OSM: Off-Chip Shared Memory for GPUs

Sina Darabi, Ehsan Yousefzadeh-Asl-Miandoab, Negar Akbarzadeh, Hajar Falahati, Pejman Lotfi-Kamran, Mohammad Sadrosadati, and Hamid Sarbazi-Azad

Abstract—Graphics Processing Units (GPUs) employ a shared memory, a software-managed cache for programmers, in each streaming multiprocessor to accelerate data sharing among the threads in a thread block. Although 60% of the shared memory space is underutilized, on average, there are some workloads that demand higher shared memory capacities. Therefore, improving shared memory utilization while satisfying the needs of shared memory intensive workloads is challenging. We make a key observation that the lifetime of each shared memory address is significantly shorter than the execution time of a thread block. In this paper, we first propose Off-Chip Shared Memory (OSM) that allocates shared memory space in the off-chip memory, and accelerates accesses to it via a small on-chip cache. Using an 8 KB cache for shared memory addresses, OSM provides almost the same performance as the baseline GPU that uses 96 KB on-chip shared memory. OSM improves GPU performance in two ways. First, it allocates higher shared memory capacities in the off-chip memory, and improves thread-level parallelism (TLP). Second, it designs a unified cache for shared memory and global address spaces, providing more caching space for global memory address space even for the workloads with high shared memory utilization. Our experimental results show an average 21% and 18% IPC improvement compared to the baseline and the state-of-the-art architectures.

Index Terms—Cache memory, GPUs, lifetime, off-chip memory, shared memory

1 Introduction

GPUs employ a shared memory in each streaming multiprocessor (SM). Shared memory is a software-managed cache (a.k.a. scratch-pad memory) that programmers usually use to accelerate data sharing across threads of a thread block. In addition, using the shared memory rather than the main memory helps to save significant amount of GPU power. It is due to the fact that every access to the main DRAM memory consumes considerable amount of energy [11], [12], [31], [52]. On the contrary, shared memory exacerbates the under-utilization of GPU resources due to two main reasons [1], [10], [20], [37], [59], [61]. First, there are many GPU workloads, e.g. graph applications [70], that under-utilize the shared memory. Our experiments across five benchmark suites indicate that most of the workloads do not fully utilize the allocated shared memory space. According to the Fig. 1, on average only 39.5% of the shared memory space is utilized across 22 workloads. Second, insufficient shared memory capacity can limit the number of allocated thread blocks in each SM, and potentially TLP [44]. This results in under-utilization of different GPU resources, e.g. execution units, register file, and memory bandwidth. Although enlarging shared memory capacity can help to improve TLP, it degrades resource-utilization of workloads that do not take advantage of a larger shared memory.

Several attempts have been accomplished by both industry and academic researches to address the aforementioned issues [10], [20], [37], [59], [61]. NVIDIA Kepler [3] architecture unifies shared memory and L1 data cache memory in each SM. This unified memory has 64 KB capacity, and programmers can select one of the 3 possible shared memory/L1 cache configurations: 16 KB/48 KB, 32 KB/32 KB, and 48 KB/16 KB. This solution is not optimal for the cases in which programmers do not utilize the minimum 16 KB portion of the shared memory. Some of the prior works use virtualization, such as [59], [61], while some others, such as [20], [37], propose new structures, and others exploit time-multiplexing approaches [71]. Shoushtari *et al.* [59] propose a virtualization framework which relocates the shared memory to off-chip memory, and hence, provides an unlimited resource illusion to all threads. Vijaykumar *et al.* [61] propose a virtualization framework that provides unlimited resource (e.g. shared memory, register, and threads) illusion. These works can improve TLP for shared memory intensive applications; however, a significant amount of large on-chip storage space is still wasted for the workloads that under-utilize the shared memory. Gebhart *et al.* [20] unify L1 data cache, shared memory, and register file. This work can increase shared memory capacity and utilization rate at the same time. NVIDIA Volta architecture [1] also provides 128 KB on-chip storage per SM unified across shared memory, L1 data cache, and texture cache,

- *Sina Darabi, Ehsan Yousefzadeh-Asl-Miandoab, and Negar Akbarzadeh are with the Department of Computer Engineering, Sharif University of Technology, Tehran 11155-9517, Iran. E-mail: {sinad1367, ehsanyousefzadehasl, n.akbarzadeh93}@gmail.com.*
- *Hajar Falahati, Pejman Lotfi-Kamran, and Mohammad Sadrosadati are with the School of Computer Science, Institute for Researches in Fundamental Sciences (IPM), Tehran 19538-33511, Iran. E-mail: {hajar.falahati, m.sadr89}@gmail.com, plotfi@ipm.ir.*
- *Hamid Sarbazi-Azad is with the Department of Computer Engineering, Sharif University of Technology, Tehran 11155-9517, Iran, and also with the School of Computer Science, Institute for Researches in Fundamental Sciences (IPM), Tehran 19538-33511, Iran. E-mail: azad@ipm.ir.*

Manuscript received 13 Jan. 2021; revised 8 Jan. 2022; accepted 7 Feb. 2022. Date of publication 24 Feb. 2022; date of current version 15 June 2022.
The work of Pejman Lotfi-Kamran was supported by a Grant from Iran National Science Foundation (INSF).
(Corresponding author: Mohammad Sadrosadati.)
Recommended for acceptance by M. Becchi.
Digital Object Identifier no. 10.1109/TPDS.2022.3154315

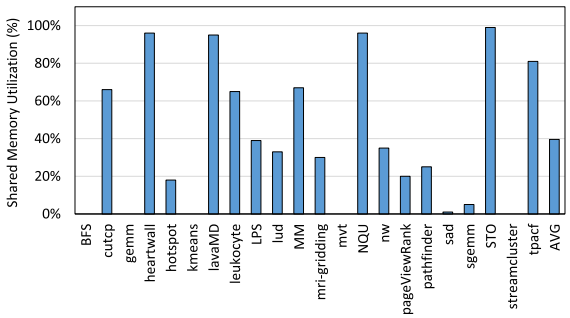


Fig. 1. Shared memory utilization across 22 workloads; 5 workloads out of 22 do not utilize shared memory at all.

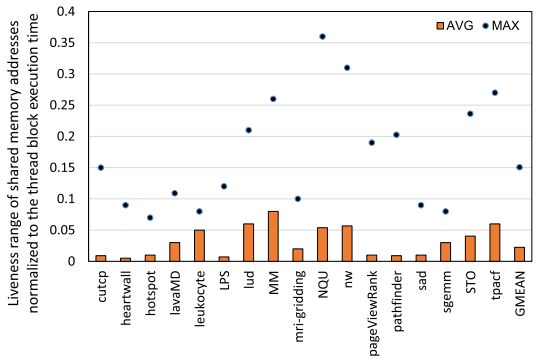


Fig. 3. Average and maximum lifetime range of shared memory addresses for the workloads that utilize shared memory.

that can support several shared memory/L1 cache/texture cache configurations. The allocatable space for shared memory per SM can be in the range of 0 KB to 96 KB. As a result, the whole 128 KB on-chip memory space can be used for L1 data and texture caches in applications that do not employ shared memory.

The main bottleneck of both works reported in [20] and [1] is that the shared memory space is allocated statically for each thread block, and thus, it will not be freed till the end of the thread block execution. This shortcoming can greatly reduce the effectiveness of these solutions if shared memory addresses experience a short lifetime range. Komuravelli *et al.* [37] propose a new on-chip memory structure, *stash*, to address visibility through specialized memory spaces in heterogeneous systems. *Stash* unifies the specialized memory components (data cache and shared memory) in a coherent address space. The unified structure needs both hardware and software supports and treats all memory accesses, either data cache or shared memory, likely. Our evaluation results show that these memory accesses have various lifetime ranges and we need to modify the cache management. However, none of the above-mentioned works have a solution to deal with various lifetime ranges in shared memory accesses.

The lifetime range of a location is the time duration between its first write and last read, as shown in Fig. 2. To compare the lifetime range of shared memory addresses with the execution time of a thread block, we measure lifetime range of shared memory addresses for each workload and normalize it to its average thread block execution time. We use Accel-Sim [34] for this experiment. Fig. 3 reports the average results for 14 workloads. Note that only the workloads that use shared memory during their execution, are evaluated. We make an observation that, on average, the lifetime range of shared memory addresses is about 50× shorter than the thread block execution time, underscoring the fact that we do not need to keep these addresses for the entire execution time of a thread block.

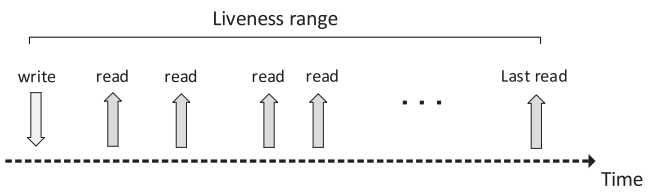


Fig. 2. lifetime range of a shared memory address.

Based on our key observation we propose Off-Chip Shared Memory (OSM) that improves capacity and utilization of shared memory at the same time. OSM allocates shared memory address space to off-chip memory and accelerates shared memory accesses using a small on-chip cache. Note that we do not need a dead shared memory address for the rest of the execution. However, since we cannot be 100% sure that a shared memory address is dead, we propose to keep shared memory addresses in the off-chip memory to be able to service any unpredictable future accesses. The key challenge is that the mechanism can increase bandwidth usage due to frequent evictions and insertions of data items in shared memory cache. Note that the first insertion has no overhead compared to the baseline GPU architecture, as the baseline also consumes memory bandwidth to move data from the global memory to the shared memory. To lessen this overhead, OSM attempts to keep addresses in the shared memory cache for their entire lifetime range, and prevent re-inserting them. Studying access patterns of shared memory addresses reveal that some addresses are accessed only a few number of times during their lifetime range. As a result, such addresses can quickly become the head of the least recently used (LRU) queue, and are evicted before elapsing their lifetime range. To deal with this issue, we devise a lock/unlock mechanism on top of LRU replacement policy. In this approach, only unlocked addresses at the head of LRU queue can be evicted. In the next step, we propose a unified on-chip memory for handling shared and global accesses; this unified on-chip memory provides a larger level-1 cache with higher bandwidth resulting in better performance.

Our experimental results show that with an 8 KB shared memory cache per SM, we can achieve almost the same performance as the baseline NVIDIA Volta GPU that can employ up to 96KB [1] on-chip shared memory per SM. We use OSM to (1) allocate larger shared memory space in the off-chip memory which allows scheduling more thread blocks for shared memory intensive workloads, and (2) design a unified cache memory for both shared memory and global memory address spaces, improving caching space and better utilization for the workloads with low shared memory utilization. Our experimental results show that our proposal improves GPU performance by 21% and 18%, on average, compared to the baseline GPU and the state-of-the-art proposal, respectively.

It should be noted that Our proposal is agnostic to NUMA in multi-chip-module GPUs. Since in such

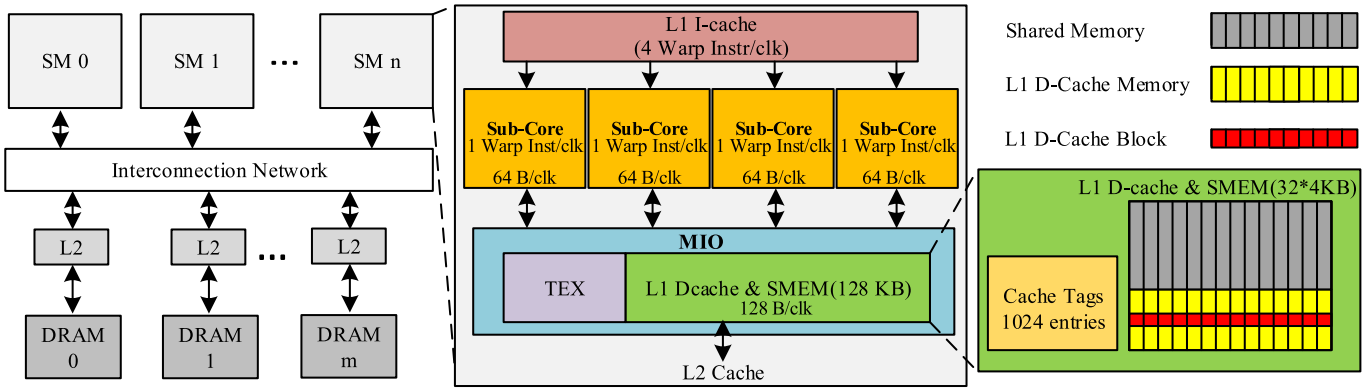


Fig. 4. GPU architecture.

architectures, each GPU has its own device memory, and we make sure to allocate shared memory addresses of each GPU to its memory. As a result, our proposal does not generate remote traffic for reading shared memory addresses.

We make the following contributions:

- We thoroughly study shared memory and the lifetime of its addresses across several workloads. We observe that the lifetime range of shared memory addresses is significantly shorter than thread block execution time.
- We propose OSM that allocates shared memory space in the off-chip memory, and accelerates its accesses using a small on-chip cache. We devise a lock/unlock mechanism on top of LRU replacement policy to prevent addresses from being evicted before their lifetime is elapsed.
- We use OSM to improve GPU's TLP and allocate more caching space to global memory address space.
- Our proposal improves GPU performance by 21% and 18%, on average, compared to the baseline GPU and the state-of-the-art proposal, respectively.

2 Background

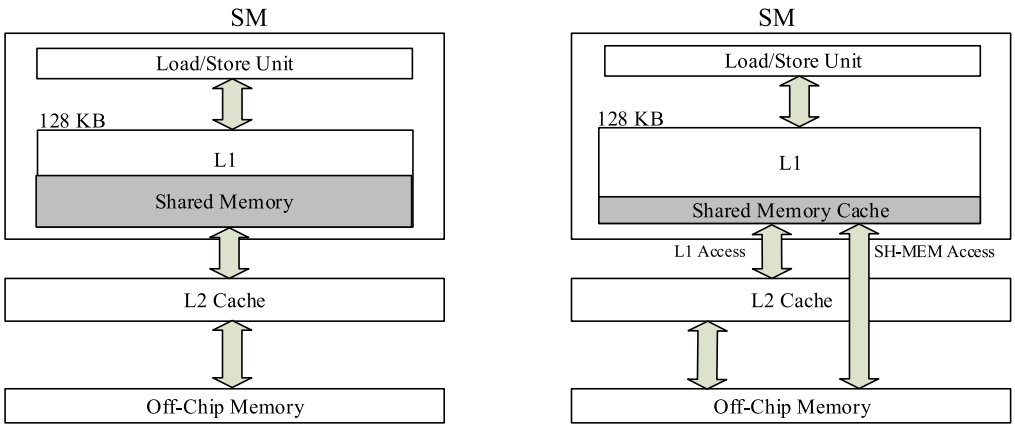
In this section, we present the needed background on the GPU architecture and shared memory.

2.1 GPU Architecture

GPU kernels are composed of many Single-Program Multiple-Data (SPMD) threads grouped by the programmer into several Thread Blocks or Cooperative Thread Arrays (CTAs). Each CTA is assigned to a Streaming Multiprocessor (SM) upon thread launch. SMs are SIMD processing units with dedicated fast memory units. During execution, threads assigned to each SM are divided into multiple fixed length (e.g., 32) groups. Each group of threads is called a warp (NVIDIA terminology) or wavefront (AMD terminology). Threads inside a warp are executed in parallel lock-step manner, where each thread executes the same instruction. SIMD units are time-multiplexed between different warps. In each cycle, the GPU selects one warp to be executed based on the GPU's warp scheduling policy following the SIMT model. In the SIMT model, all threads of a warp execute the same instruction on different data, but threads of a warp may take different control flow paths, leading to idle SIMD lanes (called branch divergence). SMs are responsible for the execution of warps. Each SM is composed of several components, including SIMD integer/floating-point units, load/store units, special function units, L1 data and instruction caches, local shared memory, and a register file that is responsible for maintaining the context of all threads inside the SM. Fig. 4 shows the GPU architecture evaluated in our study, modeled after the NVIDIA Volta GPU architecture. SMs are connected to L2 banks with an on-chip interconnection network.

2.2 GPU Memory System

A thread commonly accesses local, shared, and global data through a rich memory hierarchy shown in Fig. 4. The GPU architecture depicted in Fig. 4 is adapted from NVIDIA Volta architecture [1]. The local data is usually placed in a dense register file for fast context switching. An SM (streaming multiprocessor) is also associated with a software-managed local memory called Shared Memory for shared data accesses by the threads within a block. The shared memory/L1 is composed of 32 four-byte-width blocks. Each block contains 4 KB of data. Since a unified on-chip memory space is used for shared memory and L1-D cache, and programmer specifies the shared memory capacity, the rest of the unified on-chip memory space is dedicated to L1 cache, automatically. The gray and yellow parts of the unified on-chip memory space shown in Fig. 4 is dedicated to shared memory and L1-D cache, respectively. Since the minimum acceptable L1 cache size is 32 KB in Volta architecture, the shared memory capacity can be in the range of 0-96 KB. Therefore, in some cases that we do not need shred memory at all, we can assign the whole 128 KB on-chip cache space to the L1 cache. Since the capacity of an L1 cache block (depicted by red in Fig. 4) is 128B, assuming that the whole 128 KB on-chip cache space is dedicated to L1, we need 1024 entries for the tags. Global data refers to the data shared among all threads within a grid. When a thread requests data from off-chip main memory, the accesses pass through a two-level cache hierarchy. The L1 caches are private to SMs with no coherency among them. The L2 cache is a banked cache array that is shared by all SMs and uses write-back policy with respect to main memory. Each L2 bank communicates with L1 cache of each core through an interconnection network. Knowing the fact that each SM



(a) Volta GPU Memory System (b) OSM Memory System

Fig. 5. Baseline GPU memory system vs. OSM memory system.

consists of 4 sub-cores, the memory input/output unit (MIO) serves, at maximum, 64B requests from each sub-core per clock. If the requests are distributed evenly among the 32 blocks of L1-D cache/shared memory, the maximum bandwidth (128B/clock) can be achieved. Note that due to the 128B/clock bandwidth limit, the four sub-cores cannot serve 64B requests, simultaneously. L1-D cache and shared memory use the same 32-bank memory structure (4 KB capacity per bank) as shown in Fig. 4; however, they have some differences. We can access 32-bit shared memory arrays via a thread-index directly, while for accessing L1-D cache, we should read 128B (four 32B sectors) of the cache block. In addition, L1 cache requires an extra hardware for managing tags and implementing LRU replacement policy.

2.3 Shared Memory

GPU programmers use shared memory to accelerate data sharing and synchronization across threads in a thread block. Shared memory capacity is one of the main TLP limitation factors as it can limit the number of thread blocks allocated to each SM. Shared memory is a multi-banked memory that can serve multiple conflict-free requests in parallel. Thread-index of a thread inside a thread block is used to access its corresponding shared data arrays. However, different thread blocks cannot share their data using the shared memory. In fact, if data locality exists across thread blocks rather than inside a thread block, it is not efficient to store the data in the shared memory. To move a variable from global memory to the shared memory, GPU needs to load it to a register and then store the register value in the shared memory. Therefore, due to the high cost of moving data from global memory to the shared memory, storing data with low reusability in the shared memory would result in performance overhead instead.

Despite all the performance benefits that shared memory provides, it has some programming burden; first, programmers should handle the shared memory accesses with no bank conflicts. Second, programmers should not overestimate the shared memory capacity per thread block to provide enough TLP for the performance. The number of thread blocks that can be assigned to an SM is determined by the number of factors including available register per thread, maximum number of supportable threads, maximum number of supportable CTAs and shared memory capacity. In fact, due to the limited capacity of on-chip shared memory, over-allocating shared memory for thread blocks can limit the total number of thread blocks and accordingly sacrifice the TLP.

3 Off-Chip Shared Memory (OSM)

Fig. 3 shows that the lifetime range of shared memory addresses are too short to keep them on-chip throughout the execution time of a thread block. We propose OSM that allocates shared memory address space in the off-chip memory and accelerates its accesses using an on-chip shared memory cache. Only threads inside one thread block can access the off-chip shared memory space.

Fig. 5 indicates the difference between the baseline GPU and the proposed OSM architecture. In the baseline Volta architecture [1], we need to specify the L1 cache and shared memory space statically before running the program. However, in the proposed architecture, we can dynamically assign the available space to L1 cache or shared memory according to the requirements of the running application. For simplicity, we employ the same configuration as the L1-D cache for implementing our shared memory cache in each SM. Therefore, our shared memory cache has 64 ways. It should be noted that the accesses to the shared memory and L1 cache are still similar to the baseline architecture. The only difference is the metadata required for implementing OSM. In fact, for lock/unlock mechanism, we need to add one bit to each cache block tag.

Since shared memory addresses are shared only across threads in a thread block of an SM, we utilize a write-back policy for the added shared memory cache. Therefore, we only write data in the shared memory cache at the request time. Data will be written back to the shared memory space of off-chip memory on its eviction at some later time. If a request misses in the shared memory cache, we need to access the off-chip DRAM to service it. Note that in OSM, we need to write each *dirty* evicted shared memory address into the off-chip memory as we cannot guarantee that the

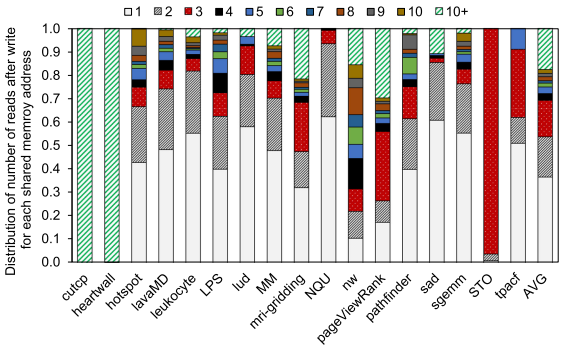


Fig. 6. Distribution of read accesses during their lifetime range for the workloads that utilize the shared memory during the execution.

address will not be accessed in the future. The shared memory accesses which are evicted or missed in the on-chip shared memory, simply bypass the L2 cache and directly go to the off-chip shared memory.

To reduce the overhead of evictions and re-insertions, OSM needs to keep each shared memory address in the cache during its lifetime range. Evicting a shared memory address before end of its lifetime period, or keeping it more than its lifetime period can potentially impose performance and energy overheads. This is due to the 2 main reasons. First, the former can potentially cause misses in the shared memory cache. This can increase the bandwidth usage and the average shared memory access latency, which indeed, imposes performance and energy overheads. Second, the latter also reduces the effectiveness of our technique and induces the need for larger shared memory caches. As a result, it is necessary to track the lifetime range of each address and consider it in the cache replacement policy.

Unfortunately, run-time measurement of lifetime ranges is impractical due to the 2 main reasons. First, we do not exactly know which shared memory read access is the last read request. Second, keeping a counter per shared memory cache block imposes a significant overhead. Therefore, we devise to predict the lifetime ranges through studying the write/read pattern of shared memory addresses. We count the number of read accesses for each shared memory address before its lifetime is elapsed for each workload. Fig. 6 shows the distribution of read accesses during their lifetime ranges for each workload. We observe that for some workloads, such as *LPS*, *NQU*, and *sad*, more than 60% of shared memory addresses have up to two reads during their lifetime range. This low number of accesses can cause quick eviction of a shared memory address, based on the baseline replacement policy (LRU). We propose to lock a shared memory address in the cache when it is written, in order not to be selected for eviction, and unlock it after a certain number of reads. We use a threshold, called *#reads-to-unlock*, to decide after how many reads a shared memory address can be unlocked. We statically set this threshold to one. We analyze the sensitivity of our proposal to this threshold value in Section 6.4. Those shared memory addresses that have frequent reads during their lifetime range, say more than 10 accesses, are likely kept in the cache based on the LRU replacement policy. As a corner case, it is probable that all cache blocks in a cache set are locked, and hence, there is no room for new insertion. In this case, OSM evicts one of these locked cache blocks based on the LRU replacement policy. It should be noted that we do not observe such a case in the tested workloads. We just consider it as a probable case that might happen in other applications.

TABLE 1
Memory Latency

Cache Type	Miss penalty (L2 hit)	Miss penalty (main memory hit)	Hit latency
Shared Memory Cache	—	310 cycle	28 cycle
L1 Data Cache	193 cycle	470 cycle	28 cycle

4 OSM Usecases

In this section, we study how OSM can be employed to improve GPU performance. OSM has two main advantages. First, since OSM moves the shared memory address space to off-chip memory, we would have an extremely larger space in the off-chip memory for shared memory addresses than the limited on-chip SRAM space. This can potentially improve the thread-level parallelism (TLP), and accordingly the performance of the workloads whose number of thread blocks is limited by the insufficient on-chip shared memory space. We analyze the effect of this advantage on the GPU performance in Section 6.1. Moreover, we analyze the access latency of the L1-D cache and shared memory cache. Table 1 reports the access latency of the L1-D cache and shared memory cache when hit/miss happens. As Volta supports GPU memory over-subscription, we need a TLB for the cases where the unified memory feature is enabled. This leads to the worst access latency of the GPU memory hierarchy since it is needed to access the TLB to find the location of the requested address. Therefore, we measure the latency of L1 cache in the worst-case scenario. Since both the L1 cache and shared memory use the same memory structure, they have the same hit latency (28 cycles). But different scenarios happen in the case of miss. For L1-D cache miss, if the data hits in L2, the miss penalty would be about 193 cycles. Otherwise, we should get the data from the main memory after 470 cycles. Fortunately, on a shared memory cache miss, the access will be sent directly to the main memory as we bypass L2. The miss penalty, in this case, would be about 310 cycles. According to the obtained results, we can conclude that the miss penalty of the shared memory cache in OSM is less than the miss penalty of the L1 cache when data also misses in L2.

Second, since we are using a cache memory for shared memory addresses, we can combine the L1 data cache and shared memory, and manage global/local memory and shared memory addresses in different spaces of off-chip memory, as shown in Fig. 7. Note that we do not allocate shared memory space inside the global memory space, rather we allocate shared memory space inside the off-chip memory (a.k.a., device memory). The off-chip memory is used for keeping different memory address spaces, such as global, local, texture, etc. In OSM, we propose to keep shared memory space in the off-chip memory as well. So,

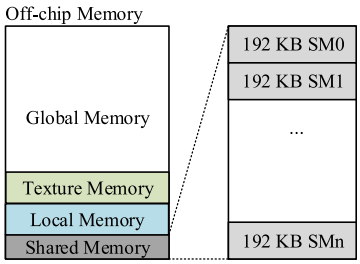


Fig. 7. Unified addressing map to off-chip memory (a.k.a., device memory).

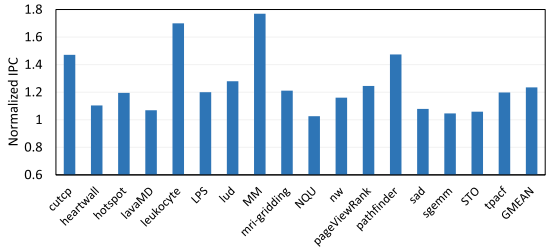


Fig. 9. Performance improvement of an idealized GPU with 96 KB shared memory and 128 KB L1 data cache compared to the baseline Volta architecture for the workloads that utilize the shared memory.

this does not need significant changes in the hardware and the address mapping. Distinguishing across these address spaces is performed based on each address range. We move forward and improve our design by unifying the shared memory cache and L1-D cache (called UCM, or unified cache memory). Despite OSM that only provides 8 KB 1-set on-chip shared memory caching space, with UCM we can have 128 KB 16-set unified on-chip shared memory/L1 cache space. UCM outperforms OSM in case of performance since it satisfies shared memory accesses dynamically using a much smaller on-chip space, while it also provides larger caching space for the global memory address space. Fig. 8 indicates the block diagram of memory access hierarchy in OSM and UCM. Since there is a limited L2 cache space, we bypass the L2 cache for accesses to the off-chip shared memory. Bypassing the L2 cache is already implemented in current GPU architectures and there is no need to any additional hardware changes. It should be noted that the depicted lines between the blocks of the figure are just showing the logical connection of the blocks.

To show the maximum achievable performance improvement for workloads that both utilize shared memory and L1 cache simultaneously, we suppose 96 KB of shared memory (like the case of baseline Volta architecture [1]) and on the other hand, increase the L1 cache capacity from 32 KB to 128 KB. Fig. 9 indicates the performance results of this ideal case. It can be seen that performance improves up to 77% (23.5%, on average) to which we should try to approach utilizing UCM architecture.

5 Methodology

Simulation. We evaluate OSM and UCM using Accel-Sim [34] modeling an NVIDIA Volta-like configuration [2], [25]. Table 2 reports the main simulation parameters. NVIDIA Volta V100 GPU comes with two configurations: 16 GB and 32 GB HBM memory [2]. However, we choose 32 GB for our experiments, so that all evaluated datasets can fit into the HBM memory. Moreover, according to the cache configurations provided in Table 2, GPU caches have relatively large associativity. This is due to the fact that (1) GPUs can tolerate higher memory access latencies and are less latency-sensitive [33], [42]. (2) GPU caches normally suffer from thrashing that can be alleviated by utilizing large associativity degrees [76]. (3) Inter-thread locality can be captured via large associativity degrees [38], [76].We modified Accel-Sim to support the proposed design. We suppose the access latency of the shared memory cache is the same as other on-chip caches in GPU. We treat off-chip shared memory accesses exactly like the off-chip memory accesses in GPUs. We calculate GPU energy consumption by multiplying the execution time extracted from Accel-Sim to GPU power consumption extracted from GPU-Wattch [29]. GPUWattch [29] reports the consumed power for all GPU components (i.e., different storage spaces, processing units, memory controllers, and network-on-chip). Compared to the baseline GPU, the proposed idea affects the power consumption of the components in the memory system and the execution units. The power consumption of the memory system is affected based on two main reasons. First, UCM adds some power consumption due to multiple insertions and evictions of shared memory addresses in the cache. Second, UCM can reduce the power consumption of the memory system due to improving the hit rate of the L1 data cache. Regarding the execution units, UCM usually

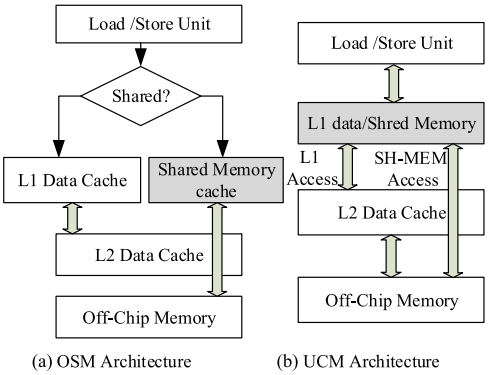


Fig. 8. Unified L1 data cache and shared memory cache.

TABLE 2
Configuration Parameters of Baseline Architecture

Parameter	Value
Number of SM	80
Core clock	1132 MHz
Scheduler	GTO
Number of schedulers per SM	4
Number of warps per SM	64
Register file size	256 KB per SM
Shared memory size	up to 96 KB per SM
L1 data cache	64-way, up to 128 KB, 128B line
L1 instruction cache	16-way, 128 KB, 128B line
LLC	24-way, 6 MB, 128B line
Memory model	32 GB HBM2.0, FR-FCFS, 850 MHZ
HBM timing	t_{CL}=12, t_{RP}=12, t_{RC}=40, t_{RAS}=28, t_{RCD}=12, t_{RRD}=3

TABLE 3
Workloads

Workload	Foot print (per SM)	Category (shared memory used)	Description
BFS [9]	433 KB	Shared memory is not used (0)	This workload performs breadth-first search on a graph
gemm [21]	3145 KB	Shared memory is not used (0)	Matrix-multiply C=alpha.A.B+beta.C
kmeans [13]	6579 KB	Shared memory is not used (0)	A clustering algorithm used extensively in data mining
mvt [21]	2164 KB	Shared memory is not used (0)	Matrix Vector Product and Transpose
streamCluster [13]	236 KB	Shared memory is not used (0)	A solver ro the online clustering problem
cutcp [60]	193 KB	Shared memory is used TLP is not limited (32 KB)	Computes the short-range component of Coulombicpotential at each grid point over a 3D gridcontaining point charges representing anexplicit-water biomolecular model.
hotspot [13]	349 KB	A Shared memory is used TLP is not limited (32 KB)	A thermal simulation tool for estimating processor temperature
leukocyte [13]	184 KB	Shared memory is used TLP is not limited (32 KB)	Medical Imaging
LPS [9]	1137 KB	Shared memory is used TLP is not limited (64 KB)	3D Laplace Solver
lud [13]	261 KB	Shared memory is used TLP is not limited (96 KB)	LU decomposition
MM [23]	274 KB	Shared memory is used TLP is not limited (64 KB)	A dense matrix multiplication using the standard BLAS format.
mri-gridding [60]	1088 KB	Shared memory is used TLP is not limited (32 KB)	Computes a regular grid of data representing an MR scan by weighted interpolation of actual acquired data points. The regular grid can then be converted into an image by an FFT.
nw [13]	194 KB	Shared memory is used TLP is not limited (96 KB)	A global optimization method for DNA sequence alignment
PageViewRank [23]	537 KB	Shared memory is used TLP is not limited (64 KB)	An algorithm to count the number of views of a web page
pathfinder [13]	220 KB	Shared memory is used TLP is not limited (16 KB)	A algorithm for finding path
sad [60]	310 KB	Shared memory is used TLP is not limited (8 KB)	Sum of absolute differences kernel, used in MPEG video encoders
sgemm [60]	3944 KB	Shared memory is used TLP is not limited (8 KB)	Single precision floating General Matrix Multiply
heartwall [13]	93 KB	Shared memory is used TLP is limited (96 KB)	Medical Imaging
lavaMD [13]	2177 KB	Shared memory is used TLP is limited (96 KB)	N-Body - Molecular Dynamics
NQU [9]	40 KB	Shared memory is used TLP is limited (96 KB)	N-Queens Solver
STO [9]	22 KB	Shared memory is used TLP is limited (96 KB)	a library that accelerates hashing-based primitives designed for middleware
tpacf [60]	176 KB	Shared memory is used TLP is limited (96 KB)	Is used to statistically analyze the spatial distribution of observed astronomical bodies. The algorithm computes a distance between all pairs of input, and generates a histogram summary of the observed distances.

increases the power consumption since UCM reduces the stall time by providing higher thread-level parallelism and higher cache hit rate for global memory address space.

Benchmark. We select 22 workloads from five benchmark suites, including CUDA SDK [48], Rodinia [14], Parboil [60], Mars [75], and PolyBench [21]. Table 3 shows the details and characterization of all the workloads used in our evaluation. The memory footprint and shared memory usage of each workload is included in Table 3. For measuring the memory footprint, we count the number of unique memory accesses throughout the execution of a kernel in each SM. We will see in Section 6 that workloads like STO and lavaMD with low and high size of memory footprints, respectively, benefit from the improved TLP; however, other workloads like nw and PageViewRank with a medium sized memory footprint benefit from the increased cache capacity. Moreover, according to the shared memory usage statistics, we can categorize workloads in 3 groups including: (1) shared memory is not used, (2) shared memory is used (TLP is not limited), and (3) shared memory is used (TLP is limited).

To better show the impact of our proposal on GPU performance, we group our workloads into 3 categories: (1) 5 workloads in "*Shared memory is not used*" category in which our proposal cannot be beneficial compared to the baseline NVIDIA Volta architecture. Since in Volta, the entire 128 KB on-chip memory space can be allocated to the L1 cache when the shared memory is not used. (2) 12 workloads in

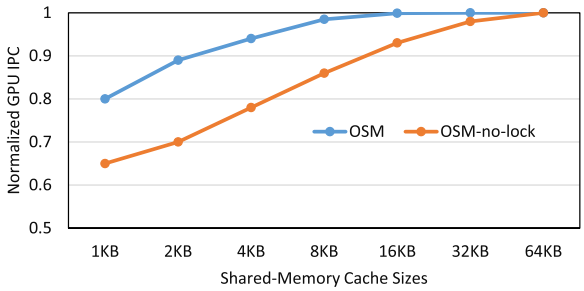


Fig. 10. GPU performance using OSM and OSM-no-lock.

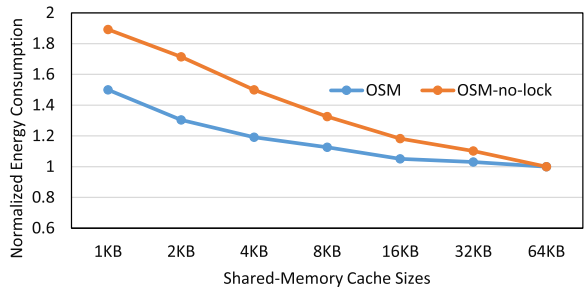


Fig. 11. Energy consumption using OSM and OSM-no-lock.

"*Shared memory is used - TLP is not limited*" category in which we expect some performance improvement compared to the prior work, mainly because our proposal can provide higher caching space for the global memory address space. (3) Five workloads in "*Shared memory is used - TLP is limited*" category in which we expect considerable performance improvement compared to the prior work since we provide not only more caching space for the global memory address space, but also more thread-level parallelism due to expanding the shared memory space in the off-chip memory.

Comparison Points. We first evaluate the OSM proposal by comparing the 3 designs: 1) the baseline GPU architecture, 2) OSM without lock/unlock mechanism (OSM-no-lock), and 3) OSM. We vary the size of shared memory cache to find the optimal point at which the performance remains unchanged. Note that all these designs have 96 KB shared memory space per SM. We then evaluate the effectiveness of the UCM proposal by comparing the 3 designs: (1) Baseline NVIDIA Volta architecture, (2) Unified on-chip memories proposed by Gebhart *et al.* [20], and (3) UCM. Please note that Gebhart *et al.* [20] propose an architecture that unifies L1 data cache, shared memory, and register file in each SM. However, to provide a fair comparison, we compare UCM with a version of Gebhart *et al.* [20] proposal that only unifies shared memory and L1 data cache. We could also unify UCM with the register file using a similar mechanism proposed by Gebhart *et al.* [20] to provide even more performance improvement. However, we do not implement this mechanism in our paper since it does not add a new contribution and it is out-of-scope of our work.

6 Evaluation

6.1 OSM Analysis

To study the effect of OSM on GPU performance, we vary the shared memory cache size from 1 KB to 64 KB and report the overall IPC in each case for OSM and OSM-no-lock designs. We then normalize the results to the BL results, as shown in Fig. 10. We make two key observations. First, using an 8 KB shared memory cache, OSM can provide almost the same performance as the baseline architecture. Second, comparing OSM and OSM-no-lock results clearly shows the importance of using lock/unlock mechanism. Without using lock/unlock mechanism, OSM needs 4× larger shared memory cache to keep the performance almost unchanged.

OSM may impose some energy overhead due to the increase in the number of accesses to off-chip DRAM. We calculate GPU energy consumption for OSM and OSM-no-lock designs using GPU-Wattch [41]. We then normalize the obtained results to those for BL architecture. Fig. 11 reports the results. We make two key observations. First, although 8 KB shared memory cache size is enough to keep the performance unchanged, we need at least 16 KB cache size to mitigate the energy overhead of our proposal. Second, lock/unlock mechanism is essential to keep the energy overhead within a reasonable range.

We conclude that (1) OSM provides almost the same performance as the baseline GPU with a reasonable energy overhead using small cache sizes, and (2) the lock/unlock mechanism is effective in increasing the performance and decreasing the consumed power. In the rest of this section, we elaborate on the opportunities of OSM to improve GPU performance.

OSM Opportunities to Improve GPU Performance. OSM provides two main opportunities. First, OSM can allocate a larger shared memory space in the off-chip memory compared to the baseline GPU architecture that uses a limited on-chip SRAM space. This can potentially improve TLP, and consequently the performance of workloads that suffer from insufficient shared memory space. Second, as we are using a cache memory for shared memory addresses, we can combine the L1 data cache and shared memory, and manage both global/local memory and shared memory addresses in the same cache. OSM can provide an L1 cache with higher capacity/bandwidth not only for workloads that do not use shared memory at all, but also for workloads that highly utilize it (e.g. *heartwall*, *NQU*, *STO*, and *tpacf*). This is due to the fact that OSM can provide almost the same performance as the baseline architecture, using a much smaller on-chip memory space, i.e., 8 KB SRAM memory, that gives the opportunity to allocate the extra 88 KB space for global/local addresses. In this paper, we analyze the effect of the first opportunity of OSM. To this end, we increase the shared memory capacity by 2× (i.e., 192 KB shared memory space per SM in the off-chip memory) as it is the maximum required capacity for our evaluated workloads. Supposing 192 KB shared memory space per SM, we can ensure that the TLP would not be limited due to the insufficient shared memory space for any of the evaluated workloads. However, by moving the shared memory off-chip (provided by OSM), actually there is almost no limitation for the shared memory capacity. Fig. 12 reports GPU performance using larger shared memory sizes. We make an observation that OSM can greatly improve GPU performance (up to 30.4%) for the workloads that their thread-level parallelism is limited by the shared memory size of

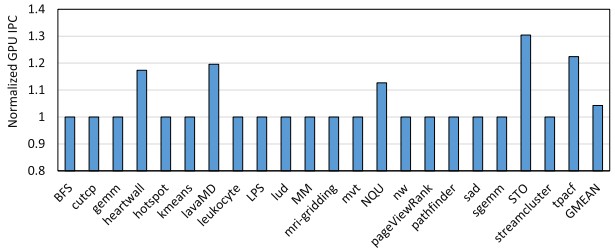


Fig. 12. GPU IPC of 2× larger shared memory using OSM.

the baseline architecture (i.e., 96 KB). 17 workloads in our experiment do not see performance improvement using this mechanism as they either do not use shared memory during the execution (such as BFS, mvt) or the baseline 96 KB shared memory capacity is enough for them (such as cutcp, hotspot).

These results ensure the effectiveness of OSM in (1) increasing logical shared memory space, realized in the off-chip memory, to remove the TLP limitation, (2) caching shared memory space on-chip to hide the long off-chip memory access latency, and (3) devising a lock/unlock mechanism to address lifetime issue.

After proving the potency of OSM, we try to prove the effectiveness of UCM and compare it with state-of-the-art industrial and academic solutions. We present the results of 3 different mechanisms: (1) Baseline NVIDIA Volta architecture, (2) Unified on-chip memories proposed by Gebhart *et al.* [20], and (3) our unified architecture. § 6.2 shows the overall effect of UCM on GPU performance. § 6.3 evaluates the GPU energy consumption.

6.2 Overall Effect on GPU Performance

We compare our architecture with Volta and the architecture proposed in [20]. Fig. 13 shows the performance results of the 3 architectures normalized to the results of the baseline Volta architecture. The figure shows the results of each category separately, and the geometric mean of each category and all the 22 workloads. Across all 22 workloads, we observe that UCM architecture provides 21% and 18% performance improvement over NVIDIA Volta [1] and [20] architectures, on average, respectively. For the workloads that do not use shared memory (Category 1), all three designs provide similar performance results. This is because the entire 128 KB caching space is allocated to L1 data cache for this category. For workloads that use shared memory but their TLP is not limited by the shared memory capacity (Category 2), UCM significantly outperforms the baseline Volta [1] and [20] architectures by about 27%, on average, since UCM satisfies shared memory accesses using a much smaller on-chip space and provides larger caching space for the global memory address space. Note that in this category, both Volta [1] and [20] architectures work almost the same since they allocate shared memory space statically based on the programmers' request. For workloads that use shared memory and their TLP is limited by the shared memory capacity (Category 3), UCM outperforms the Volta [1] and [20] architecture by 31% and 17%, on average, respectively. We provide higher improvement for this category compared to the baseline since we improve the thread-level parallelism and provide a higher caching space for the global memory address space. However, in this category, Gebhart *et al.* [20] proposal outperforms the baseline architecture since it can allocate more shared memory space using the entire 128 KB on-chip memory space, while the baseline Volta architecture [1] allocates up to 96 KB on-chip space to the shared memory.

6.3 Overall Effect on GPU Energy Consumption

We use GPU-Wattch [41] to measure the power-consumption of the proposed architecture and compare it with the NVIDIA Volta baseline [1] and the architecture proposed in [20]. Fig. 14 reports the average energy consumption by the aforementioned mechanisms normalized to the baseline architecture results. The figure reports the geometric mean of energy consumption for each category and all the 22 workloads. We observe that UCM decreases the average energy consumption compared to the NVIDIA Volta and [20] by 15% and 12%, respectively. For the workloads which do not utilize shared memory at all (Category 1), all of the 3 designs provide almost the same results. However, for Categories 2 and 3, our proposal is effective in improving the energy efficiency since it provides better performance and reduces the amount of off-chip transactions (we analyze the off-chip transaction in Section 6.6.).

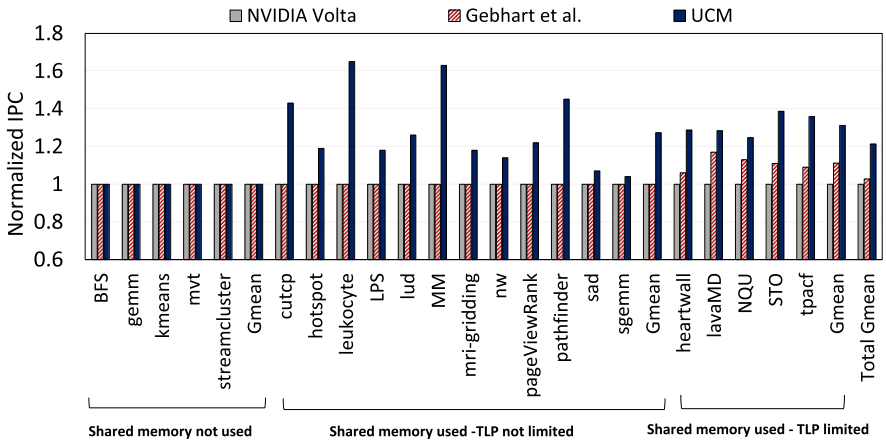


Fig. 13. GPU overall IPC results normalized to the baseline Volta architecture [1].

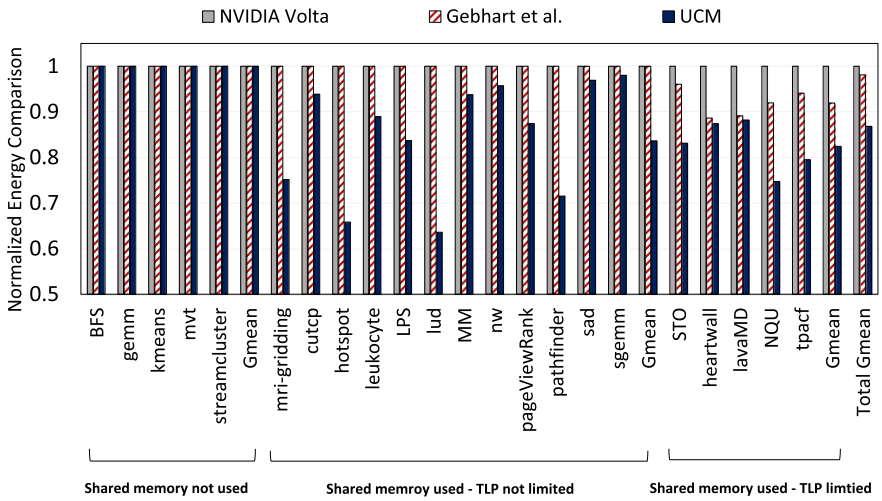


Fig. 14. Energy consumption comparison.

6.4 Analysis of Reads-to-Unlock Threshold

Based on our observations, a considerable number of shared memory addresses are read only once after they are inserted in the shared memory space. However, it may take some time for these addresses to experience their first access, mainly due to the fact that GPU SM concurrently executes several thread blocks together. In order to prevent early eviction, we devise to lock shared memory addresses in the cache once they are inserted, and unlock them only after their first read accesses happen. We do not lock these addresses for future reads, since LRU replacement policy can effectively keep the addresses with higher number of reads in the cache. Note that we devise a lock/unlock mechanism to (1) mitigate the overhead of moving shared memory addresses from the off-chip memory to the on-chip caching space, and (2) avoid keeping shared memory addresses with low usages in the on-chip space. To elaborate more upon our mechanism, we analyse the sensitivity of our proposal to the reads-to-unlock threshold. Fig. 15 indicates the performance result of the proposed method when scaling the reads-to-unlock threshold. We also compare our results with an ideal case which maintains each shared memory address in the cache for its liveness range. We choose workloads that utilize shared memory for this experiment. It can be seen that the larger threshold values are not effective since those addresses with only one read access never reach the threshold value. We also observe that our proposal results are within the margin of 5% of the ideal case's results. In fact, since we do not see significant difference between our results and the results of the ideal case, it is not worthy to utilize a more complex technique that dynamically changes the threshold values of each address.

6.5 Accuracy of Shared Memory Cache Eviction Policy

We calculate the accuracy by measuring the time each shared memory address resides in the shared memory cache before its eviction, and divide it by the actual liveness of the address. A number below "1" shows that the corresponding shared memory address has been evicted earlier than its lifetime, indicating the fact that we probably need to move this address to the shared memory cache again. On the other hand, a number larger than "1" shows that the corresponding address has been evicted after spending its lifetime, showing the fact that it might waste the shared memory cache space for some time. Fig. 16 illustrates the normalized number of cycles that an address resides in the shared memory cache with and without employing the lock/unlock mechanism. It can be seen that without employing the lock/unlock mechanism, addresses of shared memory are evicted too early which can cause performance and energy overheads since they need to be returned to the cache again. This is while with the lock/unlock mechanism, each shared memory address is evicted a bit late (with about 12% overhead) from the unified cache. This is due to the fact that after

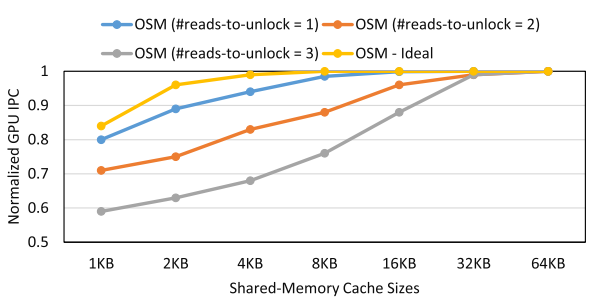


Fig. 15. Impact of read to unlock threshold on performance improvement.

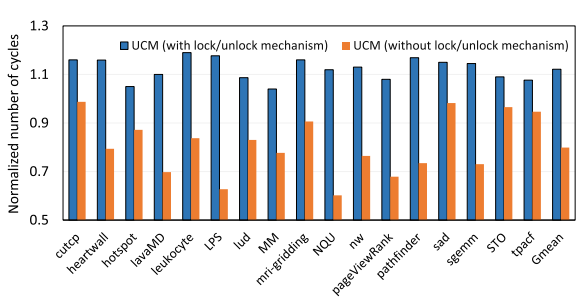


Fig. 16. Effect of lock/unlock mechanism on keeping shared memory addresses in the cache during their lifetime range.

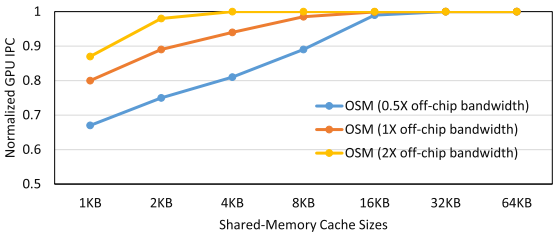


Fig. 17. Off-chip memory bandwidth sensitivity analysis.

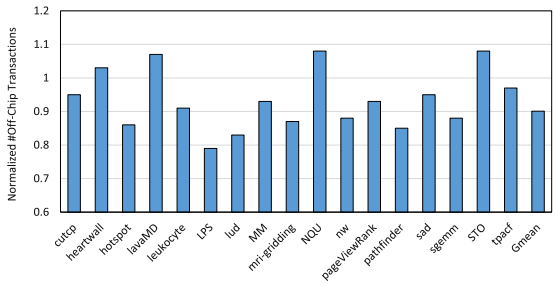


Fig. 18. Off chip transaction of workloads that utilize shared memory.

unlocking an address it cannot be evicted unless the LRU policy chooses it as a victim block.

6.6 Sensitivity on GPU Bandwidth

In this section, we analyze the sensitivity of the proposed architecture to the memory bandwidth. It should be noted that we perform this experiment for workloads that utilize the shared memory. Fig. 17 compares the effectiveness of our proposal for various amounts of memory bandwidth. We make 3 key observation. First, the effectiveness of our proposal is reduced by using 0.5× memory bandwidth, however, it still can satisfy shared memory accesses using a 16 KB shared memory cache. Second, increasing overall memory bandwidth can moderately improve the efficiency of the proposed method. This is mainly due to the fact that the overhead of accessing off-chip shared memory space becomes more negligible when we have higher memory bandwidth. As recent GPUs provide higher memory bandwidth, thanks to the 3D-stacked memories, we believe that our proposal can be more effective in the future. Third, using higher memory bandwidth, we can have similar results as the lower memory bandwidth using smaller cache sizes.

Moreover, to study the overhead of data movement of shared memory addresses, we measure the number of off-chip memory transactions for both global memory and shared memory address spaces. Fig. 18 indicates the obtained results for off-chip memory transactions. We observe that although we move shared memory address space to the off-chip memory, we reduce the number of off-chip memory transactions compared to the baseline GPU architecture. This is mainly due to the fact that our proposal significantly reduces the global memory transactions for workloads that do not fully utilize the shared memory. This is while for some workloads that highly utilize shared memory, our method imposes extra negligible data movements due to the multiple insertions and evictions of shared memory addresses in the cache.

7 Related Work

To the best of our knowledge, this paper is the first that (1) studies the access pattern of the shared memory and aliveness of addresses, and (2) proposes to move shared memory address space to off-chip memory and hide the access latency using a small cache, and then unifying the shared memory and L1 data cache to gain more performance while consuming less energy. We have extensively compared our proposal to other similar attempts employed in industry and proposed in academic studies [20]. In this section, we explain other pieces of related work in improving on-chip memory utilization, cache performance, and thread-level parallelism.

On-Chip Memory Utilization. Jing *et al.* [27] propose cache-emulated register file that utilizes the unused spaces in register file for caching, and improves utilization of the register file. Our proposal is orthogonal to this work as we are focusing on improving utilization of the shared memory. Xie *et al.* [65], [66] attempt to improve utilization of the shared memory by doing register spilling inside the shared memory. This can significantly reduce the performance and energy overheads of register spilling. Our proposal can be employed with these works as we do not eliminate shared memory address space, and the shared addresses are likely kept in the cache during their lifetime range. Multi-kernel execution [7], [16], [22], [26], [58], [63], [64], [69] is another approach to improve utilization of GPU resources, in which, kernels are run with different resource demands together with the goal of utilizing all sorts of GPU resources. Stash [37] proposes a new memory organization that combines the characteristics of caches and scratchpad memories. This work solves the under-utilization problem of on-chip shared memories, but its mechanism requires significant changes to the current microarchitecture of GPGPUs.

Cache Performance. Several works focus on improving L1/L2 cache performance, some try to reduce the pressure on the caches through using throttling [30], [38], [50], [51], [62], [65], bypassing techniques [6], [15], [40], [43], [67], [68], cache latency improvement by prefetching [17], [18], [28], [39], [49], better networking [45], [56] and improving data sharing [32], [46], [47]. Warp throttling is a cache optimization technique that prevents cache contention through decreasing the number of CTAs. In fact, with fewer number of CTAs, we can allocate a larger space to L1 cache since we reduce shared memory demand by limiting the number of CTAs. As our proposed technique enables enlarging the caching space without limiting the TLP, we can include throttling technique orthogonally in the proposed scheme to further improve the performance. Cache bypassing technique is another cache optimization technique that optimizes cache performance by determining the ideal number of threads for cache-sensitive applications. With bypassing techniques, we can decrease cache congestion through bypassing the cache for the warps whose index are beyond a threshold value. Since this technique is also orthogonal to our proposed scheme, we believe combining cache bypassing with our proposal would improve memory system throughput for cache-sensitive applications. Some other techniques attempt to increase effective cache capacity using compression techniques. There are some works that

attempt to improve cache hit rate by increasing its capacity. These works use various dense and low-power emerging memory technologies, such as STT-RAM and Domain Wall Memory, instead of SRAM to increase capacity with low area and power overheads [19], [57], [73]. Our proposal increases the cache capacity by better utilizing the on-chip memories inside each SM. Some works observe that many memory accesses are shared across threads running on the same SM or different SMs. These works use this observation to mitigate the overhead of L1 data cache miss by servicing the miss request through accessing the L1 data cache of another SM currently hosting this memory address.

Thread-Level Parallelism. Several pieces of prior work attempt to improve GPU resource utilization by improving thread-level parallelism (a.k.a. occupancy). Vijaykumar *et al.* [61] proposes Zorua, a framework for virtualizing GPU resources. Zorua can be employed for over-subscribing GPU resources in order to improve thread-level parallelism and prevent from performance cliffs. Several works attempt to improve register file scalability, as the register file capacity is the main limitation factor of thread-level parallelism [4], [5], [24], [35], [36], [53], [54], [55]. Zhao *et al.* [74] consider different types of SMs to increase thread-level parallelism in the way to increase on-chip resource utilization, and hence, improve performance. Bailey *et al.* [8] propose a hierarchical register file which increases thread-level parallelism in the way of utilizing L1 cache and scratchpad memory. Yu *et al.* [72] improve thread-level parallelism by using shared memory as register file. This work keeps some of the registers in the shared memory and employs caching to compensate the gap between the bandwidth of the shared memory and register file. We improve thread-level parallelism for the workloads that call for more shared memory space by moving the shared memory address space to the off-chip memory. [61] propose a virtualization framework that provides unlimited resource (e.g. shared memory, register, and threads) illusion. These works can improve TLP for shared memory intensive applications; however, a significantly large on-chip storage space is still wasted for the workloads that under-utilize shared memory. Yang *et al.* [71] combines two thread-blocks in one and utilizes time-multiplexed shared memory to host more thread-blocks. They propose three software approaches and one hardware approach to support the idea. Although this work increases utilization by combining thread-blocks and time-multiplexing approaches, it still suffers from lifetime mismanagement. Zorua [61] virtualizes resources that limit the TLP with a software/hardware mechanism. Although this work solves under-utilization problem of the shared memory by providing fine-grain allocation of this memory, it makes the coherence design more complex. Also, determining the threshold for over-subscription of resources can lead to uncertainties. Furthermore, this method suggests major changes to the current microarchitecture of GPUs. [71] proposes a software/hardware mechanism to multiplex a SPM among combined multiple CTAs. Although this method suggests higher TLP, it makes the programmer's work more arduous. Also, it cannot solve the under-utilization problem of the shared memory completely, because it allows only discrete amounts of shared memory allocation. Additionally, when there are not available resources, other CTAs stall until resources are available, which results in significant performance degradation.

8 Conclusion

This paper studied the utilization of shared memory and lifetime of shared memory addresses in GPUs. We observed that shared memory is not well-utilized for many GPU applications and the lifetime of shared memory addresses is much shorter than the thread block execution time. We proposed to keep shared memory address space in off-chip memory and accelerate shared memory accesses via a small fast on-chip cache. Our experimental results showed that an 8 KB on-chip shared memory cache provides almost the same performance as the baseline GPU architecture that uses 96 KB dedicated on-chip memory. In addition to improving on-chip resource utilization, our proposal provided two main advantages. First, as off-chip memory space is significantly larger than on-chip memory space, our proposal improves the thread-level parallelism for the workloads for which shared memory was a limitation factor of thread-level parallelism. Second, we proposed merging L1 data cache and shared memory aiming at increasing the on-chip cache memory capacity and bandwidth and generally increasing utilization of on-chip resources. We evaluated our proposal using several workloads from different benchmark suites and showed an average 21% and 18% performance improvement compared to the baseline and state-of-the-art proposal, respectively.

References

[1] "V100 GPU architecture," Accessed: Jan. 09, 2021. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[2] "V100 GPU configuration," Accessed: Jan. 09, 2021. [Online]. Available: https://github.com/accel-sim/gpgpu-sim_distribution/tree/dev/configs/tested-cfgs/SM7_TITANV

[3] "Nvidia kepler GK110 GK210 whitepaper." Accessed: Jun. 22, 2019. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf

[4] H. Aghilinasab, M. Sadrosadati, M. H. Samavatian, and H. Sarbazi-Azad, "Reducing power consumption of GPGPUs through instruction reordering," in *Proc. Int. Symp. Low Power Electron. Des.*, 2016, pp. 356–361.

[5] H. Asghari Esfeden, F. Khorasani, H. Jeon, D. Wong, and N. Abu-Ghazaleh, "Corf: Coalescing operand register file for GPUs," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, New York, NY, USA, 2019, pp. 701–714. [Online]. Available: http://doi.acm.org/10.1145/3297858.3304026

[6] R. Ausavarungnirun *et al.*, "Exploiting inter-warp heterogeneity to improve GPGPU performance," in *Proc. 2015 Int. Conf. Parallel Archit. Compilation*, 2015, pp. 25–38.

[7] R. Ausavarungnirun *et al.*, "Mask: Redesigning the GPU memory hierarchy to support multi-application concurrency," in *Proc. 23rd Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, New York, NY, USA, 2018, pp. 503–518. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173169

[8] J. Bailey, J. Kloosterman, and S. Mahlke, "Scratch that (but cache this): A hybrid register cache/scratchpad for GPUs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2779–2789, Nov. 2018.

[9] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed GPU simulator," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2009, pp. 163–174.

[10] P. Chakraborty *et al.*, "Partitioning and data mapping in reconfigurable cache and scratchpad memory-based architectures," *ACM Trans. Des. Automat. Electron. Syst.*, vol. 22, no. 1, pp. 1–25, 2016.

[11] K. K. Chang *et al.*, "Understanding reduced-voltage operation in modern dram devices: Experimental characterization, analysis, and mechanisms," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 1, no. 1, pp. 1–42, 2017.
[12] N. Chatterjee *et al.*, "Architecting an energy-efficient dram system for GPUs," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2017, pp. 73–84.
[13] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization*, 2009, pp. 44–54.
[14] S. Che *et al.*, "Rodinia: A. benchmark suite for heterogeneous computing," in *Proc. Proc. IEEE Int. Symp. Workload Characterization*, 2009, pp. 44–54.
[15] X. Chen, L. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W. Hwu, "Adaptive cache management for energy-efficient GPU computing," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2014, pp. 343–355.
[16] H. Dai *et al.*, "Poster: Accelerate GPU concurrent kernel execution by mitigating memory pipeline stalls," in *Proc. 26th Int. Conf. Parallel Architectures Compilation Techn.*, 2017, pp. 144–145.
[17] H. Falahati, M. Abdi, A. Baniasadi, and S. Hessabi, "ISP: Using idle sms in hardware-based prefetching," in *Proc. 17th CSI Int. Symp. Comput. Archit. Digit. Syst.*, 2013, pp. 3–8.
[18] H. Falahati, S. Hessabi, M. Abdi, and A. Baniasadi, "Power-efficient prefetching on GPGPUs," *J. Supercomput.*, vol. 71, no. 8, pp. 2808–2829, 2015.
[19] L. Gao *et al.*, "SRAM- and STT-RAM-based hybrid, shared last-level cache for on-chip CPU-GPU heterogeneous architectures," *J. Supercomputing*, vol. 74, no. 7, pp. 3388–3414, Jul. 2018. [Online]. Available: https://doi.org/10.1007/s11227-018-2389-3
[20] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally, "Unifying primary cache, scratch, and register file memories in a throughput processor," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2012, pp. 96–106.
[21] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to GPU codes," in *Proc. Innov. Parallel Comput.*, 2012, pp. 1–10.
[22] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron, "Fine-grained resource sharing for concurrent GPGPU kernels," in *Proc. 4th USENIX Workshop Hot Top. Parallelism*, Berkeley, CA, USA, 2012. [Online]. Available: https://www.usenix.org/conference/hotpar12/fine-grained-resource-sharing-concurrent-gpgpu-kernels
[23] B. He *et al.*, "Mars: A mapreduce framework on graphics processors," in *Proc. 17th Int. Conf. Parallel Archit. Compilation Techn.*, 2008, pp. 260–269.
[24] V. Jatala, J. Anantpur, and A. Karkare, "Improving GPU performance through resource sharing," in *Proc. 25th ACM Int. Symp. High-Perform. Parallel Distrib. Comput.*, New York, NY, USA, 2016, pp. 203–214. [Online]. Available: http://doi.acm.org/10.1145/2907294.2907298
[25] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA volta GPU architecture via microbenchmarking," 2018, *arXiv:1804.06826*.
[26] Q. Jiao, M. Lu, H. P. Huynh, and T. Mitra, "GPGPU energy-efficiency through concurrent kernel execution and DVFs," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2015, pp. 1–11.
[27] N. Jing *et al.*, "Cache-emulated register file: An integrated on-chip memory architecture for high performance GPGPUs," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2016, pp. 1–12.
[28] A. Jog *et al.*, "Orchestrated scheduling and prefetching for GPGPUs," in *Proc. 40th Annu. Int. Symp. Comput. Archit.*, 2013, pp. 332–343.
[29] V. Kandiah *et al.*, "Accelwattch: A power modeling framework for modern GPUs," in *Proc. MICRO-54: 54th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2021, pp. 738–753.
[30] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither more nor less: Optimizing thread-level parallelism for GPGPUs," in *Proc. 22nd Int. Conf. Parallel Archit. Compilation Techn.*, 2013, pp. 157–166.
[31] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the future of parallel computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, Sep./Oct. 2011.
[32] M. M. Keshtegar, H. Falahati, and S. Hessabi, "Cluster-based approach for improving graphics processing unit performance by inter streaming multiprocessors locality," *IET Comput. Digit. Techn.*, vol. 9, no. 5, pp. 275–282, 2015.
[33] M. Khairy, J. Akshay, T. Aamodt, and T. G. Rogers, "Exploring modern GPU memory system design challenges through accurate modeling," 2018, *arXiv:1810.07269*.
[34] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-Sim: An extensible simulation framework for validated GPU modeling," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit.*, 2020, pp. 473–486.
[35] F. Khorasani, H. A. Esfeden, A. Farmahini-Farahani, N. Jayasena, and V. Sarkar, "Regmutex: Inter-warp GPU register time-sharing," in *Proc. 45th Annu. Int. Symp. Comput. Archit.*, Piscataway, NJ, USA, 2018, pp. 816–828. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00073
[36] J. Kloosterman, J. Beaumont, D. A. Jamshidi, J. Bailey, T. Mudge, and S. Mahlke, "Regless: Just-in-time operand staging for GPUs," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchit.*, New York, NY, USA, 2017, pp. 151–164. [Online]. Available: http://doi.acm.org/10.1145/3123939.3123974
[37] R. Komuravelli *et al.*, "Stash: Have your scratchpad and cache it too," in *Proc. ACM/IEEE 42nd Annu. Int. Symp. Comput. Archit.*, 2015, pp. 707–719.
[38] G. Koo, Y. Oh, W. W. Ro, and M. Annavaram, "Access pattern-aware cache management for improving data utilization in GPU," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 307–319.
[39] N. B. Lakshminarayana and H. Kim, "Spare register aware prefetching for graph algorithms on GPUs," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit.*, 2014, pp. 614–625.
[40] S. Lee and C. Wu, "Ctrl-C: Instruction-aware control loop based adaptive cache bypassing for GPUs," in *Proc. IEEE 34th Int. Conf. Comput. Des.*, 2016, pp. 133–140.
[41] J. Leng *et al.*, "GPUWattch: Enabling energy optimizations in GPGPUs," in *Proc. 40th Annu. Int. Symp. Comput. Archit.*, 2013, pp. 487–498.
[42] B. Li, J. Wei, J. Sun, M. Annavaram, and N. S. Kim, "An efficient GPU cache architecture for applications with irregular memory access patterns," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 3, pp. 1–24, 2019.
[43] C. Li, S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou, "Locality-driven dynamic GPU cache bypassing," in *Proc. 29th ACM Int. Conf. Supercomput.*, New York, NY, USA, 2015, pp. 67–77. [Online]. Available: http://doi.acm.org/10.1145/2751205.2751237
[44] C. Li, Y. Yang, H. Dai, S. Yan, F. Mueller, and H. Zhou, "Understanding the tradeoffs between software-managed vs. hardware-managed caches in GPUs," in *IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2014, pp. 231–242.
[45] A. Mirhosseini, M. Sadrosadati, B. Soltani, H. Sarbazi-Azad, and T. F. Wenisch, "Binochs: Bimodal network-on-chip for CPU-GPU heterogeneous systems," in *Proc. 11th IEEE/ACM Int. Symp. Netw.-Chip*, 2017, pp. 1–8.
[46] N. Nematollahi *et al.*, "Efficient nearest-neighbor data sharing in GPUs," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 1, pp. 1–26, 2020.
[47] N. Nematollahi, M. Sadrosadati, H. Falahati, M. Barkhordar, and H. Sarbazi-Azad, "Neda: Supporting direct inter-core neighbor data exchange in GPUs," *IEEE Comput. Archit. Lett.*, vol. 17, no. 2, pp. 225–229, Jul.–Dec. 2018.
[48] NVIDIA, P. Vingelmann, and F. H. Fitzek, "Cuda, release: 10.2.89," 2020. [Online]. Available: https://developer.nvidia.com/cuda-toolkit
[49] Y. Oh *et al.*, "Apres: Improving cache efficiency by exploiting load characteristics on GPUs," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit.*, 2016, pp. 191–203.
[50] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-aware warp scheduling," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2013, pp. 99–110.
[51] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchit.*, Washington, DC, USA, 2012, pp. 72–83. [Online]. Available: https://doi.org/10.1109/MICRO.2012.16
[52] N. Rohbani, S. Darabi, and H. Sarbazi-Azad, "PF-DRAM: A precharge-free DRAM structure," in *Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Archit.*, 2021, pp. 126–138.
[53] M. Sadrosadati *et al.*, "ITAP: Idle-time-aware power management for GPU execution units," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 1, pp. 1–26, 2019.

[54] M. Sadrosadati *et al.*, "LTRF: Enabling high-capacity register files for GPUs via hardware/software cooperative register prefetching," in *Proc. 23th Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, 2018, pp. 489–502.
[55] M. Sadrosadati *et al.*, "Highly concurrent latency-tolerant register files for GPUs," *ACM Trans. Comput. Syst.*, vol. 37, no. 1–4, pp. 1–36, 2021.
[56] M. Sadrosadati, A. Mirhosseini, S. Roozkhosh, H. Bakhishi, and H. Sarbazi-Azad, "Effective cache bank placement for GPUs," in *Proc. Des., Automat. Test Eur. Conf. Exhib.*, 2017, pp. 31–36.
[57] M. H. Samavatian, H. Abbasitabar, M. Arjomand, and H. Sarbazi-Azad, "An efficient STT-RAM last level cache architecture for GPUs," in *Proc. 51st ACM/EDAC/IEEE Des. Automat. Conf.*, 2014, pp. 1–6.
[58] L. Shieh, K. Chen, H. Fu, P. Wang, and C. Yang, "Enabling fast preemption via dual-kernel support on GPUs," in *Proc. 22nd Asia South Pacific Des. Automat. Conf.*, 2017, pp. 121–126.
[59] M. Shoushtari *et al.*, "Shave-ice: Sharing distributed virtualized SPMs in many-core embedded systems," *ACM Trans. Embedded Comput. Syst.*, vol. 17, no. 2, pp. 1–25, 2018.
[60] J. A. Stratton *et al.*, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," Univ. Illinois Urbana-Champaign, Champaign, IL, USA, Tech. Rep. IMPACT-12-01, Mar. 2012.
[61] N. Vijaykumar *et al.*, "Zorua: A holistic approach to resource virtualization in GPUs," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2016, pp. 1–14.
[62] H. Wang, F. Luo, M. Ibrahim, O. Kayiran, and A. Jog, "Efficient and fair multi-programming in GPUs via effective bandwidth management," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2018, pp. 247–258.
[63] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Simultaneous multikernel: Fine-grained sharing of GPUs," *IEEE Comput. Archit. Lett.*, vol. 15, no. 2, pp. 113–116, Jul. 2016.
[64] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Quality of service support for fine-grained sharing on GPUs," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 269–281, Jun. 2017. [Online]. Available: http://doi.acm.org/10.1145/3140659.3080203
[65] X. Xie *et al.*, "Enabling coordinated register allocation and thread-level parallelism optimization for GPUs," in *Proc. 48th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2015, pp. 395–406.
[66] X. Xie *et al.*, "Crat: Enabling coordinated register allocation and thread-level parallelism optimization for GPUs," *IEEE Trans. Comput.*, vol. 67, no. 6, pp. 890–897, Jun. 2018.
[67] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang, "Coordinated static and dynamic cache bypassing for GPUs," in *Proc. IEEE 21st Int. Symp. High Perform. Comput. Archit.*, 2015, pp. 76–88.
[68] X. Xie, Y. Liang, G. Sun, and D. Chen, "An efficient compiler framework for cache bypassing on GPUs," in *Proc. Int. Conf. Comput.-Aided Des.*, Piscataway, NJ, USA, 2013, pp. 516–523. [Online]. Available: http://dl.acm.org/citation.cfm?id=2561828.2561929
[69] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram, "Warped-slicer: Efficient intra-sm slicing through dynamic resource partitioning for GPU multiprogramming," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit.*, 2016, pp. 230–242.
[70] Q. Xu, H. Jeon, and M. Annavaram, "Graph processing on GPUs: Where are the bottlenecks?," in *Proc. IEEE Int. Symp. Workload Characterization*, 2014, pp. 140–149.
[71] Y. Yang, P. Xiang, M. Mantor, N. Rubin, and H. Zhou, "Shared memory multiplexing: A novel way to improve GPGPU throughput," in *Proc. 21st Int. Conf. Parallel Archit. Compilation Techn.*, 2012, pp. 283–292.
[72] C. Yu, Y. Bai, Q. Sun, and H. Yang, "Improving thread-level parallelism in GPUs through expanding register file to scratchpad memory," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 4, pp. 48:1–48:24, Nov. 2018. [Online]. Available: http://doi.acm.org/10.1145/3280849
[73] J. Zhan, O. Kayiran, G. H. Loh, C. R. Das, and Y. Xie, "Oscar: Orchestrating STT-RAM cache traffic for heterogeneous cpu-GPU architectures," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2016, pp. 1–13.
[74] X. Zhao, Z. Wang, and L. Eeckhout, "Heterocore GPU to exploit TLP-resource diversity," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 1, pp. 93–106, Jan. 2019.
[75] L. Zheng *et al.*, "Mars: A video benchmark for large-scale person re-identification," in *Proc. Eur. Conf. Comput. Vis.*, 2016, vol. 9910, pp. 868–884.
[76] X. Zhu, R. Wernsman, and J. Zambreno, "Improving first level cache efficiency for GPUs using dynamic line protection," in *Proc. 47th Int. Conf. Parallel Process.*, 2018, pp. 1–10.

Sina Darabi received the BSc and MSc degrees in computer engineering from the Isfahan University of Technology, Isfahan, Iran, in 2012 and 2015, respectively. Since 2015, he has been working toward the PhD degree in computer engineering with the Sharif University of Technology. He is currently a researcher with the HPCAN Lab, Department of Computer Engineering. His research interests include high-performance and energy-efficient memory system design for GPUs.

Ehsan Yousefzadeh-Asl-Miandoab received the BSc degree in computer engineering from Tabriz University in 2016, and the MSc degree in computer engineering from the Sharif University of Technology in 2018. His reseach interests include computer architecture, particularly parallel computing systems and energy-efficient designs and their application in embedded and heterogeneous systems and hardware accelerators, especially for ML and NN applications.

Negar Akbarzadeh received the BSc and MSc degrees in electrical engineering from Shahid-Beheshti University in 2014 and 2016, respectively. Since 2017, she has been working toward the PhD degree in computer engineering with the Sharif University of Technology. She is currently a researcher with the HPCAN Lab, Department of Computer Engineering. Her research interests include computer architecture, memory systems, NoCs, and GPUs.

Hajar Falahati received the BSc degree in computer engineering from the Isfahan University of Technology in 2010, and the MSc and PhD degrees from the Sharif University of Technology in 2014 and 2016, respectively. From April 2015 to April 2016, she spent one year as a visiting researcher with the University of Southern California. She is currently a senior postdoctoral researcher with the Institute for Research in Fundamental Sciences (IPM). Her research interests include computer architecture, hardware accelerators, energy efficiency of GPUs, and machine learning.

Pejman Lotfi-Kamran received the bachelor's and master's degree in electrical and computer engineering from the University of Tehran in 2002 and 2005, respectively, and the PhD degree in computer science from EPFL in 2013. He is currently an associate professor of computer science, the head of the School of Computer Science, and the director of the Turin Cloud Services, Institute for Research in Fundamental Sciences (IPM). His research interests include computer architecture, computer systems, approximate computing, and cloud computing. He was the recipient of the Young Faculty Award from Iran's National Elites Foundation in 2016.

Mohammad Sadrosadati received the PhD degree in computer engineering from the Sharif University of Technology in 2019, under the supervision of Prof. H. Sarbazi-Azad. From April 2017 to April 2018, he spent one year as an academic guest with ETH Zurich, hosted by Prof. O. Mutlu during his PhD program. He is currently a senior researcher with SAFARI Research Group, ETH Zurich. His research interests include heterogeneous computing, processing-in-memory, memory systems, and interconnection networks. Due to his achievements and impact on improving the energy efficiency of GPUs, he was the recipeint of Khwarizmi Youth Award, one of the most prestigious awards, as the first laureate in 2020, to honor and embolden him to keep taking even bigger steps in his research career.

Hamid Sarbazi-Azad received the BSc degree in electrical and computer engineering from Shahid-Beheshti University in 1992, the MSc degree in computer engineering from the Sharif University of Technology in 1994, and the PhD degree in computing science from the University of Glasgow in 2002. He is currently a professor of computer engineering with the Sharif University of Technology and a researcher with the School of Computer Science, Institute for Research in Fundamental Sciences (IPM). His research interests include high-performance computer architecture, memory system, NoCs and SoCs, storage systems, parallel and distributed systems, performance modeling/evaluation, and social networks. He was the recipient of Khwarizmi International Award in 2006, TWAS Young Scientist Award in engineering sciences in 2007, and Sharif University Distinguished Researcher Award in 2004, 2007, 2008, 2010 and 2013. He is an associate editor for the *ACM Computing Surveys*, *IEEE Computer Architecture Letters*, *Elsevier's Computers and Electrical Engineering*, and *CSI Journal on Computer Science and Engineering*.