An Evaluation of Unified Memory Technology on NVIDIA GPUs

Wenqiang Li*, Guanghao Jin†, Xuewen Cui*, Simon See*‡
*Center for High Performance Computing, Shanghai Jiao Tong University, China
†Tokyo Institute of Technology, Japan
‡NVIDIA, Singapore
liwenqiang@sjtu.edu.cn, jin.g.ab@m.titech.ac.jp, {hp1991, simon.see}@sjtu.edu.cn

***Abstract*—Unified Memory is an emerging technology which is supported by CUDA 6.X. Before CUDA 6.X, the existing CUDA programming model relies on programmers to explicitly manage data between CPU and GPU and hence increases programming complexity. CUDA 6.X provides a new technology which is called as Unified Memory to provide a new programming model that defines CPU and GPU memory space as a single coherent memory (imaging as a same common address space). The system manages data access between CPU and GPU without explicit memory copy functions. This paper is to evaluate the Unified Memory technology through different applications on different GPUs to show the users how to use the Unified Memory technology of CUDA 6.X efficiently. The applications include Diffusion3D Benchmark, Parboil Benchmark Suite, and Matrix Multiplication from the CUDA SDK Samples. We changed those applications to corresponding Unified Memory versions and compare those with the original ones. We selected the NVIDIA Kepler K40 and the Jetson TK1, which can represent the latest GPUs with Kepler architecture and the first mobile platform of NVIDIA series with Kepler GPU. This paper shows that Unified Memory versions cause 10% performance loss on average. Furthermore, we used the NVIDIA Visual Profiler to dig the reason of the performance loss by the Unified Memory technology.**

Keywords: Unified Memory, Heterogeneous Computing, CUDA programming model

I. Introduction

GPUs have recently attracted the attention of many application developers as commodity data-parallel coprocessors. In a typical PC or cluster node today, the memories of the CPU and GPU are physically distinct and separated by the PCI-Express bus. Before CUDA 6.0, that is exactly how a programmer has to view things. Data that is shared between the CPU and GPU must be allocated in both memories, and explicitly copied between them by kernels. This adds much complexity to CUDA programs. For example, in the previous CUDA programming model[11], an application programmer transfers data from the host to GPU by calling a memory copy routine whose input parameters include a source data pointer to the host memory space, a destination pointer to the GPU memory space, and the number of bytes to be copied. The memory copy interface ensures that GPU can only access the part of the application data that is explicitly requested by the memory copy parameters.

Unified Memory is a component of the new CUDA programming model, first introduced in CUDA 6.0, which defines a new managed memory space in which all processors see a single coherent memory image with a common address space. The underlying system handles data access and locality without requiring explicit memory copy calls. Unified Memory benefits GPU programming in two ways: simplifying GPU programming and migrating data transparently. However, the workflow of the Unified Memory and the influence still remain unknown.

In this paper, we used Diffusion3D Benchmark, Parboil Benchmark Suite, and Matrix Multiplication from the CUDA SDK Samples to evaluate Unified Memory. We first changed those applications to corresponding Unified Memory versions and compared them with the original versions. We selected the Kepler K40 and the TK1[14] to evaluate the performance of Unified Memory. K40 is the latest GPU from NVIDIA. It offers a total of 12 GB of GDDR5 on-board memory, and its peak performance of double precision floating point reaches 1.43 TFLOPS. Jetson TK1 is the first mobile platform which has the powerful Kepler GPU with 192 cores and breakthrough power efficiency. The main reason that we chose the TK1 is the fact that it supports Unified Memory with 2GB physically unified memory. After comparing the performance of the original benchmarks with the Unified Memory versions, we found that while Unified Memory can simplify programming, it causes 10% performance loss on average. We used NVIDIA profiler to find the reason. Profiling results showed that kernel running time on GPU is exactly the same. We also generated PTX codes[15] (which are the pseudo-assembly codes for CUDA) and found that the PTX codes of the two versions have same structure and contents. By those profiling, we found that the performance loss is caused by redundant memory transfer (also for TK1) and page caching fault between CPU and GPU in the Unified Memory programming model. We further developed five microbenchmarks to detect when redundant memory transfer will happen.

To the best of our knowledge, the following are our original contributions:

- We introduced an evaluation methodology of the Unified Memory technology.
- We measured the performance loss of different applications when using Unified Memory on GPUs.
- We explained and validated the reason of performance loss caused by Unified Memory.

This paper is organized as follows. Section II introduces the programming model of Unified Memory programming. Section III shows the Evaluation Methodology including introduction of benchmark suites, profiling tools and test beds. Section IV discusses the result of the evaluation. Related work is introduced in Section V. Section VI concludes this paper.

II. Unified Memory Programming Model

Programming models for current heterogeneous parallel systems, such as CUDA and OpenCL[6], present CPU and GPU memories in the system as distinct memory spaces to the programmer. Applications explicitly request memory allocation on host and device memory spaces by function call (i.e. *malloc()* and *cudaMalloc()*) and perform data transfers between host and device memories by another function call (i.e. *cudaMemcpy()*). The example in Figure 1(a) illustrates this process. First, host memory is allocated for initialization (*malloc()*). Then, device memory is allocated (*cudaMalloc()*). Next, the data is copied from host memory to device memory by function call (*cudaMemcpy()*). Finally, the code is executed on the GPU side[10].

```
void compute(FILE *file, int size)
{
  dataElem *elem, *d_elem;
  elem = malloc(size);
  fread(elem, size, 1, file);
  cudaMalloc(&d_elem, size);
  cudaMemcpy(d_elem,elem,size,cudaMemcpyHostToDevice);
  kernel<<<grid,block>>>(d_elem, size);
  cudaDeviceSynchronize();
  cudaMemcpy(elem,d_elem,size,cudaMemcpyDeviceToHost);
  use_data(elem);
  cudaFree(d_elem);
  free(elem);
}
```

(a)

```
void compute(FILE *file, int size)
{
  dataElem *elem;
  cudaMallocManaged(&elem, size);
  fread(elem, size, 1, file);
  kernel<<<grid,block>>>(elem, size);
  cudaDeviceSynchronize();
  use_data(elem);
  cudaFree(elem);
}
```

(b)

Fig. 1: Sample code for normal CUDA programming model (a) and Unified Memory programming model (b). Unified Memory code is really simple and similar to CPU code.

Unified Memory removes multiple explicit requests for memory copy between CPU and GPU. Unified Memory creates one managed memory space that is shared between the CPU and GPU, bridging the CPU and GPU. The managed memory space is accessible from both CPU and GPU. The global pointer can be used to access the space without any additional function call or explicit data migration. In Unified Memory program model, programmers can allocate managed memory by two ways: via the new *cudaMallocManaged()* routine, which is semantically similar to *cudaMalloc()* ; or by defining a global *__managed__* variable.

Figure 1(b) shows an example using the Unified Memory programming model. The Unified Memory example does not call *cudaMemcpy()* but require an explicit *cudaDeviceSynchronize()* before the host program can safely use the output from the GPU. It illustrates the benefits of Unified Memory programming model which simplifies programming and maintains data coherence automatically. A GPU with compute capability 3.0 or higher (such as K20 and K40) can utilize the Unified Memory programming model[11].

Unified Memory also has some other new features:

- It is easier to control the memory space since programmers only need to allocate or delete the memory space in the global address by Unified Memory programming model.
- Unified Memory eliminates the need for deep copying when accessing structured data in GPU kernels. C++ simplifies the deep copy problem by using classes with copy constructors. It makes pass by value and pass by reference work on GPU.

III. Evaluation Approach

A. Benchmarks

We selected a series of benchmarks to compare the performance of Unified Memory version and the normal version. First, we introduce the benchmarks and the profiling tools that we used to evaluate. To simplify the implementation, we only adopted one GPU for the experiments.

1) Matrix Multiplication: the Matrix Multiplication benchmark comes from CUDA 6.0 SDK Samples. There are two versions of Matrix Multiplication. One is implemented using the cuBLAS library provided by NVIDIA. The other is an implementation of block matrix multiplication that takes advantage of shared memory[11][12].

2) Diffusion 3D Benchmark: the diffusion equation is a partial differential equation which describes density dynamics in a material undergoing diffusion. It is widely used to describe processes exhibiting diffusive behavior. Diffusion 3D Benchmark Suite includes three benchmarks: *diffusion3d_gpu_standard*, *diffusion3d_gpu_register_reuse* and *diffusion3d_gpu_shared_memory*. Different common optimization techniques are applied to solve 3D diffusion equation by the finite difference method.

3) Parboil Benchmark Suite: the Parboil Benchmark Suite[20] described in Table I is a set of computing applications which are useful for studying the performance of computer architectures and compilers. The Parboil benchmarks suite collects benchmarks from throughput computing application researchers in many different scientific and commercial fields including image processing, biomolecular simulation, fluid dynamics, and astronomy. We select part of the benchmarks because some are not suitable for porting to Unified Memory. For instance, the benchmark's memory space may increase dynamically when the program is running while programmers need to specify the memory size when

Benchmarks		Description
BFS	Breadth-First Search	The Breadth-first Search algorithm is commonly used in graph problems such as finding the shortest path between two nodes.
mri$_q$	Magnetic Resonance Imaging Q	Computation of a matrix Q, representing the scanner configuration, used in a 3D magnetic resonance image reconstruction algorithm in non-Cartesian space.
Stencil	Stencil	The importance of solving partial differential equations (PDE) numerically as well as the computationally-intensive nature of this class of application have made PDE solvers an interesting candidate for accelerators.
SpMV	Sparse Matrix-Dense Vector Multiplicatio	Sparse matrix-vector multiplication is the core of many iterative solvers. SpMV is memory-bandwidth bound when the matrix is large.
cutcp	Cutoff-limited Coulombic Potential	Cutoff-limited Coulombic Potential (CUTCP) computes a short range component of this map, in which the potential at a given map point comes only from atoms within a cutoff radius of 12 $\mathring{A}$

TABLE I: Parboil Benchmarks Description

using `cudaMallocManaged()`.

B. Profiling Tool

Profiling tools and APIs can help programmers to understand and optimize the performance of applications. A number of different techniques may be used by profilers, such as event-based, statistical, instrumented, and simulation methods. Profiling tools are important for understanding program behavior. NVIDIA Visual Profiler[16] is a graphical profiling tool that displays the timeline for an application's CPU and GPU activities. The nvprof profiling tool enables programmers to collect and view profiling data from the command-line. Unified Memory is fully supported by both the Visual Profiler and nvprof. Both profilers can capture the Unified Memory-related memory traffic to and from each GPU on the system.

C. Test bed

We conducted the experiments on our supercomputer π and NVIDIA TK1. We use one GPU node of π which has two NVIDIA Kepler K40 with two Intel Sandy Bridge CPU E5-2670 2.6GHz. TK1 is NVIDIA's embedded platform with 192 Kepler GPU cores and four ARM Cortex-A15 cores. Figure 2 and 3 show the the structure of π GPU node and TK1. Table II demonstrates the detailed features for K40 and the GPU of TK1.

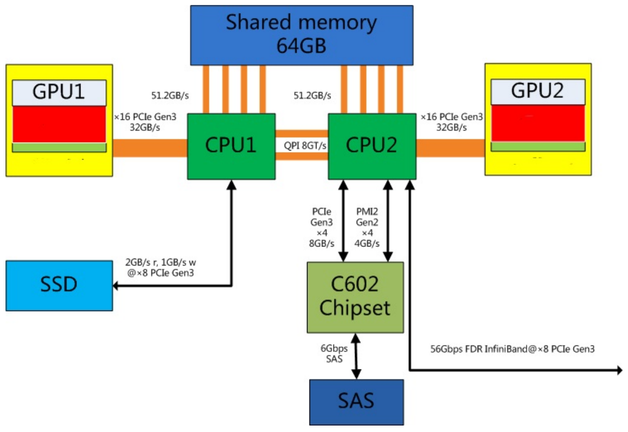


Fig. 2: Structure diagram of GPU Node

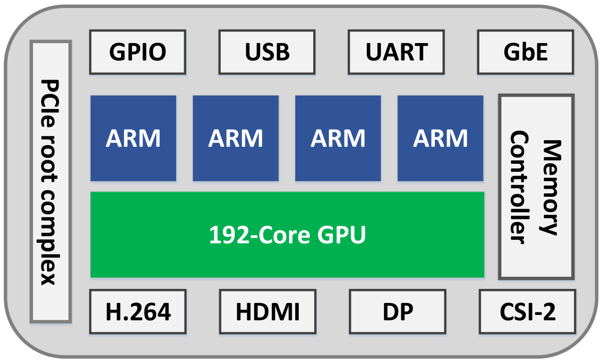


Fig. 3: Structure diagram of TK1

Features	Tesla K40	GPU of TK1
GPU	1 Kepler GK110B	1 Kepler GK20A
Computational Capability	3.5	3.2
CUDA cores	2880	192
Memory size	12GB(DDR5)	Shared 2GB(DDR3L)

TABLE II: GPU Features

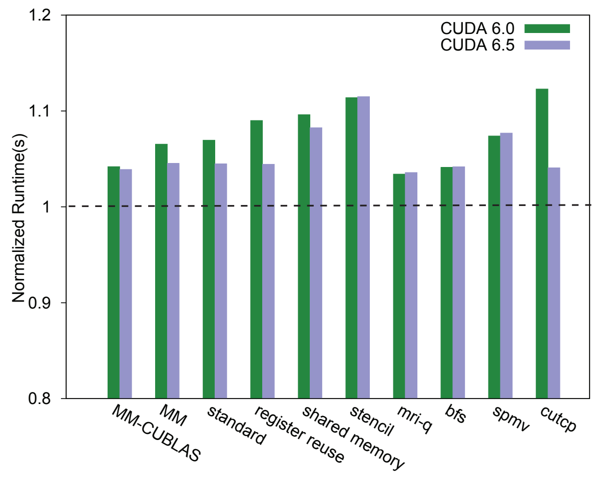
IV. Results and Analysis

A. Performance Results

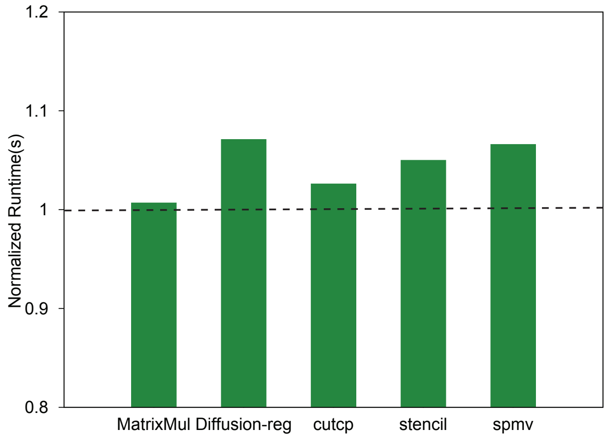
This section presents our experimental environment and the results of the evaluation for Unified Memory.

The experiments were tested both on CUDA 6.0 and CUDA 6.5 for K40 and on CUDA 6.0 for TK1. As there is no available CUDA 6.5 for TK1 now, we only show the results for TK1 on CUDA 6.0 in this evaluation.

The porting work only involved removing explicit data transfers and handling the allocation of data structure in the Unified Memory programming model. The porting process did not involve any optimization. After being ported to Unified Memory, the total number of lines of code (LOC) decreased in all benchmarks. It also indicates that Unified Memory simplifies CUDA programming.



(a)



(b)

Fig. 4: Performance of different Benchmark Suites on K40 (a). Part of the experiment results based on TK1 are presented in subfigure (b). Green and purple histograms correspond to Unified Memory version with CUDA 6.0 and CUDA 6.5

Figure 4 shows the performance results for each of the benchmarks, normalized to the case without using Unified Memory.

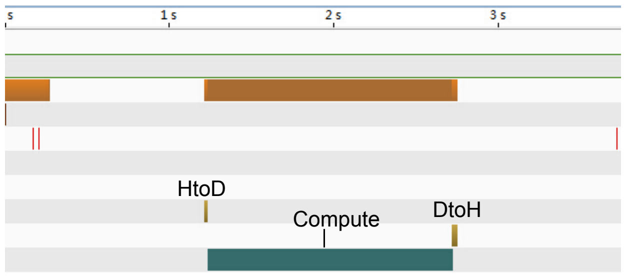
B. Discussion

According to Figure 4, the use of Unified Memory decreased performance. For instance, Figure 4(a) shows that the performance of Diffusion3D Benchmark Suite is reduced by 6.9%, 9.0%, and 9.6% to CUDA 6.0, respectively. Though CUDA 6.5 improves the performance compared to CUDA 6.0, the Unified Memory version still takes more time than the original one. For CUDA 6.5, we found that there is no significant change in the kernel running time, but the performance of launching and synchronizing has been improved according to the profiling results[13]. However, the workflow of Unified Memory is just the same as in CUDA 6.0. Similar results could be found for TK1 according to the Figure 4(b).

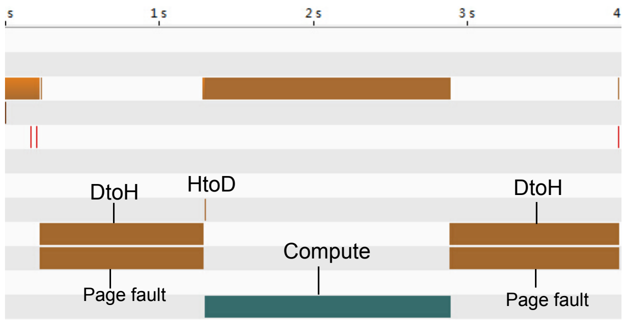
In order to figure out why Unified Memory decreases the performance. We first measured the kernel time for each benchmark. The profiling results indicate that the kernel time are exactly the same. By generating PTX codes, we found that the PTX codes of the two versions are just the same.

Next we paid our attention to the memory copy process. We firstly profiled the Diffusion3D_Standard benchmark and generated the timeline using NVIDIA Visual Profiler. Figure 5 shows that the two timelines for CUDA 6.0 are quite different. Firstly, the Unified Memory version has three memory copy operations while the normal one only has two memory copy operations. Secondly, data migration in Unified Memory costs much more time than the original version with lots of page faults[7]. In this case, we reviewed the source code and analyzed the workflow of the original version:

1) An array of $256 \times 256 \times 256$ floating point numbers is allocated both on host and on GPU memory. The total size of each array is $4\text{B} \times 256 \times 256 \times 256 = 67108864\text{B}$
2) Initialization the data on host.
3) Transfer data from host to device explicitly using `cudaMemcpy()`.
4) Launch CUDA kernel.
5) After finishing computing on GPU, copy the array from device to host.
6) Check accuracy.



(a) Normal timeline



(b) Unified Memory version timeline

Fig. 5: Diffusion3D_standard Timeline

By comparing these two timelines, We noticed that in Unified Memory version the initialization process coincided with page faults and memory copy. It means that the array is invalid to CPU at the beginning, and it triggers CPU page faults and extra memory transfers. Next, the data is transferred to GPU memory, and kernel is launched as the original version. In step 6, CPU reads the data after the GPU kernel completion. In Unified memory version, the memory migration extends to the whole checking routine and is also overlapped with page faults. Sum up the timeline of Unified Memory version: three memory migrations occur including two DtoH (device to host) copy coupled with 32768 times page faults and one HtoD (host to device) copy. The memory size of each transfer is equal to the size of array 67108864B. In our testbed, the page size of the Linux system is 4096B[21]. Do a simple calculation: $32768 \times 4096\text{B} = 2 \times 67108864\text{B}$. We can find that the total memory page faults requested equals to double size of the array corresponding to the two DtoH memory transfers.

As mentioned above, the CPU and GPU of TK1 share a 2 GB physically unified memory. However, Figure 4(b) shows that Unified Memory cannot improve the performance. Data movements still exist. Namely, the CPU and GPU of the TK1 use memory separately with the Unified Memory programming model.

We also found that the time of memory copy from host to device in the Unified Memory version is much less than the original one while the data size is the same. For the Unified Memory version, it cost 2.066 ms and the original version cost 20.796 ms in the above Diffusion3D_standard case. This result means that the data on system memory is pinned (page-locked) memory[18]. Pinned memory is stored in physical memory (RAM), and it enables the Direct Memory Access (DMA) [7]. With DMA, GPU can request host memory transfers without the involvement of CPU. Pinned memory is much more expensive to allocate and deallocate but provides higher transfer throughput for large memory transfers[2].

C. Redundant Memory transfers

In order to figure out when redundant memory transfers will happen when using Unified Memory, we developed five microbenchmarks categorized by the memory access behavior of the GPU kernel and the CPU component.

1) No kernel calls: The microbenchmark allocates a Unified variable x and initializes it on the CPU. The size of x is 16 KB, or four pages. The nvprof output is:

Device to Host (bytes)	16384
CPU Page faults	4

The profiling result shows that the microbenchmark contains a memory transfer from GPU to CPU. Thus initializing data on CPU will cause redundant DtoH transfers with Unified Memory. Although it is a very common pattern that the data is initialized on CPU and transferred to GPU for computation.

2) The data is read-only for GPU: As previous microbenchmark, the microbenchmark allocates a Unified variable x and initializes it on the CPU. Then we developed a kernel which only reads the managed memory in this microbenchmark. Next, the host reads the data again after kernel return. The nvprof output is:

Host to Device (bytes)	16384
Device to Host (bytes)	32768
CPU Page faults	4

We can find that the size of DtoH memory transfer is twice bigger than the *read-only* memory. Since the kernel did not change x, there is no need to update the memory on CPU. The Unified Memory always assumes the GPU has the newest data. So it will also cause redundant memory transfers.

3) kernel modified the data: This microbenchmark is similar to the previous except that the kernel modified the variable. The elements of x are incremented by one in GPU. nvprof output is:

Host to Device (bytes)	16384
Device to Host (bytes)	32768
CPU Page faults	4

The profiler shows the same results as the *read-only* benchmarks. If we never use the data modified by kernel, the DtoH transfer should be redundant memory transfer.

4) no kernel accesses the data: This microbenchmark allocates two Unified variable x, y and initializes them on the CPU. The sizes of x, y are equal to 16KB and 4KB, respectively. This microbenchmark only contains one kernel, and y is not argument of the kernel. The nvprof output is:

Host to Device (bytes)	20480
Device to Host (bytes)	20480
CPU Page faults	5

We can find that the size of HtoD transfer equals to the total size of x and y. Although y is not list in the kernel arguments, its data is still transferred to the device. Unified Memory seems to assume that any active kernel may use any managed memory. So the HtoD transfer is redundant memory transfer.

5) cpu never reads and writes the data: The nvprof does not show any memory transfers. CPU never read the data, and it is unnecessary to transfer data between CPU and GPU. There in no redundant memory transfers.

D. Summary

To explain the memory transfer between CPU and GPU, We can describe the Unified Memory with three different states. *Invalid* means that the data is invalid to CPU. Data must be transferred back if the CPU needs to read the data after an accelerator kernel returns. *Dirty* means that the CPU has an updated copy of the data, and this memory must be transferred back to the GPU when a kernel is called. *Valid* means that the CPU and the accelerator have the same copy of data, so the memory region does not need to be transferred before the next method invocation on the accelerator. According to our microbenchmarks results and discussion, we can use Figure 6 to describe the data states transition.

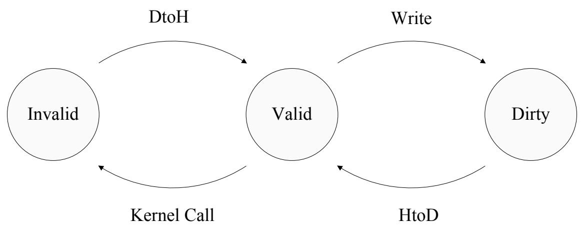


Fig. 6: States transition diagram

According to the states transition, we can explain why page faults happen. When CPU reads a piece of invalid data and raises page faults, the CUDA runtime detects the page faults and automatically migrates the data from device to host to deal with the page faults. This process may repeat many times until the CPU finishes reading. Then, the data is valid to CPU. Figure 7 describes this process. The overhead of page faults will also have an impact on performance.

In summary, we can find that the two major reasons for performance loss are:

- Redundant memory transfer
- CPU page faults

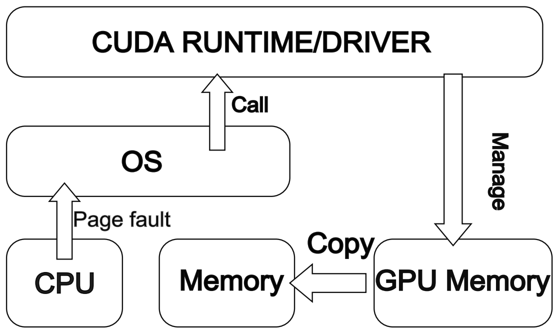


Fig. 7: The process how page faults happen

V. Related Work

With the development of heterogeneous computing, many s-tudies have been carried out to investigate the data communication between CPU and co-processor. Ryoo *et al.* [17] published on general principles for optimizing applications for GPU architecture. The key strategy is to use massive multithreading to utilize the large number of cores and hide global memory latency. Gelado *et al.* [5] presented a new programming model for heterogeneous computing, called *Asymmetric Distributed Shared Memory* (ADSM), that maintains a shared logical memory space for CPUs to access objects in the accelerator physical memory. Nickolls *et al.*[9] investigated the Unified Memory programming model and evaluate the performance. However, he only tested one benchmark suite and did not analyze the reason for the performance loss. Gelado *et al.* [4] came up with an architecture model CUBA for CPU/co-processors which is based on mapping the data structures required by the co-processor into a memory local to the co-processor. CUBA allows the CPU to cache hosted data structures with a selective write-through cache policy, allowing the CPU to access hosted data structures while supporting efficient communication with the co-processors. Jablin *et al.* [8] published a new data management model between CPU and GPU. This model manages complex and recursive data-structures without static analyses. Hartej *et al.* [19] presented a reconfigurable computing system which allows co-processors to access the system main memory using DMA.

VI. Conclusion and Future Work

This paper introduced an evaluation methodology of Unified Memory and presented a performance evaluation for the Unified Memory programming model. We ported Diffusion3D Benchmark, Parboil Benchmark Suite, and Matrix Multiplication from the CUDA SDK Samples to Unified Memory versions and selected NVIDIA Kepler K40 and TK1 as test devices. The results showed that Unified Memory will cause some performance loss although it can simplify programming.

We also validated that the performance loss is caused by redundant memory transfers and page faults when adopting the Unified Memory programming model. Profiling results and PTX codes showed that the Unified Memory programming model has no influence on kernel execution. We developed five microbenchmarks to summarize when redundant memory transfers will happen. We further proposed a memory states transition diagram and explained the reason why page fault happens based on the diagram.

In addition, we studied the memory behavior on TK1. Although there is 2GB physical unified memory on the TK1, the CPU parts and GPU parts are separated. There still exists memory transfers between CPU and GPU in Unified Memory programming model.

Today a typical system has one or more GPUs connected to a CPU using PCI Express[1](8 Giga-transfers per second per lane) which limits the GPU's ability to access the CPU memory system. The resource contention gets even worse when peer-to-peer GPU traffic is factored in. At the 2014 GPU Technology Conference, NVIDIA announced a new interconnect called NVLink which will provide between 80 and 200 GB/s of bandwidth[3], allowing GPU full-bandwidth access to the CPU's memory system. Since the data transfers between the CPU memory and GPU memory are much faster with NVLink, Unified Memory may release its potential. In our future study, we will focus on NVLink and evaluate the performance of Unified Memory.

As we have understood how Unified Memory work, future work will be likely to utilize these features and to optimize CUDA code to reduce the performance loss. We are also interested in the performance in multi-GPU systems. For now, multi-GPU systems are able to use managed memory, but managed memory allocation behaves identically to unmanaged memory allocated using `cudaMalloc()`. In the future, we will evaluate on multiple GPUs when Unified Memory works as well as investigate optimization techniques to reduce the performance loss caused by Unified Memory.

VII. Acknowledgement

This research is supported by CUDA Center of Excellence (CCOE) and Japan JSPS RONPAKU Fellowship. We would like to thank Minhua Wen and Dr. Eric Germaneau of SJTU for their sincere help.

References

[1] J. Ajanovic. Pci express*(pcie*) 3.0 accelerator features. *Intel Corporation, White Paper*, 10, 2008.

[2] S. Cook. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2013.

[3] D. Foley. NVLink, Pascal and Stacked Memory: Feeding the Appetite for Big Data. *Nvidia.com*, 2014.

[4] I Gelado, J.H. Kelm, S Ryoo, S.S. Lumetta, N. Navarro, and W.W. Hwu. Cuba: An architecture for efficient cpu/co-processor data communication. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, pages 299–308. ACM, 2008.

[5] I. Gelado, J.E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. *SIGPLAN Not.*, 45(3):347–358, March 2010.

[6] Khronos OpenCL Working Group. The OpenCL Specification, v1.2.15. 2011.

[7] J.L. Hennessy and D.A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[8] T.B. Jablin, P. Prabhu, J.A. Jablin, N.P. Johnson, S.R. Beard, and D.I. August. Automatic cpu-gpu communication management and optimization. *SIGPLAN Not.*, 46(6):142–151, June 2011.

[9] R. Landaverde, T.S Zhang, A.K Coskun, and M. Herbordt. An investigation of unified memory access performance in cuda. 2014.

[10] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[11] NVIDIA. CUDA C Programming Guide, v6.0. 2014.

[12] NVIDIA. CUDA Samples, v6.0. 2014.

[13] NVIDIA. CUDA Toolkit Release Notes, v6.5. 2014.

[14] NVIDIA. NVIDIA Tegra K1 Whitepaper, v1.1. 2014.

[15] NVIDIA. Parallel Thread Execution ISA, v4.0. 2014.

[16] NVIDIA. Profiler User's Guide, v6.0. 2014.

[17] S. Ryoo, C.I. Rodrigues, S.S. Baghsorkhi, S.S. Stone, D.B. Kirk, and W.W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 73–82. ACM, 2008.

[18] D. Schaa and D. Kaeli. Exploring the multiple-gpu design space. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12, May 2009.

[19] H. Singh, M.H. Lee, G.M. Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M. Chaves Filho. Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *Computers, IEEE Transactions on*, 49(5):465–481, May 2000.

[20] J.A. Stratton, C. Rodrigues, I.J. Sung, N. Obeid, L.W. Chang, N. Anssari, G.D. Liu, and W.W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.

[21] A.S. Tanenbaum and A.S. Woodhull. *Operating Systems Design and Implementation (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.