OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance

Adwait Jog† Onur Kayiran† Mahmut T. Kandemir† Nachiappan Chidambaram Nachiappan† Asit K. Mishra§
Onur Mutlu* Ravishankar Iyer§ Chita R. Das†
The Pennsylvania State University† Carnegie Mellon University*
University Park, PA 16802 Pittsburgh, PA 15213
(adwait, onur, nachi, kandemir, das)@cse.psu.edu onur@cmu.edu (asit.k.mishra, ravishankar.iyer)@intel.com

Abstract
Emerging GPGPU architectures, along with programming models like CUDA and OpenCL, offer a cost-effective platform for many applications by providing high thread level parallelism at lower energy budgets. Unfortunately, for many general-purpose applications, available hardware resources of a GPGPU are not efficiently utilized, leading to lost opportunity in improving performance. A major cause of this is the inefficiency of current warp scheduling policies in tolerating long memory latencies.

In this paper, we identify that the scheduling decisions made by such policies are agnostic to thread-block, or cooperative thread array (CTA), behavior, and as a result inefficient. We present a coordinated CTA-aware scheduling policy that utilizes four schemes to minimize the impact of long memory latencies. The first two schemes, CTA-aware two-level warp scheduling and locality aware warp scheduling, enhance per-core performance by effectively reducing cache contention and improving latency hiding capability. The third scheme, bank-level parallelism aware warp scheduling, improves overall GPGPU performance by enhancing DRAM bank-level parallelism. The fourth scheme employs opportunistic memory-side prefetching to further enhance performance by taking advantage of open DRAM rows. Evaluations on a 28-core GPGPU platform with highly memory-intensive applications indicate that our proposed mechanism can provide 33% average performance improvement compared to the commonly-employed round-robin warp scheduling policy.

Categories and Subject Descriptors C.1.4 [Computer Systems Organization]: Processor Architectures—Parallel Architectures; D.1.3 [Software]: Programming Techniques—Concurrent Programming

General Terms Design, Performance

Keywords GPGPUs; Scheduling; Prefetching; Latency Tolerance

1. Introduction
General Purpose Graphics Processing Units (GPGPUs) have recently emerged as a cost-effective computing platform for a wide range of applications due to their immense computing power compared to CPUs [1, 6, 7, 25, 28, 48]. GPGPUs are characterized by numerous programmable computational cores and thousands of simultaneously active fine-grained threads. To facilitate ease of programming on these systems, programming models like CUDA [46] and OpenCL [39] have been developed. GPGPU applications are typically divided into several kernels, where each kernel is capable of spawning many threads. The threads are usually grouped together into thread blocks, also known as cooperative thread arrays (CTAs). When an application starts its execution on a GPGPU, the CTA scheduler initiates scheduling of CTAs onto the available GPGPU cores. All the threads within a CTA are executed on the same core typically in groups of 32 threads. This collection of threads is referred to as a warp and all the threads within a warp typically share the same instruction stream, which forms the basis for the term single instruction multiple threads, SIMT [7, 8, 37].

In spite of the high theoretically achievable thread-level parallelism (TLP) (for example, GPGPUs are capable of simultaneously executing more than 1024 threads per core [48]), GPGPU cores suffer from high periods of inactive times resulting in underutilization of hardware resources [24, 44]. Three critical reasons for this are: 1) on-chip memory and register files are limiting factors on parallelism, 2) high control flow divergence, and 3) inefficient scheduling mechanisms. First, GPGPUs offer a limited amount of programmer-managed memory (shared memory) and registers. If the per-CTA requirements for these resources are high, then the effective number of CTAs that can be scheduled simultaneously will be small, leading to lower core utilization [5, 28]. Second, when threads within a warp take different control flow paths, the number of threads that can continue execution in parallel reduces. Recent works that have tackled this problem include [14, 15, 44, 49]. Third, the inefficiency of the commonly-used round-robin (RR) scheduling policy [5, 15, 44] to hide long memory fetch latencies, primarily caused by limited off-chip DRAM bandwidth, contributes substantially to the under-utilization of GPGPU cores.

With the RR scheduling policy, both the CTAs assigned to a core and all the warps inside a CTA are given equal priority, and are executed in a round-robin fashion. Due to this scheduling policy, most of the warps arrive at long latency memory operations roughly at the same time [44]. As a result, the GPGPU core becomes inactive because there may be no warps that are not stalling due to a memory operation, which significantly reduces the capability of hiding long memory latencies. Such inactive periods are especially prominent in memory-intensive applications. We observe that out of 38 applications covering various benchmarks suites, 19 applications suffer from very high core inactive times (on average 62% of total cycles are spent with no warps executing). The primary
cause of the high core inactivity is the large amount of on-chip and off-chip traffic caused by the burst of long-latency memory operations coming from all warps, leading to high round-trip fetch latencies. This in turn is mainly attributed to limited off-chip DRAM bandwidth available in GPGPUs. It is anticipated that this problem will be aggravated in emerging heterogeneous architectures, where the main memory is unified and shared by both CPU and GPGPU cores [3, 29–31]. This problem also becomes more severe with core scaling and the increase in the number of simultaneously executing threads [24], in a way that is similar to the memory bandwidth problem in multi-core systems [38, 42].

The goal of this paper is to tackle the under-utilization of cores for improving the overall GPGPU performance. In this context, we propose the c(O)operative thread array a(W)are warp scheduling policy, called OWL. OWL is based on the concept of focused CTA-aware scheduling, which attempts to mitigate the various components that contribute to long memory fetch latencies by focusing on a selected subset of CTAs scheduled on a core (by always prioritizing them over others until they finish). The proposed OWL policy is a four-pronged concerted approach:

First, we propose a CTA-aware two-level warp scheduler that exploits the architecture and application interplay to intelligently schedule CTAs onto the cores. This scheme groups all the available CTAs (N CTAs) on a core into smaller groups (of n CTAs) and schedules all groups in a round-robin fashion. As a result, it performs better than the commonly-used baseline RR warp scheduler because 1) it allows a smaller group of warps/threads to access the L1 cache in a particular interval of time, thereby reducing cache contention, 2) improves latency hiding capability and reduces inactive periods as not all warps reach long latency operations around the same time. This technique improves the average L1 cache hit rate by 8% over RR for 19 highly memory intensive applications, providing a 14% improvement in IPC performance.

Second, we propose a locality aware warp scheduler to improve upon the CTA-aware two-level warp scheduler, by further reducing L1 cache contention. This is achieved by always prioritizing a group of CTAs (n CTAs) in a core over the rest of the CTAs (until they finish). Hence, unlike the base scheme, where each group of CTAs (consisting of n CTAs) is executed one after another and thus, does not utilize the caches effectively, this scheme always prioritizes one group of CTAs over the rest whenever a particular group of CTA is ready for execution. The major goal is to take advantage of the locality between nearby warps and cores (associated with the same CTA) [21]. With this scheme, average L1 cache hit rate is further improved by 10% over the CTA-aware two-level warp scheduler, leading to an 11% improvement in IPC performance.

Third, the first two schemes are aware of different CTAs but do not exploit any properties common among different CTAs. Across 38 GPGPU applications, we observe that there is significant DRAM page locality between consecutive CTAs. On average, the same DRAM page is accessed by consecutive CTAs 64% of the time. Hence, if two consecutive CTA groups are scheduled on two different cores and are always prioritized according to the locality aware warp scheduling, they would access a small set of DRAM banks more frequently. This increases the queuing time at the banks and reduces memory bank level parallelism (BLP) [41]. On the other hand, if non-consecutive CTA groups are scheduled and always prioritized on two different cores, as we propose, they would concurrently access a larger number of banks. This reduces the contention at the banks and improves BLP. This proposed scheme (called the bank-level parallelism aware warp scheduler), increases average BLP by 11% compared to the locality aware warp scheduler, providing a 6% improvement in IPC performance.

Fourth, a drawback of the previous scheme is that it reduces DRAM row locality. This is because rows opened by a CTA cannot be completely utilized by its consecutive CTAs since consecutive CTAs are not scheduled simultaneously any more. To recover the loss in DRAM row locality, we develop an opportunistic prefetching mechanism, in which some of the data from the opened row is brought to the nearest on-chip L2 cache partition. The mechanism is opportunistic because the degree of prefetching depends upon the number of pending demand requests at the memory controller.

We evaluate the performance of the OWL scheduling policy, consisting of the four components integrated together, on a 28-core GPGPU platform simulated via GPGPU-Sim [5] and a set of 19 highly memory intensive applications. Our results show that OWL improves GPGPU performance by 33% over the baseline RR warp scheduling policy. OWL also outperforms the recently-proposed two-level scheduling policy [44] by 19%.

2. Background and Experimental Methodology

This section provides a brief description of GPGPU architecture, typical scheduling strategies, main memory layouts of CTAs, application suite and evaluation metrics.

2.1 Background

Our Baseline GPGPU Architecture: A GPGPU consists of many simple cores (streaming multiprocessors), with each core typically having a SIMT width of 8 to 32 (NVIDIA’s Fermi series has 16 streaming multiprocessors with a SIMT width of 32 [48] and AMD’s ATI 5870 Evergreen architecture has 20 cores with a SIMT width of 16 [2]). Our target architecture (shown in Figure 1 (A)) consists of 28 shader cores each with a SIMT width of 8, and 8 memory controllers. This configuration is similar to the ones studied in recent works [4, 5]. Each core is associated with a private L1 data cache and read-only texture and constant caches along with a low latency shared memory (scratchpad memory). Every memory controller is associated with a slice of the shared L2 cache for faster access to the cached data. We assume write-back policies for both L1 and L2 caches and optimistic performance model for atomic instructions [5, 16]. The minimum L2 miss latency is assumed to be 120 compute core cycles [56]. The actual miss latency could be higher because of queuing at the memory controllers and variable DRAM latencies. Cores and memory controllers are connected via a two-dimensional mesh. We use a 2D mesh topology, as it

Figure 1. (A) GPGPU architecture, (B) CTA data layout, and (C) Main memory layout with CTA’s data mapped.
is scalable, simple, and regular [4, 5, 43]. A detailed baseline platform configuration is described in Table 1, which is simulated on GPGPU-Sim 2.1.2, a cycle-accurate GPGPU simulator [5].

**Canonical GPGPU Application Design:** A typical CUDA application consists of many kernels (or grids) as shown in Figure 2 (A). These kernels implement specific modules of an application. Each kernel is divided into groups of threads, called cooperative thread arrays (CTAs) (Figure 2 (B)). A CTA is an abstraction which encapsulates all synchronization and barrier primitives among a group of threads [28]. Having such an abstraction allows the underlying hardware to relax the execution order of the CTAs to maximize parallelism. The underlying architecture in turn, sub-divides each CTA into groups of threads (called warps) (Figure 2 (C) and (D)). This sub-division is transparent to the application programmer and is an architectural abstraction.

**CTA, Warp, and Thread Scheduling:** Execution on GPGPUs starts with the launch of a kernel. In this work, we assume sequential execution of kernels, which means only one kernel is executed at a time. After a kernel is launched, the CTA scheduler schedules available CTAs associated with the kernel in a round-robin and load balanced fashion on all the cores [5]. For example, CTA 1 is assigned to core 1, CTA 2 is assigned to core 2 and so on. After assigning at least one CTA to each core (provided that enough CTAs are available), if there are still unassigned CTAs, more CTAs can be assigned to the same core in a similar fashion. The maximum number of CTAs per core \( N \) is limited by core resources (number of threads, size of shared memory, register file size, etc. [5, 28]).

Given a baseline architecture, \( N \) may vary across cores depending on how much resources are needed by a CTA of a particular kernel. If a CTA of a particular kernel needs more resources, \( N \) will be smaller compared to that of another kernel whose CTAs need fewer resources. For example, if a CTA of kernel X needs 16KB of shared memory and the baseline architecture has 32KB of shared memory available, a maximum of 2 CTAs of kernel X can be executed simultaneously.

The above CTA assignment policy is followed by per-core GPGPU warp scheduling. Warps associated with CTAs are scheduled in a round-robin (RR) fashion on the assigned cores [5, 44] and get equal priority. Every 4 cycles, a warp ready for execution is selected in a round-robin fashion and fed to the 8-way SIMT pipeline of a GPGPU core. At the memory stage of the core pipeline, if a warp gets blocked on a long latency memory operation, the entire warp (32 threads) is scheduled out of the pipeline and moved to the pending queue. At a later instant, when the data for the warp arrives, it proceeds to the write-back stage, and then fetches new instructions.

**CTA Data Layout:** Current GPU chips support \( \sim 10 \times \) higher memory bandwidth compared to CPU chips [25]. In order to take full advantage of the available DRAM bandwidth and to reduce the number of requests to DRAM, a kernel must arrange its data accesses so that each request to the DRAM is for a large number of consecutive DRAM locations. With the SIMT execution model, when all threads in a warp execute a memory operation, the hardware typically detects if the threads are accessing consecutive memory locations; if they are, the hardware coalesces all these accesses into a single consolidated access to DRAM that requests all consecutive locations at once. To understand how data blocks used by CTAs are placed in the DRAM main memory, consider Figure 1 (B). This figure shows that all locations in the DRAM main memory form a single, consecutive address space. The matrix elements that are used by CTAs are placed into the linearly addressed locations according to the row major convention as shown in Figure 1 (B). That is, the elements of row 0 of a matrix are first placed in order into consecutive locations (see Figure 1 (C)). The subsequent row is placed in another DRAM bank. Note that, this example is simplified for illustrative purposes only. The data layout may vary across applications (our evaluations take into account different data layouts of applications).

**2.2 Workloads and Metrics**

**Application Suite:** There is increasing interest in executing various general-purpose applications on GPGPUs in addition to the traditional graphics rendering applications [5, 32]. In this spirit, we consider a wide range of emerging GPGPU applications implemented in CUDA, which include NVIDIA SDK [47], Rodinia [10], Parboil [53], MapReduce [19], and a few third party applications. In total, we study 38 applications. While Rodinia applications are mainly targeted for heterogeneous platforms, Parboil benchmarks primarily stress throughput computing focused architectures. Data-intensive MapReduce and third party applications are included for diversity. We execute these applications on GPGPU-Sim, which simulates the baseline architecture described in Table 1. The applications are run until completion or for 1 billion instructions (whichever comes first), except for Tux where we execute only 400 million instructions because of infrastructure limitations.

**Evaluation Metrics:** In addition to using **instructions per cycle (IPC)** as the primary performance metric for evaluation, we also consider auxiliary metrics like bank level parallelism and row buffer locality. Bank level parallelism (BLP) is defined as the number of average memory banks that are accessed when there is at least one outstanding memory request at any of the banks [26, 27, 41, 42]. Improving BLP enables better utilization of DRAM bandwidth. Row-buffer locality (RBL) is defined as the average hit-rate of the row buffer across all memory banks [27]. Improving RBL increases the memory service rate and hence also enables better DRAM bandwidth utilization.

**3. Motivation and Workload Analysis**

Round robin (RR) scheduling of warps causes almost all warps to execute the same long latency memory operation (with different addresses) at roughly the same time, as previous work has shown [44]. For the computation to resume in the warps and the core to become active again, these long-latency memory accesses need to be completed. This inefficiency of RR scheduling hampers the latency hid-
Table 2. GPGPU application characteristics: (A) PMEM: IPC improvement with perfect memory (All memory requests are satisfied in L1 caches), Legend: H = High (>= 1.4x), L = Low (< 1.4x); (B) CINV: The ratio of inactive cycles to the total execution cycles of all the cores. 

<table>
<thead>
<tr>
<th>#</th>
<th>App.</th>
<th>Type-1 Applications</th>
<th>Abbr.</th>
<th>PMEM</th>
<th>CINV</th>
<th>#</th>
<th>App.</th>
<th>Type-2 Applications</th>
<th>Abbr.</th>
<th>PMEM</th>
<th>CINV</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Parboil</td>
<td>Sum of Abs. Differences</td>
<td>SAD</td>
<td>H (3.9x)</td>
<td>91%</td>
<td>20</td>
<td>CUDA SDK</td>
<td>Separable Convolution</td>
<td>CON</td>
<td>L (1.23x)</td>
<td>20%</td>
</tr>
<tr>
<td>2</td>
<td>MapReduce</td>
<td>PageViewCount</td>
<td>PVC</td>
<td>H (4.99x)</td>
<td>93%</td>
<td>21</td>
<td>CUDA SDK</td>
<td>AES Cryptography</td>
<td>AES</td>
<td>L (1.23x)</td>
<td>51%</td>
</tr>
<tr>
<td>3</td>
<td>MapReduce</td>
<td>SimilarityScore</td>
<td>SSS</td>
<td>H (4.60x)</td>
<td>85%</td>
<td>22</td>
<td>Rodinia</td>
<td>SKADI</td>
<td>SDD</td>
<td>L (1.15x)</td>
<td>20%</td>
</tr>
<tr>
<td>4</td>
<td>CUDA SDK</td>
<td>Breadth First Search</td>
<td>BFS</td>
<td>H (2.73x)</td>
<td>85%</td>
<td>23</td>
<td>CUDA SDK</td>
<td>Blackholes</td>
<td>BLK</td>
<td>L (1.16x)</td>
<td>17%</td>
</tr>
<tr>
<td>5</td>
<td>CUDA SDK</td>
<td>MMUmerGPU</td>
<td>MUM</td>
<td>H (2.66x)</td>
<td>72%</td>
<td>24</td>
<td>Rodinia</td>
<td>HotSpot</td>
<td>HS</td>
<td>L (1.15x)</td>
<td>21%</td>
</tr>
<tr>
<td>6</td>
<td>Rodinia</td>
<td>CFD Solver</td>
<td>CFD</td>
<td>H (2.46x)</td>
<td>66%</td>
<td>25</td>
<td>CUDA SDK</td>
<td>Scan of Large Arrays</td>
<td>SLA</td>
<td>L (1.13x)</td>
<td>17%</td>
</tr>
<tr>
<td>7</td>
<td>Rodinia</td>
<td>Knemics Clustering</td>
<td>KMN</td>
<td>H (2.33x)</td>
<td>65%</td>
<td>26</td>
<td>3rd Party</td>
<td>Denoise</td>
<td>DN</td>
<td>L (1.12x)</td>
<td>22%</td>
</tr>
<tr>
<td>8</td>
<td>CUDA SDK</td>
<td>Scalar Product</td>
<td>SCP</td>
<td>H (2.23x)</td>
<td>58%</td>
<td>27</td>
<td>CUDA SDK</td>
<td>3D Laplace Solver</td>
<td>LPS</td>
<td>L (1.10x)</td>
<td>12%</td>
</tr>
<tr>
<td>9</td>
<td>CUDA SDK</td>
<td>Fast Walsh Transform</td>
<td>FWT</td>
<td>H (2.29i)</td>
<td>58%</td>
<td>28</td>
<td>CUDA SDK</td>
<td>Neural Network</td>
<td>NN</td>
<td>L (1.10x)</td>
<td>13%</td>
</tr>
<tr>
<td>10</td>
<td>MapReduce</td>
<td>InvertedIndex</td>
<td>IX</td>
<td>H (2.29i)</td>
<td>65%</td>
<td>29</td>
<td>Rodinia</td>
<td>Particle Filter (Native)</td>
<td>PFN</td>
<td>L (1.08x)</td>
<td>10%</td>
</tr>
<tr>
<td>11</td>
<td>Parboil</td>
<td>Sparse-Matrix-Mul.</td>
<td>SPMV</td>
<td>H (2.19x)</td>
<td>65%</td>
<td>30</td>
<td>Rodinia</td>
<td>Leukocyte</td>
<td>LYTE</td>
<td>L (1.08x)</td>
<td>15%</td>
</tr>
<tr>
<td>12</td>
<td>3rd Party</td>
<td>JPEG Decoding</td>
<td>JPEG</td>
<td>H (2.12x)</td>
<td>54%</td>
<td>31</td>
<td>Rodinia</td>
<td>LU Decomposition</td>
<td>LUD</td>
<td>L (1.05x)</td>
<td>64%</td>
</tr>
<tr>
<td>13</td>
<td>Rodinia</td>
<td>Breadth First Search</td>
<td>BFSR</td>
<td>H (2.09x)</td>
<td>64%</td>
<td>32</td>
<td>Parboil</td>
<td>Matrix Multiplication</td>
<td>MM</td>
<td>L (1.04x)</td>
<td>4%</td>
</tr>
<tr>
<td>14</td>
<td>Rodinia</td>
<td>Streamcluster</td>
<td>SC</td>
<td>H (1.94x)</td>
<td>52%</td>
<td>33</td>
<td>CUDA SDK</td>
<td>StoreGPU</td>
<td>STO</td>
<td>L (1.02x)</td>
<td>3%</td>
</tr>
<tr>
<td>15</td>
<td>Parboil</td>
<td>FFT Algorithm</td>
<td>FFT</td>
<td>H (1.36x)</td>
<td>37%</td>
<td>34</td>
<td>CUDA SDK</td>
<td>Coulombic Potential</td>
<td>CP</td>
<td>L (1.01x)</td>
<td>4%</td>
</tr>
<tr>
<td>16</td>
<td>Rodinia</td>
<td>SKAD2</td>
<td>SD2</td>
<td>H (1.53x)</td>
<td>36%</td>
<td>35</td>
<td>CUDA SDK</td>
<td>N-Queens Solver</td>
<td>NQU</td>
<td>L (1.01x)</td>
<td>95%</td>
</tr>
<tr>
<td>17</td>
<td>CUDA SDK</td>
<td>Weather Prediction</td>
<td>WP</td>
<td>H (1.50x)</td>
<td>37%</td>
<td>36</td>
<td>Parboil</td>
<td>Distance-Cutoff CP</td>
<td>DCP</td>
<td>L (1.01x)</td>
<td>2%</td>
</tr>
<tr>
<td>18</td>
<td>MapReduce</td>
<td>PageViewRank</td>
<td>PVR</td>
<td>H (1.41x)</td>
<td>46%</td>
<td>37</td>
<td>Rodinia</td>
<td>Heartwall</td>
<td>HW</td>
<td>L (1.01x)</td>
<td>9%</td>
</tr>
<tr>
<td>19</td>
<td>Rodinia</td>
<td>Backpropagation</td>
<td>BP</td>
<td>H (1.40x)</td>
<td>33%</td>
<td>38</td>
<td>Parboil</td>
<td>Angular Correlation</td>
<td>TPAF</td>
<td>L (1.01x)</td>
<td>6%</td>
</tr>
</tbody>
</table>

Across Type-1 applications, average core inactive time (CINV) is 62% of the total execution cycles of all cores (Table 2). During this inactive time, no threads are being executed in the core. The primary reason behind this high core inactivity is what we call MemoryBlockCycles, which is defined as the number of cycles during which all the warps in the core are stalled waiting for their memory requests to come back from L2 cache/DRAM (i.e., there are warps on the core but they are all waiting for memory). Figure 3 shows the fraction of MemoryBlockCycles of all the cores out of the total number of cycles taken to execute each application. Across all 38 applications, MemoryBlockCycles constitute 32% of the total execution cycles, i.e., 70% of the total inactive cycles. These results clearly highlight the importance of reducing the MemoryBlockCycles to improve the utilization of cores, and thus GPGPU performance.

Another major constituent of inactive cycles is NoWarpCycles, which is defined as number of cycles during which a core has no warps to execute, but an application has not completed its execution as some other cores are still executing warps. This might happen due to two reasons: (1) availability of a small number of CTAs within an application (due to an inherently small amount of parallelism) [24] or (2) the CTA load imbalance phenomenon [5], where some of the cores finish their assigned CTAs earlier than the others. We find that NoWarpCycles is prominent in LUD and NQU, which are Type-2 applications. From Table 2, we see that although core inactive time is very high in LUD and NQU (64% and 95%, respectively), MemoryBlockCycles is very low (Figure 3). We leave

Figure 3. Fraction of total execution cycles (of all the cores) during which all the warps launched on a core are waiting for their respective data to come back from L2 cache/DRAM. We call the number of cycles where all warps are stalled due to memory MemoryBlockCycles. AVG-T1 is the arithmetic mean value across all Type-1 applications. AVG is the average value across all 38 applications.
improving the performance of Type-2 applications for future work, and focus on improving the performance of Type-1 applications where the main cause of core idleness is waiting on memory.

Note that Type-1 applications are present across all modern workload suites like MapReduce, Parboil, Rodinia, and CUDA SDK, indicating that memory stalls are a fundamental bottleneck in improving the performance of these applications. We have found that Type-1 applications are most affected by limited off-chip DRAM bandwidth, which leads to long memory stall times. Our goal is to devise new warp scheduling mechanisms to both reduce and tolerate long memory stall times in GPGPUs.

4. The Proposed OWL Scheduler

In this section, we describe OWL, cOoperative thread array at(W)arp scheduling policy, which consists of four schemes: CTA-aware two-level warp scheduling, locality aware warp scheduling, bank-level parallelism aware warp scheduling, and opportunistic prefetching, where each scheme builds on top of the previous.

4.1 CTA-Aware: CTA-aware two-level warp scheduling

To address the problem posed by RR scheduling, we propose a CTA-aware two-level warp scheduler, where all the available CTAs launched on a core ($N$ CTAs) are divided into smaller groups of $n$ CTAs. Assume that the size of each CTA is $k$ warps (which is pre-determined for an application kernel). This corresponds to each group having $n \times k$ warps. CTA-Aware selects a single group (having $n$ CTAs) and prioritizes the associated warps ($n \times k$) for execution over the remaining warps ($N - n \times k$) associated with the other group(s). Warps within the same group have equal priority and are executed in a round-robin fashion. Once all the warps associated with the first selected group are blocked due to the unavailability of data, a group switch occurs giving opportunity to the next CTA group for execution (and this process continues in a round-robin fashion among all the CTA groups). This is an effective way to hide long memory latencies, as now, a core can execute the group(s) of warps that are not waiting for memory while waiting for the data for the other group(s).

How to choose $n$: A group with $n$ CTAs should have enough warps to keep the core pipeline busy in the absence of long latency operations [44]. Based on the GPU core’s scheduling model described in Section 2, we set the minimum number of warps in a group to the number of pipeline stages (5 in our case). It means that, the minimum value of $n \times k$ should be 5. Since $k$ depends on the GPGPU application kernel, the group size can vary for different application kernels. As each group can only have integral number of CTAs ($n$), we start with $n = 1$. If $n \times k$ is still smaller than the minimum number of warps in a group, we increase $n$ by 1 until we have enough warps in the group for a particular application kernel. After the first group is formed, remaining groups are also formed in a similar fashion. For example, assume that the total number of CTAs launched on a core is $N = 10$. Also, assume that the number of pipeline stages is 5, and the number of warps in a CTA ($k$) is 2. In this case, the size of the first group ($n$) will be set to 3 CTAs, as now, a group will have 6 ($3 \times 2$) warps, satisfying the minimum requirement of 5 (number of pipeline stages). The second group will follow the same method and have 3 CTAs. Now, note that the third group will have 4 CTAs to include the remaining CTAs. The third group cannot have only 3 CTAs ($n = 3$), because that will push the last CTA ($10^{th}$ CTA) to become the fourth group by itself, violating the minimum group size (in warps) requirement for the fourth group. We call this scheme CTA-aware two-level scheduling (CTA-Aware), as the groups are formed taking CTA boundaries into consideration and a two-level scheduling policy is employed, where scheduling within a group (level 1) and switching among different groups (level 2) are both done in a round-robin fashion.

The need to be CTA-aware: Two types of data locality are primarily present in GPGPU applications [21, 44, 51]: (1) Intra-warp data locality, and (2) Intra-CTA (inter-warp) data locality. Intra-warp locality is due to the threads in a warp that share contiguous elements of an array, which are typically coalesced to the same cache line. This locality is exploited by keeping the threads of a warp together. Intra-CTA locality results from warps within the same thread-block sharing blocks or rows of data. Typically, data associated with one CTA is first moved to the on-chip memories and is followed by the computation on it. Finally, the results are written back to the main global memory. Since the difference between access latencies of on-chip and off-chip memories is very high [5], it is critical to optimally utilize the data brought on-chip and maximize reuse opportunities. Prioritizing some group of warps agnostic to the CTA boundaries may not utilize the data brought on-chip to the full extent (because it may cause eviction of data that is reused across different warps in the same CTA). Thus, it is important to be CTA-aware when forming groups.

4.2 CTA-Aware-Locality: Locality aware warp scheduling

Although CTA-Aware scheduling we just described is effective in hiding the long memory fetch latencies, it does not effectively utilize the private L1 cache capacity associated with every core. Given the fact that L1 data caches of the state-of-the-art GPGPU architectures are in the 16-64 KB range [48] (as well as in CMPs [22]), in most cases, the data brought by a large number of CTAs executing simultaneously does not fit into the cache (this is true for a majority of the memory-intensive applications). This hampers the opportunity of reusing the data brought by warps, eventually leading to a high number of L1 misses. In fact, this problem is more severe with the RR scheduling policy, where the number of simultaneously executing CTAs taking advantage of the caches in a given interval of time is more than that with our CTA-Aware scheduling policy. In many-core and SMT architectures, others [11, 55] also observed a similar phenomenon, where many simultaneously executing threads (which share the same cache) cause cache contention, leading to increased cache misses. One might argue that, this situation can be addressed by increasing the size of L1 caches, but that would lead to (1) higher cache access latency, and (2) reduced hardware resources dedicated for computation, thereby hampering parallelism and the ability of the architecture to hide memory latency further.

Problem: In order to understand the problem with CTA-Aware scheme, consider Figure 4 (A). Without the loss of generality, let us assume that the group size is equal to 1. Further, assume that at core 1, CTA 1 belongs to group 1, CTA 3 belongs to group 2, etc., and each CTA has enough warps to keep the core pipeline busy ($1 \times k \geq$ number of pipeline stages). According to CTA-Aware, the warps of group 1 are prioritized until they are blocked waiting for memory. At this point, the warps of CTA 3 are executed. If the warps of CTA 1 become ready to execute (because their data

![Figure 4](https://example.com/figure4.png)
arrives from memory) when the core is executing warps of CTA 5 (Figure 4 (A)), CTA-Aware will keep executing the warps of CTA 5 (and will continue to CTA 7 after that). It will not choose the warps from CTA 1 even though they are ready because it follows a strict round-robin policy among different CTAs. Thus, the data brought by the warps of CTA 1 early on (before they were stalled) becomes more likely to get evicted by other CTAs’ data as the core keeps on executing the CTAs in a round-robin fashion. This strict round robin scheduling scheme allows larger number of threads to bring data to the relatively small L1 caches, thereby increasing cache contention due to the differences in the data sets of different CTAs and hampering the effective reuse of data in the caches. Although CTA-Aware performs better in utilizing L1 caches compared to RR (because it restricts the number of warps sharing the L1 cache simultaneously), it is far from optimal.

**Solution:** To achieve better L1 hit rates, we strive to reduce the number of simultaneously executing CTAs taking advantage of L1 caches in a particular time interval. Out of N CTAs launched on a core, the goal is to *always* prioritize only one of the CTA groups of size n. n is chosen by the method described in Section 4.1. In general, on a particular core, CTA-Aware-Locality starts scheduling warps from group 1. If warps associated with group 1 (whose size is n CTAs) are blocked due to unavailability of data, the scheduler can schedule warps from group 2. This is essential to keep the core pipeline busy. However, as soon as any warps from group 1 are ready (i.e., their requested data has arrived), CTA-Aware-Locality again prioritizes these group 1 warps. If all warps belonging to group 1 have completed their execution, the next group (group 2) is chosen and is always prioritized. This process continues until all the launched CTAs finish their execution.

The primary motivation of using this scheme is that, in a particular time interval, only n CTAs are given higher priority to keep their data in the private caches such that they get the opportunity to reuse it. Since this scheme reduces contention and increases reuse in the L1 cache, we call it locality aware warp scheduling (CTA-Aware-Locality). Note that, as n is chosen by the method described in Section 4.1, in a particular core, CTA-Aware-Locality starts scheduling warps from group 1. If warps associated with group 1 (whose size is n CTAs) are blocked due to unavailability of data, the scheduler can schedule warps from group 2. This is essential to keep the core pipeline busy. However, as soon as any warps from group 1 are ready (i.e., their requested data has arrived), CTA-Aware-Locality again prioritizes these group 1 warps. If all warps belonging to group 1 have completed their execution, the next group (group 2) is chosen and is always prioritized. This process continues until all the launched CTAs finish their execution.

In the previous section, we discussed how CTA-Aware-Locality helps in hiding memory latency along with reducing L1 miss rates. In this section, we propose CTA-Aware-Locality-BLP, which not only incorporates the benefits of CTA-Aware-Locality, but also improves DRAM bank-level parallelism (BLP) [41].

**Problem:** In our study of 38 applications, we observe that the same DRAM row is accessed (shared) by consecutive CTAs 64% of the time. Table 4 shows these row sharing percentages for all the Type-1 applications. This metric is determined by calculating the average fraction of consecutive CTAs (out of total CTAs) accessing the same DRAM row, averaged across all rows. For example, if a row is accessed by CTAs 1, 2, 3, and 4, its consecutive CTA row sharing percentage is deemed to be 100% (as all CTAs are consecutive). We observe that for many GPGPU applications, the consecutive CTA row sharing percentages are very high (up to 99% in JPEG). For example, in Figure 1 (B), we observe that the row sharing percentage is 100%, as CTA 1 opens 2 rows in Bank 1 (A(0,0) and A(0,1)) and Bank 2 (A(1,0) and A(1,1)); and, CTA 2 opens the same rows again as the data needed by it to execute is also mapped to the same rows. These high consecutive CTA row sharing percentages are not surprising, as CUDA programmers are encouraged to form CTAs such that the data required by the consecutive CTAs is mapped to the same DRAM row for high DRAM row locality, improving DRAM bandwidth utilization [28]. Also, many data layout optimizations are proposed to make CTA conform to the DRAM layout [54] to get the maximum performance.

### Table 3. Reduction in combined L1 miss rates (texture, constant, data) with our warp scheduling mechanisms over baseline RR scheduling.

<table>
<thead>
<tr>
<th>App</th>
<th>CTA-Aware</th>
<th>CTA-Aware-Locality</th>
<th>#</th>
<th>App</th>
<th>CTA-Aware</th>
<th>CTA-Aware-Locality</th>
<th>#</th>
<th>App</th>
<th>CTA-Aware</th>
<th>CTA-Aware-Locality</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSAD</td>
<td>10%</td>
<td>42%</td>
<td>7</td>
<td>KMN</td>
<td>27%</td>
<td>49%</td>
<td>14</td>
<td>SC</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>PVC</td>
<td>89%</td>
<td>90%</td>
<td>8</td>
<td>SCP</td>
<td>0%</td>
<td>0%</td>
<td>15</td>
<td>FF1</td>
<td>1%</td>
<td>1%</td>
</tr>
<tr>
<td>SSC</td>
<td>1%</td>
<td>8%</td>
<td>9</td>
<td>FWT</td>
<td>0%</td>
<td>0%</td>
<td>16</td>
<td>SD2</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>BPF</td>
<td>6%</td>
<td>17%</td>
<td>10</td>
<td>BX</td>
<td>27%</td>
<td>96%</td>
<td>12</td>
<td>WP</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>WM</td>
<td>1%</td>
<td>5%</td>
<td>11</td>
<td>SPARK</td>
<td>5%</td>
<td>9%</td>
<td>18</td>
<td>PVR</td>
<td>1%</td>
<td>2%</td>
</tr>
<tr>
<td>CPD</td>
<td>1%</td>
<td>2%</td>
<td>12</td>
<td>JPEG</td>
<td>0%</td>
<td>0%</td>
<td>19</td>
<td>BP</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>BSFR</td>
<td>1%</td>
<td>2%</td>
<td>13</td>
<td>AVG-TI</td>
<td>16%</td>
<td>8%</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 4. GPGPU application characteristics: Consecutive CTA row sharing: Fraction of consecutive CTAs (out of all CTAs) accessing the same DRAM row. CTAs/Row: Average number of CTAs accessing the same DRAM row.

<table>
<thead>
<tr>
<th>#</th>
<th>App</th>
<th>Cons. CTA row sharing</th>
<th>#</th>
<th>App</th>
<th>Cons. CTA row sharing</th>
<th>1</th>
<th>App</th>
<th>Cons. CTA row sharing</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SAD</td>
<td>42%</td>
<td>1</td>
<td>14</td>
<td>SC</td>
<td>1%</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>PVC</td>
<td>36%</td>
<td>2</td>
<td>8</td>
<td>SCP</td>
<td>0%</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>SSS</td>
<td>20%</td>
<td>2</td>
<td>9</td>
<td>FWT</td>
<td>85%</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>BFS</td>
<td>23%</td>
<td>5</td>
<td>10</td>
<td>IX</td>
<td>95%</td>
<td>2</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td>MUM</td>
<td>17%</td>
<td>32</td>
<td>11</td>
<td>SPMV</td>
<td>98%</td>
<td>6</td>
<td>18</td>
</tr>
<tr>
<td>6</td>
<td>CFD</td>
<td>81%</td>
<td>10</td>
<td>12</td>
<td>JPEG</td>
<td>99%</td>
<td>16</td>
<td>19</td>
</tr>
</tbody>
</table>

In Section 4.2, we proposed CTA-Aware-Locality where a subset of CTAs (one group) is always prioritized over others. Although this scheme is effective at reducing cache contention and improving per-core performance, it takes decisions agnostic to inter-CTA row sharing properties. Consider a scenario where two consecutive CTA groups are scheduled on two different cores and are being always prioritized according to CTA-Aware-Locality. Given that the consecutive CTAs (in turn warps) share DRAM rows, the CTA groups access a small set of DRAM banks more frequently. This increases the queuing time at the banks and reduces the bank level parallelism (BLP). To understand this problem in-depth, let us revisit Figure 1 (C), which shows the row-major data layout of CTAs in DRAM [28]. The elements in row 0 of the matrix in Figure 1 (B) are mapped to a single row in bank 1, elements in row 1 are mapped to bank 2, and so on. To maximize row locality, it is important that the row that is loaded to a row buffer in a bank is utilized to the maximum, as row buffer hit latency (10 DRAM cycles \( t_{CL} \)) is almost twice cheaper than row closed latency (22 DRAM cycles \( t_{RC} + t_{CL} \)).

Figure 5 (A) depicts this phenomenon pictorially. Since consecutive CTAs (CTAs 1 and 2) share the same rows, prioritizing them in different cores enables them to access these same rows concurrently, thereby providing high row buffer hit rate. Unfortunately, for the exact same reason, prioritizing consecutive CTAs in different cores leads to low BLP because all DRAM banks are not utilized as consecutive CTAs access the same banks (In Figure 5 (A), two banks stay idle). Our goal is to develop a series of techniques that achieve both high BLP and high row buffer hit rate. First, we describe a bank-level parallelism aware warp scheduling mechanism, CTA-Aware-Locality-BLP, which improves BLP at the expense of row locality.

**Solution:** To address the above problem, we propose CTA-Aware-Locality-BLP, which not only inherits the positive aspects of CTA-Aware-Locality (better L1 hit rates), but also improves DRAM bank level parallelism. The key idea is to still always prioritize one CTA group in each core, but to ensure that non-consecutive CTAs (i.e., CTAs that do not share rows) are always prioritized in different cores. This improves the likelihood that the executing CTA groups (warps) in different cores access different banks, thereby improving bank level parallelism.

Figure 5 (B) depicts the working of CTA-Aware-Locality-BLP pictorially with an example. Instead of prioritizing consecutive CTAs (CTAs 1 and 2) in the two cores, CTA-Aware-Locality-BLP prioritizes non-consecutive ones (CTAs 1 and 4). This enables all four banks to be utilized concurrently, instead of two banks staying idle, which was the case with CTA-Aware-Locality (depicted in Figure 5 (A)). Hence, prioritizing non-consecutive CTAs in different cores leads to improved BLP. Note that this comes at the expense of row buffer locality, which we will restore with our next proposal, Opportunistic Prefetching (Section 4.4).

One way to implement the key idea of CTA-Aware-Locality-BLP is to prioritize different-numbered CTAs in consecutive cores concurrently, instead of prioritizing the same-numbered CTA groups in each core concurrently. In other words, the warp scheduler in each core prioritizes, for example, the first CTA group in core 1, the second CTA group in core 2, the third CTA group in core 3, and so on. Since different-numbered CTA groups are unlikely to share DRAM rows, this technique is likely to maximize parallelism. Algorithm 1 more formally depicts the group formation and group priority assignment strategies for the three schemes we have proposed so far.

**Discussion:** Figure 6 shows the change in BLP and row buffer hit rate with CTA-Aware-Locality-BLP compared to CTA-Aware-Locality. Across Type-1 applications, there is an 11% average increase in BLP (AVG-T1), which not only reduces the DRAM queuing latency by 12%, but also reduces overall memory fetch latency by 22%. In JPEG, the BLP improvement is 46%. When CTA-Aware-Locality-BLP is incorporated, we observe 14% average reduction in row locality among all Type-1 applications. We note that, even though there is a significant increase in BLP, the decrease in row locality (e.g., in JPEG, SD2) is a concern, because reduced row locality.

![Figure 5](image-url) An illustration showing (A) the under-utilization of DRAM banks with CTA-Aware-Locality, (B) improved bank-level parallelism with CTA-Aware-Locality-BLP, (C1, C2) the positive effects of Opportunistic Prefetching.
If these CT As do not access the row when the row is fetched into DRAM, in the previous section, we discussed how opportunistic prefetching can hinder performance. Our goal is to restore row buffer locality and hence efficiently utilize an open row as much as possible while keeping the benefits of improved BLP. Solution: We observe that prefetching cache blocks of an already open row can achieve this goal: if the prefetched cache blocks are later needed by other CT As, these CT As will find the prefetched data in the cache and hence do not need to access DRAM. As such, in the best case, even though CT As that access the same row get scheduled at different times, they would not re-open the row over and over because opportunistic prefetching would prefetch all the needed data into the caches.

The key idea of opportunistic prefetching is to prefetch the so-far-unfetched cache lines in an already open row into the L2 caches, just before the row is closed (i.e., after all the demand requests to the row in the memory request buffer are served). We call this opportunistic because the prefetcher, sitting in the memory controller, takes advantage of a row that was already opened by a demand request, in an opportunistic way. The prefetched lines can be useful for both currently executing CT As, as well as, CT As that will be launched later. Figure 5 (C1, C2) depicts the potential benefit of this scheme. In Figure 5 (C1), during the execution of CT As 1 and 4, our proposal prefetches the data from the open rows so-far-unfetched cache lines in an already open row into the L2 cache slice associated with the memory controller. The intuition is that a demand request to a different row arrives. Therefore, our second scheme of demand requests (after the row was opened the last time) from the same row to the L2 cache slice associated with the memory controller. In addition, additional latency incurred by the demand request if prefetches were continued to be issued to the open row even after the demand arrives can be hidden in GPGPUs due to the existence of a large number of warps. Hence, it may be worthwhile to keep prefetching even after a demand to a different row arrives. Therefore, our second scheme prefetches at least a minimum number of cache lines (C) regardless of whether or not a demand arrives. The value of C is set to a value lower initially. The prefetcher continuously monitors the number of demand requests at the memory controller queue. If that number is less than a threshold, the value of C is set to a value higher. The idea is that if there are few demand requests waiting, it could be beneficial to keep prefetching. In our baseline implementation, we efficiently utilized. In fact, since CTA-Aware-Locality-BLP tries to schedule different CTAs that access the same row at different times to improve BLP, these different CTAs will need to re-open the row over and over before accessing it. Hence, large losses in row locality are possible (and we have observed these, as shown in Figure 6), which can hinder performance. Our goal is to restore row buffer locality (and hence efficiently utilize an open row as much as possible) while keeping the benefits of improved BLP.

### Algorithm 1 Group formation and priority assignment

- k is the number of warps in a CTA
- n is the number of CTAs scheduled on a core
- g.size is the minimum number of warps in a group
- g.core is the number of groups scheduled on a core
- num.core is the total number of cores in GPGPU
- group.size[i] is the group size (in number of CTAs) of the ith group
- g.pri[i][j] is the group priority of the jth group scheduled on the ith core.
- The lower the g.pri[i][j], the higher the scheduling priority. Once a group is chosen, the scheduler cannot choose warps from different group(s) unless all warps of the already-chosen group are blocked because of unavailability of data.

```plaintext
procedure FORM_GROUPS
    n ← 1
    while (n × k) < g.size do
        n ← n + 1
        g.core ← ⌈N/n⌉
        for g.num = 0 to (g.core - 1) do
            group.size[g.num] ← n
        if (N mod n) ≠ 0 then
            group.size[g.core - 1] ← group.size[g.core - 1] + (N mod n)
    
    procedure CTA-AWARE FORM_GROUPS
        for core.ID = 0 to (num.core - 1) do
            g.pri[0][core.ID] ← 0
        for g.num = 0 to (g.core - 1) do
            g.pri[g.num][core.ID] ← g.num
    
    procedure CTA-AWARE-LOCALITY FORM_GROUPS
        for core.ID = 0 to (num.core - 1) do
            g.pri[g.num][core.ID] ← (g.num - core.ID) mod g.core
```

This problem, we propose our final scheme, memory-side Opportunistic Prefetching.

**Figure 6.** Effect of CTA-Aware-Locality-BLP on DRAM bank-level parallelism and row locality, compared to CTA-Aware-Locality.

### 4.4 Opportunistic Prefetching

In the previous section, we discussed how CTA-Aware-Locality-BLP improves bank-level parallelism, but this comes at the cost of row locality. Our evaluations show that, on average, 15 CTAs access the same DRAM row (shown under CTAs/row in Table 4). If these CTAs do not access the row when the row is fetched into the row buffer the first time, data in the row buffer will not be efficiently utilized. In fact, since CTA-Aware-Locality-BLP tries to schedule different CTAs that access the same row at different times to improve BLP, these different CTAs will need to re-open the row over and over before accessing it. Hence, large losses in row locality are possible (and we have observed these, as shown in Figure 6), which can hinder performance. Our goal is to restore row buffer locality (and hence efficiently utilize an open row as much as possible) while keeping the benefits of improved BLP.
set lower to 8, higher to 16, and threshold to the average number of pending requests at the memory controller. Section 5.1 explores sensitivity to these parameters. More sophisticated mechanisms are left as part of future work.

4.5 Hardware Overheads

CTA-aware scheduling: The nVIDIA warp scheduler has low warp-switching overhead [37] and warps can be scheduled according to their pre-determined priorities. We take advantage of such priority-based warp scheduler implementations already available in existing GPGPUs. Extra hardware is needed to dynamically calculate the priorities of the warps using our schemes (Algorithm 1). In addition, every core should have a group formation mechanism similar to Narasimam et al.’s proposal [44]. We synthesized the RTL design of the hardware required for our warp scheduler using the 65nm TSMC libraries in the Synopsys Design Compiler. For a 28-core system, the area overhead is 0.18 mm².

Opportunistic prefetching: Opportunistic prefetching requires the prefetcher to know which cache lines in a row were already sent to the L2. To keep track of this for the currently-open row in a bank, we add n bits to the memory controller, corresponding to n cache lines in the row. When the row is opened, the n bits are reset. When a cache block is sent to the L2 cache from a row, its corresponding bit is set. For 8 MCs, each controlling 4 banks, with a row size of 32 cache blocks (assuming column size of 64B), the hardware overhead is 1024 bits (8 × 4 × 32 bits). The second prefetching mechanism we propose also requires extra hardware to keep track of the average number of pending requests at the memory controller. This range of this register is 0-127 and its value is computed approximately with the aid of shift registers.

5. Experimental Results

In this section, we evaluate our proposed scheduling and memory-side prefetching schemes with 19 Type-1 applications, where main memory is the main cause of core idleness.

5.1 Performance Results

We start with evaluating the performance impact of our scheduling schemes (in the order of their appearance in the paper) against the Perfect-L2 case, where all memory requests are L2 cache hits. We also show results with Perfect-L1 (PMEM), which is the ultimate upper bound of our optimizations. Recall that each scheme builds on top of the previous.

Effect of CTA-Aware: We discussed in Section 4.1 that this scheme not only helps in hiding memory latency, but also partially reduces cache contention. Figure 7 shows the IPC improvements of Type-1 applications (normalized to RR). Figure 8 shows the impact of our scheduling schemes on MemoryBlockCycles as described in Section 3. On average (arithmetic mean), CTA-Aware provides 14% (9% harmonic mean (hmean), 11% geometric mean (gmean)) IPC improvement, with 9% reduction in memory waiting time (MemoryBlockCycles) over RR. The primary advantage comes from the reduction in L1 miss rates and improvement in memory latency hiding capability due to CTA grouping. We observe significant IPC improvements in PVC (2.5×) and IIX (1.22×) applications, as the miss rate drastically reduces by 89% and 27%, respectively. As expected, we do not observe significant performance improvements in SD2, WP, and SPMV as there is no reduction in miss rate compared to RR. We see improvements in JPEG (6%) and SCP (19%), even though there is no reduction in miss-rates (see Table 3). Most of the benefits in these benchmarks are due to the better hiding of memory latency, which comes inherently from the CTA-aware two-level scheduling. We further observe (not shown) that CTA-Aware achieves similar performance benefits compared to the recently proposed two-level warp scheduling [44]. In contrast to [44], by introducing awareness of CTAs, our CTA-Aware warp scheduling mechanism provides a strong foundation for the remaining three schemes we develop.

Effect of CTA-Aware-Locality: The main advantage of this scheme is further reduced L1 miss rates. We observe 11% average IPC improvement (6% decrease in MemoryBlockCycles) over CTA-Aware, and 25% (17% hmean, 21% gmean) over RR. We observe 81% IPC improvement in IIX, primarily because of 69% in L1 miss rates. Because of the row locality and BLP trade-off (this scheme sacrifices BLP for increased row locality), we observe that some applications may not attain optimal benefit from CTA-Aware-Locality. For example, in SC, IPC decreases by 4% and MemoryBlockCycles increases by 3% compared to CTA-Aware, due to a 26% reduction in BLP (7% increase in row locality). We also observe similar results in MUM: 1% increase in row locality, 10% reduction in BLP, which causes 3% reduction in performance compared to CTA-Aware. In SD2, we observe a 7% IPC improvement over CTA-Aware on account of a 14% increase in row-locality, with a 21% reduction in BLP. Nevertheless, the primary advantage of CTA-Aware-Locality is the reduced number of memory requests due to better cache utilization (Section 4.2), and as a result of this, we also observe an improvement in DRAM bandwidth utilization due to reduced contention in DRAM banks.

Effect of CTA-Aware-Locality-BLP: In this scheme, we strive to achieve better BLP at the cost of row locality. Using this scheme, on average, we observe 6% IPC (4% hmean, 4% gmean) improvement, and 3% decrease in MemoryBlockCycles over CTA-Aware-Locality. BLP increases by 11%, which also helps in the observed 22% reduction in overall memory fetch latency (12% reduction in queuing latency). In SD2, we see a significant increase in BLP (48%) over CTA-Aware-Locality, but performance still reduces (by

Figure 7. Performance impact of our schemes on Type-1 applications. Results are normalized to RR.
10%) compared to CTA-Aware-Locality, due to a 46% reduction in row locality. In contrast, in JPEG, the effects of the 62% reduction in row locality is outweighed by the 46% increase in BLP, yielding a 10% IPC improvement over CTA-Aware-Locality. This shows that both row locality and BLP are important for GPGPU performance.

Combined Effect of OWL (Integration of CTA-Aware-Locality-BLP and opportunistic prefetching): The fourth bar from the left in Figure 7 shows the performance of the system with OWL. We can draw four main conclusions from this graph. First, using opportunistic prefetching on top of CTA-Aware-Locality-BLP consistently either improves performance or has no effect. Second, on average, even a simple prefetching scheme like ours can provide an IPC improvement of 2% over CTA-Aware-Locality-BLP, which is due to a 12% improvement in L2 cache hit rate. Overall, OWL achieves 19% (14% hmean, 17% gmean) IPC improvement over CTA-Aware and 33% (23% hmean, 28% gmean) IPC improvement over RR. Third, a few applications, such as JPEG, gain significantly (up to 15% in IPC) due to opportunistic prefetching, while others, such as FWT, SPMV, and SD2, gain only moderately (around 5%), and some do not have any noticeable gains, e.g., SAD, PVC, and WP. The variation seen in improvements across different applications can be attributed to their different memory latency hiding capabilities and memory access patterns. It is interesting to note that, in SCP, FWT, and KMN, some rows are accessed by only one or two CTAs. The required data in these rows are demanded when they are opened for the first time. In these situations, even if we prefetch all the remaining lines, we do not observe significant improvements. Fourth, we find that the scope of improvement available for opportunistic prefetching over CTA-Aware-Locality-BLP is limited: Perfect-L2 can provide only 13% improvement over CTA-Aware-Locality-BLP. This is mainly because if an application inherently has a large number of warps ready to execute, the application will also be able to efficiently hide the long memory access latency. We observe that prefetching might not be beneficial in these applications even if the prefetch-accuracy is 100%.

We conclude that the proposed schemes are effective at improving GPGPU performance by making memory less of a bottleneck. As a result, OWL enables the evaluated GPGPU to have performance within 11% of a hypothetical GPGPU with a perfect L2.

5.2 Sensitivity Studies

In this section, we describe the critical sensitivity studies we performed related to group size, DRAM configuration and opportunistic prefetching.

Sensitivity to group size: In Section 4.1, we mentioned that the minimum number of warps in a group should be at least equal to the number of pipeline stages. Narasiman et al. [44] advocated that, if the group size is too small, the data fetched in DRAM row buffers is not completely utilized, as fewer warps are prioritized together. If the group size is too large, the benefits of two-level scheduling diminishes. Figure 9 shows the effect of the group size on performance. The results are normalized to RR and averaged across all Type-1 applications. We observe that when minimum group size is 8 warps, we get the best IPC improvements (14% for CTA-Aware, 25% for CTA-Aware-Locality and 31% for CTA-Aware-Locality-BLP over RR), and thus, throughout our work, we have used a minimum group size of 8, instead of 5 (which is the number of pipeline stages).

Sensitivity to the number of DRAM banks: Figure 10 shows the change in performance of CTA-Aware-Locality-BLP with the number of DRAM banks per MC. We observe that as the number of banks increases, the effectiveness of CTA-Aware-Locality-BLP increases. This is because having additional banks enables more benefits from exposing higher levels of BLP via our proposed techniques. As a result, the performance improvement of our proposal is 2% higher with 8 banks per MC than with 4 banks per MC (our baseline system). We conclude that our techniques are likely to become more effective in future systems with more banks.

Sensitivity to Opportunistic Prefetching Parameters: We experimented with all combinations of lower and upper values for the prefetch degree in the range of 0 (no-prefetching) to 32 (prefetching all the columns in a row) with a step size of 8. The value of threshold is also varied similarly, along with the case when it is equal to the average memory controller queue length. Figure 11 shows the best case values achieved across all evaluated combinations (Best OWL). We find that the average performance improvement achievable by tuning these parameter values is only 1% (compare Best OWL vs. OWL). This can possibly be achieved by implementing a sophisticated prefetcher that can dynamically adjust its parameters based on the running application’s characteristics, which comes at the cost of increased hardware complexity. We leave the design of such application-aware memory-side prefetchers as a part of the fu-
6. Related Work

To our knowledge, this is the first paper in the context of GPGPUs to propose (1) CTA-aware warp scheduling techniques to improve both cache hit rates and DRAM bank-level parallelism, and (2) memory-side prefetching mechanisms to improve overall GPPGU performance by taking advantage of open DRAM row buffers. We briefly describe the closely related works in this section.

Scheduling in GPGPUs: The two-level warp scheduling mechanism proposed by Narasiman et al. [44] increases the core utilization by creating larger warps and employing a two-level warp scheduling scheme. This mechanism is not aware of CTA boundaries. In our work, we propose CTA-aware warp scheduling policies, which improve not only L1 hit rates, but also DRAM bandwidth utilization. We find that the combination of all of our techniques, OWL, provides approximately 19% higher performance than two-level warp scheduling. Gebhart et al. [17] also proposed a two-level warp scheduling technique. Energy reduction is the primary purpose of their approach. Even though we do not evaluate it, OWL is also likely to provide energy benefits as reduced execution time (with low hardware overhead) is likely to translate into reduced energy consumption. Concurrent work by Rogers et al. [51] proposed a cache-conscious warp scheduling policy. Their work improves L1 hit rates for cache-sensitive applications. OWL not only reduces cache contention, but also improves DRAM bandwidth utilization for a wide range of applications. Recent work from Kayiran et al. [24] dynamically estimates the amount of thread-level parallelism that would improve GPPGU performance by reducing cache and DRAM contention. Our approach is orthogonal to theirs as our CTA-aware scheduling techniques improve cache and DRAM utilization for a given amount of thread-level parallelism.

BLP and Row Locality: Bank-level parallelism and row buffer locality are two important characteristics of DRAM performance. Several memory request scheduling [12, 26, 27, 33–35, 41, 50, 58, 59] and data partitioning [20, 40, 57] techniques have been proposed to improve one or both within the context of multi-core, GPPGU, and heterogeneous CPU-GPU systems [3]. Our work can be combined with these approaches. Mutlu and Moschibroda [41] describe parallelism-aware batch scheduling, which aims to preserve each thread’s BLP in a multi-core system. Hassan et al. [18] suggest that optimizing BLP is more important than improving row buffer hits, even though there is a trade-off. Our work uses this observation to focus on enhancing BLP, while restoring the lost row locality by memory-side prefetching. This is important because, in some GPPGU applications, we observe that both BLP and row locality are important. Similar to our proposal, Jeong et al. [20] observe that both BLP and row locality are important for maximizing benefits in multi-core systems. The memory access scheduling proposed by Yuan et al. [58] restores the lost row access locality caused by the in-order DRAM scheduler, by incorporating an arbitration mechanism in the interconnection network. The staged memory scheduler of Ausavathurungnirun et al. [3] batches memory requests going to the same row to improve row locality while also employing simple in-order request scheduling at the DRAM banks. Lakshminarayana et al. [33] propose a potential function that models the DRAM behavior in GPPGU architectures and a SJF DRAM scheduling policy. The scheduling policy essentially chooses between SJF and FR-FCFS at run-time based on the number of requests from each thread and their potential of generating a row buffer hit. In our work, we propose low-overhead warp scheduling and prefetching schemes to improve both row locality and BLP. Exploration of the combination of our warp scheduling techniques with memory request scheduling and data partitioning techniques is a promising area of future work.

Data Prefetching: To our knowledge, OWL is the first work that uses a memory-side prefetcher in GPUs. Our opportunistic prefetcher complements the CTA-aware scheduling schemes by taking advantage of open DRAM rows. The most relevant work on hardware prefetching in GPUs is the L1 prefetcher proposed by Lee et al. [36]. Carter et al. [9] present one of the earliest works done in the area of memory-side prefetching in the CPU domain. Many other prefetching mechanisms (e.g., [13, 23, 45, 52]) have been proposed within the context of CPU systems. Our contribution in this work is a specific prefetching algorithm (in fact, our proposal can potentially use the algorithms proposed in literature), but to employ the idea prefetching in conjunction with new BLP-aware warp scheduling techniques to restore row buffer locality and improve L1 hit rates in GPPGUs.

7. Conclusion

This paper proposes a new warp scheduling policy, OWL, to enhance GPPGU performance by overcoming the resource under-utilization problem caused by long latency memory operations. The key idea in OWL is to take advantage of characteristics of cooperative thread arrays (CTAs) to concurrently improve cache hit rate, latency hiding capability, and DRAM bank parallelism in GPPGU. OWL achieves these benefits by 1) selecting and prioritizing a group of CTAs scheduled on a core, thereby improving both L1 cache hit rates and latency tolerance, 2) scheduling CTA groups that likely do not access the same memory banks on different cores, thereby improving DRAM bank parallelism, and 3) employing opportunistic memory-side prefetching to take advantage of already-open DRAM rows, thereby improving both DRAM row locality and cache hit rates. Our experimental evaluations on a 28-core GPPGU platform demonstrate that OWL is effective in improving GPPGU performance for memory-intensive applications: it leads to 33% IPC performance improvement over the commonly-employed baseline round-robin warp scheduler, which is not aware of CTAs. We conclude that incorporating CTA awareness into GPPGU warp scheduling policies can be an effective way of enhancing GPPGU performance by reducing resource under-utilization.

Acknowledgments

We thank the anonymous reviewers, Rachata Ausavathurungnirun, Kevin Kai-Wei Chang, Mahshid Sedghi, and Bikash Sharma for their feedback on earlier drafts of this paper. This research is supported in part by NSF grants #1213052, #1152479, #1147388, #1139023, #1017882, #0963839, #0811687, #0953246 and grants from Intel and nVIDIA.
References