An Investigation of Unified Memory Access Performance in CUDA

Raphael Landaverde, Tiansheng Zhang, Ayse K. Coskun and Martin Herbordt
Electrical and Computer Engineering Department, Boston University, Boston, MA, USA
{soptnrs, tszhang, acoskun, herbordt}@bu.edu

***Abstract*—Managing memory between the CPU and GPU is a major challenge in GPU computing. A programming model, Unified Memory Access (UMA), has been recently introduced by Nvidia to simplify the complexities of memory management while claiming good overall performance. In this paper, we investigate this programming model and evaluate its performance and programming model simplifications based on our experimental results. We find that beyond on-demand data transfers to the CPU, the GPU is also able to request subsets of data it requires on demand. This feature allows UMA to outperform full data transfer methods for certain parallel applications and small data sizes. We also find, however, that for the majority of applications and memory access patterns, the performance overheads associated with UMA are significant, while the simplifications to the programming model restrict flexibility for adding future optimizations.**

I. Introduction

GPUs have been used extensively in the past 7-8 years for a wide variety of computational acceleration. For many applications, the level of parallelism introduced by the GPU architecture and enabled by the use of Nvidia CUDA have allowed for orders of magnitude of acceleration [1]. Examples of a few problem spaces accelerated by GPU acceleration include molecular docking [2], numerical weather prediction [3], and geophysical signal processing [4].

Although GPUs provide many mechanisms for accelerating a wide variety of programs, its use is not a silver bullet for time consuming calculations. There are significant limitations, particularly concerning memory bandwidth latency and GPU utilization. Along with these difficulties, acceleration over mature and highly optimized CPU implementations of computations for many problem spaces may not provide the order of magnitude improvement that people have come to expect from GPUs [5]. These issues are exacerbated by the already difficult nature of mapping existing algorithms to the unique and parallel design of a GPU.

To partially alleviate this issue, Nvidia has introduced Unified Memory Access (UMA) in their most recent CUDA 6 SDK [6]. UMA is primarily a programming model improvement created to simplify the complicated methods which GPUs require for memory communication with a host device, typically a CPU. Nvidia's primary goal in this design is to create an SDK feature that enables quick acceleration of simple applications, while providing high bandwidth for data transfers at runtime for shared CPU and GPU data.

In this paper, we investigate the performance and behavior of UMA on a variety of common memory access patterns, especially the communication behavior between a host CPU and GPU. In particular, we investigate the behavior of UMA memory transfers and analyze whether UMA provides better performance over the standard data transfer implementation as was done prior to the introduction of CUDA 6. We also analyze whether certain sparse memory access patterns provide an immediate and simple performance benefit with UMA usage. To test this feature, we develop multiple customized microbenchmarks for the GPU architecture. Furthermore, to investigate UMA performance on representative problems and applications, we provide a brief classification of the Rodinia benchmark suite [7], categorize the benchmarks by their behavior, and then create UMA implementations for a subset of them to investigate the changes in performance. We find that for the vast majority of applications, UMA generates significant overhead and results in notable performance loss. Furthermore, the UMA model only marginally simplifies the programming model for most applications.

The rest of this paper is organized as follows. We first introduce the background of current GPU architecture as well as the means of communication between CPU and GPU in section II. Section III presents our general experimental methodology, including the benchmarks we develop and experimental setup we use in this paper. In section IV, we show the classification of Rodinia benchmarks based on our setup. We evaluate and discuss our experimental results in section V and section VI concludes this paper.

II. GPU Memory Model and UMA

GPUs originated from the need to dedicate off-chip processors for handling the computationally intensive tasks of rendering computer graphics. However, this dramatically different design for a microprocessor enabled massive gains in parallelism that could accelerate time-consuming computations unrelated to graphics. For this reason, Nvidia introduced compute unified device architecture (CUDA), a language and programming interfaces for interacting with GPUs using C/C++, providing the mechanisms for organizing threads on to the GPU architecture [6], [1].

Typically, the huge performance improvement gained from a GPU lies in the massive quantity of cores that behave in a single instruction multiple threads (SIMT) manner [1]. However, in order to keep these cores active, data must remain local to the GPU. The memory hierarchy of a Kepler generation GPU, the current state of the art, is shown in Figure 1. Global memory is the bulk of the memory on the GPU device, stored in off-chip DRAM, and with the slowest latency on board. The fastest memory is shared memory for each of the Steaming Multiprocessors (SMs), accessible directly by threads and programmer managed. This hierarchy enables a programmer to control data flow and minimize access latency.

978-1-4799-6233-4/14/$31.00

However, the data to be operated on is always generated on the host CPU first, as the GPU is simply a slave device when using CUDA. Thus, the common programming model is as follows:

1) CPU serial code and data initialization
2) Data transfer to the GPU
3) Parallel computation execution
4) Data transfer back to the CPU.
5) CPU serial code

Without significant changes to the design of the parallel computation for the purposes of optimization, there is no possible overlap between data transfer and GPU computation. The introduction of Nvidia UMA does not change this, but rather simplifies the model to:

1) CPU serial code and data initialization
2) Parallel kernel execution
3) Synchronization between CPU and GPU.
4) CPU serial code

Here there is no more need for explicit data transfer in the code. Nvidia claims that the data transfer occurs on demand for both GPU and CPU, requiring the use of only one array of data, and no need for duplicate pointers as data is transferred between the two processors. Furthermore, they emphasize that this new method both simplifies the programming model and enables close to maximum bandwidth for the data transfer [6], [9]. This model is best seen in Figure 2, where from the developer's point of view, the CPU and GPU memory are the same. The simplified data model expands the ease of GPU programming and ideally, provides immediate performance benefits of full bandwidth data transfer to the GPU. Along with this, Nvidia hints that the future of UMA lies in providing more automatic performance benefits within a simplified programming model [9]. Unfortunately, the details of UMA are hidden from the programmer's perspective. When profiling applications with Nvidia tools, neither the destination of transferred data nor the actual size or length of transfers is observable. Thus, it is important to analyze the performance and behaviors of UMA for different memory access patterns from a raw latency perspective.

Other than Nvidia's UMA, there is very little work in the field of improving the communication between CPU and GPU. A fine grained GPU-CPU synchronization scheme using full/empty bits is one of the only articles directly addressing the communication pattern [10]. However, their work requires both modifications to the GPU architecture, and a more robust and complicated programming model in order to yield raw performance gains. We instead focus on examining UMA and discussing the potential of UMA to become a more performance oriented tool.

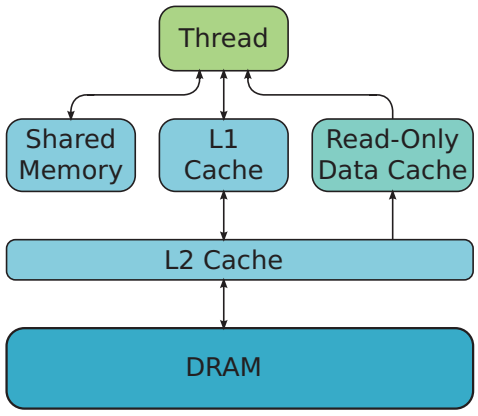

Fig. 1: Memory hierarchy of the Kepler GPU architecture [8].

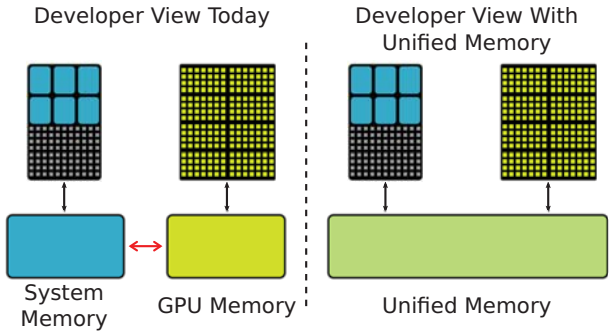

Fig. 2: Traditional and unified memory access model [9].

Our goal in this paper is analyzing the performance of this simplified model, and discovering the details of how UMA transfers data between GPU and CPU. Since UMA operates similarly to a standard paging mechanism [6], we focus on analyzing the particular effects this mechanism has on the overall performance.

III. Methodology

A. GPU Microbenchmarks

To analyze the behavior of UMA, we develop microbenchmarks that address common computation and memory access patterns between the GPU kernel and CPU code. In particular, we choose that the data is always organized as a matrix of floating point values arranged in a one-dimensional array. The computation then follows the following steps:

1) Memory allocation on the Host CPU
2) Initialization of data on the Host CPU
3) (*Non UMA*) Transfer data to the GPU
4) Perform matrix computation on the GPU
5) (*Non UMA*) Transfer data to the Host CPU
6) Iterate over matrix on the Host CPU

The items listed as *non UMA* are not performed for the UMA version of the benchmarks, as UMA removes the need for explicit data transfers. Our goal is generalizing the process of GPU acceleration of a core computation, followed by further analysis performed on the same data on the host CPU.

Since UMA operates similarly to a paging mechanism [6], a UMA enabled version of a CUDA program has the potential to optimize certain memory transfer operations. If CUDA UMA were to transfer data in small blocks between devices, there is the potential that UMA may perform better than the standard data transfer methods while being simpler to implement for cases in which only a subset of the total data is used on the CPU or GPU. Thus, we analyze various access patterns that may alter the data transfers created by UMA.

To simulate these access patterns, we create 5 microbenchmarks categorized by the memory access behavior of the GPU kernel and the CPU component. With the exception of one microbenchmark, the GPU kernels are created simply to access and quickly change the data in memory. We do not implement complex algorithms because the focus is isolating the memory transfer behavior using UMA, not the computation performance. Our microbenchmarks are:

- **All GPU All CPU:** All of the elements in the matrix are accessed by the GPU during computation. Each element is simply incremented by one in place. Data is transferred to the CPU and all of the matrix elements are incremented by one again to simulate an exhaustive post processing.

- **All GPU All CPU SOR:** Identical to the previous method, with the exception that the kernel implements a single step of *Successive Over Relaxation (SOR)*, a common linear algebra algorithm. For each element of the matrix, a thread averages its neighbors with the element itself, and modifies the element value as a weighted sum of the average and its current value. This kernel requires significantly more memory accesses, more threads, and is performed out of place, requiring an input and output matrix. We implement this kernel to observe the effect of more complex memory access patterns on the runtime of UMA. After the GPU kernel completes, the CPU computes the difference between the output and input for each element, and compares the difference to a threshold.

- **Subset GPU All CPU:** On the host CPU, three elements are chosen at random and provided to the GPU kernel at launch. On the GPU, only the threads which pass over the matching elements perform the computation: a simple in place addition and multiplication to the element. Only three elements of the matrix are ever accessed by the GPU. After the GPU kernel returns, the CPU adds one to every element as an exhaustive post processing.

- **Subset GPU All CPU RAND:** Similar to the previous microbenchmark, with the exception that the elements accessed by the GPU are a random subset of the input matrix. This is achieved by performing the computation out of place, with an input and output matrix. An element of the output matrix is assigned if the input matrix value is below a threshold. The threshold is used to control the percentage of elements in the output that are touched, since A is given random values within a range.

- **All GPU Subset CPU:** This microbenchmark uses the same GPU kernel as the *All GPU All CPU* case. The CPU computation is changed so that only a subset of the output matrix is accessed at random on the CPU.

We run these microbenchmarks for various matrix dimensions ranging from 32 to 8192, and discuss the results in greater details in section V.

B. Rodinia Benchmarks

Rodinia is a benchmark suite designed for heterogeneous computing infrastructures with OpenMP, OpenCL and CUDA implementations [7]. It provides benchmarks that exhibit various types of parallelism, data-access patterns, and data-sharing characteristics while covering a diverse range of application domains. In this paper, we use this benchmark suite (CUDA version) to further investigate other common GPU and CPU patterns when utilizing UMA, as well as investigate the performance effect of UMA on a full benchmark suite alongside our microbenchmarks. We classify Rodinia benchmarks into three categories according to their memory access patterns and pick representative ones from each category as targets for analysis. The chosen benchmarks are modified to use UMA in CUDA 6 and are run to test the performance difference between the *UMA* and *non-UMA* version.

C. Experimental Setup

We run the experiments using a system that contains Intel Xeon E5530 4-core CPUs (2.4GHz), 24GB main memory, Kepler era K20c GPUs and uses the latest CUDA 6 to enable UMA. UMA is only enabled to run on Kepler era GPUs. The operating system is CentOS 6.4. We use runtime as the performance metric, to keep in line with the ultimate goal of GPU computations: speed. To get accurate timings from our experiments, we apply several different mechanisms for different cases. For our microbenchmarks and the *UMA* vs *non-UMA* testing of our Rodinia benchmark modifications, we put $gettimeofday$ timing functions inside the programs to extract the execution time for memory copies and kernel computations. For Rodinia benchmark classification, we use *nvprof*, a profiling software provided by Nvidia, to get the detailed timing of each function call (e.g. *cudaMalloc, cudaMemcpy* and kernel function). We do not use *nvprof* for timing any UMA computations as data transfers are hidden from profiling when using UMA. Since the runtime of experiments varies for different runs, we average the results of ten runs for all experiments.

IV. Rodinia Benchmark Characterization

The Rodinia benchmark suite contains 19 benchmarks in total. Due to compiling issues, we do not include *mummergpu* and *lavaMD* in our experiments. For *Speckle Reducing Anistropic Diffusion (SRAD)*, there are two versions in the benchmark suite released and these two versions expose quite different behaviors, so we include both of them in the experiments. Since these benchmarks only support up to CUDA 5, we need to modify them to use UMA for memory management. Thus, we first categorize the benchmarks according to their memory access patterns and then select representatives from each category to address the modifications.

We use the reference input size for each benchmark (provided by the developers) and *nvprof* to profile the execution period. As introduced in section III, this profiling software gives us the time spent in kernel computation, memory copy between device (GPU) and host (CPU) respectively. Figure 3 displays the profiling results (distribution of runtime to different tasks) for all 19 benchmarks we test, ranked in descending order based on the proportion of time spent on the kernel computation. In this figure, red blocks represent the percentage of time spent on kernel computation; yellow blocks stand for that of the *memset* function; white ones represent device to device (DtoD) copy, typically referring to data duplication in

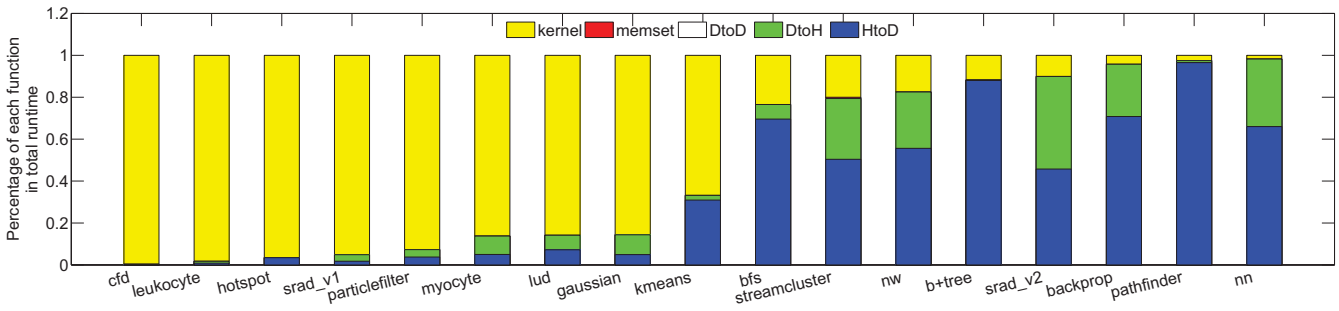

Fig. 3: GPU behavior characterization of Rodinia benchmarks.

TABLE I: Categorization of Rodinia benchmarks.

Types		Benchmarks
kernel		leukocyte, hotspot, srad_v1, myocyte, cfd, particlefilter, **lud**, gaussian, kmeans
memory	HtoD	bfs, b+tree, **pathfinder**
	Balanced	streamcluster, nw, srad_v2, backprop, **nn**

GPU memory when only one GPU is being used; green blocks indicate device to host (DtoH) data transfers; and blue blocks show host to device (HtoD) data transfers. Since *memset* and DtoD always take less than 1% of execution time for the benchmarks, they do not stand out in this figure.

From this figure, we see that the benchmarks express significant variances on runtime composition. For example, *cfd, luekocyte* etc. spend more than 90% of execution time on kernel computation while for benchmarks such as *pathfinder* and *nn*, data transfer time predominates over the kernel computation time. Therefore, we categorize the first type as kernel-intensive benchmarks and the second type as memory-intensive benchmarks. Additionally, it is clear to see that among memory-intensive ones, *b+tree* and *pathfinder* have far more HtoD transfers than DtoH transfers while the rest ones show more comparable transfers between HtoD and DtoH. Thus we further categorize memory-intensive benchmarks into HtoD-memory-intensive and Balanced-memory-intensive benchmarks. The benchmark categorization is shown in Table I. Due to such variation in the Rodinia benchmark suite, we pick one benchmark from each category and modify them for the UMA test. We pick *lud*, *pathfinder* and *nn* as the representative kernel-intensive, HtoD-memory-intensive and Balanced-memory-intensive benchmarks, respectively.

To modify these benchmarks to use UMA, we remove all references to device pointers and memory transfers, and convert all host pointers containing working data to CUDA managed pointers. Of note is that these conversions are not always trivial, as some algorithms were designed to work with multiple pointers for output and input, or merged data into single pointers. Furthermore, the number of lines saved between the two versions of code (UMA vs non-UMA) is in the single digits for these benchmarks. While the programming model is easier to understand when using UMA, as memory is treated as a single space, it is not a significant amount of code saved, nor does it necessarily simplify the design of the program.

V. Experimental Results

A. Results for GPU Microbenchmarks

For a subset of our microbenchmarks, *Subset GPU All CPU*, *Subset GPU All CPU RAND*, and *All GPU Subset CPU*, our expectation is that the *UMA* version would perform better across all sizes. This is anticipated if UMA operates similarly to a paging mechanism, because these benchmarks operate on only subsets of the input data and would have a distinct advantage in *UMA*, requiring only a subset of data to be transferred. Interestingly, the experimental results do not support this theory, but rather, the performance of the *UMA* version is consistently worse than *non-UMA* version.

Figure 4 shows the performance of the microbenchmarks for a varying matrix dimension normalized to the runtime of the *non-UMA* version. The red line is the performance of the *UMA* version and the black dashed line shows the performance of the *non-UMA* version. The performance is measured as the time between the initial transfer to the GPU up to the finish of the CPU computation that follows kernel execution. The CPU portion of the computation is included in the timing in order to ensure that all UMA transfer latencies are also included. From this figure, it is clear that for almost all cases, the performance of the *UMA* version is worse than the performance of the *non-UMA* version. Furthermore, the difference in performance is not static, but changes as the data set size changes.

Although each microbenchmark has different performance scaling between the *UMA* and *non-UMA* versions, there are evident trends in the data that reveal some details on the UMA mechanism. First, the performance of UMA appears to be split up into two phases based on the dimension of the matrix. As soon as the matrix dimension hits 1024, there is an immediate performance loss. Thereafter, the performance stabilizes as a constant factor differences compared to the *non-UMA* version, up to sizes of 8192. One key observation we can glean from these experiments is that 1024 corresponds to the paging value UMA uses: 4 KB. Prior to that size, the performance is more variable due to shifting access patterns for small matrices where a page of data may span multiple rows of the input matrix. This also implies that controlling the page size to match the problem size can improve UMA performance in general.

A further observation is that the benchmarks that are expected to perform well, marked c to e in Figure 4, at times perform worse than *All GPU All CPU* and *All GPU All CPU*

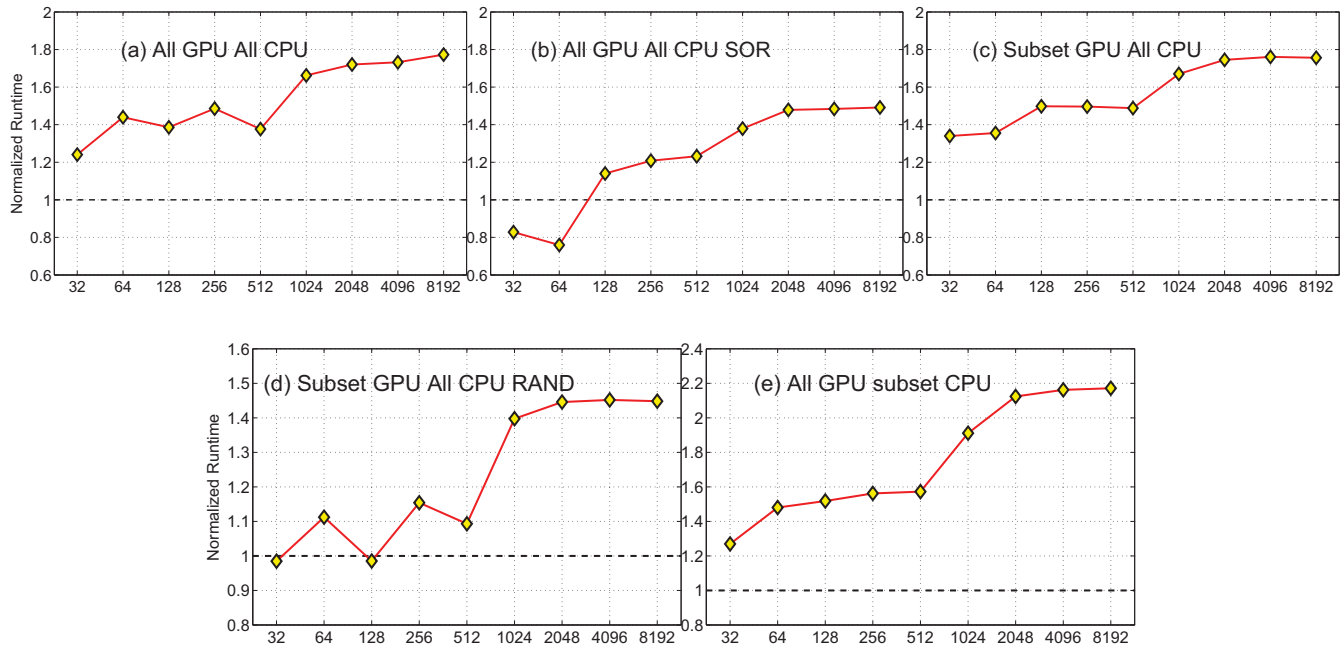

Fig. 4: *UMA* vs *non-UMA* performance for the GPU microbenchmarks. The *X axis* represents the dimension of the input matrix for the microbenchmarks.

SOR. This is most evident with the *All GPU Subset CPU* case, which at its worse, is 2.2x slower than the *non-UMA* version. For high performance computing applications, a case like this would be unacceptable. The *Subset GPU All CPU RAND* case displays some favorable results for smaller matrix sizes up to a dimension of 512, and has the best scaling for large sizes as well, but it still performs worse than the *non-UMA* version.

Unusually, the *All GPU All CPU SOR* case performs better than the *non-UMA* version for small matrix sizes. This may be because UMA attempts to place data in a position in memory to best improve performance and maintain consistency [9], and for SOR which requires multiple memory accesses, this is highly advantageous. In particular, these small sizes are close to the UMA page size, and may have more efficient transfers compared to the *non-UMA* version.

Overall, these results demonstrate that the performance hit taken by these microbenchmarks when using UMA is not trivial at large sizes, and weigh heavily against the positives of a simplified memory model.

B. UMA Tests for Rodinia Benchmarks

After implementing UMA in representative Rodinia benchmarks *LUD*, *Nearest Neighbor (NN)* and *Pathfinder*, we perform similar timing as with our microbenchmarks, but using varying parameters as is appropriate for each of the benchmarks. Figure 5 presents the experimental results for these benchmarks, normalized to the case without using UMA. We find interesting trends that demonstrate that UMA may indeed have a use case in which it performs better than *non-UMA* CUDA.

For *Pathfinder* and *LUD*, with smaller input sizes, the *UMA* version performs comparably, or better than the *non-UMA* version. These results differ from those seen with the microbenchmarks. In particular, these two benchmarks represent a special case which is not included in the microbenchmarks: subset data access on the GPU with multiple consecutive kernels before the need for data transfers back to the host.

In this case, a subset of the data is moved to the GPU via UMA and is repeatedly accessed across multiple kernels before other data is ever referenced by the GPU. CUDA UMA prioritizes memory coherence and locality when transferring data to GPU, and the location of data is invisible to the program [9]. Because of this, if a subset of data is repeatedly being operated on, our results demonstrate that the location of the data via UMA may provide a performance benefit over a *non-UMA* version of memory transfer. Along these lines, *LUD*'s performance appears to decrease as the input matrix dimension approaches and passes the page size we mentioned in the previous section. The memory locality benefit attained from UMA thus seems linked to the data dimensions, how the data is accessed on GPU, and how many times the data is accessed before data is required to be returned to the CPU.

LUD is a kernel intensive application, and the performance degradation due to the migration of data when using UMA for large data sizes is larger than that of *Pathfinder*, a memory intensive application. This indicates that kernel intensive applications are more adversely affected when memory overhead becomes larger.

The performance of *NN*, however, aligns much more closely to results from the microbenchmarks. *NN* is a balanced memory intensive application which operates on all elements of the input data, but only on a subset of the output. Thus, for the different input sizes, the *UMA* version is simply a scale factor worse than the *non-UMA* version, similarly to our results from the microbenchmarks.

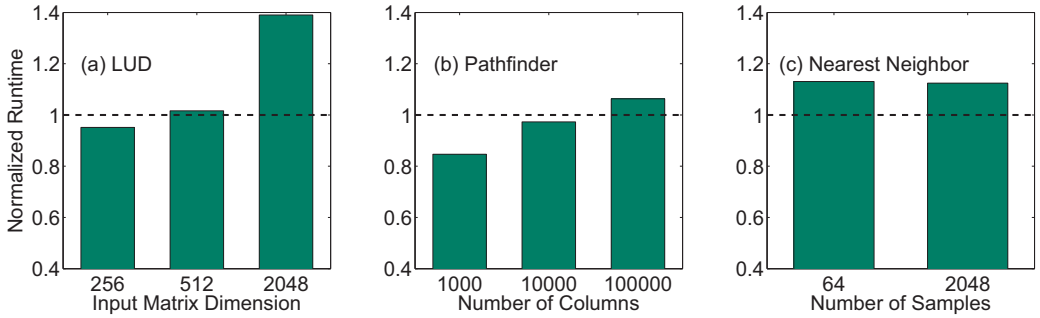

Fig. 5: Performance results of the *UMA* version of the Rodinia benchmarks normalized to the *non-UMA* version.

C. Implications for UMA

These results demonstrate that the performance of UMA varies greatly based on program design, kernel intensity, and memory access patterns. In particular, we demonstrate that there are indeed cases where a *UMA* version can perform better than a *non-UMA* implementation. One particular note is that in both our microbenchmarks and the Rodinia benchmarks, the difference in code complexity between the *UMA* and *non-UMA* version is little, with no more than 10 lines of code changing across versions. For the majority of GPU applications which operate on arrays of data, the introduction of UMA does not provide a large simplification. However, for more complex data structures where porting to the GPU may prove difficult, UMA provides a simplification, albeit at a potentially large performance cost. In our experimentation, we see an up to 2.2x decrease in performance.

In order to improve UMA performance, we find that the application must use kernels which operate on subsets of the output data at a time. In this way, the paging mechanism provided by UMA provides the most benefit. Beyond this, UMA should only be used when the data structures of a CPU program are particularly difficult to arrange on the GPU, and further optimizations are not necessary. Once a program is designed for UMA, the use of streams for data overlap, as well as various other data transfer optimizations, is made more difficult and less obvious.

UMA can become a more performance oriented tool if it is extended to support more features and controls. Dynamically specifying the paging size between CPU and GPU memory would be critical, as the data transfer can then be tuned to match memory access behavior for a given application. As is seen in Figure 4, once the input hits the page size, performance decreases notably. Along with this, if CUDA were more transparent in the data transfer mechanism using UMA, the program design could be better optimized to match data transfer patterns. Finally, if UMA allowed kernels to access data on demand during transfer, similarly to the work done in [10], significant performance benefits over a *non-UMA* version would be attained.

VI. Conclusions

In this work, we present an investigation of unified memory access performance in the latest CUDA version. We have designed GPU microbenchmarks to probe the performance features of UMA. Based on the results we collect from a system with Kepler GPUs, we have demonstrated that the performance of UMA varies significantly based on the memory access patterns. We also show that there are cases in which *UMA* versions can perform better than a *non-UMA* implementation. However, in its current form, UMA has limited utility due to its high overhead and marginal improvement in code complexity. Further work on UMA should include greater transparency in the paging mechanism, greater control of the internal mechanism, as well as further analytical tools for optimizing UMA behaviors.

References

[1] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, March 2008.

[2] B. Sukhwani and M. C. Herbordt, "GPU acceleration of a production molecular docking code," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2, 2009, pp. 19–27.

[3] J. Michalakes and M. Vachharajani, "GPU acceleration of numerical weather prediction," *Parallel Processing Letters*, vol. 18, no. 04, pp. 531–548, 2008.

[4] S.-C. Wei and B. Huang, "GPU acceleration of predictive partitioned vector quantization for ultraspectral sounder data compression," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 4, no. 3, pp. 677–682, Sept 2011.

[5] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney, and A. Shringarpure, "On the limits of GPU acceleration," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism*, 2010, pp. 13–13.

[6] Nvidia, "CUDA," https://developer.nvidia.com/cuda-toolkit, 2014, accessed: 2014-4-29.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of IEEE International Symposium on Workload Characterization*, Oct 2009, pp. 44–54.

[8] Nvidia, "Nvidia kepler GK110 architecture," http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2012, accessed: 2014-3-31.

[9] ——, "Unified memory in CUDA 6," https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/, 2014, accessed: 2014-4-29.

[10] D. Lustig and M. Martonosi, "Reducing GPU offload latency via fine-grained CPU-GPU synchronization," in *Proceedings of IEEE International Symposium on High Performance Computer Architecture*, 2013, pp. 354–365.