# An Attempt to Realize Unified Memory Extending GPGPU-Sim

Debashis Ganguly, Mohammad Hasanzadeh Mofrad
*Department of Computer Science*
*School of Computing and Information*
*University of Pittsburgh*
*Pittsburgh, USA*
*Email:* {debahis, hasanzadeh}@cs.pitt.edu

***Abstract*—The current GPGPU-Sim is fully functional with CUDA 4.0. It is a complete functional and timing simulator publicly available. During the course of this project, we came across multiple simulators, extending GPGPU-Sim, which demand to achieve one or the other theoretical concepts. From our detailed investigation, we realized that all of them miss either functional or timing simulation aspect of a simulator. Thus, we felt the need of extending GPGPU-Sim to realize Unified Memory both functionally and from timing simulation perspective.**

***Keywords*-GPGPU, on-demand paging, unified memory**

## I. Introduction

General-purpose computing workloads such as MapReduce [1], Graph Processing [2], and Deep Learning [3], [4] on Graphics Processing Units (GPUs) are seeing increasingly wide use. As a result, GPUs have been widely adopted in HPC-clusters [5] and cloud-based data centers [6]. This is because GPUs exploit thread-level parallelism (TLP) with its massive number of simple compute cores that even multiple CPUs can not offer due to the specialization in each chip.

While CPUs traditionally rely on caches and low-latency memory systems to improve performance of single-threaded execution, GPUs rely on multi-threading to hide memory latency and guarantee high throughput of heterogeneous compute workloads. However, in both discrete and integrated CPU-GPU systems, there are high overheads associated with kernel launch, memory management, and synchronization. Developers are more concerned with optimizing the execution of general purpose kernels offloaded from host CPU to the neighboring GPUs as these kernels typically belong to large, highly parallel, and throughput oriented workloads. Thus, traditionally, it was application programmers' responsibility to explicitly migrate data from host to guest memory and also take care of memory over-subscription. In the recent years, NVIDIA Pascal GPUs [7], with the support of CUDA 8.0, [8], truly enables Unified Memory where pages are transparently migrated from host memory to device memory on-demand. With on-demand paging, GPU is not left idle and can begin execution even before the complete dataset is copied to the local memory. However, this improved programmability comes at the cost of performance [9], [10]. With on-demand paging, the kernel execution incorporates the latency of individual page transfer on every page-fault [11]. Further, during this transfer, GPU cores grouped as Streaming Multi-processors (SMs) stall [12]. To address this, several works [12], [13] tried to overlap between kernel execution and page-fetching. Hence, there is an increasing trend to outweigh page-fetch latency by executing more long-running kernels or in other words saturate GPU compute units. Naturally, this necessitates sharing GPUs by multiple-kernels [14] to achieve better resource utilization, overall throughput [15], and power efficiency [16].

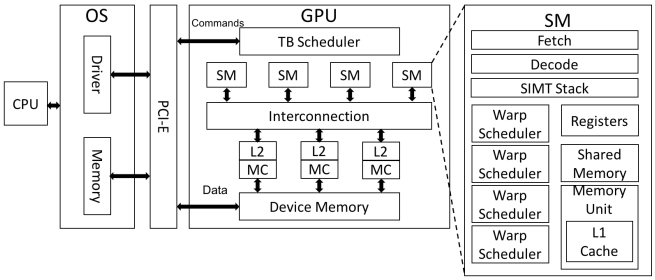


Figure 1. System Overview

This report lists out our current achievements towards realizing Unified Memory extending GPGPU-Sim. Although, the project goal is not fully realized due to limited background knowledge and timing constraints, we take pride in presenting our framework for demand-aging and also the insights, gained over the course of time, to integrate the framework to realize a fully functional timing simulator.

## II. Background

In this section, we describe the architecture, execution model, and the memory management of a GPU. We use NVIDIA CUDA terminology, but the concepts can also be applied to the other GPU vendors and runtimes.

### A. GPU Architecture

Figure 1 shows an overview discrete CPU/GPU system architecture where the subsytems are connected through PCIe bus. A GPU consists of a massive number of simple compute cores grouped into SMs. The SM is the main execution unit which performs as a compute unit as a

whole. Each SM has additional resources such as registers, shared memory, and L1 cache to enable computing on the set of cores. Each discrete GPU chip has its own local GDDR5 memory. SMs share this device memory through an interconnection network. Memory requests are distributed to the memory controllers according to requested address. Each memory controller has its own L2 cache. All these L2 cache together give an illusion of unified last level cache to the SMs.

### B. GPU Execution Model

Traditionally, a GPU program has two components: a host code and a device code, also known as GPU kernel. Kernels are Single Instruction Multiple Threads (SIMT) programs where programmer writes code for one thread, and the GPU generates user specified number of threads to execute the same code. Threads are grouped in Thread Blocks (TBs), which are dispatched to the GPU one at a time. Once a TB is dispatched to an SM, its threads are batched into warps by the hardware, 32 at a time also known as SIMD width of the GPU. An SM has one or more warp schedulers, andthe warps are distributed equally to them. The schedulers select instructions from the ready warps to keep the pipeline busy. This is because warps can be stalled due to long latency operations. The main objective of GPU is to hide long memory latency by exploiting massive thread-level parallelism (TLP).

GPUs are typically treated as co-processors or slave devices to the host CPU. Thus, GPUs are initialized and controlled by the designated runtime, like CUDA, resident to the Operating System running on the CPU.

### C. GPU Memory Management

In both discrete and integrated CPU/GPU systems, the host or CPU memory and the device or GPU memory can be either laid out as either partitioned address space or unified address space. In pre-Pascal GPUs, partitioned address space are prevalent. Kernels can only reference data physically residing on the local GDDR5 memory. Thus, commonly, application programmers explicitly copy data up front from host to device memory and then launch kernels to operate on that data. Further, upon completion of kernel execution, the results need to be copied back from device to host memory. As a result, memory copy and kernel execution are serialized.

However, with the introduction of paged memory, this restriction is relaxed. Unified Memory allows greater programmability upon providing a single memory space directly accessible by all GPUs and CPUs in the system, with automatic page migration for data locality [17]. NVIDIA Pascal GPU architecture [7] comes with larger virtual memory address space and Page Migration Engine, enabling true virtual memory demand paging. On Pascal and later GPUs, cudaMallocManaged API of CUDA 8 [8] returns a pointer to a memory, accessible to both CPU and GPU, which may not be physically allocated. Pages and page table entries are populated on-demand as accessed by the GPU or the CPU. As a result, kernels do not need to wait for the whole dataset to be copied over, and can start execution as soon as the first referred page is accessible.

## III. Extending GPGPU-Sim for CUDA 8.0

The stable release of GPGPU-Sim, released in 2009, currently works with CUDA 4.2 and gcc/g++ 4.4. One of the achievement of this project is to make the compilation of GPGPU-Sim possible with CUDA 8.0 and gcc/g++ 5.2 released in 2017. To achieve this, we used the `dev` branch of GPGGPU-Sim and made some changes in its `Makefile, Config files, and Environment files`. Also, GPGPU-Sim is highly dependant on the `NVIDIA_GPU_COMPUTING_SDK` belonging to CUDA 4.2 and not compatible with the newer versions of CUDA toolkit. We extened GPGPU-Sim such that it points to CUDA 8.0 and generates `ptx` executable on Pascal micro-architecture with CUDA 8.0.

## IV. New APIs for Unified Memory

In this project, we implemented the hooks for two CUDA APIs `cudaMallocManaged` and `cudaMemPrefetchAsync`. These two APIs are originally introduced in CUDA 7 with the purpose of realizing Unified Memory. However, NVIDIA makes them fully functional with the introduction of CUDA 8 and Pascal micro-architecture. The new methods enable on-demand paging across a unified space of memory. The stubs are implemented without any side effect during compilation of the GPGPU-Sim and the sample CUDA programs. Moreover, the GPGPU-Sim does not implement `cudaFree` API. We also implemented this API to make it compatible with Unified Memory implementation.

## V. Framework for On-demand Paging

GPGPU-Sim currently does not support Unified Memory. Before introduction of UVM, an user application had to initialize data sets on both CPU and GPU memory using malloc and cudaMalloc respectively. Then, user had to explicitly copy data between these two memory addresses using cudaMemcpy. Unified Memory removes the redundant declaration of two pieces of memory on both CPU and GPU and thus invariably removes the need to copy data between them. With cudaMallocManaged, the program gets a virtual address of an allocation that is addressable by both CPU and GPU. The first challenge, we faced while implementing UVM, is to allocate memory and return virtual address to the same which can be accessed by both CPU and GPU. For that we introduced a new data-structure called Managed_Page that represents a page in UVM allocated by cudaMallocManaged. It encompasses few major members,

namely (i) base address to the page, (ii) a bit to represent whether the page is valid in GPU or not, (iii) a dirty bit to denote whether the content of the page requires write-back to CPU, and (iv) a counter for least-recently-used policy. The second data-structure introduced is called Managed_Allocation. It represents a cudaMallocManaged allocation or list of Managed_Page. It has - (i) base address to the allocation, (iii) total size of the allocations in bytes, and (iii) a dictionary of Managed_Page keyed by base address of each page starting from the base address of the allocation. The major component of UVM in our implementation is GMMU which represents GPU Memory Management Unit and is responsible for managed UVM allocations. It has a dictionary of Managed_Allocation keyed by the base address of the allocation per concurrent application. Thus, this represents page table for each application on GPU side.

## VI. Future Directions

Although, we could not finish implementing the On-demand Paging, we found some sweat spots in the GPGPU-Sim code-base which are promising for our future research. We found out, the parser implementation of GPGPU-Sim is the first place that the memory addresses are generated for an `opcode`. We think further investigation of this component will help us find the memory addresses used in `ptx`. Furthermore, the `read` and `write` methods under the `memory_space_impl` are called by individual instructions such as load, store, add, etc. The `memory_space_impl` has a key-value data structure from where it returns a `storage` data type corresponding to a virtual address. The way GPGPU-Sim operated on this class is cryptic and unknown to us at this moment. Hence, further investigation on this implementation will also help us crack the GPGPU-Sim and find how to attach the `read` and `write` methods to our Unified Memory implementation. We also need to emulate page fault for first access to a managed page, stalling warp threads waiting on far-faults, coalescing memory accesses on PCI-E, pre-fetching of pages and page eviction under oversubscription.

## References


[1] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 260–269.

[2] J. Zhong and B. He, "Medusa: Simplified graph processing on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1543–1552, 2014.

[3] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang, "Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 27–40.

[4] M. Wang, T. Xiao, J. Li, J. Zhang, C. Hong, and Z. Zhang, "Minerva: A scalable and highly efficient training platform for deep learning," in *NIPS Workshop, Distributed Machine Learning and Matrix Computations*, 2014.

[5] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "Gpu cluster for high performance computing," in *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*. IEEE, 2004, pp. 47–47.

[6] A. Herrera, "Nvidia grid: Graphics accelerated vdi with the visual performance of a workstation," *Nvidia Corp*, 2014.

[7] "Whitepaper: NVIDIA Tesla P100," https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016, [Online; accessed 04-Dec-2017].

[8] "CUDA RUNTIME API: API Reference Manual," http://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf, July, 2017, [Online; accessed 04-Dec-2017].

[9] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt, "An investigation of unified memory access performance in cuda," in *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*. IEEE, 2014, pp. 1–6.

[10] M. Dashti and A. Fedorova, "Analyzing memory management methods on integrated cpu-gpu systems," in *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*. ACM, 2017, pp. 59–69.

[11] M. Harris, "Unified Memory for CUDA Beginners," https://devblogs.nvidia.com/parallelforall/unified-memory-cuda-beginners/, June, 2017, [Online; accessed 04-Dec-2017].

[12] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler, "Towards high performance paged memory for gpus," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 345–357.

[13] D. Lustig and M. Martonosi, "Reducing gpu offload latency via fine-grained cpu-gpu synchronization," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE, 2013, pp. 354–365.

[14] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving gpgpu concurrency with elastic kernels," in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 407–418.

[15] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 358–369.

[16] G. Wang, Y. Lin, and W. Yi, "Kernel fusion: An effective method for better power efficiency on multithreaded gpu," in *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*. IEEE Computer Society, 2010, pp. 344–350.

[17] N. Sakharnykh, "Beyond GPU Memory Limits with Unified Memory on Pascal," https://devblogs.nvidia.com/parallelforall/beyond-gpu-memory-limits-unified-memory-pascal/, December, 2016, [Online; accessed 04-Dec-2017].