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

Adwait Jog[†] Onur Kayiran[†] Nachiappan Chidambaram Nachiappan[†] Asit K. Mishra[§]
Mahmut T. Kandemir[†] Onur Mutlu[*] Ravishankar Iyer[§] Chita R. Das[†]

The Pennsylvania State University[†]
University Park, PA 16802
(adwait, onur, nachi, kandemir, das)@cse.psu.edu

Carnegie Mellon University[*]
Pittsburgh, PA 15213
onur@cmu.edu

Intel Labs [§]
Hillsboro, OR 97124
(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

ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.

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 schedu(L)ing policy*, called OWL[1]. 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 threads and warps (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 polices 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

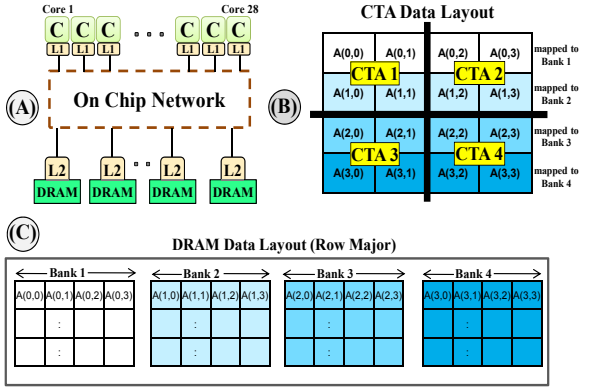

Figure 1. (A) GPGPU architecture, (B) CTA data layout, and (C) Main memory layout with CTA's data mapped.

[1] Owl is a bird known for exceptional *vision* and *focus* while it hunts for food. Our proposed scheduling policy also follows an owl's philosophy. It intelligently selects (*visualizes*) a subset of CTAs (out of many CTAs launched on a core) and *focuses* on them to achieve performance benefits.

is scalable, simple, and regular [4, 5, 43]. A detailed baseline platform configuration is described in Table 1, which is simulated on GPGPU-Sim 2.1.2b, 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 kernels 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.

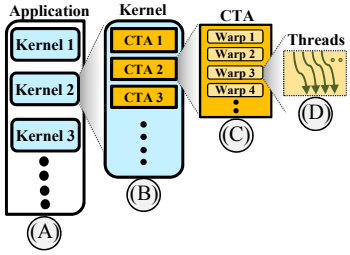

Figure 2. GPGPU application hierarchy.

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).

Table 1. Baseline configuration

Shader Core Config.	1300MHz, 5-Stage Pipeline, SIMT width = 8
Resources / Core	Max. 1024 Threads, 32KB Shared memory, 32684 Registers
Caches / Core	32KB 8-way L1 Data cache, 8KB 4-way Texture cache 8KB 4-way Constant cache, 64B line size
L2 Cache	16-way 512 KB/Memory channel, 64B line size
Scheduling	Round-robin warp scheduling, (among ready warps), Load balanced CTA scheduling
Features	Memory coalescing enabled, 32 MSHRs/core, Immediate post dominator based branch divergence handling
Interconnect	2D Mesh (6 $\times$ 6; 28 cores + 8 Memory controllers), 650MHz, 32B channel width
DRAM Model	FR-FCFS (Maximum 128 requests/MC), 8MCs, 4 DRAM banks/MC, 2KB row size
GDDR3 Timing	800MHz, t_{CL} = 10, t_{RP} = 10, t_{RC} = 35, t_{RAS} = 25 t_{RCD} = 12, t_{RRD} = 8, t_{CDLR} = 6, t_{WR} = 11

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 `IIX` 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-

ing capability of GPGPUs. To understand it further, let us consider 8 CTAs that need to be assigned to 2 cores (4 CTAs per core). According to the load-balanced CTA assignment policy described in Section 2.1, CTAs 1, 3, 5, 7 are assigned to core 1 and CTAs 2, 4, 6, 8 are assigned to core 2. With RR, warps associated with CTAs 1, 3, 5 and 7 are executed with equal priority on core 1 and are executed in a round-robin fashion. This execution continues until all the warps are blocked (when they need data from main memory). At this point, there may be no ready warps that can be scheduled, making core 1 inactive. Typically, this inactive time is very significant in memory intensive applications, as multiple requests are sent to the memory subsystem by many cores in a short period of time. This increases network and DRAM contention, which in turn increases queuing delays, leading to very high core inactive times.

To evaluate the impact of RR scheduling on GPGPU applications, we first characterize our application set. We quantify how much IPC improvement each application gains if *all* memory requests magically hit in the L1 cache. This improvement, called PMEM, is depicted in Table 2, where the 38 applications are sorted in descending order of PMEM. Applications that have high PMEM ($>= 1.4\times$) are classified as Type-1, and the rest as Type-2. We have observed that the warps of highly memory intensive applications (Type-1) wait longer for their data to come back than warps of Type-2 applications. If this wait is eliminated, the performance of `SAD`, `PVC`, and `SSC` would improve by 639%, 499% and 460%, respectively (as shown by the PMEM values for these applications).

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

Table 2. GPGPU application characteristics: *(A) PMEM*: IPC improvement with perfect memory (All memory requests are satisfied in L1 caches), Legend: H = High ($>= 1.4$x) , L = Low (< 1.4x); *(B) CINV*: The ratio of inactive cycles to the total execution cycles of all the cores.

#	App. Suite	**Type-1 Applications**	Abbr.	PMEM	CINV	#	App. Suite	**Type-2 Applications**	Abbr.	PMEM	CINV
1	Parboil	Sum of Abs. Differences	SAD	H (6.39x)	91%	20	CUDA SDK	Separable Convolution	CON	L (1.23x)	20%
2	MapReduce	PageViewCount	PVC	H (4.99x)	93%	21	CUDA SDK	AES Cryptography	AES	L (1.23x)	51%
3	MapReduce	SimilarityScore	SSC	H (4.60x)	85%	22	Rodinia	SRAD1	SD1	L (1.17x)	20%
4	CUDA SDK	Breadth First Search	BFS	H (2.77x)	81%	23	CUDA SDK	Blackscholes	BLK	L (1.16x)	17%
5	CUDA SDK	MUMerGPU	MUM	H (2.66x)	72%	24	Rodinia	HotSpot	HS	L (1.15x)	21%
6	Rodinia	CFD Solver	CFD	H (2.46x)	66%	25	CUDA SDK	Scan of Large Arrays	SLA	L (1.13x)	17%
7	Rodinia	Kmeans Clustering	KMN	H (2.43x)	65%	26	3rd Party	Denoise	DN	L (1.12x)	22%
8	CUDA SDK	Scalar Product	SCP	H (2.37x)	58%	27	CUDA SDK	3D Laplace Solver	LPS	L (1.10x)	12%
9	CUDA SDK	Fast Walsh Transform	FWT	H (2.29x)	58%	28	CUDA SDK	Neural Network	NN	L (1.10x)	13%
10	MapReduce	InvertedIndex	IIX	H (2.29x)	65%	29	Rodinia	Particle Filter (Native)	PFN	L (1.08x)	10%
11	Parboil	Sparse-Matrix-Mul.	SPMV	H (2.19x)	65%	30	Rodinia	Leukocyte	LYTE	L (1.08x)	15%
12	3rd Party	JPEG Decoding	JPEG	H (2.12x)	54%	31	Rodinia	LU Decomposition	LUD	L (1.05x)	64%
13	Rodinia	Breadth First Search	BFSR	H (2.09x)	64%	32	Parboil	Matrix Multiplication	MM	L (1.04x)	4%
14	Rodinia	Streamcluster	SC	H (1.94x)	52%	33	CUDA SDK	StoreGPU	STO	L (1.02x)	3%
15	Parboil	FFT Algorithm	FFT	H (1.56x)	37%	34	CUDA SDK	Coulombic Potential	CP	L (1.01x)	4%
16	Rodinia	SRAD2	SD2	H (1.53x)	36%	35	CUDA SDK	N-Queens Solver	NQU	L (1.01x)	95%
17	CUDA SDK	Weather Prediction	WP	H (1.50x)	54%	36	Parboil	Distance-Cutoff CP	CUTP	L (1.01x)	2%
18	MapReduce	PageViewRank	PVR	H (1.41x)	46%	37	Rodinia	Heartwall	HW	L (1.01x)	9%
19	Rodinia	Backpropogation	BP	H (1.40x)	33%	38	Parboil	Angular Correlation	TPAF	L (1.01x)	6%

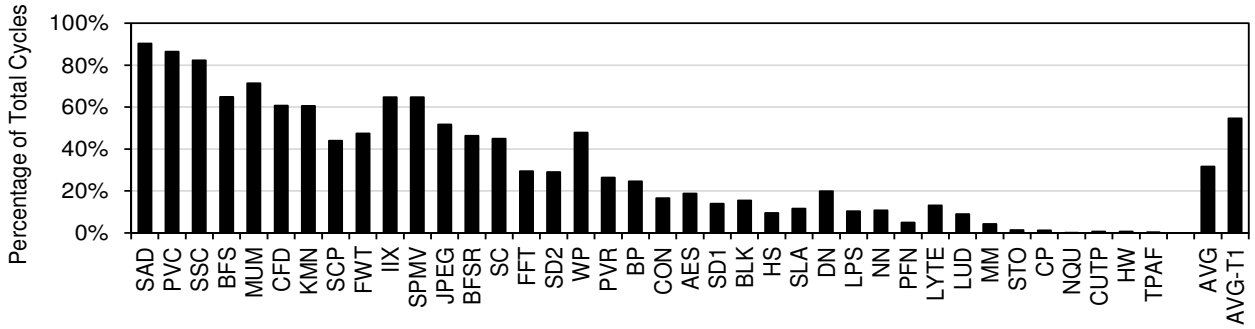

Figure 3. Fraction of total execution cycles (of all the cores) during which *all* the warps launched on a core are waiting for their respective data to come back from L2 cache/DRAM. We call the number of cycles where all warps are stalled due to memory *MemoryBlockCycles*. AVG-T1 is the average (arithmetic 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, c(O)operative thread array a(W)are warp schedu(L)ing 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 (3×2) 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

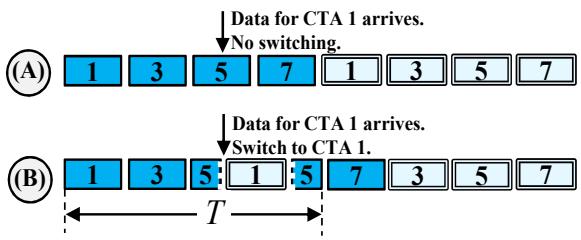

Figure 4. An illustrative example showing the working of (A) CTA-aware two-level warp scheduling (*CTA-Aware*) (B) Locality aware warp scheduling (*CTA-Aware-Locality*). Label in each box refers to the corresponding CTA number.

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

#	App.	*CTA-Aware*	*CTA-Aware-Locality*	#	App.	*CTA-Aware*	*CTA-Aware-Locality*	#	App.	*CTA-Aware*	*CTA-Aware-Locality*
1	SAD	6%	42%	7	KMN	27%	49%	14	SC	0%	0%
2	PVC	89%	90%	8	SCP	0%	0%	15	FFT	1%	1%
3	SSC	1%	8%	9	FWT	0%	0%	16	SD2	0%	0%
4	BFS	1%	17%	10	IIX	27%	96%	17	WP	0%	0%
5	MUM	1%	2%	11	SPMV	0%	8%	18	PVR	1%	2%
6	CFD	1%	2%	12	JPEG	0%	0%	19	BP	0%	0%
				13	BFSR	2%	16%		**AVG-T1**	**8%**	**18%**

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 closer to N, *CTA-Aware-Locality* degenerates into RR, as there can be only one group with N CTAs.

Typically, a GPGPU application kernel does not require fairness among the completion of different CTAs. CTAs can execute and finish in any order. The only important metric from the application's point of view is the total execution time of the kernel. A fair version of *CTA-Aware-Locality* can also be devised, where the CTA group with highest priority is changed (and accordingly, priorities of all groups will change) in a round-robin fashion (among all the groups) after a fixed interval of time. We leave the design of such schemes to future work.

Figure 4 (B) shows how *CTA-Aware-Locality* works. Again, without loss of generality, let us assume that the group size is equal to 1. We show that, *CTA-Aware-Locality* starts choosing warps belonging to CTA 1 (belonging to group 1) once they become ready, unlike *CTA-Aware*, where scheduler keeps on choosing warps from CTA 5 (group 3), 7 (group 4) and so on. In other words, we *always* prioritize a small group of CTAs (in this case, group 1 with $n = 1$) and shift the priority to the next CTA only after CTA 1 completes its execution. During time interval T, we observe that only 3 CTAs are executing and taking advantage of the private caches, contrary to 4 CTAs in the baseline system (Figure 4 (A)). This implies that a smaller number of CTAs gets the opportunity to use the L1 caches concurrently, increasing L1 hit rates and reducing cache contention.

Discussion: *CTA-Aware-Locality* aims to reduce the L1 cache misses. Table 3 shows the reduction in L1 miss rates (over baseline RR) when *CTA-Aware* and *CTA-Aware-Locality* schemes are incorporated. On average, for Type-1 applications, *CTA-Aware* reduces the overall miss rate by 8%. *CTA-Aware-Locality* is further able to reduce the overall miss rate (by 10%) by scheduling warps as soon as the data arrives for them, rather than waiting for their turn, thereby reducing the number of CTAs currently taking advantage of the L1 caches. With *CTA-Aware-Locality*, we observe maximum benefits with Map-Reduce applications `PVC` and `IIX`, where the reduction in L1 miss rates is 90% and 96%, respectively, leading to significant IPC improvements (see Section 5). Since these applications are very memory intensive (highly ranked among Type-1 applications in Table 2) and exhibit good L1 data reuse within CTAs, they significantly benefit from *CTA-Aware-Locality*. Interestingly, we find that 8 out of 19 Type-1 applications show negligible reduction in L1 miss rates with both *CTA-Aware* and *CTA-Aware-Locality*. Detailed analysis shows that these applications do not exhibit significant cache sensitivity, thus, do not provide sufficient L1 data reuse opportunities. In `WP`, because of resource limitations posed by the baseline architecture (Section 2), there are only 6 warps that can be simultaneously executed. This restriction eliminates the possibility of getting benefits from *CTA-Aware-Locality*, as only one group (with 6 warps) can be formed, and no group switching/prioritization occurs.

4.3 *CTA-Aware-Locality-BLP*: BLP aware warp scheduling

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 7, its consecutive CTA row sharing percentage is deemed to be 75% (as CTAs 1, 2, 3 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 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.

#	App.	Cons. CTA row sharing	CTAs/Row	#	App.	Cons. CTA row sharing	CTAs/Row	#	App.	Cons. CTA row sharing	CTAs/Row
1	SAD	42%	32	7	KMN	66%	2	14	SC	1%	2
2	PVC	36%	2	8	SCP	0%	1	15	FFT	14%	5
3	SSC	20%	2	9	FWT	85%	2	16	SD2	98%	35
4	BFS	23%	5	10	IIX	36%	2	17	WP	93%	7
5	MUM	17%	32	11	SPMV	98%	6	18	PVR	38%	2
6	CFD	81%	10	12	JPEG	99%	16	19	BP	99%	4
				13	BFSR	71%	8		**AVG**	**64%**	**15**

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_{RCD} + t_{CL})), and almost three times cheaper than row conflict latency (32 DRAM cycles ($t_{RP} + t_{RCD} + t_{CL}$)) [41]. *CTA-Aware-Locality* prioritizes CTA 1 (group 1) at core 1 and CTA 2 (also, group 1) at core 2. When both the groups are blocked, their memory requests access the same row in both bank 1 and bank 2, as CTA 1 and CTA 2 share the same rows (row sharing = 100%).

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 CTA groups 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

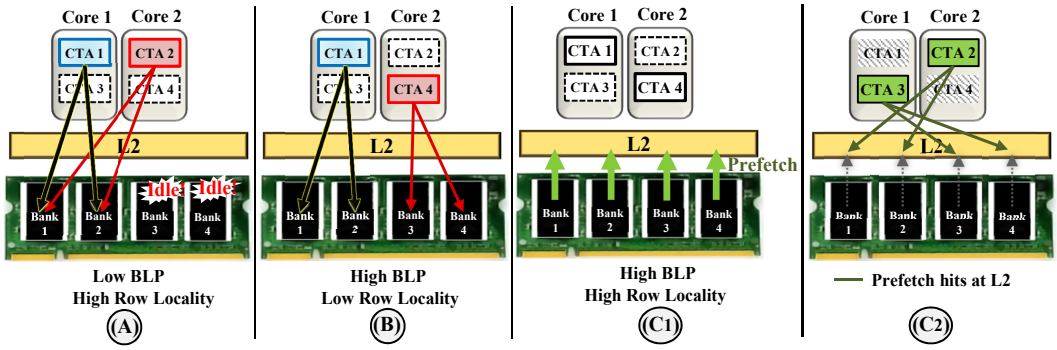

Figure 5. An example illustrating (A) the under-utilization of DRAM banks with *CTA-Aware-Locality*, (B) improved bank-level parallelism with *CTA-Aware-Locality-BLP*, (C1, C2) the positive effects of *Opportunistic Prefetching*.

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
▷ n is the minimum number of CTAs in a group
▷ g_size is the minimum number of warps in a group
▷ g_core is the number of groups scheduled on a core
▷ num_cores is the total number of cores in GPGPU
▷ $group\_size[i]$ is the group size (in number of CTAs) of the i^{th} group
▷ $g\_pri[i][j]$ is the group priority of the i^{th} group scheduled on the j^{th} 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.

procedure FORM_GROUPS
  $n \leftarrow 1$
  while $(n \times k) < g\_size$ **do**
    $n \leftarrow n + 1$
  $g\_core \leftarrow \lfloor N/n \rfloor$
  for $g\_num = 0 \rightarrow (g\_core - 1)$ **do**
    $group\_size[g\_num] \leftarrow n$
  if $(N \bmod n) \neq 0$ **then**
    $group\_size[g\_core - 1] \leftarrow group\_size[g\_core - 1] + (N \bmod n)$

procedure CTA-AWARE
  FORM_GROUPS
  for $core\_ID = 0 \rightarrow (num\_cores - 1)$ **do**
    for $g\_num = 0 \rightarrow (g\_core - 1)$ **do**
      $g\_pri[g\_num][core\_ID] \leftarrow 0$
        ▷ All groups have equal priority and executed in RR fashion.

procedure CTA-AWARE-LOCALITY
  FORM_GROUPS
  for $core\_ID = 0 \rightarrow (num\_cores - 1)$ **do**
    for $g\_num = 0 \rightarrow (g\_core - 1)$ **do**
      $g\_pri[g\_num][core\_ID] \leftarrow g\_num$

procedure CTA-AWARE-LOCALITY-BLP
  FORM_GROUPS
  for $core\_ID = 0 \rightarrow (num\_cores - 1)$ **do**
    for $g\_num = 0 \rightarrow (g\_core - 1)$ **do**
      $g\_pri[g\_num][core\_ID] \leftarrow (g\_num - core\_ID) \bmod g\_core$

locality adversely affects DRAM bandwidth utilization. To address this problem, we propose our final scheme, memory-side *Opportunistic Prefetching*.

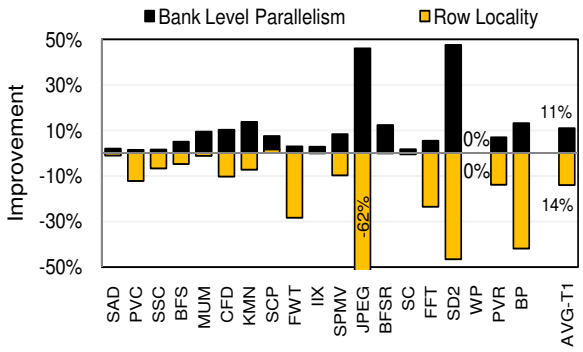

Figure 6. Effect of *CTA-Aware-Locality-BLP* on DRAM bank-level parallelism and row locality, compared to *CTA-Aware-Locality*.

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.

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 CTAs, these CTAs will find the prefetched data in the cache and hence do not need to access DRAM. As such, in the best case, even though CTAs 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 CTAs, as well as, CTAs that will be launched later. Figure 5 (C1, C2) depicts the potential benefit of this scheme. In Figure 5 (C1), during the execution of CTAs 1 and 4, our proposal prefetches the data from the open rows that could potentially be useful for other CTAs (CTAs 2 and 3 in this example). If the prefetched lines are useful (Figure 5 (C2)), when CTAs 2 and 3 execute and require data from the same row, their requests will hit in the L2 cache and hence they will not need to access DRAM for the same row.

Implementation: There are two key design decisions in our opportunistic prefetcher: *what cache lines to prefetch* and *when to stop prefetching*. In this paper, we explore simple mechanisms to provide an initial study. However, any previously proposed prefetching method can be employed (as long as they generate requests to the same row that is open) – we leave the exploration of such sophisticated techniques to future work.

What to prefetch? The prefetcher we evaluate starts prefetching when there are no more demand requests to an open row. It sequentially prefetches the cache lines that were *not* accessed by demand requests (after the row was opened the last time) from the row to the L2 cache slice associated with the memory controller.

When to stop opportunistic prefetching? We study two possible schemes, although there are many design choices possible. In the first scheme, the prefetcher stops immediately after a demand request to a *different* row arrives. The intuition is that a demand request is more critical than a prefetch, so it should be served immediately. However, this intuition may not hold true because servicing useful row-hit prefetch requests before row-conflict demand requests can eliminate future row conflicts, thereby improving performance (also shown by Lee et al. [34]). 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

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 Narasiman 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 power overhead is 57 mW and the area overhead is 0.18 mm^2, which is less than 0.05% of the nVIDIA GeForce GTX 285 area.

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 4 MCs, each controlling 4 banks, with a row size of 32 cache blocks (assuming column size of 64B), the hardware overhead is 512 bits ($4 \times 4 \times 32$ 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

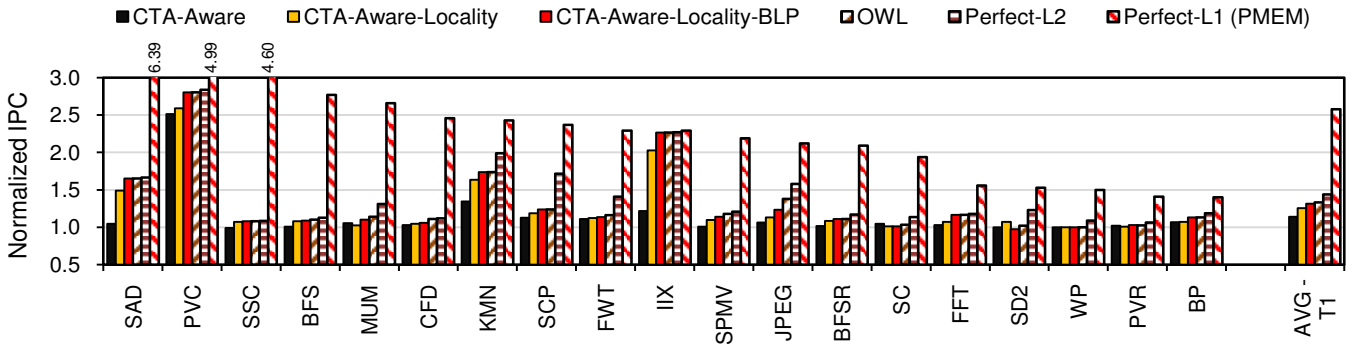

Figure 7. Performance impact of our schemes on Type-1 applications. Results are normalized to RR.

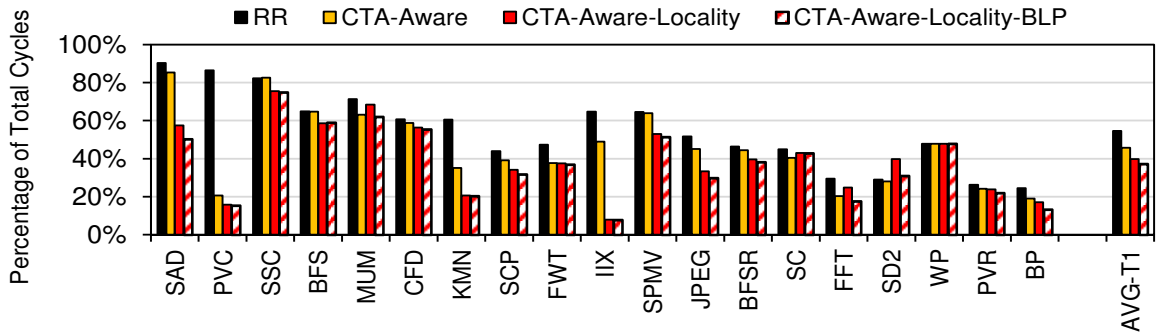

Figure 8. Impact of different scheduling schemes on *MemoryBlockCycles* for Type-1 applications. Results are normalized to the total execution cycles with baseline RR scheduling.

in queuing latency). In SD2, we see a significant increase in BLP (48%) over *CTA-Aware-Locality*, but performance still reduces (by 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.

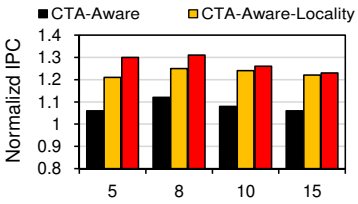

Figure 9. Sensitivity of IPC to group size (normalized to RR).

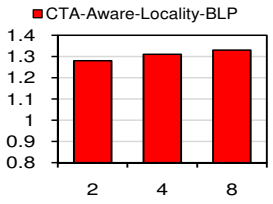

Figure 10. Sensitivity of IPC to the number of banks (normalized to RR).

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

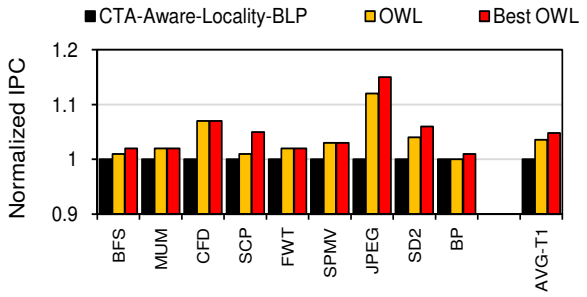

Figure 11. Prefetch degree and throttling threshold sensitivity.

the cost of increased hardware complexity. We leave the design of such application-aware memory-side prefetchers as a part of the future work, along with more sophisticated techniques to determine what parts of a row to prefetch.

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 GPGPU 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 GPGPU 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 [3, 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, GPGPU, and heterogeneous CPU-GPU systems [3]. Our work can be combined with these approaches. Mutlu and Moscibroda [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 GPGPU 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 Ausavarungnirun 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 GPGPU 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 GPGPUs.

7. Conclusion

This paper proposes a new warp scheduling policy, OWL, to enhance GPGPU 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 GPGPUs. 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 GPGPU platform demonstrate that OWL is effective in improving GPGPU 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 GPGPU warp scheduling policies can be an effective way of enhancing GPGPU performance by reducing resource under-utilization.

Acknowledgments

We thank the anonymous reviewers, Rachata Ausavarungnirun, 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

[1] AMD. Radeon and FirePro Graphics Cards, Nov. 2011.

[2] AMD. Heterogeneous Computing: OpenCL and the ATI Radeon HD 5870 (Evergreen) Architecture, Oct. 2012.

[3] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu. Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems. In *ISCA*, 2012.

[4] A. Bakhoda, J. Kim, and T. Aamodt. Throughput-effective On-chip Networks for Manycore Accelerators. In *MICRO*, 2010.

[5] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *ISPASS*, 2009.

[6] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. In *ICS 2008*.

[7] M. M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA Code Generation for Affine Programs. In *CC/ETAPS 2010*.

[8] M. Bauer, H. Cook, and B. Khailany. CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization. In *SC*, 2011.

[9] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a Smarter Memory Controller. In *HPCA*, 1999.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IISWC*, 2009.

[11] X. E. Chen and T. Aamodt. Modeling Cache Contention and Throughput of Multiprogrammed Manycore Processors. *IEEE Trans. Comput.*, 2012.

[12] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt. Parallel Application Memory Scheduling. *MICRO*, 2011.

[13] E. Ebrahimi, O. Mutlu, and Y. N. Patt. Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems. In *HPCA*, 2009.

[14] W. Fung, I. Sham, G. Yuan, and T. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *MICRO*, 2007.

[15] W. W. L. Fung and T. M. Aamodt. Thread Block Compaction for Efficient SIMT Control Flow. In *HPCA*, 2011.

[16] W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt. Hardware Transactional Memory for GPU Architectures. In *MICRO*, 2011.

[17] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron. Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors. In *ISCA*, 2011.

[18] S. Hassan, D. Choudhary, M. Rasquinha, and S. Yalamanchili. Regulating Locality vs. Parallelism Tradeoffs in Multiple Memory Controller Environments. In *PACT*, 2011.

[19] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A MapReduce Framework on Graphics Processors. In *PACT*, 2008.

[20] M. K. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez. Balancing DRAM Locality and Parallelism in Shared Memory CMP Systems . In *HPCA*, 2012.

[21] W. Jia, K. A. Shaw, and M. Martonosi. Characterizing and Improving the Use of Demand-fetched Caches in GPUs. In *ICS*, 2012.

[22] A. Jog, A. K. Mishra, C. Xu, Y. Xie, V. Narayanan, R. Iyer, and C. R. Das. Cache Revive: Architecting Volatile STT-RAM Caches for Enhanced Performance in CMPs. In *DAC*, 2012.

[23] D. Joseph and D. Grunwald. Prefetching Using Markov Predictors. *IEEE Trans. Comput.*, 1999.

[24] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das. Neither More Nor Less: Optimizing Thread-Level Parallelism for GPGPUs. In *CSE Penn State Tech Report, TR-CSE-2012-006*, 2012.

[25] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 2011.

[26] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. ATLAS: A Scalable and High-performance Scheduling Algorithm for Multiple Memory Controllers. In *HPCA*, 2010.

[27] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *MICRO*, 2010.

[28] D. Kirk and Wen-mei. W. Hwu. Programming Massively Parallel Processors. 2010.

[29] K. Krewell. Amd's Fusion Finally Arrives. *MPR*, 2011.

[30] K. Krewell. Ivy Bridge Improves Graphics. *MPR*, 2011.

[31] K. Krewell. Most Significant Bits. *MPR*, 2011.

[32] K. Krewell. Nvidia Lowers the Heat on Kepler. *MPR*, 2012.

[33] N. B. Lakshminarayana, J. Lee, H. Kim, and J. Shin. DRAM Scheduling Policy for GPGPU Architectures Based on a Potential Function. *Computer Architecture Letters*, 2012.

[34] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt. Prefetch-Aware DRAM Controllers. In *MICRO*, 2008.

[35] C. J. Lee, V. Narasiman, O. Mutlu, and Y. N. Patt. Improving Memory Bank-Level Parallelism in the Presence of Prefetching. In *MICRO*, 2009.

[36] J. Lee, N. Lakshminarayana, H. Kim, and R. Vuduc. Many-thread Aware Prefetching Mechanisms for GPGPU Applications. In *MICRO*, 2010.

[37] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 2008.

[38] T. Moscibroda and O. Mutlu. Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems. In *USENIX SECURITY*, 2007.

[39] A. Munshi. The OpenCL Specification, June 2011.

[40] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda. Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning". In *MICRO*, 2011.

[41] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *ISCA*, 2008.

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

[43] N. Chidambaram Nachiappan, A. K. Mishra, M. Kandemir, A. Sivasubramaniam, O. Mutlu, and C. R. Das. Application-aware Prefetch Prioritization in On-chip Networks. In *PACT*, 2012.

[44] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. Improving GPU Performance Via Large Warps and Two-level Warp Scheduling. In *MICRO*, 2011.

[45] K. J. Nesbit, and J. E. Smith. Data Cache Prefetching Using a Global History Buffer. In *HPCA*, 2004.

[46] NVIDIA. CUDA C Programming Guide, Oct. 2010.

[47] NVIDIA. CUDA C/C++ SDK code samples, 2011.

[48] NVIDIA. Fermi: NVIDIA's Next Generation CUDA Compute Architecture, Nov. 2011.

[49] M. Rhu and M. Erez. CAPRI: Prediction of Compaction-Adequacy for Handling Control-Divergence in GPGPU Architectures. In *ISCA 2012*.

[50] S. Rixner, W. J. Dally, U. J. Kapasi, P. R. Mattson, and J. D. Owens. Memory Access Scheduling. In *ISCA*, 2000.

[51] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-conscious Wavefront Scheduling. In *MICRO*, 2012.

[52] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. In *HPCA*, 2007.

[53] J. A. Stratton et al. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. 2012.

[54] I. J. Sung, J. A. Stratton, and W.-M. W. Hwu. Data Layout Transformation Exploiting Memory-level Parallelism in Structured Grid Many-core Applications. In *PACT*, 2010.

[55] R. Thekkath, and S. J. Eggers. The Effectiveness of Multiple Hardware Contexts. In *ASPLOS*, 1994.

[56] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU Microarchitecture Through Microbenchmarking. In *ISPASS*, 2010.

[57] H. Yoon, J. Meza, R. Ausavarungnirun, R. Harding, and O. Mutlu. Row Buffer Locality Aware Caching Policies for Hybrid Memories. In *ICCD*, 2012.

[58] G. Yuan, A. Bakhoda, and T. Aamodt. Complexity Effective Memory Access Scheduling for Many-core Accelerator Architectures. In *MICRO*, 2009.

[59] W. K. Zuravleff and T. Robinson. Controller for a Synchronous DRAM that Maximizes Throughput by Allowing Memory Requests and Commands to Be Issued Out of Order. *U.S. Patent Number 5,630,096*, 1997.