ScaleGPU: GPU Architecture for Memory-Unaware GPU Programming

Youngsok Kim, Jaewon Lee, Donggyu Kim, and Jangwoo Kim
Department of Computer Science and Engineering, POSTECH
{elixir,spiegel0,vteori,jangwoo}@postech.ac.kr

Abstract—Programmer-managed GPU memory is a major challenge in writing GPU applications. Programmers must rewrite and optimize an existing code for a different GPU memory size for both portability and performance. Alternatively, they can achieve only portability by disabling GPU memory at the cost of significant performance degradation. In this paper, we propose ScaleGPU, a novel GPU architecture to enable high-performance memory-unaware GPU programming. ScaleGPU uses GPU memory as a cache of CPU memory to provide programmers a view of CPU memory-sized programming space. ScaleGPU also achieves high performance by minimizing the amount of CPU-GPU data transfers and by utilizing the GPU memory's high bandwidth. Our experiments show that ScaleGPU can run a GPU application on any GPU memory size and also improves performance significantly. For example, ScaleGPU improves the performance of the *hotspot* application by ~48% using the same size of GPU memory and reduces its memory size requirement by ~75% maintaining the target performance.

1 Introduction

Graphics processing units (GPUs) are becoming popular for general purpose computations due to their high throughput. GPUs achieve orders-of-magnitude speed-up over multi-core CPUs by running thousands of concurrent threads [11].

Although GPUs promise high computational throughput, the programmer-managed GPU memory and slow data transfers through PCI Express (PCIe) make it hard to achieve the optimal throughput. If programmers wish to run GPU codes written and optimized for a large GPU memory on a small GPU memory, they must perform a series of *manual code modifications* based on the characteristics of the codes and algorithms (e.g., memory access pattern). An example of such application includes GPU-accelerated large scale data analysis [12]. Alternatively, programmers may use the *zero-copy* scheme [8] to avoid manual code modifications by having GPUs use CPU memory directly. However, the zero-copy scheme incurs prohibitive performance degradation due to redundant long-latency data transfers between CPU and GPU as the GPU memory is disabled. As a result, current GPU architectures and programming models do not provide high-performance memory-unaware GPU programming.

This paper proposes ScaleGPU, a novel GPU architecture which can run applications on any GPU memory size while providing high performance. To achieve both portability and performance, ScaleGPU uses GPU memory as a cache of CPU memory. First, ScaleGPU achieves portability by providing programmers a view of CPU memory-sized programming space. Second, ScaleGPU achieves high performance by minimizing the amount of long-latency CPU-GPU memory transfer as GPU performs most of the memory accesses on the cached data in GPU memory. Our detailed simulations show that ScaleGPU not only runs GPU applications written for a large GPU memory on a small GPU memory without any programming efforts, but also improves performance significantly. For example, ScaleGPU achieves ~48% speedup for the same size of GPU memory or it consumes ~25% of GPU memory for the same target performance in the case of *hotspot* application. ScaleGPU outperforms the zero-copy scheme significantly as expected.

• *Manuscript submitted: 02-Apr-2013. Manuscript accepted: 31-May-2013. Final manuscript received: 18-Jun-2013.*

Date of publication 15 Jul. 2013; date of current version 31 Dec. 2014.
Digital Object Identifier 10.1109/L-CA.2013.19

2 Using GPU Memory as a Cache

2.1 Motivation

Current GPU programming models require programmers to group threads into thread blocks manually so that the threads within a thread block access contiguous memory regions to take advantage of intra-core cache resources (e.g., shared memory, L1 cache). To reduce the memory bandwidth usage, current GPU architectures also exploit the contiguous memory access patterns by coalescing memory accesses to adjacent data regions. Therefore, the current GPU programming models and architectures lead to very high spatial and temporal localities among GPU memory accesses, which motivate us to use the GPU memory as a cache of the CPU memory.

To achieve such programming portability, programmers may consider using the zero-copy scheme existing in modern GPUs, which forwards all memory requests from GPU cores directly to CPU memory. Therefore, the zero-copy scheme disables GPU memory completely. The zero-copy scheme can also perform thread execution and data transfers in parallel. However, the zero-copy scheme suffers from significant performance degradation when either the data is reused (e.g., spatial and temporal localities,) or there exist non-coalescable memory transfer accesses (i.e. indirect memory accesses). These limitations motivate using GPU memory as a cache of CPU memory to provide both portability and performance by exploiting the GPU's high access localities and memory bandwidth.

2.2 Advantages

Utilizing GPU memory as a cache of CPU memory has two main benefits. First, caching the data improves performance by removing unnecessary memory transfers. Therefore, GPU does not have to forward memory requests directly to CPU as long as the data is present in GPU memory. Instead, GPU can get the data from its local memory without paying the long-latency CPU-GPU communication overhead.

Second, GPU memory only stores the application's current working set which is typically small due to high spatial and temporal localities by the nature of GPU applications. As GPU applications tend to touch only a small portion of memory, the size of GPU memory can be significantly reduced without incurring a performance loss.

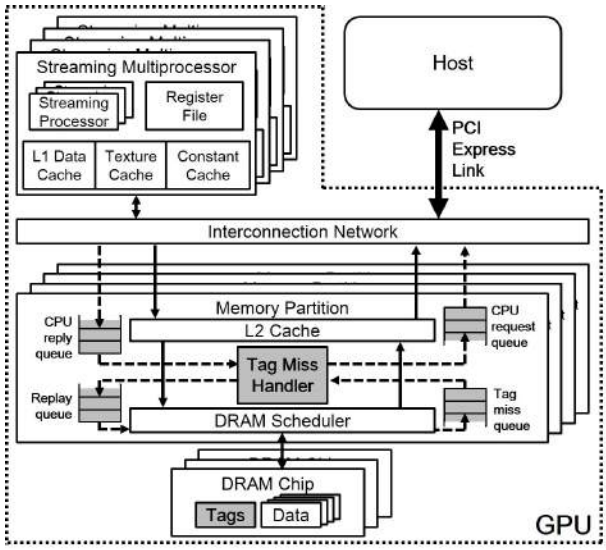

Fig. 1: ScaleGPU architecture.

Therefore, CPU and GPU memories become dynamically balanced by the application's behavior.

3 ScaleGPU Architecture

Figure 1 shows the architecture of ScaleGPU, where the grey components are newly introduced on top of the existing GPU architecture. ScaleGPU implements the GPU memory as a DRAM cache [13] of the CPU memory by storing tags and data in GPU DRAM chips and adding a hardware module to handle tag misses and message queues to buffer requests and replies. This section describes each hardware component of ScaleGPU in detail.

3.1 Assigning tags to data

When the GPU memory functions as a cache of the CPU memory, conflict misses can occur in the GPU memory. In order to detect such conflict misses, the GPU memory in ScaleGPU must store the *tag* of the memory address. ScaleGPU implements tags as the most significant bits of the virtual address instead of those of the physical address.

ScaleGPU associates a single tag to each DRAM row and increases the memory partition interleaving granularity to the DRAM row size for exploiting the high spatial locality inherent in GPU memory accesses. The overhead of the increased interleaving granularity is compensated by the increased row buffer hits and the overlapping of data transfers and computations. The size of the GPU memory determines the number of bits used for the tag. The rest of the virtual address indicates the physical location such as the target memory chip, bank, row, or column. ScaleGPU places tags and data in separate banks to perform tag comparisons and DRAM row accesses in parallel.

3.2 Tag miss handler

When the DRAM scheduler receives a memory request, it reads the tag from tag banks and the data from data banks. In the case of a tag hit, the DRAM scheduler processes the memory request and sends a reply back to the sender. Otherwise, the memory request is forwarded to the *tag miss handler (TMH)*, indicating one of the following cases: cold/conflict-missed read/atomic requests, conflict-missed write requests, and cold-missed write requests.

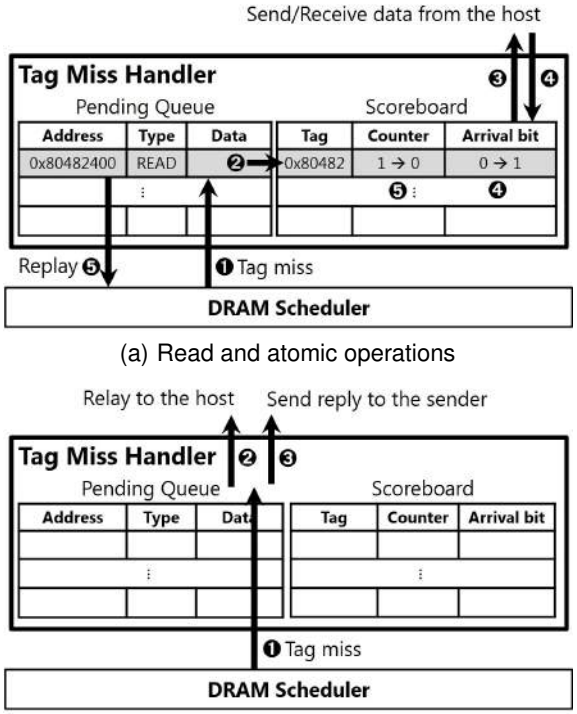

Fig. 2: Tag miss handler (TMH).

The TMH consists of the pending queue and the scoreboard, as shown in Figure 2. The pending queue holds the tag-missed memory requests. Each scoreboard entry has a target tag, a counter, and an arrival bit. The counter stores the number of memory requests in the pending queue whose tag match the target tag. The arrival bit indicates whether the data of the target tag are loaded on GPU memory. By storing recent tags, the TMH exploits the scoreboard to avoid a burst of data fetch requests to the host.

Figure 2(a) shows how the TMH handles the cold/conflict-missed read/atomic request. First, the TMH pushes the request into the pending queue. Next, it searches the scoreboard to find an entry whose target tag matches that of the request. When no matching entry exists in the scoreboard, the TMH generates a new data fetch request for the tag, sends it to the host, and creates a new scoreboard entry with the counter set to one and the arrival bit set to zero. If a matching entry exists in the scoreboard, the TMH does not send a new data fetch request to the host and only increases the counter of the matching entry. When the request reaches the head of the pending queue, it periodically checks the arrival bit of the scoreboard entry to keep track of the availability of its data. When the reply from the host arrives at the TMH, it changes the arrival bit from zero to one. When the request detects that the arrival bit has changed to one, the counter of the matching entry is decreased and the request is sent to the DRAM scheduler for replay. The matching entry is removed from the queue when its counter becomes zero.

Figure 2(b) shows how the TMH handles the conflict-missed write request. The TMH simply relays the conflict-missed write request to the host by bypassing the pending queue, and sends a reply to the sender immediately. Simply relaying the conflict-missed write requests to the host is suitable for GPU applications because the write data are not likely to be accessed by other GPU threads following the nature of relaxed GPU memory model [8]. On the other hand, the cold-missed write requests are treated as tag hits and their data

TABLE 1: GPGPU-Sim configuration

Cores	14 cores, 1150MHz, SIMT width = 32
Resources / core	Max. 1536 threads, 32768 registers, 48KB shared memory,
Caches / core	16KB L1 data, 2KB L1 instruction, 12KB texture, 8KB constant
Scheduling	Round-robin warp scheduling, loose round-robin block scheduling
Branch divergence	PDOM [3]
Interconnection	Crossbar, 1150MHz, flit size = 32B
Memory partition	Six memory partitions, 1150MHz, 128KB L2 data cache, partition interleaving granularity = 256B for baseline, 2KB for ScaleGPU
DRAM controller	FR-FCFS, 64B minimum access granularity, max. 16 DRAM requests per memory controller, two DRAM chips per memory controller
GDDR5 organization	16 banks, row size = 2KB
GDDR5 timing	750MHz, $t_{CL} = 12$, $t_{RP} = 12$, $t_{RC} = 40$, $t_{RAS} = 28$, $t_{RCD} = 12$, $t_{RRD} = 6$, $t_{CDLR} = 5$, $t_{WR} = 12$
PCI Express	2.0 with 16 lanes, 5GT/s, peak bandwidth = 8GB/s, PCIe packet header size = 20B, max. payload size = 128B, CPU-GPU round-trip latency = 1200ns,
TMH	max. pending queue entries = 32, max. scoreboard entries = 32

TABLE 2: Benchmarks

Name	Data size	Benchmark suite
vectorAdd	48MB	NVIDIA SDK [9]
histogram	48MB	NVIDIA SDK
hotspot	48MB	Rodinia [2]
Shortest Path	41MB	Based on *bfs* from Rodinia
Bank Account	47MB	KILO-TM [4]

are written to the DRAM directly. However, to ensure consistency between GPU memory and CPU memory, the TMH also sends the requests to the host.

3.3 Overlap of data transfers and GPU execution

ScaleGPU allows to overlap data transfers and GPU execution by executing GPU threads whose data has been delivered by TMH from CPU memory to GPU memory following the spatial and temporal localities. In addition, ScaleGPU can adjust the size of data transfer by changing the number of tags associated to a DRAM row while fully utilizing the PCIe bandwidth. On the other hand, existing GPU architectures must either serialize the transfers and the execution, or disable GPU memory by forwarding every memory access from GPU to CPU. Both methods incur a significant performance degradation.

4 Evaluation

4.1 Methodology

We implement ScaleGPU on top of GPGPU-Sim version 3.1.2 [1] with the parameters shown in Table 1. In particular, we carefully model the CPU-GPU communication via PCIe as both ScaleGPU and the zero-copy scheme communicate with CPU during kernel execution. We obtain the PCIe parameters from a reference machine using an NVIDIA Tesla C2050 GPU, and validated the performance of both the PCIe link and the zero-copy scheme against the reference machine using various pointer-chasing microbenchmarks. For the zero-copy scheme, we enable only L2 caches [10]. We model a tag access as a 64B DRAM access due to the minimal access granularity.

Table 2 shows the three distinct groups of benchmarks used to evaluate ScaleGPU. First, *vectorAdd* and *histogram* contain touch-once data only and their memory accesses can be easily split. These workloads are expected to run on a smaller GPU memory without performance degradation. Using the zero-copy scheme on these workloads can improve the performance as they do not reuse data.

Second, *hotspot* can be manually modified to run on a smaller GPU memory, but it incurs a significant performance degradation due to the increased amount of data transfer. The overall amount of data transfer increases as the amount of data required for each iteration is larger than the smaller GPU memory. The zero-copy scheme is expected to incur significant performance degradation as each iteration uses the data generated by the previous iteration, resulting in redundant data transfers between CPU and GPU.

Third, *Shortest Path* and *Bank account* cannot be modified to run on a smaller GPU memory due to their indirect memory accesses, even though they touch only a small portion of the large memory space. The zero-copy scheme is expected to incur performance degradation due to both the reused data and a large number of uncoalesced memory accesses (i.e., atomic operations). As a summary, we perform manual code modifications only for *vectorAdd*, *histogram*, and *hotspot* to run on small GPU memory sizes.

Due to the excessive simulation times, we also scale the data sizes of the benchmarks to 48MB. However, we ensure the scaled benchmarks preserve their original GPU core utilization and memory throughput usage by monitoring their dynamic resource usages using GPGPU-Sim.

4.2 High performance using smaller GPU memory

Figure 3 shows the execution time of benchmarks when we vary the size of the GPU memory. The baseline and zero-copy scheme run programmer-modified codes for a smaller GPU memory and for bypassed GPU memory, respectively. We evaluate the runtime of ScaleGPU by decreasing the size of the GPU memory from 100% to 12.5% of the data size. We normalize the runtime of smaller GPU memory to that of the baseline with 100% GPU memory. Each runtime is also divided into host-to-device (H2D) transfer latency, device-to-host (D2H) transfer latency, and GPU kernel execution latency. It should be noted that *Shortest Path* and *Bank Account* cannot be manually modified to fit in smaller memory due to the nature of their algorithm.

First, ScaleGPU achieves an average of 8% speedup for split-friendly workloads such as *vectorAdd* and *histogram* by overlapping the data transfers and the GPU kernel executions. The zero-copy scheme achieves an average of 58% speedup because these applications do not reuse data and all memory accesses are coalesced. ScaleGPU does not reach the zero-copy scheme's performance as it accesses the GPU memory before forwarding the request to CPU memory.

Next, ScaleGPU achieves an average of 32% speedup for *hotspot* over the manually modified codes. Whereas both the manually modified codes and the zero-copy scheme suffer from significant performance losses due to the increased memory transfer, ScaleGPU maintains baseline performance using only 25% of the memory. Such performance improvement comes from frequent data reuse in GPU memory as well as the overlapping of the data transfers and the GPU execution.

Finally, *Shortest Path* and *Bank Account* fail to run on baseline with smaller GPU memory because these applications cannot be split to fit in a smaller GPU memory. Although the zero-copy scheme successfully runs on smaller GPU memory, it suffers from a

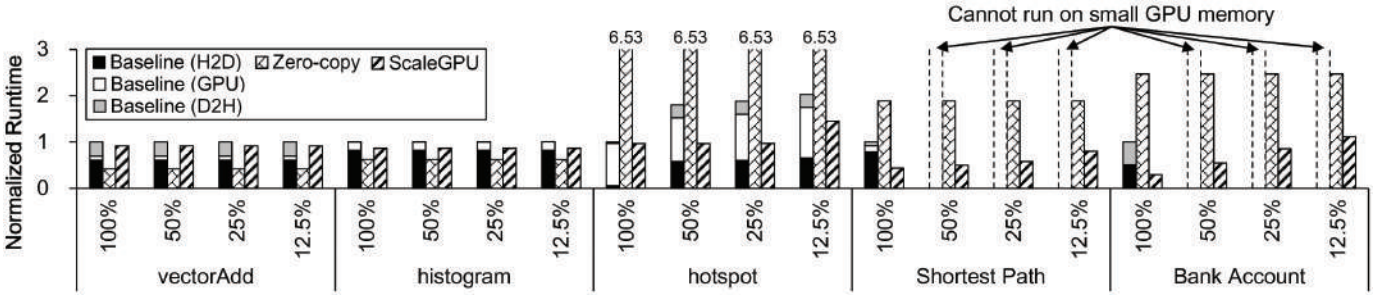

Fig. 3: Runtime comparison of the baseline, zero-copy scheme, and ScaleGPU.

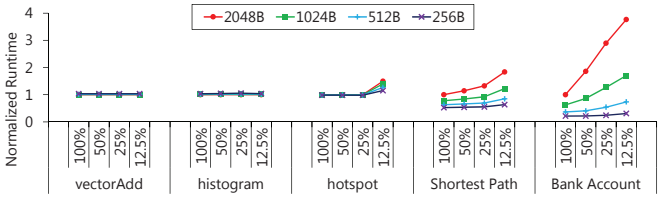

Fig. 4: Impact of the amount of data per tag.

large number of uncoalesced memory accesses. On the other hand, ScaleGPU runs the benchmarks without any code modification, and even achieves an average of 42% speedup by minimizing the amount of data transfer significantly. In particular, *Shortest Path* achieves a 21% speedup using only 12.5% of the GPU memory in ScaleGPU when compared to the baseline with 100% of GPU memory.

4.3 The amount of data per tag

When the GPU memory is too small compared to the working set, ScaleGPU may suffer from the increased number of tag conflict misses. Many tag conflict misses can reduce the performance improvement by transferring the same data multiple times. We observe that *hotspot*, *Shortest Path*, and *Bank Account* suffer from tag conflict misses when the GPU memory becomes too small. In such cases, ScaleGPU can mitigate the performance loss by reducing the amount of data per tag (or increasing the number of tags per a DRAM row), which effectively reduces tag conflict misses.

Figure 4 shows the performance impact when varying the amount of data per tag from 2048 bytes to 256 bytes which is large enough to exploit spatial locality. Runtimes are normalized to that of 2048 bytes of data per tag with 100% GPU memory. For *vectorAdd* and *histogram*, varying the amount of data per tag does not affect the performance of ScaleGPU because these workloads do not reuse data in GPU memory. However, we observe that *hotspot*, *Shortest Path*, and *Bank Account* benefit from decreasing tag conflict misses as the amount of data per tag decreases.

As *Shortest Path* and *Bank Account* initiate unnecessary data transfers mainly due to a lot of small-sized indirect memory accesses, ScaleGPU improves performance by using smaller amount of data per tag with 100% GPU memory due to the reduced amount of data transfers. Note that a single DRAM bank is sufficiently large enough to store the tags even with 256B of data per tag as the data size is still significantly larger than the tag size.

5 Related Work

Programmers must modify existing GPU codes manually to run on smaller GPU memory or disable GPU memory [8]. Compiler-assisted static partitioning [6] can be applied only to static analysis-friendly workloads. ScaleGPU does not require any code modification and can be applied to any GPU workload. Lustig and Martonosi [7] propose programmer-managed overlapping of data transfer and GPU execution for a fixed size of memory. ScaleGPU does not require any programmer annotation and can be applied to any GPU memory size. To the best of our knowledge, near-future VM supports from industry will require non-trivial translation and page table overhead to support fine-grained paging for discrete GPUs [5]. In fact, ScaleGPU acts as a light-weight VM, which enables fine-grained paging without incurring such overheads.

6 Acknowledgments

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2011-0014817) and NRF Grant funded by the Korean Government (NRF-2012-Global Ph.D. Fellowship Program).

7 Conclusion

We propose ScaleGPU, a novel GPU architecture to enable high-performance memory-unaware GPU programming. ScaleGPU uses GPU memory as a cache of the CPU memory to enable memory-unaware GPU programming. ScaleGPU achieves a significant performance improvement, while also reducing the memory requirement significantly.

References

[1] A. Bakhoda *et al.*, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *ISPASS*, 2009.
[2] S. Che *et al.*, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IISWC*, 2009.
[3] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," in *MICRO*, 2007.
[4] W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt, "Hardware Transactional Memory for GPU Architectures," in *MICRO*, 2011.
[5] M. Harris, "Future Directions for CUDA," in *GTC*, 2013.
[6] J. Kim, H. Kim, J. H. Lee, and J. Lee, "Achieving a Single Compute Device Image in OpenCL for Multiple GPUs," in *PPoPP*, 2011.
[7] D. Lustig and M. Martonosi, "Reducing GPU Offload Latency via Fine-Grained CPU-GPU Synchronization," in *HPCA*, 2013.
[8] NVIDIA, "CUDA C Programming Guide," http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.
[9] NVIDIA, "GPU Computing SDK," https://developer.nvidia.com/gpu-computing-sdk.
[10] NVIDIA Developer Zone, "Custom CPU to GPU ringbuffer," http://devtalk.nvidia.com/default/topic/508723/cudaprogramming-and-performance/custom-cpu-to-gpuringbuffer/.
[11] S. Ryoo *et al.*, "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA," in *PPoPP*, 2008.
[12] R. Wu, B. Zhang, and M. Hsu, "GPU-Accelerated Large Scale Analytics," http://www.hpl.hp.com/techreports/2009/HPL-2009-38.pdf.
[13] L. Zhao, R. Iyer, R. Illikkal, and D. Newell, "Exploring DRAM Cache Architectures for CMP Server Platforms," in *ICCD*, 2007.