TR-CSE-2013-014 CSE-PennState Tech Report.
This is a 2-page summary of the original paper – OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance, ASPLOS-2013

Memory-Aware Warp Scheduling

Adwait Jog[†], Onur Kayiran[†], Nachiappan CN[†], Asit Mishra[§] , Mahmut Kandemir[†], Onur Mutlu[*] , Ravi Iyer[§] , Chita Das[†]
Pennsylvania State University[†], Intel Labs[§], Carnegie Mellon University[*]

1. ABSTRACT

A modern Graphics Processing Unit (GPU) is characterized by numerous programmable computational cores and thousands of simultaneously active fine-grained threads. These threads are grouped into *thread blocks*, also known as *cooperative thread arrays (CTAs)*. All the threads within a CTA are scheduled on the same core at the granularity of *warps*. In spite of having numerous resident threads and theoretically high thread-level parallelism (TLP), GPU cores still suffer from high periods of idle times, resulting in under-utilization of hardware resources. These idle times are primarily a result of the inability of the commonly-employed warp scheduling policies in facilitating a GPU core to completely tolerate the long memory fetch latencies, which are primarily attributed to: (1) contention in caches caused by multiple concurrent threads, (2) DRAM contention caused by various concurrent threads from multiple GPU cores, and (3) limited off-chip DRAM bandwidth available in GPUs. The commonly-employed warp scheduling policies, for example, round-robin (RR) scheduler is shown to be ineffective in alleviating all these three sources of long memory fetch latencies [3–5]. Although the recently-proposed two-level warp scheduler [5] performs better than the RR scheduler on all three aspects, it is far from optimal [2–4]. The cache-conscious warp scheduler [6], developed concurrently with our ASPLOS 2013 paper, was shown to outperform both the two-level and RR schedulers, but it addresses contention in only caches, not in DRAM.

In this paper, we propose a comprehensive *c(O)operative thread array a(W)are warp schedu(L)ing policy*, called OWL[1], which is a four-pronged concerted approach for improving overall performance in GPUs. OWL cohesively: (1) improves latency tolerance, (2) alleviates contention in caches and memory, (3) improves DRAM bandwidth utilization via improving memory bank-level parallelism and row-buffer locality, and (4) facilitates effective incorporation of memory-side prefetching techniques. To the best of our knowledge, this is the first unified warp scheduler that cohesively takes cache locality, DRAM row-buffer locality, and DRAM bank-level parallelism into account, and orchestrates with memory-side prefetching techniques.

2. SUMMARY

2.1 Problem I: Cache contention

In GPUs, each core is capable of executing thousands of threads concurrently. In most cases, the data brought by a *large number* of threads executing simultaneously does not fit into the cache. This hampers the opportunity of reusing the data brought by threads, eventually leading to a high number of L1 misses. In fact, this problem is more severe with the commonly-employed baseline round-robin (RR) warp scheduling policy, where the CTAs assigned to a core as well as all the warps inside a CTA are given equal priority, and are executed in a round-robin fashion. As a result, a large group of warps/threads access the L1 cache in a short interval of time, thereby increasing the cache contention. Further, with RR, most of the warps arrive at long latency memory operations roughly at the same time. As a result, the GPU core becomes idle because there may be no warps that are *not* stalling due to a memory operation, which significantly reduces the latency tolerance of GPUs.

Solutions and Key Insights: OWL tackles this problem using two schemes.

`(a) Scheme-1`: OWL groups all the available CTAs on a core into smaller groups and schedules all *groups* in a round-robin fashion. Figure 1(A) illustrates this scheme. It assumes 4 CTAs are launched on a GPU core, and each group has 1 CTA, resulting in 4 groups in total. First, the warps of group 1 (CTA-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 after their data arrives from memory, and if the core is executing warps of CTA 5, it will keep executing the warps of CTA 5, and will continue with CTA 7 after finishing the execution of CTA5. 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. The advantage of this scheme is that it improves latency hiding capability and reduces idle periods as not all the warps reach long latency operations around the same time. On the flip side, the data brought by the warps of CTA 1 earlier, 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.

`(b) Scheme-2`: To address the drawback of `Scheme-1`, second scheme of OWL *always* prioritizes a group of CTAs in a core over the rest of the CTAs until the prioritized group finishes its execution. Unlike `Scheme-1`, where each group of CTAs is executed one after another, this scheme always prioritizes one group of CTAs over the rest whenever a particular group of CTAs is ready for execution. This significantly reduces cache contention, as in a particular time interval, only a small number of CTAs are given higher priority to keep their data in the private caches such that they get the opportunity to reuse it and take advantage of the locality between nearby threads and warps (associated with the same CTA) [1]. Figure 1 (B) shows this pictorially. This scheme starts choosing warps belonging to CTA 1 (group 1) once they become ready, whereas in `Scheme-1` the scheduler keeps on choosing warps from CTA 5 (group 3), and then CTA 7 (group 4). In other words, we *always* prioritize a small group of CTAs and shift the priority to the next group only after the highest priority group completes its execution.

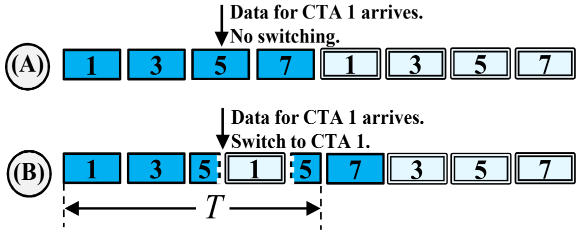

Figure 1: An illustrative example showing how OWL can reduce cache contention. Label in each box refers to the corresponding CTA number.

[1] **Original paper:** OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance, in the Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems **(ASPLOS-2013)**, pp 395-406, Houston, TX, March 2013. An animated presentation summarizing the paper can be downloaded from: `http://www.cse.psu.edu/~axj936/docs/OWL-ASPLOS-Slides.pptx` The full version of ASPLOS paper can be downloaded from: `http://www.cse.psu.edu/~axj936/docs/OWL-ASPLOS-2013.pdf`

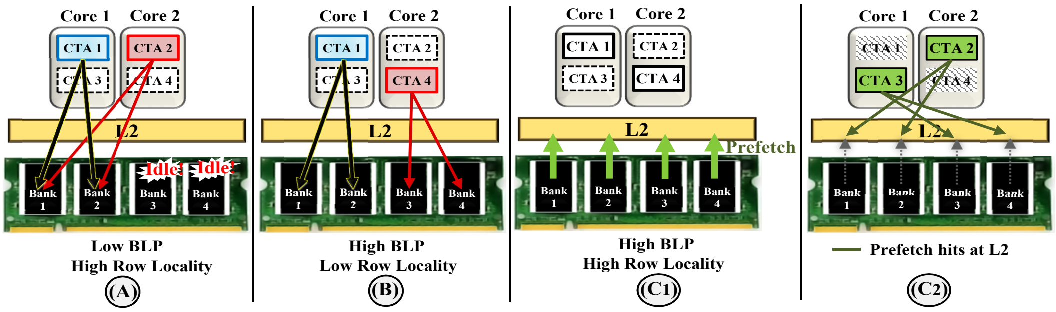

Figure 2: An example illustrating 4 CTAs assigned to 2 cores: (A) the underutilization of banks with `Scheme-2`, (B) improved memory bank-level parallelism with `Scheme-3`, (C1, C2) the positive effects of memory-side prefetching `Scheme-4`.

Figure 1 shows that during time interval T, only 3 CTAs are executing and taking advantage of the private caches, contrary to 4 CTAs in `Scheme-1` (Figure 1 (A)). This implies that a smaller number of CTAs gets the opportunity to use the L1 caches concurrently, thereby reducing the cache contention.

2.2 Problem II: DRAM contention and limited DRAM bandwidth

Massive multi-threading not only creates contention in the private cache of a *single* core, but also manifests itself at the DRAM, due to the memory requests sent by *multiple* cores. One thread's memory requests can cause DRAM bank conflicts, row-buffer conflicts, and data/address bus conflicts with another memory requests. As the number of GPU cores increases, the pressure on the DRAM system and hence, the interference among the threads that share the DRAM increases. We identify that the current warp scheduling techniques are not effective in managing the contention in DRAM systems.

In order to understand the reasons behind the DRAM contention, we studied 38 applications across various benchmark suites. Our studies show that that there is significant DRAM page locality between consecutive CTAs (see Sec. 4.3 in [3]). 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 `Scheme-2`, they would access a *small* set of DRAM banks more frequently. This reduces memory bank-level parallelism (BLP) and increases the queuing time at the banks. Figure 2 (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 rows concurrently, thereby providing high row buffer hit rate. Unfortunately, for the exact same reason, this approach leads to low BLP because not all the DRAM banks are utilized (In Figure 2 (A), two banks stay idle).

Solutions and Key Insights: We develop two schemes to achieve high BLP and high row-buffer hit rate.

`(a) Scheme-3:` For improving BLP, we would like to schedule CTAs so that the CTAs which do not share rows are *always* prioritized in different cores. We attempt to achieve this by *always* prioritizing non-consecutive CTAs in different cores, since consecutive CTAs are likely to access the same rows. Figure 2 (B) depicts the working of this scheme. Instead of prioritizing consecutive CTAs (CTAs 1 and 2) in two cores, `Scheme-3` prioritizes non-consecutive ones (CTAs 1 and 4). This enables all four banks to be utilized concurrently, whereas `Scheme-2` utilizes only two banks (Figure 2 (A)).

`(b) Scheme-4:` A drawback of `Scheme-3` is that it reduces DRAM row-buffer 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-buffer locality, we develop a memory-side prefetching mechanism, in which some of the data from an *already* opened row is brought to the nearest on-chip L2 cache partition. The key idea of memory-side prefetching is to prefetch the so-far-unfetched cache lines from an already open row into the L2 caches, after all the demand requests to the row in the memory request buffer are served. The prefetched lines can be useful for both currently executing CTAs, as well as the CTAs that will be launched later. Figure 2 (C1, C2) illustrates how this scheme works. In Figure 2 (C1), during the execution of CTAs 1 and 4, `Scheme-4` 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 2 (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.

2.3 Evaluation

Hardware Overheads: We synthesized the RTL design of the hardware required for OWL scheduler, and for a 28-core system, the area overhead is 0.18 mm^2 (see Sec. 4.5 of [3]).

Key Results: OWL provides benefits via (a) `Schemes 1 and 2:` selecting and prioritizing a group of CTAs scheduled on a core, thereby improving both L1 cache hit rates and latency tolerance, (b) `Scheme-3:` scheduling CTA groups that likely do not access the same memory banks on different cores, thereby improving DRAM bank parallelism, and (c) `Scheme-4:` employing memory-side prefetching to take advantage of already-open DRAM rows, thereby improving both DRAM row locality and cache hit rates. We evaluate the performance of the OWL scheduling policy, consisting of the four components integrated together, on a 28-core simulated GPU platform with 38 applications. For a set of 19 highly memory-intensive applications, `Schemes 1 and 2` together improve the average L1 cache hit rate by 18% over the baseline RR policy, thereby providing 25% improvement in IPC performance. `Scheme-3` improves average BLP by 11% and reduces row-buffer locality by 14% compared to `Scheme-2`, resulting in an additional 6% improvement in IPC performance. `Scheme-4` restores the row-buffer locality (while preserving BLP), leading to overall IPC performance improvement performance of 33% over the baseline RR scheduling policy. OWL also outperforms the recently-proposed two-level scheduling policy [5] by 19%.

References

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

[2] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. Orchestrated Scheduling and Prefetching for GPGPUs. In *ISCA*, 2013.

[3] A. Jog, O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *ASPLOS*, 2013.

[4] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das. Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *PACT*, 2013.

[5] 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.

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