ASHA: An adaptive shared-memory sharing architecture for multi-programmed GPUs

Hamed Abbasitabar¹,*, Mohammad Hossein Samavatian¹, Hamid Sarbazi-Azad¹

HPCAN Laboratory, Computer Engineering Department, Sharif University of Technology, Tehran, Iran

A R T I C L E   I N F O

Article history:
Received 14 December 2015
Revised 20 June 2016
Accepted 12 July 2016
Available online xxx

Keywords:
GP-GPU
Multi-programming
Resource sharing
Shared-memory

A B S T R A C T

Spatial multi-programming is one of the most efficient multi-programming methods on Graphics Processing Units (GPUs). This multi-programming scheme generates variety in resource requirements of stream multiprocessors (SMs) and creates opportunities for sharing unused portions of each SM resource with other SMs. Although this approach drastically improves GPU performance, in some cases it leads to performance degradation due to the shortage of allocated resource to each program. Considering shared-memory as one of the main bottlenecks of thread-level parallelism (TLP), in this paper, we propose an adaptive shared-memory sharing architecture, called ASHA. ASHA enhances spatial multi-programming performance and increases utilization of GPU resources. Experimental results demonstrate that ASHA improves speedup of a multi-programmed GPU by 17%–21%, on average, for 2- to 8-program execution scenarios, respectively.

© 2016 Published by Elsevier B.V.

1. Introduction

General purpose computing has increasingly deployed GPUs. Programming languages like NVIDIA CUDA [1] and OpenCL [2] facilitate developing high throughput general purpose applications on GPUs. Moreover, general-purpose GPUs (GPGPUs) have been growingly exploited in supercomputing applications in high performance and cloud computing centers [3–6]. Compared to other computational resources, the number of GPUs is bounded in these centers. Limited number of GPUs highlights the importance of resource management in GPUs.

Many GPU applications cannot fully utilize GPU resources. In order to resolve this shortcoming, different multi-tasking approaches have been introduced for simultaneously running multiple programs on one GPU [7–12]. Threads of one program have similar sequence of instructions leading to similar resource requirements. But when running multiple programs on a GPU, the variety of resource requirements over time can lead to an opportunity to yield unused resources of some SMs by others that need more resources. This capability would be exploited for resource sharing among SMs. In other words, SMs running a program which does not exploit their resources completely, can lend their remaining resources to other SMs that are in shortage of these types of resources. Thus, multi-programming with resource sharing on GPUs would benefit from higher throughput via resource utilization improvement and TLP enhancement.

GPUs have various resources including memories and processing elements within SMs, and multiple thread blocks (TBs) that can be run on these resources. The number of TBs that can be simultaneously run on each SM is restricted by available SM resources such as shared-memory. Furthermore, GPU design trends have shown a move toward having less number of SMs and more processing elements in recent years. This trend keeps the accessible amount of shared-memory on each SM unchanged besides the increased processing ability, which makes shared-memory one of the main TLP bottlenecks.

Based on our observations, two major issues are raised by spatial multi-programming. First, some combinations of programs lead to performance degradation on GPU. Second, unlike sequential execution, where all SMs are assigned to a single task at a time, spatial multi-programming divides up the SMs among multiple tasks at once. Since increasing the number of TBs per SM mostly results in higher throughput, managing SMs resources in order to run more TBs per SM would be beneficial.

In this paper, we design an Adaptive Shared-Memory Sharing Architecture (ASHA) which facilitates running multiple programs simultaneously by sharing the shared-memory among the SMs of a GPU. ASHA enhances the spatial multi-programming

* Corresponding author.
E-mail addresses: abbasitabar@ce.sharif.ir (H. Abbasitabar), samavatian@ce.sharif.ir (M.H. Samavatian), azad@sharif.ir, azad@ipm.ir (H. Sarbazi-Azad).

¹ No 434, Azadi Ave., Tehran, Iran

http://dx.doi.org/10.1016/j.micpro.2016.07.012
0141-9331/© 2016 Published by Elsevier B.V.

Please cite this article as: H. Abbasitabar et al., ASHA: An adaptive shared-memory sharing architecture for multi-programmed GPUs, Microprocessors and Microsystems (2016), http://dx.doi.org/10.1016/j.micpro.2016.07.012
Table 1
List of acronyms and shortened terms.

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A/P</td>
<td>Absence/presence register</td>
</tr>
<tr>
<td>ASHA</td>
<td>Adaptive shared-memory sharing architecture</td>
</tr>
<tr>
<td>B/S-M-Id</td>
<td>Borrower stream multiprocessor identifier</td>
</tr>
<tr>
<td>B/T-B-Id</td>
<td>Borrower thread block identifier</td>
</tr>
<tr>
<td>DeMUX</td>
<td>Demultiplexer</td>
</tr>
<tr>
<td>GPGPU</td>
<td>General purpose graphics processor unit</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics processor unit</td>
</tr>
<tr>
<td>Hspeedup</td>
<td>Harmonic mean of speedups</td>
</tr>
<tr>
<td>L/S-M-Id</td>
<td>Lender stream multiprocessor identifier</td>
</tr>
<tr>
<td>L/T-B-Id</td>
<td>Lender thread block identifier</td>
</tr>
<tr>
<td>MUX</td>
<td>Multiplexer</td>
</tr>
<tr>
<td>RTR</td>
<td>Remaining TB ratio</td>
</tr>
<tr>
<td>SM</td>
<td>Stream multiprocessor</td>
</tr>
<tr>
<td>SP</td>
<td>Stream processor</td>
</tr>
<tr>
<td>TB</td>
<td>Thread block</td>
</tr>
<tr>
<td>TLP</td>
<td>Thread level parallelism</td>
</tr>
<tr>
<td>Wspeedup</td>
<td>Weighted speedup</td>
</tr>
</tbody>
</table>

performance and utilizes utilization of GPU resources. It also reduces the performance degradations caused by spatial multi-programming for those combinations of programs that experience performance degradation without resource sharing. Experimental results demonstrate that ASHA improves speedup of a multi-programmed GPU by 17%, 19% and 21%, on average, for 2-, 4- and 8-program execution scenarios, respectively.

The rest of the paper is organized as follows. Section 2 describes a short background of GPGPU programs running scheme on hardware. Section 3 discusses previous related works. The proposed multi-programming method, from motivation to architectural supports, is presented in Section 4. The evaluation platform including benchmarks, metrics and experimental methodology is described in Section 5. Section 6 discusses performance results. Finally, Section 7 concludes the work and highlights future works. To help readers of the paper, a short list of acronyms and shortened terms used in the paper are presented in Table 1.

2. Background

State-of-the-art GPUs consist of many simple processors, named stream processors (SPs) in NVIDIA CUDA terminology, to do high throughput computations. A specified number of stream processors stand in groups named stream multiprocessor (SM). SMs own various kinds of memories, including shared-memory, register file, instruction and texture caches. GPU architectures have a private L1 data cache for each SM and a shared L2 cache. Fig. 1 shows the architecture of a GPU.

GPU programs have a special hierarchy for appropriate execution on GPU hardware. A special function, which is called kernel, describes thread functionality in the body of a program. Based on the functionality of threads and communication requirements between them, they will be grouped in thread blocks (TBs). TBs form warps in order to execute on SPs of each SM. Warps include a number of threads of a TB, specified by the GPU hardware, which are executed at the same time. In contrast to TBs and kernels, warps are entirely managed by the GPU hardware.

It is possible to run multiple TB concurrently on each SM through TLP. Each kernel, depending on its resource requirements, can execute a maximum number of TBs on an SM. The degree of TLP for a kernel is defined as the maximum number of concurrent TBs per SM. Limited amount of SM resources cause the kernels having different degrees of TLP and this in turn prevents GPU resources from being fully utilized. The maximum possible number of TBs per SM is determined by resource requirements of TBs. Therefore, limited resources in SMs can be TLP bottleneck. Bottlenecks may be classified as: (1) shared-memory, (2) register file, (3) number of threads, and (4) maximum TBs per SM.

Among SM resources which may be a TLP bottleneck, shared-memory and register file units are storages that are allotted to each TB at its issuing time and the allocated size is released only when the TB finishes running on the SM. Shared-memory is a space used for communications between threads of a TB or a programmer controlled cache for preventing accesses to long latency memories (at different levels of memory hierarchy).

3. Related work

Adriaens et al. [7] introduced a spatial multi-tasking method for GPUs that partitions SMs between concurrent applications to utilize more resources. Authors in [8] proposed a hardware-software approach for efficient spatial-temporal multitasking and presented the advantages over spatial multitasking on GPUs. Gregg et al. [13] introduced a KernelMerge scheduler for concurrent running of kernels, which merges the kernels in a smart manner to better utilize the resources of GPU. However, it does not provide any mechanism to benefit from unused portion of resources of an SM.

Shared-memory in GPUs has been previously researched on several aspects with their main focus being on gaining performance by more coalesced memory accesses [14,15] and using shared-memory as a software managed cache [16]. In [17], software and hardware approaches were introduced for throughput enhancement of applications for which shared-memory is a TLP bottleneck. Gebhart et al. [18] also introduced a unified memory structure which can dynamically change the partitioning among registers, cache, and shared-memory on a per application basis. As the memories are unified into each SM, this mechanism cannot benefit from different demands of shared-memory among SMs when multi-programming is used.
4. The proposed architecture

4.1. Motivation

Each kernel of GPGPU programs creates TBs. TBs that are dispatched onto SMs take their required resources and then will be executed. Since TBs of each kernel encompass the same resource needs, execution of one kernel on GPU causes SMs having almost similar resource utilization level. Therefore, SMs usually cannot take advantage of resource sharing. In contrast to sequential running, the multi-programming method makes SMs having different resource requirements. Therefore, this capability may be exploited for resource sharing among SMs.

Shared-memory management can be categorized to software and hardware level. Software level management, which usually is done by programmer or compiler, as well as hardware level could gain speedup through better usage of this resource. Subsequently, it prevents slower memory accesses and achieves acceptable degrees of TLP. On the other hand, the higher amount of shared-memory usage of a TB may cause a lower TLP degree. Hence, there is a trade-off between memory access latency and TLP degree. In other words, the more amount of shared-memory used by each TB, the higher the probability of shared-memory to be TLP bottleneck; however, it can catch speedups obtained by lower access latency of shared-memory. By sharing the shared-memory, we can create an opportunity of enhancement in shared-memory usage per TB while maintaining the level of TLP degree.

Furthermore, Table 2 expresses GPU design trends toward higher capacity of shared-memory per SM. This growth is brought up by higher number of processing elements in each SM including SPs and floating point units. Besides, Kepler and Fermi architectures use a combined shared-memory and L1 cache. In contrast, the memory architecture of Maxwell is changed and SMs feature a 96 K dedicated shared-memory and posses a combined L1 and texture cache memory which share their memory space with each other. Based on reported results by NVIDIA [19], performance and performance per watts are improved greatly encouraging them to put about twice SMs in Maxwell. Wrapping up, managing the shared-memory in right way can enhance GPUs performance drastically.

Additionally, when shared-memory size (e.g. 16 KB) is not divisible to the required shared-memory space of each TB (e.g. 6 KB), a portion of shared-memory (e.g. 4 KB) is wasted during kernel execution. Meanwhile, when TLP is limited by a resource (e.g. shared-memory), other resources may be under-utilized.

4.2. Architecture

Fig. 2 shows an example of running two programs with shared-memory sharing capability. It is assumed that the GPU has 2-SM clusters and both of them run on all clusters. Each SM has 16 KB shared-memory and all TBs in all kernels occupy 4 KB space. We show the nth kernel of program m as (m, n). At the starting moment, (P1, 1) has a TLP degree of 4 limited by shared-memory, and kernel (P2, 1) has a TLP degree of 3 which is limited by some TLP bottlenecks (other than shared-memory). The SMs running (P1, 1), borrow 4 KB shared-memory space from the SMs of (P2, 1). As a result, (P1, 1) reaches to the TLP degree of 5, limited by TLP bottlenecks which are not shared-memory (\(\text{TLP} 1\)). Thus, (P1, 1) runs faster and (P2, 1) continues running the same as the baseline multi-programming. After finishing (P1, 1), the lent 4 KB space will be released and given back to the SMs of (P2, 1) (\(\text{TLP} 2\)).

Kernel (P1, 2) is triggered immediately using 8 KB of shared-memory space. After a while, when (P2, 1) is finished, (P2, 2) starts and borrows 8 KB of shared-memory space to reach the TLP degree of 6 (\(\text{TLP} 3\)). Before (P2, 2) finishes, (P1, 3) arrives and requests for

<table>
<thead>
<tr>
<th>Table 2: Specification of three generations of GPUs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year Launched</td>
</tr>
<tr>
<td>---------------</td>
</tr>
<tr>
<td># of SMs</td>
</tr>
<tr>
<td># of SMs</td>
</tr>
<tr>
<td># of Register Files</td>
</tr>
<tr>
<td>Max. shared-memory size (KB)</td>
</tr>
</tbody>
</table>

Fig. 2. An example of running two programs with shared-memory sharing capability.
32 KB shared-memory in order to obtain a TLP degree of 8 (maximum possible value). But at this moment, there is only 8 KB available, thus (P1, 3) is executed with a TLP degree of 2 (8). After finishing (P2, 2), (P1, 3) continues running with all 32 KB available space of its SM and the SM belonged to P2, and reaches to the TLP degree of 8 (8). Based on above example, the constraints for sharing shared-memories among SMs are:

- Each SM has the highest priority for using its own shared-memory space.
- If an SM needs its shared-memory while it is lent to another SM, it has to wait for that SM to finish the execution of its current kernel to get its shared-memory back.
- Each TB can only borrow its required shared-memory from one SM and is not allowed to use the memory space of multiple SMs.
- Multiple TBs from distinct SMs can borrow the shared-memory of a single SM.

To realize resource sharing, borrowing and lending SMs need to communicate with each other. Allowing all SMs to communicate with each other reduces the efficiency of proposed method though, due to area and power overheads of the added hardware and performance overheads due to memory conflicts. However, when too few SMs are able to share their memory banks, few programs would be running in parallel and thus, less performance improvement is achieved. Therefore, it is desirable to cluster a proper number of SMs to share their memories. Fig. 3 shows an architecture where SMs are clustered together in groups of four and share their memory spaces.

We use some multiplexers (MUXs) and demultiplexers (DeMUXs) in each SM in the proposed architecture for inter-SM communication. The MUX of each SM connects the write and read buffers of its shared-memory unit to other SMs in the cluster through their Demuxs (3). For this purpose, a small internal table inside each SM saves the status of its shared-memory unit (4). When a TB arrives and requires shared-memory, start and end addresses of the allocated shared-memory space with the corresponding SM-Identifier (SM-Id) are stored in this table. Shared-Memory Sharing Controller (5) forms the path by controlling MUX and DeMUX select signals based on the internal table information.

A common 32 × 30 bit table saves the status of shared-memory allocations among all TBs of SMs in a cluster (4). Each row of this table has an address decoded by SM-Identifier, which is unique for each SM in its cluster, and a TB-Identifier (TB-Id). Start and End addresses of allotted shared-memory spaces are registered in the table lines. Due to the possible usage of the shared-memory units of other adjacent SMs, the SM-Id of the SM lending (L/SM-Id) its shared-memory is also registered in the table.

When a TB wants shared-memory space upon starting its execution, its SM sends a request to shared-memory allocator (4) of the cluster that serves requests one by one (one request at a time). For each SM, an absence/presence (A/P) register (5) exists that shows which TBs of the cluster are using which SMs shared-memory units. This register also acts like valid flag for lines of the common table (8 bits of the register are used by the corresponding SM itself; note that each SM can run at most 8 TBs).

The first-fit algorithm tries to find an appropriate memory space for any TB requesting shared-memory in the cluster (4). It examines the common table lines using SM-Id, TB-Id and A/P register to find out the status of shared-memory units in the cluster. It checks unallocated memory spaces listed in the common table line by line, and when finds the first line indicating a proper free

![Fig. 3. The proposed architecture for supporting shared-memory sharing between SMs.](image-url)
memory space for the requesting TB, allocates it and updates the corresponding table lines and informs the requesting and lending SMs about its allocation to update their local tables. Virtually, implementation of the allocator can be realized through some parallel combinational logic for checking the availability of each line of each candidate TB. In next stage of the logic, a simple combinational logic selects an available TB with highest priority.

Based on the proposed architecture, accesses to all shared-memory banks within each cluster are equal when output paths of MUXs and DeMUXs are not changed. The GPU firmware determines kernels execution information including TLP limits and the number of TBs per SM. At the beginning of each kernel, it is checked which SM needs more shared-memory to run extra TBs. Then, SMs are checked to determine which one can provide the required shared-memory space. These steps are not on the critical-path, hence, there is no performance degradation. The latency of the formed path for inter-SM shared-memory accesses (between MUXs and DeMUXs) and their switching time is lower than 0.08ns (calculated by Synopsis Design Compiler) plus 0.55ns access latency of shared-memory (calculated by CACTI). That means the access latency is less than one cycle (1.4ns @700 MHz). Thus, there are no extra stall cycles from non-local memory accesses.

Our area estimation using Synopsys Design and Memory Compiler considering a die size of 529 mm$^2$ and 3.2 billion transistors for GTX480, shows that the proposed architecture for sharing the shared-memory has lower than 1% area overhead with respect to the total chip area.

In terms of performance and power consumption overheads, the worst case is where a program which employs some of its shared-memory space lends the unused portions to other programs. In these cases, the MUXs and DeMUXs may switch between SMs of the clusters which lead to some slight power consumption overhead. We have partially minimized the occurrence of such cases by, prohibiting SMs from borrowing shared-memory from multiple SMs. Also, due to the fact that the switching time of MUXs/DeMUXs is less than one cycle, (about 3.25% for a 2-program case as the worst case), overall performance remains intact.

**5. Evaluation platform**

**5.1. Baseline multi-programming architecture**

Many GPGPU applications cannot fully utilize GPU resources. The issue also remains when temporal multi-tasking is exploited for running multiple programs. While a program is running on GPU through cooperative multi-tasking, it is not possible for other programs to access GPU resources until the current program releases the GPU voluntarily. Moreover, switching between programs could only occur between the execution of kernels. This issue can potentially cause starvation and system failure by malicious codes or malfunctioning programs. For preventing this problem, earlier OS versions limit the time a program can possess GPU resources and terminate violating programs. To address the problem, there are some approaches, but they are not performance efficient for GPGPUs. As an instance, preemptive multitasking has solved this problem, but it degrades the performance due to prohibitive overhead of context switching due to massive register file, caches and shared memory per SM in GPU.

The multi-programming method was introduced to address the above mentioned problems. Instead of allotting all resources of GPU to one program for a while and switching among different programs, all programs run on their own portion of resources simultaneously. It results in greatly improving GPU utilization and subsequently, throughput improvement without the overhead of context switching.

We implemented the multi-programming scheme based on spatial multitasking method in [7]. Also, we added some modifications to better utilize the resources to address some issues which were not mentioned in previous works. The most important modification is partitioning GPU L2 cache among programs in order to prevent thrashing. The multi-programming method partitions SMs among different programs and each program executes its kernels on allotted SMs similar to [7], but the multi-programming scheme implemented in this paper partitions SMs evenly among programs and when the number of SMs is not dividable to the number of programs, extra SMs will be allotted to the programs that have larger remaining TB ratio (RTR) calculated by:

$$\text{RTR}_P = \frac{\# \text{ of remaining TBs of } P}{\text{total TBs of } P}. \quad (1)$$

Fig. 4 depicts an example of running 3 programs, P1, P2 and P3, under the proposed multi-programming scheme. Assume that there are 5 SMs and P1, P2 and P3 consist of 64, 30 and 20 TBs with a maximum TBs per SM of 4, 3 and 2, respectively. First, P1 starts its execution at interval#1 and simply all of 5 SMs are allotted to it. After a while, at interval#2, P2 arrives. The SM allocator allocates 2 SMs to each of P1 and P2, and the 5th SM will be assigned to the program with greater RTR. P2 takes the 5th SM because in our example RTR of P1 and P2 are $30/4 = 0.47$ and $30/5 = 1.00$, respectively. Indeed, when the number of SMs is not dividable by the number of programs, the program recently started running will always take one of the remaining SMs, since its RTR equals 1 (maximum possible value). At this point, 3 SMs are assigned to P2 and the rests are allocated to P1. At interval#3, these 3 SMs are released gradually and P2 starts running on them. Later, P3 arrives at interval#4. The SM partitioning algorithm calculates RTR values of $4/0.47 = 0.08$, $5/0.53$ and $5/1.00$ for P1, P2 and P3, respectively. P1 and P2 release some SMs after finishing TBs on them and P3 takes them at interval#5. Finally at interval#6, P1 finishes running and its SM is allocated to P3 based on calculated RTR values of $3/0.17$ and $3/0.80$ for P2 and P3, respectively.

This simple partitioning algorithm tries to balance the execution time of different programs. In other words, it favours maximizing the time different kernels are executing together to benefit from aforementioned multi-programming advantages.

The partitioning algorithm makes decisions for the number of SMs that should be allocated to each program. When a new program arrives, the partitioning procedure will be repeated with the new set of programs and the decision is applied gradually. It means that when a new program wants to start on the GPU, the partitioning procedure decides to allot some SMs to it. Therefore, current running programs must release their extra SMs when execution of their existing TBs on these SMs is finished.

Beside SM partitioning, exterior SM resources would be smartly allocated to programs. GPU L2-cache, which has a limited size, is shared among SMs. If multiple programs run in parallel and each program access the whole space of L2-cache, programs might thrash each others data in L2-cache. The multi-programming method partitions L2-cache among programs to address this problem.

Dividing L2-cache by ways reduces cache associativity and does not require any modifications to cache configuration. Thus, the multi-programming scheme divides L2-cache by ways, and cache controller only permits programs to evict and access their own data. This can be easily done by small registers indicating the owner of each way. The L2-cache allocation to each set is realized by a policy similar to the approach for partitioning of SMs. In order to evaluate the proposed scheme, we employed GPGPU-Sim [20] and applied the required changes to implement our idea. Table 3 shows our baseline GPU architecture configuration.

---

Please cite this article as: H. Abbasitabar et al., ASHA: An adaptive shared-memory sharing architecture for multi-programmed GPUs, Microprocessors and Microsystems (2016), http://dx.doi.org/10.1016/j.micpro.2016.07.012
Fig. 4. An example of running 3 programs, P1, P2 and P3, under multi-program.

Table 3
Baseline GPU configuration (based on GTX480).

<table>
<thead>
<tr>
<th>SM</th>
<th>No. of SMs</th>
<th>SPs per SM</th>
<th>SP working frequency</th>
<th>SM working frequency</th>
<th>Max. no. of TBs</th>
<th>Shared-memory size</th>
<th>Max. no. of threads</th>
<th>Register file</th>
<th>L1-cache size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
<td>32</td>
<td>1400 MHz</td>
<td>700 MHz</td>
<td>8</td>
<td>16–48 KB</td>
<td>1536</td>
<td>16K 32-bit</td>
<td>16–48 KB</td>
</tr>
</tbody>
</table>

Inter-Connection Network

<table>
<thead>
<tr>
<th>Inter-connection between SMs and L2-cache partitions</th>
<th>Butterfly</th>
</tr>
</thead>
</table>

L2-Cache

<table>
<thead>
<tr>
<th>Size</th>
<th>768 KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of partitions</td>
<td>6</td>
</tr>
<tr>
<td>Partition configuration</td>
<td>64 sets, 128B blocks, 16 ways</td>
</tr>
</tbody>
</table>

5.2. Applications

We selected 12 applications in order to characterize GPU applications behavior running on the baseline and proposed architectures. All of these programs are written in CUDA and chosen from 3 benchmark suites: RODINIA [21], Parboil [22] and GPGPU-Sim [20].

Table 4 lists the investigated programs based on their kernels behavior. The kernels can be categorized into two groups based on whether shared-memory is their only TLP bottleneck or not. The first 7 applications (G1) benefit from shared-memory sharing when running simultaneously. However, for the last five applications (G2), shared-memory does not limit TLP. Such a grouping helps us better explain the evaluation results.

5.3. Metrics

Since GPUs are exploited for high throughput applications, normalized instructions per cycle (normalized IPC) is usually used to report throughput improvement. Eq. (2) depicts how to measure IPC for concurrent programs; it means that overall IPC can be obtained by summation of individual IPCs of N programs running in parallel [23].

\[
\text{throughput} = \sum_{i=0}^{N-1} \text{IPC}_i
\]

IPC cannot solely depict performance enhancements when multiple programs are executing on GPU simultaneously. Hence, two other metrics, weighted speedup (Wspeedup) [24] and harmonic
mean of speedups (Hspeedup) [25], are used to evaluate our proposed method. Wspeedup makes the impact of all programs in the job mixture equal by normalizing their multi-program IPC to the single-program IPC. Hspeedup benefits from the advantages of harmonic mean and implicitly considers fairness in IPC improvement of each program. Eqs. (3) and (4) express Hspeedup and Wspeedup, where $IPCP^S$ and $IPCMP^S$ are IPC of the $i$-th program under single-program and multi-program execution, respectively. Finally, geometric means of both Hspeedup and Wspeedup are reported for the proposed approaches.

$$Hspeedup = \frac{N}{\sum_{i=0}^{N-1} \frac{IPCP^S}{IPCP^M}}$$  \hspace{1cm} (3)

$$Wspeedup = \sum_{i=0}^{N-1} \frac{IPCMP^S}{IPCP^M}$$  \hspace{1cm} (4)

5.4. Experimental methodology

In order to evaluate the proposed approaches, we employed GPGPU-Sim [20] and applied required changes to implement our idea. Simulations of the multi-programming method were done through isolated executions of each program on their allotted SMs and other private resources. Moreover, resource sharing method was simulated by an in-house small simulator for working with information adopted from GPGPU-Sim kernel executions.

NVIDIA GTX480 is used as our simulation configuration (see Table 3) with some changes. The number of SMs was altered from 15 to 16 in order to simplify the simulation of resource partitioning between programs. In addition, current benchmarks are released so that they can run on a wide range of GPUs including older ones with 16 KB fixed shared-memory size.

As it is shown in Fig. 5, GPGPU-Sim determines kernels execution information including the number of run cycles, number of executed instructions, TLP limits, number of TBs per SM, etc. At the beginning of each kernel run, our simulator checks which SM needs more shared-memory to run extra TBs. TLP limits and the number of TBs on this SM are checked then if shared-memory is the only TLP bottleneck, the SM should get its needed amount of shared-memory in order to run the extra concurrent TB.

In the next step, SMs located in a cluster are checked to determine which SM can provide the required shared-memory space. If there are several candidate SMs, first-fit algorithm selects the lender SM and shared-memory space will be possessed by the borrower SM. These steps (checking TLP bottlenecks and finding lender SM in clusters) are repeated until shared-memory would not be the only bottleneck of TLP. As each kernel is running on all or some of the SMs allocated to the program to which it belongs, borrowed resources will be released on the kernel finishing time. Moreover, the number of executed instruction and execution cycles for each kernel are calculated and performance metrics such as normalized IPC, Hspeedup and Wspeedup, in addition to other detailed statistics, are computed using this information. The exploited toolset and its source codes as well as full description of using it are available in http://ce.sharif.ir/~abbasitabar/rs_gpgpu.html.

6. Experimental results

In our experiments, we use the multi-programming technique that was described in Section 5, as the baseline to show performance enhancement attained by ASHA. We assume the number of SMs in each cluster equals the number of programs. The obtained IPC results are averaged over all possible combinations of the programs executed in parallel.

It is expected that the multi-programming scheme improves overall GPU throughput and running more programs in parallel should increase the possibility of achieving higher throughputs. However, the multi-programming scheme did not improve the performance of GPU for a few set of programs, especially 2-program mixes as evidenced in Fig. 6. This is because running more programs in parallel reduces the probability of performance degradation through multi-programming. Indeed, about 30% of 2-program execution scenarios experienced performance degradation. This value is 12% and 0.1% for 4-program and 8-program execution scenarios, respectively. ASHA, in the presence of multi-programming, reduces program combinations that experience performance degradation by 25%, 24% and 39% in 2-, 4- and 8-program executions, respectively. To emphasize that ASHA could resolve performance degradation of multi-programming, we used sequential execution of the programs as the baseline in Fig. 6. As it is shown, ASHA increases the IPC median by shifting the speedup toward higher values.

Fig. 7 indicates to what extent each program can benefit from ASHA. In other words, it presents the average speedup of each
Fig. 5. Simulation platform used for simulating multi-program and resource sharing.

Fig. 6. Cumulative distribution of speedups of all program combinations for multi-programming method with and without shared-memory sharing capability. All speedup values are normalized by the sequential run IPC. The baseline marker illustrates the baseline multi-programming execution without sharing the shared-memory.

Please cite this article as: H. Abbasitabar et al., ASHA: An adaptive shared-memory sharing architecture for multi-programmed GPUs, Microprocessors and Microsystems (2016), http://dx.doi.org/10.1016/j.micpro.2016.07.012
program for all combinations that it had with other programs. By comparing average values in 2-, 4- and 8-program scenarios for G1 (solely) and for G1 & G2 together, it can be seen that 8-program scenario resulted in the highest average speedup improvement. The main reason for higher speedups achieved is due to the opportunities created for shared-memory sharing when more programs run in parallel. That is, when more programs are executed simultaneously, the possibility of sharing more shared-memory space by programs that can lend their shared-memory space in the same cluster is increased.

When ASHA mechanism is added to the proposed architecture, experimental results show that for all 2-, 4- and 8-program execution scenarios, performance in improved.

Table 5 reveals that shared-memory sharing has not experienced any performance degradation for all combinations of programs, although theoretically performance degradation is undeniable. Note that Table 5 compares ASHA with a spatial multi-programming baseline, but in Fig. 6 the baseline of all charts is sequential execution of programs.

TLP improvement per program is presented in Fig. 8. The considered applications have TLP degrees with average of 2.14 for G1 programs and 2.84 for G1 & G2 programs. The ASHA method has enhanced the average degree of TLP to 3.94, 4.16 and 4.29 for 2-, 4- and 8-program scenarios, respectively, which means 1.39X, 1.47X and 1.51X performance improvement, respectively. When only G1 programs are considered, for 2-, 4- and 8-program scenarios, performance improvement even grows to 1.75X, 1.94X, and 2.04X, respectively.

Fig. 9 illustrates that both Hspeedup and Wspeedup are enhanced by the proposed multi-programming method. Quadruple program execution in terms of Hspeedup and octuple program execution in terms of Wspeedup achieved the most of improvements.

As Wspeedup and Hspeedup measure the overall improvement of the system based on the division of the throughput of each program in multi-program execution by its single-program execution, it can better show the GPU speedup by tending to smaller values. Considering that ASHA usually keeps the speedups of the lender programs fixed, and improves IPC of the borrowers, both Hspeedup and Wspeedup are increased for all combinations of programs.

7. Conclusion

In this paper, we proposed an adaptive shared-memory sharing architectural support for GPUs. The proposed method was built upon spatial multi-programmed GPUs, called ASHA. In addition to the capability of multi-tasking, the multi-programming scheme creates opportunities for resource sharing among SMs. This capability was attained by producing variety in SMs resource requirements, such as shared-memory, during programs runtime. SMs running programs which need more shared-memory space may borrow unutilized portions of shared-memory of other SMs and therefore, handle more TBs. Our experiments showed that the proposed ASHA method improved speedup of a multi-programmed GPU by 17%, 18% and 21% on average, for 2-, 4- and 8-program execution scenarios, respectively. Additionally, weighted speedup were improved for all 2-, 4- and 8-program executions by 6, 7 and 7% as well harmonic mean of speedups by 7, 9 and 10%, respectively.

![Fig. 7. IPC improvement for each program individually, achieved by shared-memory sharing technique normalized to the baseline multi-programming IPC.](image1)

![Fig. 8. Overall average enhancement of TLP degree for each program using shared-memory sharing method in comparison with baseline multi-programming.](image2)

### Table 5

<table>
<thead>
<tr>
<th>Running programs</th>
<th>Geometric mean of IPCs</th>
<th>Max. IPC</th>
<th>Min. IPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.17</td>
<td>1.60</td>
<td>1.00</td>
</tr>
<tr>
<td>4</td>
<td>1.13</td>
<td>1.78</td>
<td>1.00</td>
</tr>
<tr>
<td>8</td>
<td>1.10</td>
<td>1.39</td>
<td>1.02</td>
</tr>
</tbody>
</table>
Fig. 9. Wspeedup and Hspeedup improvements compared to the baseline multi-programming architecture. Thick lines indicate sequential run values.

References