Evaluating Unified Memory Performance in HIP

Zheming Jin
Oak Ridge National Laboratory
jinz@ornl.gov

Jeffrey S. Vetter
Oak Ridge National Laboratory
vetter@computer.org

2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) | 978-1-6654-9747-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/IPDPSW55747.2022.00096

***Abstract*— Heterogeneous unified memory management between a CPU and a GPU is a major challenge in GPU computing. Recently, unified memory (UM) has been supported by software and hardware components on AMD computing platforms. The support could simplify the complexities of memory management. In this paper, we attempt to have a better understanding of UM by evaluating the performance of UM programs on an AMD MI100 GPU. More specifically, we evaluate data migration using UM against other data transfer techniques for the overall performance of an application, assess the impacts of three commonly used optimization techniques on the kernel execution time of a vector add sample, and compare the performance and productivity of selected benchmarks with and without UM. The performance overhead associated with UM is not trivial, but it can improve programming productivity by reducing lines of code for scientific applications. We aim to present early results and feedback on the UM performance to the vendor.**

Keywords—Unified memory, Performance evaluation, GPU

I. Introduction

Unified memory (UM) allows modern graphics processing units (GPUs) and central processing units (CPUs) to share heterogeneous unified memory address space by GPU runtime support [1]. In a conventional heterogeneous programming model, developers need to explicitly copy data between a CPU and a GPU before and after executing a task on a GPU [2, 3, 4]. Compared to the explicit copy-then-execute approach, the runtime, operating system, and underlying hardware work together to automatically migrate data to destinations in UM, relieving a developer from managing data migration between a host and a device explicitly for complex applications.

However, conventional programming models are still widely used because UM sacrifices performance for flexibility [5, 6, 7, 8, 9]. UM incurs performance overhead because a GPU runtime must trace memory accesses, determine the granularity of data migration, and handle page faults, etc. To mitigate performance issues of UM, the GPU computing communities have mainly focused on developing optimization techniques with the CUDA programming model on NVIDIA GPUs [10, 11, 12, 13].

HIP is a C++ runtime application programming interface (API) and kernel language that allows developers to write portable applications for AMD GPUs from a single source. HIP now supports and automatically manages heterogeneous memory management (HMM) allocation on AMD GPUs [14]. Previous studies evaluated the performance of UM applications and benchmarks on NVIDIA GPUs [5-9]. In this work, we modify existing benchmarks for UM support with HIP and evaluate their performance on a recent AMD discrete GPU. Rather than proposing new optimization techniques for UM, our experimental results aim to provide timely feedback on the performance of UM benchmarks in HIP on an AMD GPU and invite discussions about performance optimization techniques for the software stack and hardware devices on AMD computing platforms.

More specifically, we evaluate the data migration methods, including UM, pageable memory, pinned memory, and zero-copy, and show their impacts upon the performance of the matrix multiplication and vector addition. Then, we evaluate the effectiveness of the UM optimization techniques in reducing the kernel execution time with a performance profiler. Finally, we evaluate the performance of eight benchmarks in the high-performance computing and machine learning domains with and without UM. Despite the performance overhead incurred by UM, we argue that UM can improve programming productivity by reducing lines of code of scientific applications.

We have described the motivation and scope for our work. The rest of the paper is organized as follows. Section II introduces the UM support, the GPU execution model, and the AMD GPU architecture. Section III describes our performance evaluation of programs and benchmarks with UM support on the target GPU. Section IV summarizes related work, and Section V concludes the paper.

II. Background

A. Unified memory support in ROCm

AMD ROCm is a software stack composed of development tools, libraries, compiler toolchains, programming models, and drivers/runtime [15]. ROCm has been enhancing HIP with UM support. UM maps and migrates data seamlessly without developers' explicitly copying data between different memory allocations. To enable the feature, the "hipMallocManaged()" function is needed in a HIP program to allocate memory automatically managed by HIP. "hipMemAdvise()", "hipMemPrefetchAsync()" and related application programming interfaces (APIs) might aid the runtime with memory usage hints for performance optimization.

UM is only supported on recent Linux kernels. Before making the managed memory API call in a HIP program, it is recommended to either query the feature of a device for UM support or look for the HMM-specific message in the kernel log.

UM only works on recent AMD GPUs, including Vega10 and MI100. Older GPUs such as Fiji and Polaris are not supported. There are two flavors of the support: XNACK-enabled and XNACK-disabled. In the XNACK-enabled mode,

978-1-6654-9747-3/22/$31.00 ©2022 IEEE
DOI 10.1109/IPDPSW55747.2022.00096

a GPU can handle retry of a memory access after page-faults, which enables mapping and migrating data on demand, as well as memory overcommitment. In the XNACK-disabled mode, all memory must be resident and mapped in GPU page tables when the GPU is executing application code. The XNACK-enabled mode only has experimental support. Not all the math libraries included in ROCm support the XNACK-enabled mode on current hardware. A mode can be chosen at boot-time, and the default is XNACK-disabled. Due to the uncertainties of the XNACK-enabled mode, our evaluation is limited to the XNACK-disabled mode. We would like to investigate the XNACK-enabled mode in our future work.

B. GPU execution model and architecture

A GPU program generally consists of two integral parts: a host program and a device program. A host program runs on a host processor while a device program runs on a GPU. A kernel, which usually represents a computationally intensive task, is executed by multiple work-items (WIs) and these work-items are grouped into work-groups (WGs). The number of WGs and the number of WIs per WG are specified programmatically. These WGs are executed by an array of compute units (CUs) that form the core of a GPU. An AMD GPU has multiple CUs and each CU is a processor with program counters, scalar registers, vector registers, arithmetic logical units (ALU), etc. There is a level-one (L1) cache for each CU and all CUs share a L2 cache via an interconnect network.

The 7-nm MI100 is a discrete GPU with 120 CUs. As shown in the specification [16], each CU contains a scalar ALU, 104 scalar registers, and four single-instruction-multiple-data (SIMD) units. Each SIMD unit has 64 vector registers and 16 ALUs. The maximum clock frequency of the GPU is 1502 MHz and the maximum number of WIs in a WG is 1024. The sizes of the L1 and L2 caches are 16 KB and 8 MB, respectively. The total number of stream processors are 7680 (120 × 4 × 16).

III. Evaluation

We will describe our experiments on the MI100 GPU in the following order. Firstly, transparent data migration with UM is compared against other data transfer techniques for the overall performance of the matrix multiply and vector add applications. Secondly, we examine the impacts of three commonly used optimization techniques on the kernel execution time of a vector add example. Lastly, we compare the performance and programming productivity of selected benchmarks with and without UM.

A. Impacts of data migration techniques on the application performance

As mentioned before, explicit data migration is commonly used in many GPU applications. The UM support in CUDA or HIP was introduced after explicit data migration had been widely adopted by the GPU research community. In addition, UM's performance issues may discourage people from porting a program written with the explicit copy-then-execute approach. Despite these factors, we will evaluate the impacts of data migration approaches [17, 18] upon the performance of an application. Starting with the NVIDIA UM performance sample that contains a matrix multiplication kernel [19], we port the sample to HIP with HIPIFY [20]. Then, we add another sample with a vector add kernel. We build all HIP programs in our work with ROCm version 4.5.2.

Figures 1 and 2 show the overall performance reported as bandwidth (MB/s) across various memory sizes of the matrix multiplication and vector addition applications, respectively. The overall performance is derived from the sum of data transfer time between a host and a device, device execution time, and host access time. "UM-hint" and "UM" indicate unified memory with and without memory usage hints, respectively. "ZeroCopy" uses zero-copy buffers for data migration. "PageableCopy" copies data from pageable host memory to device memory whereas "PageLockedCopy" transfers data from page-locked

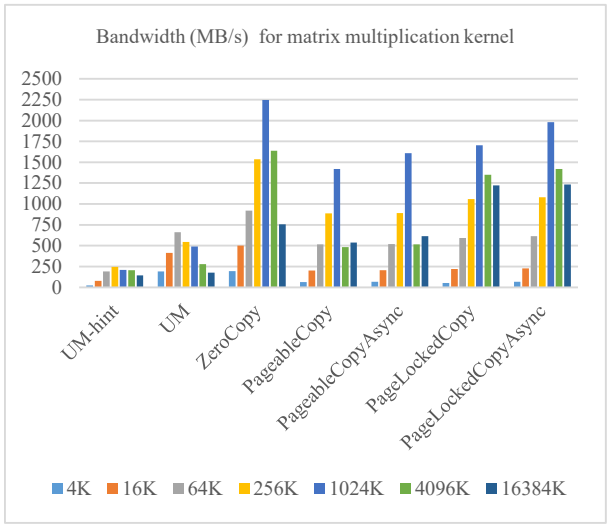

Fig 1. Overall performance of the matrix multiply application with respect to data migration methods on the MI100 GPU. The memory size ranges from 4 KB to 16384 KB.

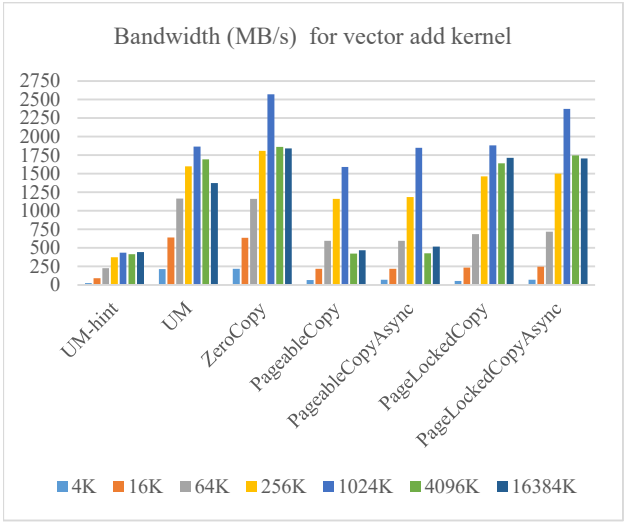

Fig 2. Overall performance of the vector add application with respect to data migration methods on the MI100 GPU. The memory size ranges from 4 KB to 16384 KB.

host memory to device memory. The suffix "Async" indicates that data transfers are performed asynchronously on a host. The bandwidth of "UM-hint" is approximately 1.2X to 7.5X slower than that of "UM" across the matrix sizes for the matrix multiplication. The bandwidth of "UM-hint" is also approximately 3.0X to 8.5X slower than that of "UM" across the vector sizes for the vector add. On the other hand, when the memory size ranges from 4 KB to 64 KB, the performance of "UM" is approximately 1.07X to 2.8X higher than that of "PageLockedCopyAsync" for the matrix multiply. It is also approximately 1.6X to 3.1X higher than that of "PageLockedCopyAsync" for the vector add. The performance of "UM" gradually decreases from the peak performance at the memory size of 64 KB for the matrix multiplication. For the vector add, the peak performance of "UM" is at 1024 KB. The experimental results show that the performance of the applications using UM is closely related to data transfer size and memory accesses of a kernel. Compared to "UM", prefetching memory as a memory usage hint leads to significant data transfers between the host and device.

B. Impacts of UM optimizations on the kernel performance

Managed memory may not be physically allocated when calling the API function to allocate memory that will be automatically managed by HIP; it may only be populated on access or prefetching. At the runtime level, pages and page table entries may not be created until they are accessed by a device. Pages could migrate to any processor's memory at any time as the runtime may not automatically copy all pages needed by a kernel to a GPU before running the kernel. While the kernel launch may not incur any migration overhead, accessing any absent pages causes a GPU to stall the execution of the accessing threads, and wait for the migrating pages to reach the device before resuming the threads.

Empirical studies showed that three optimization techniques for UM are effective in improving the kernel performance on NVIDIA Pascal and later GPUs [21]:

1. Move data initialization to a GPU into another kernel.
2. Run a kernel many times and look at the average and minimum execution time.
3. Prefetch data to GPU memory before running a kernel.

We use a vector add kernel (i.e., y = x + y) to measure the impacts of the three optimization techniques on the kernel execution time on the MI100 GPU. Without UM, the vector arrays are initialized on the host, copied to the GPU for execution, and finally the results are copied back to verify their correctness. We profile the programs with the ROCm profiler, a command-line interface for AMD GPU profiling libraries. We run the kernel with pageable memory copy (non-UM) and unified memory (UM) for 100 iterations.

Figure 3 shows that the decrease of the kernel execution time ranges from approximately 1.1X to 2.8X with respect to the vector length for the three optimization techniques. However, the execution time is still approximately 1.4X to 74.8X longer than that of the kernel that takes the copy-then-execute approach. The worst kernel performance occurs when the vector length is 4096K. The results indicate that the extra cost of data

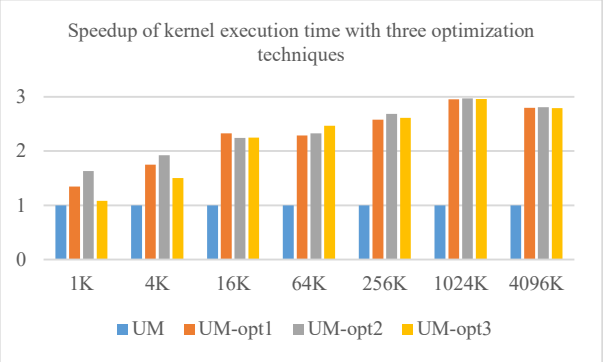

Fig 3. Performance speedup achieved by the three optimization methods on the MI100 GPU. The vector length ranges from from 1K to 4096K. The kernel execution time of the baseline (UM) is normalized to 1.

TABLE I. PERFORMANCE PROFILING METRICS

Metrics	***Descriptions***
FetchSize	The total kilobytes fetched from the global memory. This is measured with all extra fetches and any cache or memory effects are considered.
WriteSize	The total kilobytes written to the global memory. This is measured with all extra fetches and any cache or memory effects are considered.
MemUnit Stalled	The percentage of GPU time the memory unit is stalled.
L2Cache Hit	The percentage of fetch, write, atomic, and other instructions that hit the data in L2 cache.

migration, which is included in the kernel execution time, can only be partially mitigated with these optimization techniques.

To have a better understanding of the performance gaps, we profile the kernels with the collected metrics and their descriptions listed in Table I.

Figure 4 shows the fetch and write sizes over 100 iterations for the two data migration methods when the vector length is 4096K. The fetch sizes are on average 2.04X and 2.34X larger than the write sizes for UM and non-UM, respectively. While the write size is fixed at 16384 KB for UM, it fluctuates slightly between 14037 KB and 14068 KB for non-UM. This implies

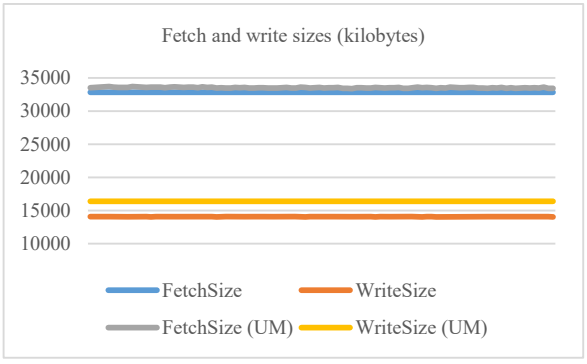

Fig 4. Fetch and write sizes over 100 iterations of kernel execution for a vector length of 4096K

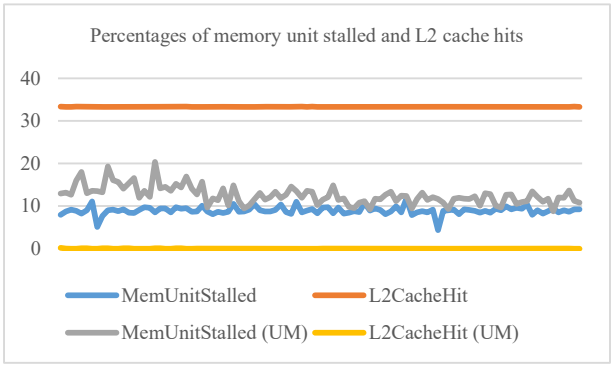

Fig 5. Memory stall rates and L2 cache hits rate over 100 iterations of kernel execution for a vector length of 4096K

that global memory writes are partly cached. The fetch size for UM is on average 2.1% larger than that for non-UM; the fetch sizes are not constant for both UM and non-UM over 100 iterations. The fetch sizes are on average 0.23% and 2.3% higher than the total sizes of the two vectors in bytes for non-UM and UM, respectively. The results suggest that UM incurs more data migration between a host and a device.

Figure 5 shows the percentages of memory unit stalled and L2 cache hits. The cache hit rate levels off at around 33% for non-UM, but it is almost zero for UM. Hence, the L2 cache is hardly utilized to improve data locality for UM. The stall rate for UM fluctuates between 10% and 20% while the rate for non-UM fluctuates between 8% and 10%. The stall rate for UM is on average 3.6% higher than that for non-UM due to the larger number of memory fetches and writes.

Figure 6 shows the percentages of memory unit stalled with respect to the vector length when the kernel is executed for 100 iterations in the UM mode. When the vector length ranges from 1K to 64K, the stall rate is almost zero over 100 iterations. Further increasing the vector length leads to fluctuating rates

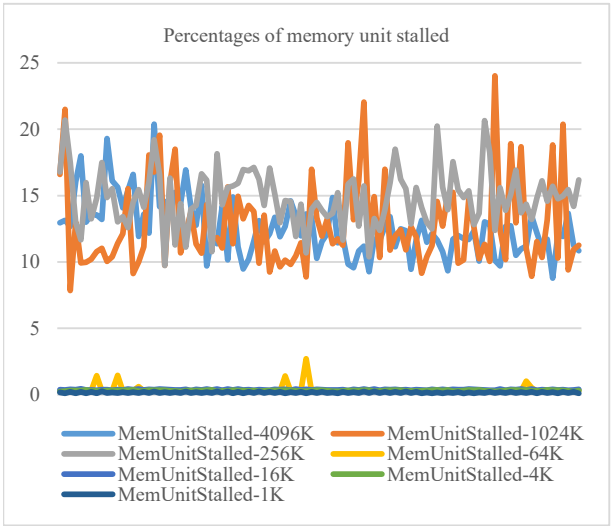

Fig 6. Memory stall rates with respect to the vector length when the kernel is executed for 100 iterations in the UM mode

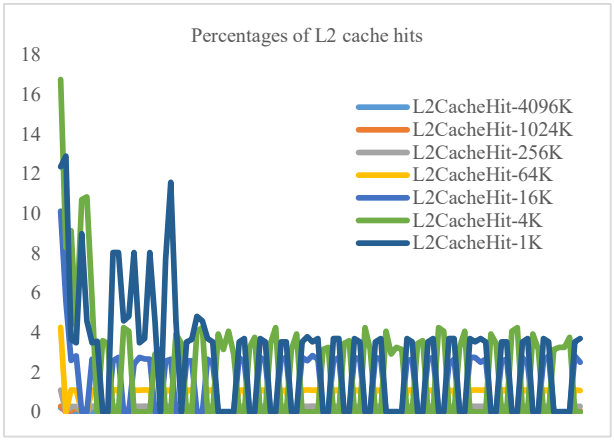

Fig 7. L2 cache hit rates with respect to the vector length when the kernel is executed for 100 iterations in the UM mode

mostly between 10% and 25%. The result shows that the stall rate is highly sensitive to the increase of memory size in UM.

Figure 7 shows the percentages of L2 cache hits with respect to the vector length when the kernel is executed for 100 iterations in the UM mode. After the hit rates peak in the beginning for the shorter vector lengths, they appear fluctuating between 0% to 4%. For larger vector lengths, the hit rates are less than 1%. The result shows that the hit rates are not constant across the iteration range. They are less sensitive to the increase of memory size in UM.

C. *Performance and productivity of UM benchmarks*

We choose eight benchmarks from different domains for evaluating the UM performance. “backprop”, “hotspot”, and “pathfinder” are grid applications in the Rodinia benchmark suite [22]. “winograd” and “mnist” are machine learning benchmarks for Winograd convolution [23] and convolution neural network [24], respectively. “deredundancy” is an award-winning bioinformatics application for gene de-redundancy [25]. “s3d” is a scientific application for combustion simulation from the SHOC benchmark suite [26]. “gpp” is a proxy application for the generalized plasmon-pole model from BerkeleyGW, a many-body perturbation theory code [27]. We write these benchmarks in HIP and convert the copy-then-execute APIs to the UM APIs. Specifically, the host and device pointers involved in data transfers between the two are replaced with the “hipMallocManaged()” function calls. All memory copy APIs (i.e., “hipMemcpy()”) explicitly called in a program are not needed in the UM model. This may involve rewriting codes around the API calls in some benchmarks to implement the equivalent functionalities. Hence, certain data structure used on the host side may be modified for the UM version. For example, a 2D host array may be flattened into a 1D array before it is allocated with UM. This is because the kernel still expects a pointer to a 1D array.

To evaluate the performance of benchmarks with or without UM fairly, we propose to measure the time of executing both host and device codes rather than the time of executing device codes alone. Hence, for the HIP benchmarks without UM, our

TABLE II. EXECUTION TIME IN SECONDS OF THE BENCHMARKS

Benchmark	*Non-UM (s)*	*UM (s)*	*Slowdown (%)*
backprop	0.68	0.73	7.3
deredundancy	100.3	103.6	3.3
gpp	267	N/A	N/A
hotspot	1.09	1.21	11
mnist	51	64	25.4
pathfinder	0.81	0.93	14.8
s3d	0.118	0.128	8.4
winograd	1.9	5.9	210

timing measurement includes allocation and initialization of data structures in host and device memory spaces, data transfers from a host to a device, execution of kernel(s) on a device, data transfers from a device to a host, and deallocation of data structures in host and device memory spaces. For the UM benchmarks, our timing measurement includes allocation and initialization of data structures in UM, execution of kernel(s), and deallocation of data structures in UM.

Table II lists the execution time in seconds of the benchmarks with or without UM and the comparison of their execution time. The result of the "gpp" benchmark with UM is not available because it causes the benchmark to hang. Among the other benchmarks, the performance of "winograd" is worst with UM. We try to have a better understanding of the significant drop in performance. Looking into "winograd", we find that the performance bottleneck lies in the loop body where Winograd convolutions are performed. The amount of CPU and GPU workloads are computed in the beginning of each loop iteration. For the first seven loop iterations, the workload size is zero on a CPU. Then, both host and device will compute their parts of Winograd convolution and write their results to a buffer allocated with UM concurrently.

Figure 8 shows the fetch and write sizes over the loop iteration range for the Winograd benchmark with and without UM. Because the CPU workload increases from zero to the maximum over the iteration range, the fetch and write sizes decrease generally over the iteration range on the GPU. However, there are three spikes in the fetch and write sizes with UM. The first spike occurs when the CPU and GPU start co-executing the Winograd convolution at the 8th iteration, causing the runtime to synchronize memory between the host and device with excessive amount of data migration. As the CPU workload continues to increase, the size of the output from the CPU convolution written to the buffer in UM reaches to a point (i.e., the second spike) where it triggers significant data migration between the host and device again. The last spike, the magnitude of which is significantly lower than the previous spikes, increases the stall rate of the memory unit from around 50% to 77%. Additional performance profiling metrics, such as memory thrashes, addresses of CPU and GPU page faults, and the correlation of memory events with application code, may be provided by the ROCm profiler to fully understand the causes of the spikes.

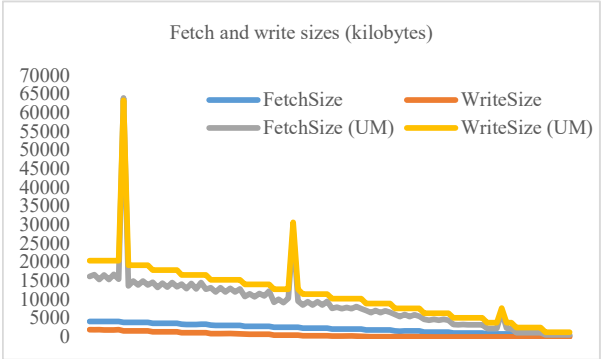

Fig 8. Fetch and write sizes of the "winograd" kernel over the loop iteration range

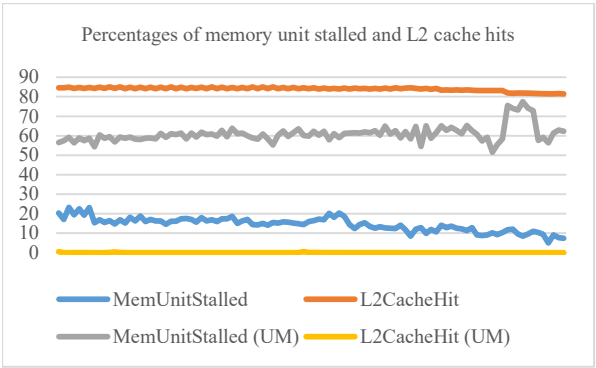

Fig 9. Memory stall rates and L2 cache hit rates of the "winograd" kernel over the loop ieration range

Figure 9 shows the percentages of memory unit stalled and L2 cache hits for the Winograd benchmark with and without UM. With the increasing amount of CPU workload, the stall rate decreases slowly from around 20% to around 7% for the non-UM implementation. However, the stall rate ranges from 50% to 75% for the UM implementation. The hit rate ranges from 81% to 85% without UM, but it is almost zero with UM. The results show that UM is unable to utilize L2 cache for performance improvement.

We argue that lines of code (LOC), which measure the size of a computer program by counting the number of lines in the text of the program's source code, can estimate programming

TABLE III. PROGRAMMING STEPS WITH AND WITHOUT UM

Step	*Non-UM*	*UM*
1	Allocate host memory	Allocate unified memory
2	Allocate device memory	
3	Initialize host memory	Initialize unified memory
4	Copy data from host memory to device memory	Launch kernel(s)
5	Launch kernel(s)	
6	Copy data from device memory to host memory	
7	Deallocate device memory	Deallocate unified memory
8	Deallocate host memory	

TABLE IV. LINES OF CODE OF THE HOST PROGRAMS IN THE BENCHMARKS

Benchmark	*Non-UM (LOC)*	*UM (LOC)*	*Reduction (LOC)*	*File of interest*
backprop	100	93	7	main.cu
deredundancy	239	221	18	main.cu
gpp	204	156	48	main.cu
hotspot	147	136	11	hotspot.cu
mnist	142	139	3	main.cu
pathfinder	186	180	6	main.cu
s3d	194	171	23	S3D.cu
winograd	137	127	10	main.cu

productivity. UM could achieve better programming productivity or simplify the complexities of a program by reducing the number of logical steps needed for writing a program for heterogeneous computing.

As shown in Table III, UM eliminates the needs of allocating host and device memory separately, and explicit data transfers between host and device memory. Hence, the number of logical steps decrease from 8 to 4. When there are many host and device memory allocations as well as data transfers in a complex application, UM could simplify the complexity more effectively.

To evaluate the productivity of UM, we compare LOC of the host programs with or without UM and list the names of the programs in these benchmarks. It should be noted that the amount of reduction in LOC with UM is closely related to how host and device codes interact in each application.

Table IV shows that UM can reduce LOC by 3 to 48 depending on the benchmarks. Although the decrease of three lines is trivial for a program, removing nearly 50 lines could improve programming productivity and program readability.

IV. RELATED WORK

In [5], the authors evaluate the performance impact of adopting UM for solving sparse systems of linear equation with CUDA 6.0. Their experience is using UM never resulted in more than a 25% slowdown over hand-tuned code. The speedup achieved by UM ranges from 0.77 to 1.39. Looking closely at the problem sizes and execution time, we observe that the UM implementation is generally faster than the non-UM implementation for small problem sizes. On the other hand, they report that UM can simplify the code structure because the CPU and GPU can access data stored in managed memory at different computing stages of the algorithm. Additionally, UM offers ease of programming for object-oriented programming with sharing of objects between host and device code.

In [7], the authors find that for many applications and memory access patterns, the performance overheads associated with UM are significant at large problem sizes on the NVIDIA K40 and Jetson TK1 GPUs, while the simplifications to the programming model restrict flexibility for adding future optimizations. For their microbenchmarks and the Rodinia benchmarks, the difference in code complexity between the UM and non-UM version is little, with no more than 10 lines of code changing across versions. They conclude that for many GPU applications which operate on arrays of data, the introduction of UM does not provide a large simplification. While their performance trend is consistent with our evaluation, we argue that UM can reduce code complexity for scientific applications that involve a lot of host and device interaction.

In [8], the authors found that the performance improvement with UM is 1% to 3% with the selected tests for heat distribution, but most applications saw performance drops or time close to those using the standard (non-UM) APIs. Their experiments showed that the performance of UM appears to be worse when data transfer time relative to kernel computations on a GPU is substantial and that more data compared to the standard version is transferred in the UM version. However, UM is a mechanism that allows people to quickly write programs like standard programs written for a CPU. Their results are consistent with the increasing fetch and write sizes for larger problem sizes in our experiment. However, porting a program written with standard APIs to a program with UM requires an understanding of how host and device code work in an application.

In [9], the authors evaluate the performance of a set of UM benchmarks over Intel-Volta/PascalPCIe based systems and the Power9-Volta-NVLink based system. They find that the effectiveness of memory usage hints and prefetching depends on the platform, but the overhead of handling page faulting is more significant compared to the execution time of the benchmarks with explicit data transfers. In their work, they also evaluate the performance of these optimization techniques when GPU memory is oversubscribed. We argue that improving the performance of in-memory execution in UM should be addressed first. They also mention the effort in operating system of providing mechanisms to mirror a CPU page table on a GPU and integrate device memory pages in a system page table. This is important because a recent Linux kernel is required for the UM support on an AMD GPU.

In [28], the authors present 32 open-source UM benchmarks in CUDA and evaluate their performance on an NVIDIA Pascal GPU. They find that across the benchmarks the performance of the UM benchmarks is on average 34.2% slower compared with the benchmarks without UM due to the cost of page fault handling. They apply the three optimization techniques summarized in our paper to the UM benchmarks that cause the highest slowdown. These techniques are effective in improving the performance of the benchmarks in their study. Their results are consistent with our findings using a vector add kernel. However, we would like to add more benchmarks and applications with UM in our future work.

Besides the extensive performance evaluation of benchmarks and applications with CUDA's UM support on NVIDIA GPUs, in-depth analyses and optimizations of the implementations of UM through GPGPU simulators are described in [10, 11, 12, 13]. Our work aims to provide early results and feedback to the vendor from the aspects of application development and performance evaluation.

V. CONCLUSION

The performance overheads associated with UM on an AMD GPU are significant based on our experimental results. Given the large amount of work on performance evaluation and optimization of UM on NVIDIA Pascal and later GPUs, such overheads may be expected for AMD GPUs. While the feature

has only been supported with recent AMD software and hardware components, there are optimization spaces of reducing fetch and write sizes for large memory sizes, improving L2 cache hit rates and decreasing memory unit stalls, and adding more performance metrics related to UM and GPU architectures in the ROCm profiler. On the other hand, our benchmark results show that UM can improve programming productivity effectively for certain scientific applications by reducing lines of code. We expect that the performance gap between UM and non-UM benchmarks will gradually shrink with the maturing software and hardware components.

ACKNOWLEDGMENT

We sincerely appreciate the reviewers for their comments and suggestions. The author would like to acknowledge people at the Advanced Computing Systems Research section in Oak Ridge National Laboratory for their generous support. This research used resources of the Experimental Computing Lab (ExCL). This research was supported by the US Department of Energy Advanced Scientific Computing Research program under Contract No. DE-AC05-00OR22725.

REFERENCES

[1] Chu, H., 2013. AMD heterogeneous uniform memory access. Proceedings of the APU 13th Developer Summit, pp.11-13.

[2] Lindholm, E., Nickolls, J., Oberman, S. and Montrym, J., 2008. NVIDIA Tesla: A unified graphics and computing architecture. IEEE Micro, 28(2), pp.39-55.

[3] Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y. and Volkov, V., 2008. Parallel computing experiences with CUDA. IEEE Micro, 28(4), pp.13-27.

[4] Nickolls, J. and Dally, W.J., 2010. The GPU computing era. IEEE Micro, 30(2), pp.56-69.

[5] Negrut, D., Serban, R., Li, A. and Seidl, A., 2014. Unified memory in CUDA 6.0. A brief overview of related data access and transfer issues. SBEL, Madison, WI, USA, Tech. Rep. TR-2014-09.

[6] Landaverde, R., Zhang, T., Coskun, A.K. and Herbordt, M., 2014, September. An investigation of unified memory access performance in CUDA. In 2014 IEEE High Performance Extreme Computing Conference (HPEC) (pp. 1-6). IEEE.

[7] Li, W., Jin, G., Cui, X. and See, S., 2015, May. An evaluation of unified memory technology on NVIDIA GPUs. In 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (pp. 1092-1098). IEEE.

[8] Jarząbek, Ł. and Czarnul, P., 2017. Performance evaluation of unified memory and dynamic parallelism for selected parallel CUDA applications. The Journal of Supercomputing, 73(12), pp.5378-5401.

[9] S. Chien, I. Peng and S. Markidis, Performance Evaluation of Advanced Features in CUDA Unified Memory, 2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC), 2019, pp. 50-57

[10] Jog, A., Kayiran, O., Mishra, A.K., Kandemir, M.T., Mutlu, O., Iyer, R. and Das, C.R., 2013, June. Orchestrated scheduling and prefetching for GPGPUs. In Proceedings of the 40th Annual International Symposium on Computer Architecture (pp. 332-343).

[11] Yu, Q., Childers, B., Huang, L., Qian, C. and Wang, Z., 2020. A quantitative evaluation of unified memory in GPUs. The Journal of Supercomputing, 76(4), pp.2958-2985.

[12] Ganguly, D., Zhang, Z., Yang, J. and Melhem, R., 2019, June. Interplay between hardware prefetcher and page eviction policy in CPU-GPU unified virtual memory. In Proceedings of the 46th International Symposium on Computer Architecture (pp. 224-235).

[13] Kim, H., Sim, J., Gera, P., Hadidi, R. and Kim, H., 2020, March. Batch-aware unified memory management in GPUs for irregular workloads. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (pp. 1357-1370).

[14] The AMD HIP Programming Guide v4.5. https://github.com/RadeonOpenCompute/ROCm/blob/master/AMD_HIP_Programming_Guide.pdf

[15] Open Source Platform for HPC and Ultrascale GPU Computing. https://github.com/RadeonOpenCompute/ROCm

[16] The AMD MI100 GPU Instruction Set Architecture. https://developer.amd.com/wp-content/resources/CDNA1_Shader_ISA_14December2020.pdf

[17] CUDA C++ Programming Guide. https://docs.NVIDIA.com/cuda/cuda-c-programming-guide

[18] CUDA Runtime API. http://docs.NVIDIA.com/cuda/cuda-runtime-api/index.html

[19] https://github.com/NVIDIA/cuda-samples

[20] https://github.com/ROCm-Developer-Tools/HIPIFY

[21] Harris M., 2013, Unified memory in CUDA 6. https://devblogs.NVIDIA.com/unifed-memory-in-cuda-6/

[22] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H. and Skadron, K., 2009, October. Rodinia: A benchmark suite for heterogeneous computing. In 2009 IEEE international symposium on workload characterization (IISWC) (pp. 44-54). IEEE.

[23] Zhang, C., Zhang, F., Guo, X., He, B., Zhang, X. and Du, X., 2020. iMLBench: A Machine Learning Benchmark Suite for CPU-GPU Integrated Architectures. IEEE Transactions on Parallel and Distributed Systems, 32(7), pp.1740-1752.

[24] https://developer.amd.com/wp-content/resources

[25] Gene sequence de-redundancy. https://devmesh.intel.com/projects/gene-sequence-de-redundancy

[26] Danalis, A., Marin, G., McCurdy, C., Meredith, J.S., Roth, P.C., Spafford, K., Tipparaju, V. and Vetter, J.S., 2010, March. The scalable heterogeneous computing (SHOC) benchmark suite. In Proceedings of the Third Workshop on General-Purpose Computation on Graphics Processing Units (pp. 63-74).

[27] Yang, C., Gayatri, R., Kurth, T., Basu, P., Ronaghi, Z., Adetokunbo, A., Friesen, B., Cook, B., Doerfler, D., Oliker, L. and Deslippe, J., 2018, November. An empirical roofline methodology for quantitatively assessing performance portability. In 2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC) (pp. 14-23). IEEE.

[28] Gu, Y., Wu, W., Li, Y. and Chen, L., 2020. UVMBench: A Comprehensive Benchmark Suite for Researching Unified Virtual Memory in GPUs. arXiv preprint arXiv:2007.09822.