only one request for one cache line can exit an intra-warp coalescer per cycle, but each coalescer can issue requests in parallel. The design of the intra-warp coalescers is detailed in [16].

When an intra-warp coalescer is ready to accept a new instruction, a load or store is popped from the highest priority instruction queue. Before the instruction moves to the coalescer, its registers are read and addresses are generated. Once in the intra-warp coalescer, one cache line per cycle is issued. For loads, the intra-warp coalescer issues into the inter-warp coalescer, and for stores, it issues directly to the cache. Metadata about which threads in the warp request which parts of the cache line are passed along with the request.

3.4 Inter-Warp Coalescing Queues

The inter-warp coalescing queues make up the window in which requests from different warps are merged. Requests coming from the intra-warp pipelines are mapped to one of many queues based on a subset of bits from their address. When a request is inserted into one of these queues, its cache line is matched against cache lines already in the queue, and requests to the same line are merged together.

Figure 7 shows the structure of the inter-warp coalescing queues. Inside each queue are two tags identifying a cache line with slots underneath that accumulate requests to that cache line. For each merged request, the queues need to track which warp they are from, which load instruction in that warp needs the data, and metadata about how to map data in the cache line to the threads in that warp. When a request is inserted into a queue, a lookup is done against the tags and the request is inserted under a tag that matches. If no tag matches, a new tag will be allocated if free. Note that in the evaluated configuration, there are only two tags per queue, so only two tags will be searched per insertion. Previous work has found that the larger cache tag lookups (as part of GPU cache power) make up a very small fraction of total GPU power [16].

The total number of tags across the queues is the window size across which requests can be merged. Structuring these tags as two per queue with many queues minimizes the number of tag lookups that need to be done and reduces the number of times intra-warp coalescers attempt to insert into the same queue. Addresses are mapped to a queue based on bits in their address,

Figure 7: A diagram of the inter-warp coalescing queues. Requests exiting the intra-warp coalescers are merged with other requests to the same cache line in these queues.

using the method described in [17]. The same bits are also used to map addresses to cache sets, and matching the two hashes simplifies future designs where *WarpPool* issues to a cache banked by sets.

3.5 Request Selector

Requests remain in the inter-warp queues until they are selected to be sent to the L1 cache. There are three competing concerns that the request selection logic must balance: first, keeping requests in the coalescing queues for longer leads to more coalescing, because requests can only merge with requests in the queues. However, the second concern is latency, ensuring requests do not stay in the queues so long that the coalescer adds latency to misses. Third, the order the requests are sent to the cache must preserve temporal locality inside warps, so that reordering requests in the inter-warp queues does not lead to cache thrashing.

For most benchmarks, the most effective strategy is to drain the *oldest* request in the queues. This is implemented with a circular queue that saves the order requests were inserted into the queues. Choosing the oldest request balances the three concerns: it keeps requests in the queues for as long as possible without adding latency, and it follows the order produced by the GTO scheduler and the queues in front of the intra-warp coalescers, both optimized to prioritize access to the cache.

For extremely cache-sensitive benchmarks, an alternate strategy that prioritizes one warp’s requests, the *warp ID* policy, leads to a lower miss rate. Because the inter-warp queues store accesses to the cache at the granularity of individual requests rather than load instructions, *WarpPool* can prioritize requests at a finer granularity than the warp scheduler can, similar to the opportunity exploited by [9]. Being able to schedule requests rather than instructions allows newly issued requests from the warp with access to the cache to interrupt requests from instructions issued by other warps, leading to fewer requests by different warps between accesses by the prioritized warp and causing less cache thrashing.

WarpPool uses performance counters to determine when to switch selection policies. The benchmarks begin execution in *oldest* mode. During quanta of 100,000 cycles, each SM tracks the L1 miss rate. If the miss rate is above 99% during a quantum, *WarpPool* toggles the policy for the next quantum. This discovers whether one of the strategies causes thrashing for a benchmark and switches if it does. Quanta of 100,000 were chosen to be sufficiently long for changes in the miss rate to stabilize before a new policy decision is made.

3.6 Metadata Tracker

Because each thread in a SIMD load can request a different word in a cache line, the LSU needs to keep metadata about which words in the request’s cache line map to which thread in each outstanding load instruction. This way the LSU has enough information to write the correct word to the correct thread’s registers when the data returns. To move the data to the correct thread, the baseline load-store unit contains a crossbar that is able to move any word in the cache line to any thread. In *WarpPool*, this metadata must be stored for every request in the inter-warp coalescing queues as well for every request in the L1 cache’s MSHRs. Because the metadata needs to be stored for so many requests, minimizing the size of the metadata is important to keep overhead reasonable. To do this, common mappings of threads to words in a cache line are recognized by the intra-warp coalescers and encoded in fewer bits.

There are four common mapping patterns that can be encoded by *WarpPool*, shown in Figure 8. In the *consecutive* mapping, the threads map 1-to-1 with the words in the cache line. In the *broadcast* mapping, every thread requests the same word. In the *single* mapping, one thread in a warp requests one word from a cache line. In the *range* mapping, a consecutive subset of the threads request a consecutive subset of the words in a cache line. Figure 9 shows what percentage of the memory requests across the benchmarks use each of these mappings. There are more *single* mappings than the other types because one load can generate up to 32 *single* mappings, each requesting one word for one thread, whereas the other types of requests are for multiple words for multiple threads. Each of these encodings requires a maximum of 10 bits, as opposed to the 320 bits otherwise needed. In the case where none of these mappings apply, an 8-entry *thread map table* entry is allocated to store the mapping. *WarpPool*’s intra-warp coalescers have an added pipeline stage to identify a mapping pattern and allocate a table entry, if necessary.

Metadata needs to be stored for all requests sent to the memory system, including the requests in MSHRs. As requests exit the coalescer to the cache, their metadata is stored in the *MSHR metadata table* until the data comes back from the cache.

3.7 Writeback

When the data for a request comes back from the cache, the writeback unit uses the metadata in the table

Figure 8: Common mapping patterns from cache line words to threads

Figure 9: Relative occurrence of mapping patterns in benchmarks

to map the data to the correct registers. If a map table entry was allocated for request, it is released at write-back. The mappings that do not require the map table can use simple selectors to move the data, whereas the mappings from the map table require the pre-existing crossbar. The registers for one warp can be written back to the registers at one time. Data returning from the cache along with its coalescing metadata is stored in a 2-entry queue as the data for each warp is sent to the register file. The cache stalls when this buffer is full.

3.8 Stores and Memory Consistency

WarpPool only performs inter-warp coalescing on loads. Stores progress through the instruction queues and intra-warp coalescers like loads, but instead of issuing into the coalescing queues, they issue directly to the L1 cache. Coalescing stores would require buffering the 128 bytes of data to be stored, and because GPU L1 caches are no-write-allocate, stores can be issued without concern about destroying any intra-warp locality. Each cycle, a selector chooses whether to allow a load from the coalescing queues or a store directly from an intra-warp coalescer to drain to the L1 cache. In order to reduce the miss penalty, this selector prioritizes loads.

CUDA has a weak memory consistency model where there are no inter-warp consistency guarantees except as provided for atomics and barriers. Inside a warp, the baseline GTO warp scheduler always sends the loads and stores in program order, so any memory reordering in *WarpPool* needs to guarantee the observed behavior is the same as executing the loads and stores in one thread in program order. Previous work has maintained consistency either by flushing the reordering buffers be-

Type	Entry Size	Entries	Total Storage	Dynamic Power	Static Power	Total Power
Instruction queues	37 bits	128	592 bytes	41.4 mW	0.4 mW	41.8 mW
Intra-warp coalescers	256 bytes	2	512 bytes	24.1 mW	0.2 mW	24.3 mW
Inter-warp coalescing queues	124 bits	64	992 bytes	43.8 mW	0.5 mW	44.3 mW
Thread map table	321 bits	8	321 bytes	9.8 mW	0.1 mW	9.9 mW
MSHR metadata table	23 bits	1024	2.9 KB	19.9 mW	2.1 mW	22.0 mW
Total per SM			5.23 KB	139.0 mW	3.3 mW	142.3 mW

Table 1: Per-SM storage and power overhead of *WarpPool* components

fore a store is sent to the cache [9], or by reordering only loads [25]. Neither of these is an option for *WarpPool*, as it would limit the coalescing window size to the interval between stores.

WarpPool guarantees memory consistency by using the warp scheduler to limit when stores can be issued to the LSU. A counter for each warp is incremented when a load is inserted into the instruction queues inter-warp coalescing queues and decremented when a request from that warp is sent to the L1 cache. When the counter is 0, there are no loads from the warp in the inter-warp coalescing queues. This counter needs to be 0 for a store to be issued, to ensure stores cannot be reordered with loads in the inter-warp coalescer, and to ensure stores cannot be reordered with other stores in the intra-warp coalescers. The scheduler will similarly wait for all stores to drain before issuing a load. A flag encodes when the previous global memory operation was a load, in which case it is safe to issue a load even when the counter is not 0.

3.9 Resource Configuration

We performed a design space sweep to determine the best number and size of each hardware resource for our workloads. Each of these resources is present in each SM. The instruction queues need to have at least 2 entries for each of 48 warps to allow for prioritization independent of the scheduler, suggesting a configuration of 48 instruction queues with 2 entries each. However, we found a configuration of 16 queues with 8 entries performed just as well but with much less selector overhead. Two intra-warp coalescers were adequate for most kernels, although some with high memory divergence like SYRK can achieve improved performance with more intra-warp coalescers. We used 32 coalescing queues with 2 tags each, as explained in Section 3.3, and allowed up to 4 inter-warp coalesces per request sent to the L1; increasing the number of coalesces increases the amount of metadata storage needed.

Table 1 describes the sizes of each of the hardware structures in our final configuration, per SM. The total overhead is 5.23 KB of storage, with over half of that used to build the MSHR metadata table.

3.10 Verilog Implementation

Since a substantial part of the hardware overhead of *WarpPool* is the connections between components on top of any storage overhead, we implemented *WarpPool* in Verilog to perform synthesis and place-and-route to accurately determine power and area overhead. We synthesized *WarpPool* in a 45nm process at 1.2GHz to best

Figure 10: Per-SM area breakdown of *WarpPool* components, with a total area of 0.36 mm^2 per SM. (* = SRAM area calculated using CACTI)

match the GTX 480 baseline system. The MSHR metadata table can be implemented as a regular SRAM, so CACTI 5.3 [28] was used to estimate its power and area. RC values from the routed design and traces of memory accesses from the kernels were used to more accurately estimate dynamic power.

WarpPool as configured has an area of 0.36 mm^2 per SM, broken down by component in Figure 10. Routing accounts for 45% of the overall area, mostly in the intra-warp coalescers. This is highest in the intra-warp coalescers because they work with full load and store instructions after address generation. There is a wide bus necessary to move the addresses and store data into the intra-warp coalescers from the address generation logic and register file, and a 160-bit bus from the intra-warp coalescers to the thread map table to allow a load with a *random* mapping pattern to issue in one cycle.

Compared to the GTX 480, with a die size of 529 mm^2 [18], *WarpPool* adds 5.4 mm^2 or 1.0% to the total GPU area. The added static and dynamic power of the routed netlist, shown in Table 1, is 142 mW per SM, or 0.8% of the GTX 480 TDP [19].

4. EVALUATION

4.1 Methodology

We use GPGPU-sim 3.2.2 [1] with the simulation parameters in Table 2 to model a Fermi-class GTX 480 GPU. Benchmarks were drawn from the Parboil [27], Rodinia [3], and PolyBench [6] benchmark suites. Rodinia and Parboil are used as a representative cross-section of GPU workloads. The Polybench benchmarks are designed to test the effectiveness of memory access

Figure 11: Speedup of GPU with banked cache, MRPB, and *WarpPool* over GTX 480 baseline.

optimizations, modelling commonly used linear algebra operations; they have been used in other GPU memory optimization works such as [9]. Out of the kernels in these suites, we used a subset that is limited by GPU memory throughput, as measured by having waiting memory requests for more than 90% of execution time. Kernels were run until completion or for hundreds of millions to billions of instructions in the steady state.

We compare *WarpPool* against other techniques for increasing L1 throughput and reducing the L1 miss rate. Banking the L1 cache increases throughput by allowing multiple hits to be serviced in parallel. We implemented an L1 cache banked eight ways, with eight cache sets per bank. This cache could perform eight tag lookups and service up to eight hits per cycle, but can only service one miss per cycle because of the need to search MSHRs. Eight banks was chosen as the throughput did not increase with more banks. Each bank has a coalescing unit that selects a line each cycle from the active load instruction that maps to that bank, which allows eight requests to be serviced in parallel by the cache.

We also compare against MRPB [9], which reorders memory requests to increase temporal locality by buffering memory requests going to the L1 cache. *WarpPool* is also able to reorder requests using the instruction queues and request selector to increase temporal locality, but adds the ability to combine requests across warps to exploit spatial locality between threads. We implemented MRPB with the configuration evaluated in [9], and calibrated the implementation against the results in that paper. The same paper also analyzes cache bypassing, which we do not implement as the bypassing technique is orthogonal to the memory reordering technique. Warp scheduler-based techniques such as CCWS [23] and Mascar [25] also reduce cache thrashing, but by reducing the number of active warps. For some workloads, adding CCWS-SWL, a warp limiting technique, on top of *WarpPool* showed added benefit.

4.2 Results

4.2.1 Speedup

Figure 11 shows the speedup over the GTX 480 baseline of *WarpPool*, MRPB, and the 8-way banked cache. *WarpPool* yields a better improvement than the other techniques with a geometric mean $1.38\times$ speedup. There were two mechanisms which produced this speedup.

The **memory divergent inter-warp** kernels bene-

Architecture	Fermi (GTX 480), 15SMs, 48 active warps per SM
L1 cache	32kB (64 sets, 128-byte lines, 4-way associative), 32 MSHRs
Warp scheduler	GTO
Instruction queues	16
Instruction queue entries	16
Intra-warp coalescers	2
Intra-warp coalescing queues	32 queues with 2 tags each, 4 merges per tag
Max. inter-warp coalesces/request	4

Table 2: Resource configuration evaluated

fitted from increased throughput to the L1 cache, created by inter-warp coalescing. The **bandwidth-limited inter-warp** kernels see speedup from more efficient utilization of the cache resources, creating more overlap between computation and L1 cache stalls. The **cache-limited** kernels see significantly fewer misses, caused by memory request prioritization.

Despite higher bandwidth to the cache, banking is not able to achieve a speedup because miss rates for GPU benchmarks are high and the banked cache could service only one miss per cycle. As well, banking creates more cache stalls because the miss is made by some banks more frequently than others, causing more cache line allocation stalls. MRPB gives a larger speedup on some of the cache-limited kernels due to its larger queue sizes, but is not as effective as *WarpPool* on the memory divergent or bandwidth-limited workloads. The following sections will examine the two mechanisms by which *WarpPool* yields speedup: increased L1 throughput and reduced L1 misses.

4.2.2 L1 Throughput

Figure 12 shows the average number of load instructions coalesced into a request to the L1. The baseline coalescer will always yield one instruction per request, so values greater than one are due to inter-warp coalescing. The number of instructions per request can be interpreted as a multiplier on the core-side throughput of the L1 cache, with the reciprocal showing the reduction in L1 accesses. For the kernels with inter-warp spatial locality, *WarpPool* allowed the L1 cache to service an average of 1.14 requests per cycle, with 1.08 across all the kernels.

The 8-way banked cache serviced an average 1.18 re-

Figure 12: Average number of requests *WarpPool* coalesced into an L1 cache request, compared against the number of requests an 8-bank cache serviced each cycle.

Figure 13: Number of misses per thousand instructions (MPKI) for MRPB and *WarpPool*, normalized to the baseline.

quests per cycle, but its increased throughput was not effective at producing speedup. The difference from *WarpPool* was because *WarpPool* uses locality to increase throughput whereas the banked cache increases throughput by looking up requests from divergent loads in parallel. This allowed the banked cache to find opportunity in the intra-warp workloads that *WarpPool* could not, but made it ineffective at increasing bandwidth for workloads without much memory divergence like *sc*. The banked cache is not able to convert increased throughput to performance for two reasons: first, the miss rate of GPU workloads is high but only one miss could be serviced per cycle. Second, the banked cache is only able to service requests from one warp at a time, so the banks are often idle. Unlike the banked cache, *WarpPool* is able to merge misses and look across multiple warps to translate higher throughput into performance.

The number of inter-warp coalesces found by *WarpPool* is limited by two factors: window size and memory consistency. The window size limits how far apart merged requests can be. In the tested configuration, the window size was 64 distinct cache lines, because the inter-warp queues have 64 tags. The kernels' hit rate can also limit window size when requests drain too quickly for the queues to fill with requests. This is why *SYR2K* has a higher number of instructions per request than *SYRK*, which has a similar access pattern: the miss rate of *SYR2K* is higher, which causes more backup in the coalescing queues and leads to more coalescing. Maintaining memory consistency limited the window size for many kernels, especially for *GEMM*, *2MM_1*, and *3MM_1*, which have a store for every two loads.

4.2.3 L1 Misses

Figure 13 shows the number of misses per thousand instructions for each kernel, normalized to the misses in the baseline. The geometric mean is 77% of the baseline, showing *WarpPool* is able to not only reduce the number of accesses from the SM to the L1 cache, but

reduce the number of requests the L1 made over the interconnect to the rest of the memory system.

This improvement is due to the prioritization schemes in *WarpPool*. The instruction queues allow *WarpPool* to prioritize warps more effectively than the scheduler by buffering requests rather than sending them immediately to the coalescer. The second prioritization scheme, using the *warp ID* selection policy, reduced the number of misses even further in a number of kernels, including *ATAX_1*, *MVT_1*, and *BICG_2*.

CORR_3 saw the number of L1 misses increase, and several others do not see a reduction in L1 misses. This is caused by *WarpPool* causing early cache evictions due to two effects. First, when requests are coalesced together, the LRU is only updated once when multiple requests would have updated the LRU status multiple times. Second, inter-warp coalescing can increase the number of unique requests in a given time interval, because duplicate requests are coalesced together, which causes more capacity pressure on the cache.

WarpPool was able to reduce the MPKI more than MRPB. The behavior of MRPB is similar to *WarpPool* always being in *warp ID* selection mode. This hurts *SYR2K* and *SYRK*, where temporal reuse is between warps in a block more than inside a thread. *kmeans_2* and *pf_1* likewise have reuse across warps which *WarpPool*'s *oldest* selection mode works better to find. MRPB reduced the miss rate more than *WarpPool* for *spmv*, *sc*, *MVT_1*, and *BICG_2*, because it has 6× the number of queue entries, and saw a corresponding speedup over *WarpPool* for those kernels.

4.3 Case Study

It can require a significant amount of programmer effort and expertise to make algorithms run efficiently on a GPU. Matrix transposes require careful implementation for GPUs because they load and store data along different dimensions of the matrix. A straightforward implementation of matrix transpose that does not use shared memory [7] has poor performance because the

Figure 14: Relative execution time of matrix transpose versions, normalized to naïve global memory version.

global memory loads are done in column-wise order, leading to high memory divergence for the loads. Copying a block of the matrix to shared memory allows both the loads and stores to be well-coalesced, although the shared memory implementation still requires padding to avoid bank conflicts.

For this benchmark, *WarpPool* runs the straightforward, unoptimized version using global memory in $0.50\times$ the time, which is near the runtime of the shared memory version, which runs in $0.38\times$ of the baseline time. This shows that *WarpPool* is able to relieve the burden of programmers to optimize for particular memory access patterns.

5. RELATED WORK

CPU request merging and cache throughput:

Juan et al. [12] investigated methods of improving bandwidth for superscalar processors, including multi-porting and banking. Davidson et al. [5] studied memory coalescing to widen memory requests for CPUs. Our work differs by needing L1 throughput to satisfy parallel threads rather than wider single thread execution. Olukotun et al. [20] propose techniques that allow data returning from the cache to satisfy loads not yet issued to it. *WarpPool* differs by merging requests before sending them to the cache, which takes advantage of the GPU’s relative latency insensitivity. Rivers et al. [22] reorder and combine requests in a CPU’s load-store queue to optimize requests to a banked cache. Our work differs because GPUs do not have the same load-store queue structures. Quintana et al. [21] use a dual-banked cache to allow unaligned loads on a vector unit integrated in a CPU to complete in one cycle.

Analysis of GPU Coalescing: Hestness et al. [8] analyze the benefits of intra-warp coalescing in GPU memory systems, finding that increasing the window of threads inside a warp can lead to a large reduction in memory accesses but little speedup. We get past this limitation by merging requests across warps. Yang et al. [29] use compiler techniques to transform GPU kernels to use memory accesses with better coalescing behavior. Baskaran et al. [2] use polyhedral analysis to improve coalescing and locality in auto-parallelized code. Our work is able to optimize memory accesses dynamically in hardware.

Improving Inter-Warp Locality: Lee et al. [14] use a block scheduling policy that assigns nearby CTAs

to the same SM, to capture inter-CTA spatial locality that is lost with a round-robin warp scheduler. Jog et al. [11] show there is benefit to scheduling spatially nearby warps temporally distant from each other so that warps will prefetch data for each other. Jog et al. [10] also propose a warp scheduling technique that divides CTAs into warp groups that have different priority access to the cache. Lee et al. [13] use compiler analysis to map patterns to GPUs in ways that best preserve locality. Lee et al. [15] perform auto-parallelization for GPUs to improve inter-warp locality. Our work builds on these techniques by providing another place where inter-warp locality can improve performance.

Warp Throttling: Scheduling only a subset of ready warps can increase the amount of intra-warp temporal locality, as it prevents cache thrashing. Rogers et al. [23] detect cache thrashing and decrease the number of warps to prevent it. Later work by Rogers et al. [24] predicts how many warps’ data will fit in the cache and limits the number of warps accordingly. Our work limits access to the cache after warp scheduling, which allows for finer granularity when choosing requests to send to the cache. Sethia et al. [26] use performance counters to detect cache sensitivity in order to reduce the number of threads. Our work also detects cache sensitivity with performance counters, but uses them to toggle the selection policy rather than the number of warps.

Cache Bypassing: Another technique to prevent cache thrashing is by routing requests from only a subset of warps to the L1 cache, forcing other warps’ requests to bypass the cache. Chen et al. [4] watch for early evictions to prevent thrashing of cache lines with high contention, using bypassing to avoid the contention. Jia et al. [9] use a combination of request prioritization and bypassing to reduce cache contention. Zheng et al. [30] separate warps into groups, only one of which can access the cache. Cache bypassing is complementary to *WarpPool*’s prioritization methods and can be added to it to further reduce cache thrashing.

6. CONCLUSION

Many memory throughput-limited benchmarks are constrained by the interface between the SM and the L1 cache. We alleviate this bottleneck by extending the window size of the memory coalescer from the threads in one warp to requests made by multiple warps. For workloads with divergent requests, this reduces the cost of serializing multiple requests. For workloads limited by memory bandwidth, this makes better use of cache resources. For cache-sensitive workloads, the coalescing window enables finer-grained request scheduling which reduces cache thrashing. This leads to a 38% speedup across a set of memory throughput-limited kernels. We also show our technique can help GPU programmability by achieving high performance without the need to optimize a workload’s memory access patterns. We implemented *WarpPool* in Verilog to show that *WarpPool* achieves these benefits with minimal power and area overhead.

7. ACKNOWLEDGEMENTS

The authors would like to thank Anoushe Jamshidi, the other members of the CCCP group, and the anonymous reviewers for their valuable comments and feedback. This work was supported by ARM Ltd. and by the Defense Advanced Research Projects Agency (DARPA) under the Power Efficiency Revolution for Embedded Computing Technologies (PERFECT) program.

8. REFERENCES

- [1] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174, April 2009.
- [2] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In R. Gupta, editor, *Compiler Construction*, volume 6011 of *Lecture Notes in Computer Science*, pages 244–263. Springer Berlin Heidelberg, 2010.
- [3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE.
- [4] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W.-M. Hwu. Adaptive cache management for energy-efficient gpu computing. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 343–355. IEEE, 2014.
- [5] J. W. Davidson and S. Jinturkar. Memory access coalescing: A technique for eliminating redundant memory accesses. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, PLDI '94*, pages 186–195, New York, NY, USA, 1994. ACM.
- [6] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *Innovative Parallel Computing (InPar)*, 2012, pages 1–10, May 2012.
- [7] M. Harris. An efficient matrix transpose in cuda c/c++. <http://devblogs.nvidia.com/parallelforall/efficient-matrix-transpose-cuda-cc>. Accessed: April 2015.
- [8] J. Hestness, S. Keckler, and D. Wood. A comparative analysis of microarchitecture effects on cpu and gpu memory system behavior. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 150–160, Oct 2014.
- [9] W. Jia, K. Shaw, and M. Martonosi. Mrpb: Memory request prioritization for massively parallel processors. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 272–283, Feb 2014.
- [10] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. Owl: Cooperative thread array aware scheduling techniques for improving gpgpu performance. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 395–406, New York, NY, USA, 2013. ACM.
- [11] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. Orchestrated scheduling and prefetching for gpgpus. *ACM SIGARCH Computer Architecture News*, 41(3):332–343, 2013.
- [12] T. Juan, J. J. Navarro, and O. Temam. Data caches for superscalar processors. In *Proceedings of the 11th International Conference on Supercomputing, ICS '97*, pages 60–67, New York, NY, USA, 1997. ACM.
- [13] H. Lee, K. Brown, A. Sajeeth, T. Rompf, and K. Olukotun. Locality-aware mapping of nested parallel patterns on gpus. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 63–74, Dec 2014.
- [14] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu. Improving gpgpu resource utilization through alternative thread block scheduling. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 260–271, Feb 2014.
- [15] S. Lee, S.-J. Min, and R. Eigenmann. Openmp to gpgpu: A compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '09*, pages 101–110, New York, NY, USA, 2009. ACM.
- [16] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi. Gpuwatch: Enabling energy optimizations in gpgpus. In *Proceedings of the 40th Annual International Symposium on Computer Architecture, ISCA '13*, pages 487–498, New York, NY, USA, 2013. ACM.
- [17] C. Nugteren, G.-J. van den Braak, H. Corporaal, and H. Bal. A detailed gpu cache model based on reuse distance theory. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 37–48, Feb 2014.
- [18] Nvidia. GeForce GTX 480. <http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-480/specifications>.
- [19] Nvidia. GeForce GTX 680. http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf.
- [20] K. Olukotun, M. Rosenblum, and K. Wilson. Increasing cache port efficiency for dynamic superscalar microprocessors. In *Computer Architecture, 1996 23rd Annual International Symposium on*, pages 147–147, May 1996.
- [21] F. Quintana, J. Corbal, R. Espasa, and M. Valero. Adding a vector unit to a superscalar processor. In *Proceedings of the 13th international conference on Supercomputing*, pages 1–10. ACM, 1999.
- [22] J. A. Rivers, G. S. Tyson, E. S. Davidson, and T. M. Austin. On high-bandwidth data cache design for multi-issue processors. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–56. IEEE, 1997.
- [23] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-45*, pages 72–83, Washington, DC, USA, 2012. IEEE.
- [24] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Divergence-aware warp scheduling. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-46*, pages 99–110, New York, NY, USA, 2013. ACM.
- [25] A. Sethia, D. Jamshidi, and S. Mahlke. Mascar: Speeding up gpu warps by reducing memory pitstops. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 174–185, Feb 2015.
- [26] A. Sethia and S. Mahlke. Equalizer: Dynamic tuning of gpu resources for efficient execution. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 647–658, Dec 2014.
- [27] J. A. Stratton et al. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign.
- [28] S. Thoziyoor, N. Muralimanohar, and N. P. Jouppi. Cacti 5.0. *HP Laboratories, Technical Report*, 2007.
- [29] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A gpgpu compiler for memory optimization and parallelism management. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '10*, pages 86–97, New York, NY, USA, 2010. ACM.
- [30] Z. Zheng, Z. Wang, and M. Lipasti. Adaptive cache and concurrency allocation on gpgpus. *Computer Architecture Letters*, PP(99):1–1, 2014.