Performance Implications of Async Memcpy and UVM: A Tale of Two Data Transfer Modes

Ruihao Li[1*] Sanjana Yadav[1] Qinzhe Wu[1] Krishna Kavi[2] Gayatri Mehta[2]
Neeraja J. Yadwadkar[1, 3] Lizy K. John[1]

[1]*The University of Texas at Austin*
[2]*University of North Texas*
[3]*VMware Research*
[*]*liruihao@utexas.edu*

Abstract

Heterogeneous systems with CPU-GPUs have become dominant parallel architectures in recent years. To optimize memory management and data transfer between CPUs and GPUs, unified virtual memory and asynchronous memory copy were introduced in recent Nvidia GPUs. With such architectural support, the entire processing flow can now be pipelined into multiple stages, thereby efficiently overlapping data transfer with computation.

*In this paper, we provide a thorough performance analysis of GPU asynchronous memory copy (**Async Memcpy**) and unified virtual memory (**UVM**) on workloads covering multiple domains. We especially study the joint effect of these two architectural features, exploring which applications benefit from one or both of these features. On a suite of 14 real-world applications, we observe an average 21% performance gain when using unified virtual memory only, and 23% gain when using both of them. In irregular programs like kmeans and lud, asynchronous memory copy provides around 20% benefits over unified virtual memory. Furthermore, we dive deep into the GPU kernel using performance counters to reveal the root causes contributing to the performance variances. We make sensitivity studies on how the number of blocks and threads, and L1-cache/shared memory partition affect the performance. We discuss future research directions to further improve the data transfer pipeline.*

1. Introduction

GPUs are widely used for machine learning (ML) and big data workloads. Through their massive parallelism, GPUs achieve improved throughput for such workloads. Given the increase in data sizes, a significant amount of work in the last decade has been dedicated to optimizing GPU architectures to provide more computation capability. For example, (i) tensor cores were introduced with Nvidia V100 to fully utilize parallelism in ML workloads [21], (ii) fine-grained structured sparsity was used in Nvidia A100 to support sparse matrix multiplication [19]. However, GPU systems need CPUs to act as central controllers for distributing work and transferring data to the GPUs. As a result, it is crucial to optimize CPUs and GPUs together as a single heterogeneous system.

Recent works have focused on reducing performance losses due to data transfer between CPU-GPU systems. For some applications with large data sets, the memory-transfer overhead is larger than 50× the GPU processing time [8]. Overlapping data transfer latency with computation is one direction to eliminate memory copy overheads, which has already been applied in different application domains, including graph processing [28], ML [25], and database systems [16]. Some approaches have proposed ways for the CPUs and GPUs to concurrently access the same memory to potentially reduce the amount of data transfer needed [11]. To amortize the data transfer overhead, recent Nvidia GPUs introduced unified virtual memory and asynchronous memory copy [19, 20] as two approaches for addressing the memory transfer issue. In this work, we draw attention to the fact that the performance implications of these different optimizations depend of application characteristics and might not be clear for users. We dive deep into *UVM* and *Async Memcpy*, these two architectural features available in recent Nvidia GPUs, and analyze and discuss their performance implications for a range of workloads.

Nvidia Unified Virtual Memory (UVM) [20] makes the memory address space shared across the CPU hosts and GPU devices. Thus, in the *UVM* system, CPUs and GPUs can access the same virtual memory space. UVM abstracts away the data transfer from the user program. So, the programmers do not need to call explicit data transfer functions. With *UVM* support, GPUs initiate the data transfer exactly at the time point when it needs the data for computation, instead of moving all data before launching the kernel. Therefore, CPUs are able to execute other jobs when not processing the CPU-GPU data transfer. In addition to *UVM*, Nvidia recently introduced a new architectural feature, asynchronous memory copy (*Async Memcpy*), with the Ampere architecture (CUDA 11) [19]. *Async Memcpy* instructions can load data directly from global memory into the streaming multiprocessor's (SM) shared memory, eliminating the need for an intermediate register file usage. *Async Memcpy* reduces register file bandwidth, uses memory bandwidth more efficiently, and reduces power consumption.

As the name implies, *Async Memcpy* can be done in the background while the SM is performing other computations.

While there is a wealth of prior research on *UVM* [2, 3, 13, 35] and *Async Memcpy* [30], questions about the performance implications of these mechanisms for a given workload remain unanswered. *UVM* is explored deeply with studies focusing on various aspects including analyzing prefetchers and over-subscriptions, developing efficient page fault handlers, and reducing data movement. Many prior works have studied the sparse units and power consumption of the Ampere architecture [4, 32], but very few of them discussed *Async Memcpy*. Moreover, no prior work has explored the intersection between *UVM* and *Async Memcpy*. It is essential to analyze the performance of these two hardware features since all kinds of architectural enhancements come with overhead. The overall system performance cannot benefit from them if the overhead is not well handled. *Async Memcpy* complicates the data transfer since additional GPU resources are required to pipeline the global memory to shared memory transfer and SM computation. In *UVM*, page faults can happen on the GPU side when the accessed data is not in the page table, which blocks the data transfer and downgrades the overall system performance.

In addition, programmers need to make a choice when writing their CUDA programs whether to write a *UMA* version or a *Async Memcpy* version. There are no automatic tools such as compilers available for converting programs to UVM or *Async Memcpy* versions. Software developers need to hand-tune the CUDA programs for better performance, making a design guideline for these two architectural features more desirable.

As *UVM* and *Async Memcpy* become available in modern GPUs, more questions will arise: (a) What kind of workloads benefit from using *Async Memcpy*? In other words, which workloads are bottlenecked by the GPU global memory to shared memory data transfer stage? Similarly, which workloads benefit from using *UVM*, i.e., bottleneck on the CPU DRAM to GPU global memory transfer? (b) What are the performance implications for the choice between *Async Memcpy* and *UVM*? Are these implications workload-agnostic? How should programmers make these choices? (c) Would the overall performance improve further if we use both *UVM* and *Async Memcpy*? Can programmers make the decision with limited profiling or intensive profiling?

To answer these questions, in this paper, we make a deep analysis of how *UVM* and *Async Memcpy* affect the performance of workloads on CPU-GPU heterogeneous systems. To the best of our knowledge, we are the first to consider the two architectural features together. The following are the key contributions of our work.

- We explore the performance implications of CUDA programming choices for data transfer (*UVM* and/or *Async Memcpy*). We break down the execution time into GPU kernel time, data transfer time, and data allocation time on 21 workloads and use performance counters to reveal the root cause of the performance differences. We make sensitivity studies on the number of blocks and threads, and L1-Cache/shared memory partition, to further understand the impact of *UVM* and *Async Memcpy*.
- Our analysis on *UVM* and *Async Memcpy* can help CUDA programmers understand these two hardware features better and develop more efficient GPU codes.
- We create and make available a benchmark suite for *Async Memcpy* and/or *UVM* studies, including 7 microbenchmarks and 14 real-workload applications, which cover multiple domains. We implement the *Async Memcpy* and/or *UVM* versions of each workload that weren't available already. We believe that releasing this benchmark publicly will enable further research in this domain. Our code is available at https://github.com/UT-LCA/UVMAsyncBench.

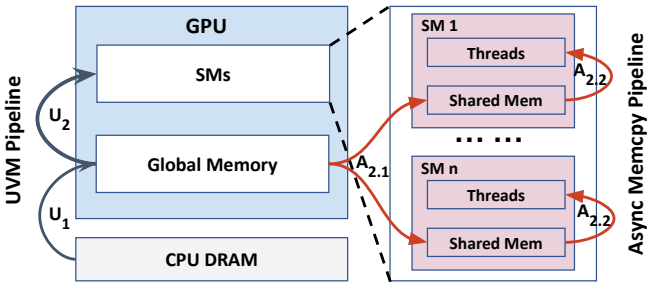


Figure 1: CPU-GPU system data transfer pipeline. With *UVM*, the pipeline contains U_1 and U_2 stages. Adding *Async Memcpy* atop of *UVM*, stage U_2 can be pipelined into $A_{2.1}$ and $A_{2.2}$.

2. Background and Related Work

Figure 1 summarizes state-of-the-art CPU-GPU heterogeneous system memory architectures with *UVM* and *Async Memcpy*. We describe details about *UVM* in Section 2.1 and *Async Memcpy* in Section 2.2 . We also discuss prior performance characterization studies on *UVM* and *Async Memcpy* in this section.

2.1. Unified Virtual Memory

UVM [20] is a powerful technology introduced in Nvidia GPUs, which provides a single memory space and automates memory management and data migration between the CPU host and GPU device physical memory modules. In Figure 1, for *UVM*, the CPU DRAM to GPU global memory data transfer is marked U_1 and the global memory to shared memory data transfer during SM execution is marked U_2. An enhanced *UVM* version supports *prefetch* data from global memory to L2 cache [2, 13], reducing the global memory to shared memory time (U_2).

UVM is designed to be transparent to the application, making CUDA programming using UVM more concise and understandable (See example in Figure 2). With a unified virtual memory space, the CPU and GPU heap memory management can be overlapped. Additionally, applications can easily leverage the combined memory resources of multiple GPUs to perform data-intensive computations, such

Without UVM

```
DataType *h_a, *d_a;
h_a = malloc(size);
cudaMalloc(&d_a, size);
cudaMemcpy(d_a, host_a, size, cudaMemcpyHostToDevice);
cudaKernel<<<...>>>(d_a);
cudaMemcpy(host_a, host_a, size, cudaMemcpyDeviceToHost);
cudaDeviceSynchronize();
print(h_a);
```

With UVM

```
DataType *uvm_a;
cudaMallocManaged(&uvm_a, size);
cudaKernel<<<...>>>(uvm_a);
cudaDeviceSynchronize();
print(uvm_a);
```

Figure 2: CUDA programming without/with UVM (Pseudo-code). Coding with UVM simplifies programming.

Without Async Memcpy

```
__shared__ DataType data[size];
for (; tile < end; tile++) {
     memcpy(data[0:size-1], input[tile]);
     compute on data[0:size-1];
}
```

With Async Memcpy

```
__shared__ DataType data[size * 2];
for (compute = fetch; compute < end; compute++) {
     for (; fetch < end && fetch < compute + 2; fetch++)
          memcpy_async(data[0:size-1], input[fetch]);
     compute on data[size:size*2-1];
}
```

Figure 3: CUDA programming without/with Async Memcpy (Pseudo-code). Async Memcpy necessitates careful programmer intervention.

as ML and high-performance computing. However, *UVM* also brings performance overhead. GPUs require a copy of the CPU virtual memory physical memory mapping if a unified virtual memory space is used. GPU page faults must be synchronized with CPUs as well.

The benefit and overhead of *UVM* have led to attention from research communities, which can be divided into two categories:

(1) Architecture Optimizations that focus on reducing the additional performance overhead by introducing hardware enhancement. For example, prior works have improved *UVM* system performance by batch processing page faults [13], improving GPU cache utilization [14], and dynamically managing variable-sized pages [15]. All these architectural improvements can be used in next-generation GPU designs.

(2) Characterization and Analysis that reveal bottlenecks in current systems, guiding programmers to develop more hardware-friendly applications and libraries. For example, Zheng et al. [33] compared *UVM* and traditional memory management methodology and found the possibility of using *UVM* with minimal overhead. Allen et al. [2, 3] dived into the software and hardware-based root causes of the internal behaviors of page fault generation and servicing. Shao et al. [29] revealed the reasons behind the diverse sensitivities to oversubscription among different workloads.

2.2. Asynchronous Memcpy

Pipelining computation and data transfer can boost the CPU-GPU heterogeneous system performance. In the last decade, efforts have been put into overlapping GPU computation time with CPU DRAM-GPU global memory data transfer time [8, 11]. This technique has already been used in the *UVM* system, by enabling GPU-driven fine-granularity transfer while freeing CPU cycles for other jobs, instead of blocking the CPU to transfer the entire chunk of allocated memory.

In addition to CPU DRAM-GPU global memory data transfer, the GPU global memory to shared memory data transfer latency can also be pipelined and optimized, as long as the GPU hardware architecture supports it. Fortunately, starting with Ampere architecture with CUDA 11, Nvidia GPUs support this asynchronous copying of data from global memory to shared memory. In Figure 1, for *Async Memcpy*[1], copying data from global memory to shared memory is marked as $A_{2.1}$, and fetching data from shared memory during processing is marked $A_{2.2}$. The *Async Memcpy* allows the programmer to initiate a transfer of data from global to shared memory, without blocking GPU thread execution (code snippets shown in Figure 3). Additional primitives are then provided which enable waiting for the asynchronous memory operation to complete.

Though there are many works that studied the Ampere architecture before, the majority of the works focused on the sparse units [6, 7] and power efficiency [32], but not *Async Memcpy*. A few prior studies on *Async Memcpy* can also be divided into two categories: **(1) Software Optimizations** that focus on enhancing compilers and system libraries to make full use of the new *Async Memcpy* hardware feature. For example, *Async Memcpy* has been used in deep learning compilers recently to optimize the pipeline of tensor programs [12, 31]. **(2) Characterization and Analysis** that study the performance of *Async Memcpy* and compare it against its predecessor architecture. For example, Svedin et al. [30] compared A100 performance with four previous generations of GPUs, and in their experiments on A100, they observed up to 1.25× performance improvement from *Async Memcpy*.

2.3. UVM vs Async Memcpy

As shown in Figure 1, *Async Memcpy* can be used together with *UVM*. These two architectural features can be used together and make the CPU-GPU heterogeneous system data transfer into a 3-stage pipeline: (1) from CPU DRAM to GPU global memory (U_1), (2) from GPU global memory to shared memory ($A_{2.1}$), and (3) from shared memory to each thread ($A_{2.2}$).

1. In this paper *Async Memcpy* only refers to asynchronous copying of data from global memory to shared memory.

TABLE 1: Hardware configurations used in the study.

Hardware	Description
CPU	64× AMD EPYC 7742 @ 3.2GHz 4 MB L1-dcache, 4 MB L1-icache 64 MB L2-cache, 512 MB L3-cache 16× 64GB DDR4 @ 3200 MT/s
GPU	Nvidia Tesla A100 @ 1410MHz 40GB HBM2 @ 1215 MHz

All pipelines come with overhead. The overall system throughput can only be improved with an acceptable number of pipeline bubbles. Whether *Async Memcpy* together with *UVM* is able to boost GPU system performance more is worth exploring, considering the sophisticated 3-stage data transfer pipeline.

3. Experimental Methodology

In this section, we first provide details of the experimental hardware and software setup. We then give an overview of the 21 workloads in the benchmark suite we created. Lastly, we discuss how to determine the configuration of each workload since the performance can be affected dramatically with different configurations.

3.1. Experimental Setup

3.1.1. Hardware. We conduct our characterization study on an Nvidia A100 server with AMD CPUs. We list the hardware configurations of the CPU-GPU heterogeneous system in Table 1.

3.1.2. Software. We use Ubuntu 20.04 with Linux kernel 5.4.0 as the operating system. We use GCC 9.4.0 and CUDA 11.4 as the compilers for the heterogeneous system. For the profiling tools, we use Linux perf [1] and Nvidia CUPTI [17] for performance counter collections of CPUs and GPUs, respectively.

3.1.3. *UVM* and *Async Memcpy* Configurations. We use the following five architecture configurations in our experiments:
(a) **standard** (Without *UVM* or *Async Memcpy*),
(b) **async** (using *Async Memcpy* only),
(c) **uvm** (using *UVM* only),
(d) **uvm_prefetch** (using *UVM* with *prefetch*), and
(e) **uvm_prefetch_async** (using *UVM* with *prefetch* and *Async Memcpy*).

3.2. Overview of Benchmarks

We use 14 real-world applications and 7 microbenchmarks in our performance studies, and illustrate the two groups of benchmarks in Table 2. These 21 workloads cover the domain of linear algebra, physics simulation, data mining, image processing, and machine learning. We elaborate on these benchmarks in detail in this section.

3.2.1. Microbenchmarks. We use a set of microbenchmarks to gain a better understanding of the performance of *UVM* and *Async Memcpy*. Each workload in the Microbenchmark suite uses one single CUDA kernel. The *vector_seq* and *vector_rand* are workloads built atop of benchmarks used in the prior study [30]. We use the CUDA *Pipeline* API, since it showed better performance than *Arrive/Wait Barriers* [30]. In addition to Vector-to-Constant, we include 5 additional microbenchmarks from Polybench [24][2]. Vector-to-Vector (*saxpy*), Matrix-to-Vector (*gemv*), and Matrix-to-Matrix (*gemm*) are considered as extensions to the two Vector-to-Constant workloads, each of which shares similar computation patterns but different computation densities. 2D convolutions (*2DCONV*) and 3D convolutions (*3DCONV*) are fundamental kernels of a large number of computer vision and ML workloads, which have been gaining increasing popularity in the last decade.

3.2.2. Real-world Applications. Our benchmark suite includes the widely used *Rodinia* [5] benchmark suite, which contains 29 applications covering domains of multimedia, arithmetic, signal/image processing, biological computing, and big data applications. Instead of using the entire *Rodinia* suite, 8 diverse benchmarks are selected. We select *lavaMD*, *NW*, *Kmeans*, *Srad*, *Backprop*, and *Pathfinder* based on the representativeness of the 6 workloads. They were classified into different groups based on prior performance characterization studies [27]. We also include *HostSpot* and *LUD*, since they were used in prior *Async Memcpy* studies [30].

We use workloads *bayesian* and *KNN* from *Uvmbench* [9], which is a comprehensive benchmark suite for *UVM* studies (other workloads in *Uvmbench* are overlapped with *Polybench* and *Rodinia*). We implement the *Async Memcpy* version of them as well.

We also study ML workloads since they are widely used in CPU-GPU heterogeneous systems, especially in the last decade. Instead of using the *CNN* and other ML workloads in *Uvmbench*, we choose workloads (networks) in *darknet* [26], since the scalability of *darknet* is better than the *Uvmbench* implementation and is more widely used in the ML community. In addition, *darknet* is implemented in C rather than Python, making it easier for profiling-based studies.

3.3. Benchmark Configurations

It is well known that the benchmark performance depends on the chosen configuration. In this section, we explore the input size search space and rationalize our choice of input size.

We define six input sizes in Table 3, from 1MB to 32GB memory footprint. We also list the reference size (assuming float32 data type, the numbers are subject to change, e.g., if there are 2 vectors, the size of each vector is 128K on the

2. We adjusted the Polybench codes to make them scalable for large input sizes. We also compared the performance of our own implementation with cutlass [18] to guarantee the efficacy of our kernel implementations.

TABLE 2: Benchmark programs.

Suites	Source	Program Name	Description	Inputs
Micro	Svedin et al. [30]	vector_seq	Vector-to-Constant, element-wise arithmetic operations on vector (sequential access)	Vector (1D)
		vector_rand	Vector-to-Constant, element-wise arithmetic operations on vector (random access)	Vector (1D)
	PolyBench [24]	saxpy	Vector-to-Vector multiplication and addition	Vector (1D)
		gemv	general Matrix-to-Vector multiplication	Matrix (2D)
		gemm	general Matrix-to-Matrix multiplication	Matrix (2D)
		2DCONV	general 2D convolution	Grid (2D)
		3DCONV	general 3D convolution	Grid (3D)
Apps	Rodinia [5]	LavaMD	The code calculates particle potential and relocation due to mutual forces between particles within a large 3D space.	Box (3D)
		NW	Needleman-Wunsch, a nonlinear global optimization method for DNA sequence alignments.	Sequence (2D)
		Kmeans	K-means is a clustering algorithm used extensively in data-mining and elsewhere, important primarily for its simplicity.	Points (1D)
		Srad	Speckle Reducing Anisotropic Diffusion is a diffusion method for ultrasonic and radar imaging applications based on partial differential equations (PDEs).	Grid (2D)
		Backprop	Back Propagation is an ML algorithm that trains the weights of connecting nodes on a layered neural network.	Nodes (1D)
		Pathfinder	PathFinder uses dynamic programming to find a path on a 2-D grid.	Grid (2D)
		HotSpot	A widely used tool to estimate processor temperature based on an architectural floorplan and simulated power measurements.	Grid (2D)
		LUD	LU Decomposition is an algorithm to calculate the solutions of a set of linear equations.	Grid (2D)
	UVMBench [9]	bayesian	Bayesian network learning algorithm	Nodes (1D)
		KNN	K-Nearest Neighbors Algorithm	Points (1D)
	Darknet [26]	Resnet18	Residual Network with 18 convolution layers	ImageNet dataset
		Resnet50	Residual Network with 50 convolution layers	
		Yolov3-tiny	Yolov3-tiny	COCO dataset
		Yolov3	Yolov3	

TABLE 3: Parameter configurations. Numbers are rounded up to the lower bound for, e.g., *Mem* for *Large* is 512MB $\sim$ 4GB.

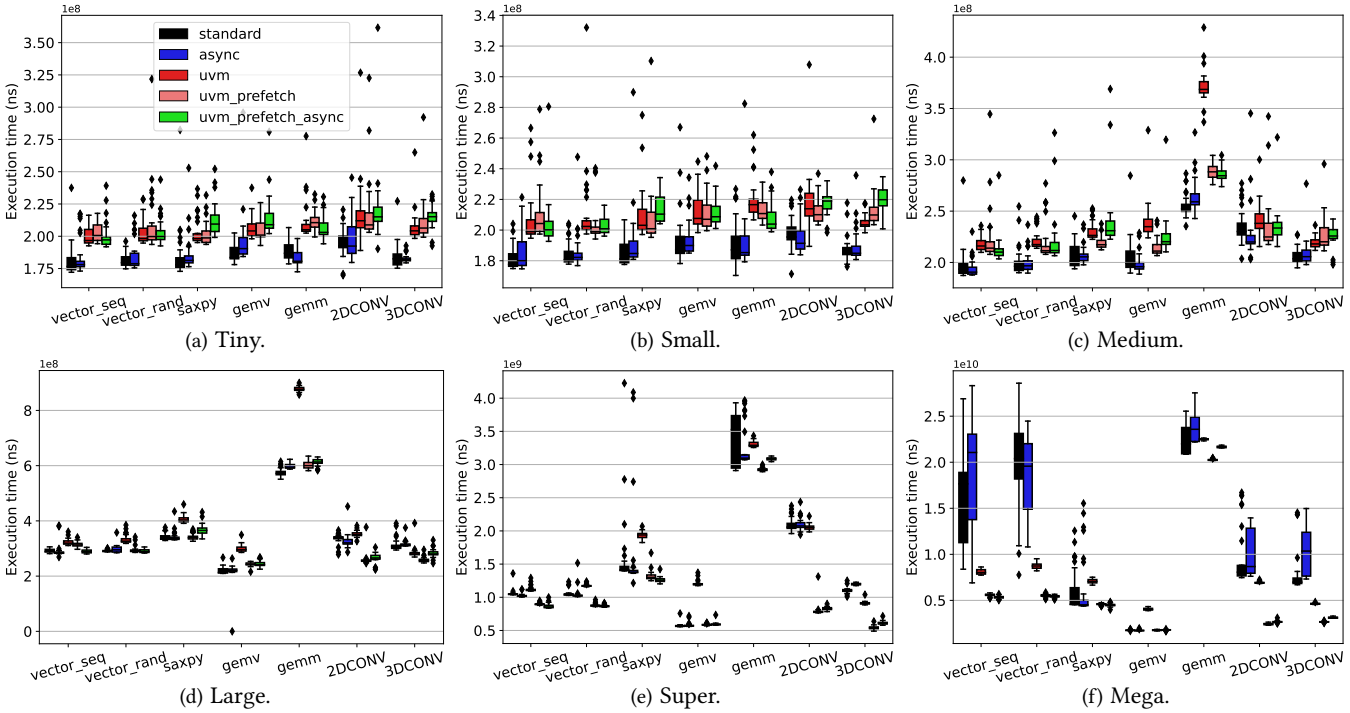
	Tiny	Small	Medium	Large	Super	Mega
Mem	1MB	8MB	64MB	512MB	4GB	32GB
1D Grid	256K	2M	16M	128M	1G	8G
2D Grid	512^2	$1K^2$	$4K^2$	$8K^2$	$32K^2$	$64K^2$
3D Grid	64^3	128^3	256^3	512^3	$1K^3$	$2K^3$

Tiny input) of each input dimension, for 1D, 2D, and 3D inputs. We need to use an input size that is large enough to capture the performance difference between using *UVM* and *Async Memcpy*. The input size also needs to be large enough to make the execution time of the region of interest long enough to amortize system overhead. However, an extremely long execution program is not the best candidate for performance studies. For a deep hardware profiling-based analysis, it is easy to trigger more than 10$\times$ overhead due to the limitation of profiling tools [34]. Therefore, the input size should not be either too small or too large.

The majority of prior characterization works focused on the GPU kernel execution time. Since we consider CPU-GPU as one entire heterogeneous system, we use the sum of data allocation time (*cudaMalloc*() or *cudaMallocManaged*() + *cudaFree*()), the data transfer time (*cudaMemcpy*() or explicit unified memory data transfer time), and GPU kernel execution time as the overall execution time.

We ran each workload 30 times and plotted the distribution of each run. Figure 4 shows the overall execution time distribution of the 7 workloads in the microbenchmark suite on 6 different input sizes. We note that the confidence interval becomes narrower when the input data size is larger than *Medium*. This is aligned with our assumption that with a constant system overhead, a larger input size can amortize the noise in measurements. To have a better panoramic view of the data, we also list the average (of the 5 setups) standard deviation of the 30 runs over the mean in Figure 5. Considering the geometric-mean of the 7 workloads, when the input size increased from *Tiny* to *Large*, the overall execution time became more stable, which is as expected.

However, the *Mega* input size shows more performance variance than *Large* and *Super*, which is counter-intuitive. We pick the most abnormal benchmark, *vector_seq*, and visualize the breakdown of its execution time of the 30 runs in Figure 6. We see that the data allocation time and GPU kernel time almost remain the same, but the data transfer time between CPU and GPU varies a lot. The reason is that the input size is close to the capacity of a single DRAM chip (64GB in our evaluation platform). There is a large chance that part of the data is stored in the other DRAM chip, which adds more randomness to the overall



(a) Tiny. (b) Small. (c) Medium. (d) Large. (e) Super. (f) Mega.

Figure 4: Execution time (distribution of 30 runs) on microbenchmarks with different input sizes. *Large* and *Super* are the most stable.

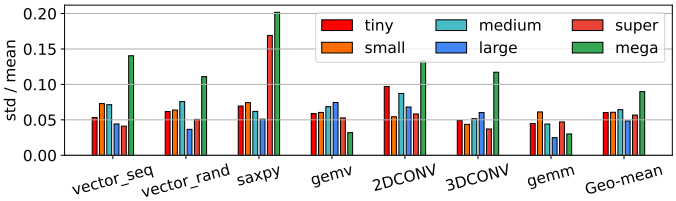


Figure 5: Standard deviation (over mean) of 30 runs on different input sizes. *Large* and *Super* are the smallest.

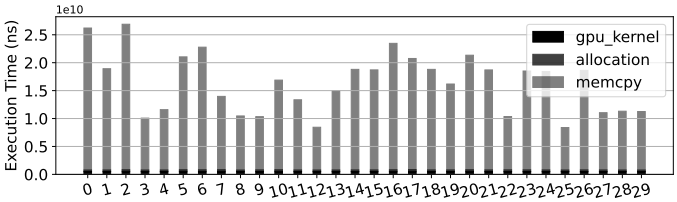


Figure 6: Execution time breakdown of 30 runs for *Mega* inputs. *Memcpy* time is not stable.

system performance. As a result, we choose to use *Large* and *Super* input sizes for the rest of our experiments.

Takeaway 1: Big input sizes are typically used for benchmarking to avoid noisy measurements. However, our analysis suggests that big input sizes do not always guarantee stable performance in CPU-GPU heterogeneous systems. Instead, for generalizable benchmarking results in GPUs, input sizes should be chosen considering the memory capacity.

4. Results

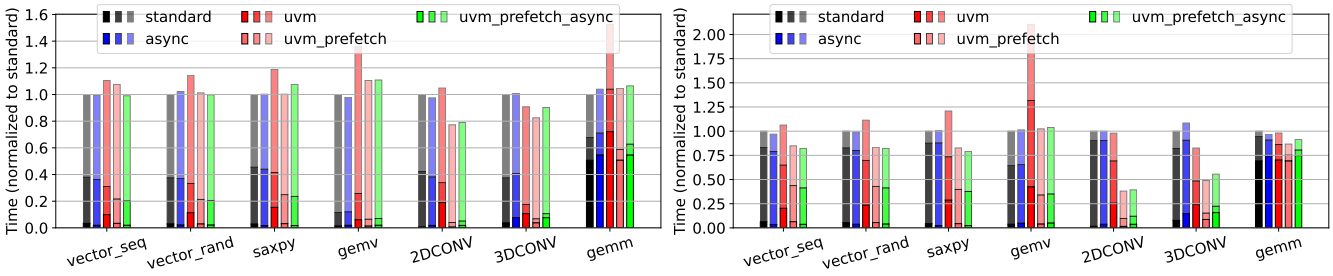
In this section, we make a side-by-side comparison between the five setups for all benchmarks, by breaking down the overall execution time into GPU kernel, CPU-GPU data transfer, and data allocation time. We dive deep into workloads with unexpected behaviors by measuring GPU performance counters. We summarize our insights on the results and provide key takeaways for both GPU programmers and computer architects.

4.1. Performance Comparison

In this section, we compare and analyze the performance of microbenchmarks and real-world applications for various configurations of *UVM* and *Async Memcpy*. This comparison can also indicate whether a workload is bottlenecked by CPU DRAM to GPU global memory data transfer, or GPU internal data transfers (global memory to shared memory).

4.1.1. Microbenchmarks. We used both *Large* and *Super* input sizes since both of them show relatively stable performance among multiple runs. In addition, the GB-level memory footprint is large enough to make system noise negligible. We used the average of the 30 runs and make a side-by-side comparison between the 7 microbenchmarks, shown in Figure 7.

When considering the overall execution time, there is almost no performance difference between *standard* and *async*. Considering the geo-mean of the 7 workloads, the *async* has only 0.27% and 0.36% performance over *standard*, for *Large* and *Super* respectively. However, when comparing the pure GPU kernel time, there is an appreciable difference.



(a) *Large* input. (b) *Super* input.

Figure 7: Comparisons on Microbenchmarks. From bottom (darkest) to top (lightest), each shade shows gpu_kernel, memcpy, and allocation. The combination of *Async Memcpy* and *UVM* benefits *vector_seq*, but not *3DCONV* and *gemm*.

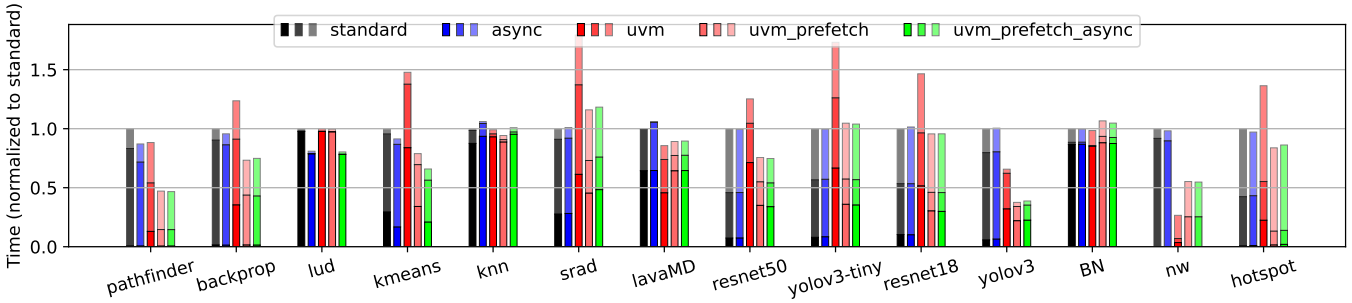


Figure 8: Comparisons on real-world applications. From bottom (darkest) to top (lightest), each shade shows gpu_kernel, memcpy, and allocation. *Async Memcpy* works better than *UVM* for *lud*, but the combination of *Async Memcpy* and *UVM* does not benefit *yolov3*.

For example, *async* achieves 41.78% GPU kernel time reduction over *standard* in the *vector_seq* workload while 146.02% increment over *standard* in the 2*DCONV* workload (both with *Large* input sizes). For memory-bounded workloads, the 41.78% kernel time reduction only results in less than 1% overall performance improvement. For compute-bounded workloads, the overhead due to the additional pipeline stages in *async* leads to significant performance degradation.

The overall system performance cannot be improved by merely using *UVM*. As shown in Figure 7, considering the geo-mean of the 7 workloads, *uvm* slows down the overall performance 16.79% and 13.23% over *standard*, for *Large* and *Super* respectively, though *uvm* achieves 31.46% and 35.19% time savings over *standard* on data transfer between GPUs and CPUs for the two input sizes. The cost of using *UVM*, e.g., additional page walking, is non-negligible [2]. Such overhead leads to 2.0× and 2.2× GPU kernel time increases over *standard* on *Large* and *Super*, counteracting the benefit of data transfer time savings.

Therefore, we use *UVM* with *prefetch* (*uvm_prefetch*), which is able to improve the performance by reducing the number of page faults [3], for a more fair comparison. When using *uvm_prefetch*, there is 3.07% and 28.40% performance improvement over *standard* for *Large* and *Super* input sizes, respectively. The reason for the limited overall performance improvement on *Large* is the nearly constant data allocation overhead. For GPU kernel and CPU-GPU data transfer time, *uvm_prefetch* achieves 54.07% and 45.41% time savings over *standard* on the *Large* input, and 57.50% and 47.90% time savings over *standard* on *Super*.

It is also interesting and worth exploring whether the asynchronous memory copy can be used together with unified virtual memory. We introduce our last setup, *uvm_prefetch_async*, by applying these two architectural features together. The average performance improvement of *uvm_prefetch_async* over *standard* is 27.01% for the *Super* input size, which is slightly lower than *uvm_prefetch* over *standard* (28.40%). However, that does not mean asynchronous memory copy should be forbidden for all scenarios. For memory-bounded workload *vector_seq* and *vector_rand*, *uvm_prefetch_async* can make the time savings over *standard* of the two workloads to 17.81% and 17.87%, which is better than the 15.20% and 16.66% time savings over *standard* when using *uvm_prefetch*. However, for workloads with higher computation intensity, e.g., *2DCONV*, *3DCONV*, and *gemm*, the additional control logic overhead in asynchronous memory copy can hurt the performance. For example, in *gemm*, *uvm_prefetch* spends only 0.06% more time over *standard* on GPU kernels, while *uvm_prefetch_async* spends 7.86% more.

4.1.2. RealWorld Applications. In addition to microbenchmarks, we show the execution time breakdown of the 14 real-world applications as well. We use the average of 30 runs and *Super* input sizes, shown in Figure 8.

Considering the geo-mean of the 14 workloads, there is 2.81%, –4.41%, 20.96%, and 22.52% performance improvement achieved by *async*, *uvm*, *uvm_prefetch*, and *uvm_prefetch_async* over *standard*, respectively. The majority of the speedups are coming from a reduction of CPU-GPU data transfer time. The *uvm*, *uvm_prefetch*, and

uvm_prefetch_async achieve 32.70%, 64.24%, and 64.18% memcpy time savings compared with *standard*.

In real-world applications, *Async Memcpy* together with *UVM* can bring more performance benefits, as *uvm_prefetch_async* achieves the highest speedups. *Async Memcpy* pipelines the computation and global memory to shared memory data transfer, so *uvm_prefetch_async* has less GPU kernel overhead compared with *uvm_prefetch*. The *uvm_prefetch_async* spends 21.72% more GPU kernel time over *standard*, while *uvm_prefetch* spends 27.50% more.

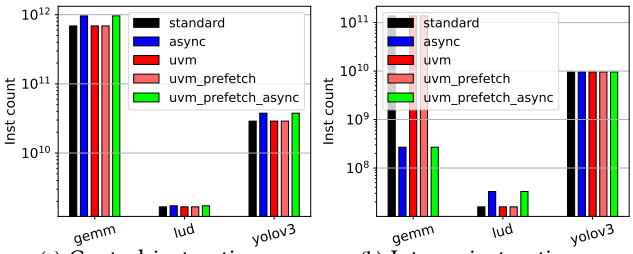
Though *uvm_prefetch_async* beats the performance of *uvm_prefetch* in most scenarios, there is one exception, *yolov3*. The *yolov3* workload is not compute-bounded, since the GPU kernel time counts only 5.81% of the overall execution time. The reason *uvm_prefetch_async* performed worse is that the GPU global memory to shared memory data transfer is not the bottleneck of the system data transfer pipeline, so the *async* setup does not bring any performance benefit. In addition, the *yolov3* uses the *gemm* kernel we studied in the microbenchmark suite, which follows a very regular and predictable data access pattern, making *uvm_prefetch* more powerful under this scenario. The other two interesting data points are *nw* and *lud*. For *nw*, *prefetch* can downgrade the performance, despite using *Async Memcpy* or not. The reason is that *nw* has two CUDA kernels operating on the same data object, making *prefetch* data on one kernel affect the performance of the other kernel. For *lud*, the performance only benefits from *Async Memcpy* but not *UVM* (with *prefetch*). The reason is that *lud* follows an irregular data access pattern, making prefetchers not able to predict the next data access accurately. When combining these two techniques together, *lud* maintains the same speedup as *Async Memcpy* only; it is not affected by *UVM* overhead.

Takeaway 2: (1) *UVM* without *prefetch* does not provide significant performance improvement, but with *prefetch* there is a 21% performance gain on real-world applications. Workloads with regular data access patterns, e.g., *2DCONV* can benefit more from *UVM* (with *prefetch*) (up to 2.63× speedups over *standard*). (2) Workloads with less regular data access patterns, e.g., *lud* can benefit more from *Async Memcpy* (up to 1.24× speedups over *UVM*).

4.2. In-Depth Analysis

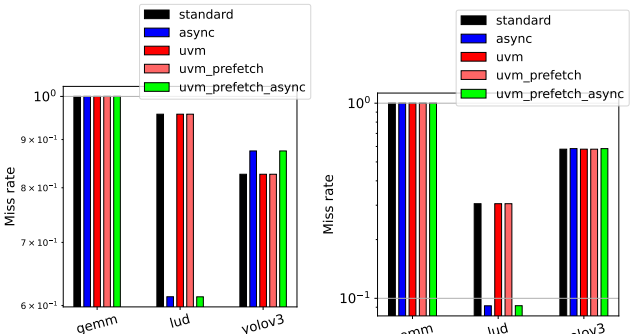
Async Memcpy brings additional computation overhead on GPU kernels to control data transfer pipelines. Based on the results on *gemm*, *yolov3*, and *lud*, it is not clear whether a compute-bound workload can benefit from *Async Memcpy* if we only consider GPU kernel time as the metric. In this section, we will dive deep into the GPU kernel to reveal the real cause affecting *Async Memcpy* performance.

Analysis of performance counters is essential for optimizing system performance and identifying performance bottlenecks, which have already been used extensively in prior performance studies [10, 22]. In this section, we explore the effects of *Async Memcpy* and *UVM* by comparing the 2 groups of hardware counters. We pick instruction mix-ups and cache miss rates since they are most correlated with the performance based on our measurement results.



(a) Control instructions. (b) Integer instructions.

Figure 9: Instruction mix comparison. In *gemm* and *yolov3*, *Async Memcpy* increases control instruction count, which hurts the performance.



(a) Load miss rate. (b) Store miss rate.

Figure 10: Cache miss rate comparison. *Async Memcpy* significantly reduces the miss rate for both loads and stores in *lud*.

4.2.1. Instruction Mix. We first use the GPU instruction mix to explore the potential cost when using *UVM* and *Async Memcpy* . We compared the total number of memory access, floating point, integer, and control instructions. There is a notable difference in integer and control instructions between the five setups, so we plot them in Figure 9. It is shown that the instruction counts would not be affected too much when *UVM* (with and without prefetch) is used, but *Async Memcpy* does affect the instruction mix.

Based on the results in Section 4, *Async Memcpy* can hurt the performance if used with *UVM* (with *prefetch*) on *gemm* and *yolov3*, but not on *lud*. The additional performance overhead of *Async Memcpy* comes from the increase in control instructions (39.98% on *gemm* and 30.13% on *yolov3*), which is the result of the additional stage on the data transfer pipeline.

4.2.2. Global Cache Miss. In addition to the instruction mix, we also measured other performance counters. There is no conspicuous difference on most of the counters, except global cache miss, which is the miss rate for global load and store in unified L1/texture cache. As shown in Figure 10, there is 35.96% and 69.99% load and store miss rate reduction when *Async Memcpy* is applied on *lud*, which is the root cause contributing to the performance improvement.

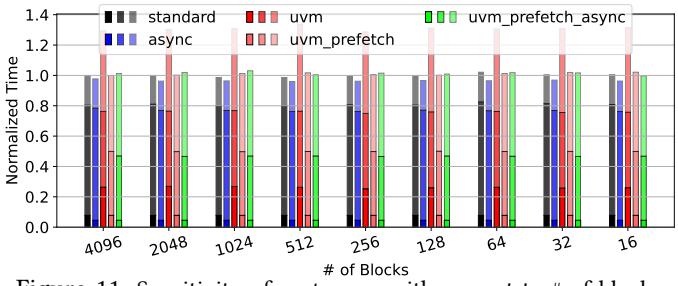


Figure 11: Sensitivity of *vector_seq* with respect to # of blocks. From bottom (darkest) to top (lightest), each shade shows gpu_kernel, memcpy, and allocation. Performance remains stable when # of blocks changes.

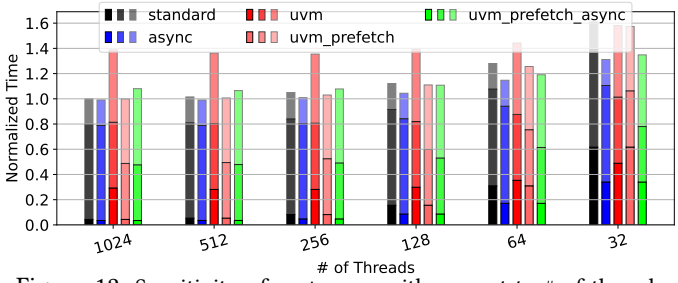


Figure 12: Sensitivity of *vector_seq* with respect to # of threads per block. From bottom (darkest) to top (lightest), each shade shows gpu_kernel, memcpy, and allocation. Performance varies when # of threads changes.

Takeaway 3: Additional control instruction results in overhead in *Async Memcpy*. The performance improvements of *Async Memcpy* come from reduced cache load and store miss rates.

5. Sensitivity Studies

Even when we use the same input size, the overall performance of a workload can still be affected by GPU program hyperparameters, including the number of CUDA blocks, threads, and L1-Cache/shared memory partition. These configurations cannot be determined by compilers automatically and have to be assigned by CUDA programmers. In this section, we further explore how *Async Memcpy* and/or *UVM* are sensitive to these configurations.

5.1. CUDA Block and Thread

Programmers assign the parallelism of each CUDA program by following the GPU resource hierarchy (Grid, Block, Thread) as the guideline. There is almost no limitation[3] on the number of blocks in the entire GPU grid. Such transparency makes large workloads easily programmed on GPUs without considering real hardware resource limitations.

In Nvidia GPUs, one CUDA block is mapped on one SM unit (A100 has 108 SM units, each of which contains

3. CUDA uses 16-bit integers as the block index in the 3D grid. As long as there is no overflow on the block index, there will be no compiling error.

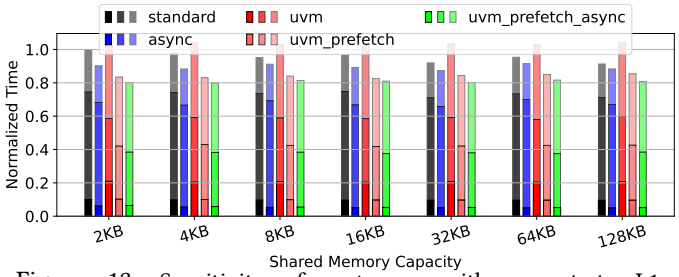


Figure 13: Sensitivity of *vector_seq* with respect to L1-Cache/Shared Mem partition. From bottom (darkest) to top (lightest), each shade shows gpu_kernel, memcpy, and allocation. Increasing shared memory hurts *UVM* performance.

64 CUDA cores). How *Async Memcpy* and/or *UVM* would affect the CUDA block and thread to the real GPU core mapping is worth exploring. With more blocks, the entire input space is partitioned into finer granularity chunks. With more threads in one block, the parallelism can be increased but the per-thread shared memory resource gets reduced, due to the limited shared memory capacity (164KB per block on A100).

We first explore the effect of the number of blocks on the overall system performance. We set the number of threads per block as 256, and change the number of blocks from 4096 to 16. We used the *vector_seq* workload since the computation pattern of this workload is simple and can be flexibly partitioned into a different number of blocks (and threads). In addition, the performance of *vector_seq* can benefit from both *Async Memcpy* and *UVM*. We plot the execution time breakdown of *vector_seq* in Figure 11. Interestingly, there is no obvious performance change ($\sim$ 2%) on all 5 configurations when using a different number of blocks. On average, *async*, *uvm_prefetch*, and *uvm_prefetch_async* achieves 2.77%, 21.34%, and 22.38% performance improvement over *standard*, respectively.

Once the total number of cores is fixed (set as 64), changing the number of threads within each block can affect the performance as well. As shown in Figure 12, the performance is sensitive to the number of threads per block (more than 50%). The reason is that when there are fewer than 128 threads, GPU resources remain underutilized (A100 has 6,912 CUDA cores). The execution time breakdown also confirms this. The GPU kernel execution time of 32 threads is 3.95$\times$ as 128 threads. Though the performance downgrades when using fewer threads, *async* performs much better than *standard* (1.01% speedups on 1024 threads, but 16.51% speedups on 32 threads). The reason is that fewer threads in one block can lead to a deeper buffer for each thread, with the same per-block shared memory capacity. As long as the buffer becomes deeper, *Async Memcpy* shows more efficiency in improving performance than *UVM* (with *prefetch*).

Takeaway 4: The performance of *UVM* and *Async Memcpy* is not sensitive to the number of CUDA blocks, but very sensitive to the number of threads per block.

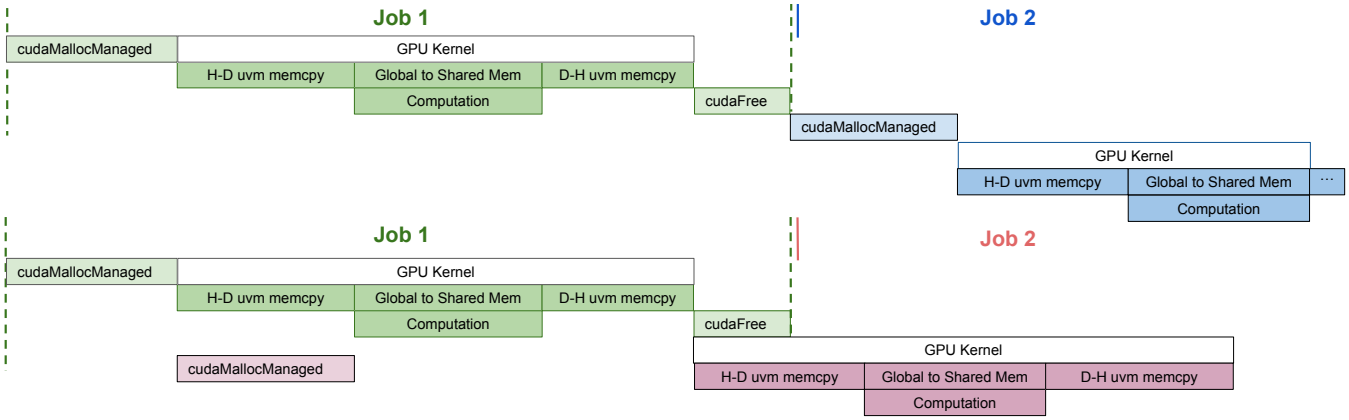


Figure 14: Without/with the inter-job pipeline. H-D and D-H is the abbreviation of Host-to-Device and Device-to-Host.

5.2. L1-Cache/Shared Memory Partition

Nvidia Ampere architecture features a unified L1 cache, texture cache, and shared memory for a total of 192KB per SM. The shared memory can be configured to use up to 164KB of the unified memory while the rest is used for both the L1 and texture cache [19]. The L1-cache/shared memory partition is decided by CUDA programmers, so it is important to understand the trade-offs in making the partition decisions.

It is interesting to know *UVM* and *Async Memcpy* are sensitive to the L1-cache/shared memory partition. The 164KB shared memory capacity is usually larger than a single page size (4KB for x86 systems). In *UVM*, multiple page faults could be triggered when fetching data to shared memory. *Async Memcpy* pipelines the thread computation and data transfer by double buffering the shared memory, making an efficient partition more essential.

We increase the shared memory capacity from 2KB to 128 KB[4] and compare the performance of the 5 configurations (shown Figure 13). After allocating more than 4KB of shared memory, the per-thread buffer depth will be enough for pipelining the kernel computation and global to shared memory data transfer. The performance of using *Async Memcpy* and *UVM* together can be hurt if too much shared memory is allocated, which reduces the L1-cache capacity and bottlenecks the system performance. A decent amount of L1-cache has to be reserved, in order to protect current cache lines from being evicted by the *UVM* prefetcher.

Takeaway 5: *Async Memcpy* and *UVM* are sensitive to shared memory/L1-cache partition. Allocating too small of a shared memory (too large L1-cache) capacity can hurt *Async Memcpy* performance, while too large of a shared memory (too small L1-cache) can hurt *UVM* performance.

6. Discussion

Though *Async Memcpy* and *UVM* can improve the CPU-GPU heterogeneous system performance, there are still limitations in current data transfer pipelines. In this section, we discuss a new data transfer model that overlaps multiple jobs. We use the profiling results shown earlier in this paper to derive preliminary estimates for the performance gain of this new data transfer model.

6.1. Limitations of *UVM* and *Async Memcpy*

Async Memcpy and *UVM* improve the system performance by overlapping data transfer and computation. Based on the execution time breakdown in Section 4, the CPU-GPU data transfer time decreases from 55.86% to 24.55% of the overall execution time. Since less time is spent on data transfer, GPU occupancy is improved as well (from 25.15% to 37.79%).

Though helped by *Async Memcpy* and *UVM*, the GPU occupancy is still relatively low. GPU computation units are idle during more than 60% of cycles. In addition, since *Async Memcpy* and *UVM* do not improve the CPU side, now the overall system time is bounded by data allocation (*cudaMalloc*() and *cudaFree*()) time. Data allocation only counts for 18.99% of the overall execution time before but with *Async Memcpy* and *UVM* it increases to 37.66%.

6.2. A New Data Transfer Model

Overlapping data allocation with other tasks (data transfer and GPU kernel) is the future direction to further improve the data pipeline. However, data allocation has to be done before data transfer and GPU kernel for every single workload, so overlapping data allocation time with other tasks is infeasible for most scenarios. The exception can be there if GPU kernels are processed in batches. Data allocation of the second kernel can happen while GPUs are processing the first kernel, which can be used in *Kaas* (Kernel-as-a-Service) [23] systems. Overlapping data allocation across kernels is a new data transfer model that can be used in future research.

To have a better illustration of how this new data transfer model works atop of *Async Memcpy* and *UVM*, we visualize the current model of the batch processing pipeline (top half) and the new data transfer model (bottom half)

4. Allocating more than 32KB shared memory requires dynamic allocations. Results in Figure 13 are inconsistent with results in previous sections since we used 32KB static allocated shared memory before.

in Figure 14. With the help of *UVM* and *Async Memcpy*, the CPU-GPU and global memory-shared memory data transfer can be overlapped with GPU kernel computation. The new data transfer model attempts to reduce overall execution time by minimizing the time the CPU and GPU spend idle. This is achieved through overlapping CPU and GPU execution for different job processes. Once the GPU kernel of job 1 begins running, job 2 is able to start data allocation (*cudaMallocManaged*()), utilizing the idle CPU. Once the GPU kernel of job 1 has finished execution, job 2 can run its own kernel, while job 1 proceeds with the data deallocation (*cudaFree*()) on the CPU. In the idealist case, the 37.66% data allocation (and deallocation) time can be overlapped with the 37.79% GPU kernel time. Therefore, an additional more than 30% performance improvement can be achieved if the new data transfer model is used, which is a promising future research direction.

7. Conclusion

In this paper, we investigate the performance implications of *Async Memcpy* and *UVM*. We conduct a deep characterization study by creating a benchmark suite, including 7 microbenchmarks and 14 real-workload applications. We believe the benchmark suite has the potential to enable further research in this domain. So, we plan to release our benchmark suite publicly.

We witness 21% performance gain on real-world applications when using *UVM*. With the help of *Async Memcpy*, the GPU computation can be pipelined with global memory to shared memory data transfer, which gives irregular programs, e.g., *kmeans* and *lud*, around 20% benefits atop of *UVM*. By breaking down the execution time, we give guidelines to software developers on making programming decisions. They could consider using both *UVM* (with *prefetch*) and *Async Memcpy* for GB-level memory-bounded applications. *UVM* (with *prefetch*) can make the performance benefit more if the workload follows regular data access patterns, while the data access pattern would not have an impact on *Async Memcpy*.

Furthermore, we perform a sensitivity study and make both software programmers and hardware architects aware that the *Async Memcpy* and *UVM* performance can be affected by the number of threads per block, and L1-cache/shared memory partition strategy. These parameters should be considered in the compiler and resource mapping designs. We also discuss future research directions that need contributions from both the system software and hardware architecture communities to further improve the data transfer pipelines.

Acknowledgement

We thank anonymous reviewers for their constructive feedback and insightful comments. This research was supported in part by NSF grant numbers 1763848 and 1828105, and gifts from Cisco and Amazon Web Services (AWS). The authors would also like to acknowledge the computing servers donated by Ampere and Nvidia, and the support from the iMAGINE Consortium, the ML Labs and the Texas Advanced Computing Center (TACC). Any opinions, findings, conclusions, or recommendations are those of the authors and not of the National Science Foundation or other sponsors.

References

[1] "Linux perf tool," 2023. [Online]. Available: https://perf.wiki.kernel.org/

[2] T. Allen and R. Ge, "Demystifying gpu uvm cost with deep runtime and workload analysis," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 141–150.

[3] T. Allen and R. Ge, "In-depth analyses of unified virtual memory system for gpu accelerated computing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2021, pp. 1–15.

[4] H. Anzt, Y. M. Tsai, A. Abdelfattah, T. Cojean, and J. Dongarra, "Evaluating the performance of nvidia's a100 ampere gpu for sparse and batched computations," in *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 2020, pp. 26–38.

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

[6] J. Choquette and W. Gandhi, "Nvidia a100 gpu: Performance & innovation for gpu computing," in *2020 IEEE Hot Chips 32 Symposium (HCS)*. IEEE Computer Society, 2020, pp. 1–43.

[7] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky, "Nvidia a100 tensor core gpu: Performance and innovation," *IEEE Micro*, vol. 41, no. 2, pp. 29–35, 2021.

[8] C. Gregg and K. Hazelwood, "Where is the data? why you cannot debate cpu vs. gpu performance without the answer," in *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 2011, pp. 134–144.

[9] Y. Gu, W. Wu, Y. Li, and L. Chen, "Uvmbench: A comprehensive benchmark suite for researching unified virtual memory in gpus," *arXiv preprint arXiv:2007.09822*, 2020.

[10] Y. Hao, N. Jain, R. Van der Wijngaart, N. Saxena, Y. Fan, and X. Liu, "Drgpu: A top-down profiler for gpu applications," in *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*, 2023, pp. 43–53.

[11] J. Hestness, S. W. Keckler, and D. A. Wood, "Gpu computing pipeline inefficiencies and optimization opportunities in heterogeneous cpu-gpu processors," in *2015 IEEE International Symposium on Workload Characterization*. IEEE, 2015, pp. 87–97.

[12] G. Huang, Y. Bai, L. Liu, Y. Wang, B. Yu, Y. Ding, and Y. Xie, "Alcop: Automatic load-compute pipelining in deep learning compiler for ai-gpus," *Proceedings of Machine Learning and Systems*, vol. 5, 2023.

[13] H. Kim, J. Sim, P. Gera, R. Hadidi, and H. Kim, "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*, 2020, pp. 1357–1370.

[14] J. B. Kotra, M. LeBeane, M. T. Kandemir, and G. H. Loh, "Increasing gpu translation reach by leveraging under-utilized on-chip resources," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1169–1181.

[15] J. Lee, J. M. Lee, Y. Oh, W. J. Song, and W. W. Ro, "Snakebyte: A tlb design with adaptive and recursive page merging in gpus," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 1195–1207.

[16] C. Lutz, S. Breß, S. Zeuch, T. Rabl, and V. Markl, "Pump up the volume: Processing large data on gpus with fast interconnects," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1633–1649.

[17] Nvidia, "Cupti," 2023. [Online]. Available: https://docs.nvidia.com/cuda/cupti/

[18] Nvidia, "Cutlass 3.0," 2023. [Online]. Available: https://github.com/NVIDIA/cutlass/

[19] Nvidia, "Nvidia a100 gpu architecture," https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, 2023.

[20] Nvidia, "Nvidia p100 gpu architecture," https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2023.

[21] Nvidia, "Nvidia v100 gpu architecture," https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2023.

[22] R. Panda, S. Song, J. Dean, and L. K. John, "Wait of a decade: Did spec cpu 2017 broaden the performance horizon?" in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 271–282.

[23] N. Pemberton, A. Zabreyko, Z. Ding, R. Katz, and J. Gonzalez, "Kernel-as-a-service: A serverless interface to gpus," *arXiv preprint arXiv:2212.08146*, 2022.

[24] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite," 2012. [Online]. Available: http://www.cs.ucla.edu/%7Epouchet/software/polybench

[25] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, "Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.

[26] J. Redmon, "Darknet: Open source neural networks in c," http://pjreddie.com/darknet/, 2013–2016.

[27] J. H. Ryoo, S. J. Quirem, M. Lebeane, R. Panda, S. Song, and L. K. John, "Gpgpu benchmark suites: How well do they sample the performance spectrum?" in *2015 44th International Conference on Parallel Processing*. IEEE, 2015, pp. 320–329.

[28] A. H. N. Sabet, Z. Zhao, and R. Gupta, "Subway: Minimizing data transfer during out-of-gpu-memory graph processing," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[29] C. Shao, J. Guo, P. Wang, J. Wang, C. Li, and M. Guo, "Oversubscribing gpu unified virtual memory: Implications and suggestions," in *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, 2022, pp. 67–75.

[30] M. Svedin, S. W. Chien, G. Chikafa, N. Jansson, and A. Podobas, "Benchmarking the nvidia gpu lineage: From early k80 to modern a100 with asynchronous memory transfers," in *Proceedings of the 11th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, 2021, pp. 1–6.

[31] Y. Wang, B. Feng, Z. Wang, T. Geng, K. Barker, A. Li, and Y. Ding, "Mgg: Accelerating graph neural networks with fine-grained intra-kernel communication-computation pipelining on multi-gpu platforms," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 779–795.

[32] K. Yoshida, R. Sageyama, S. Miwa, H. Yamaki, and H. Honda, "Analyzing performance and power-efficiency variations among nvidia gpus," in *Proceedings of the 51st International Conference on Parallel Processing*, 2022, pp. 1–12.

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

[34] K. Zhou, J. Anderson, X. Meng, and J. Mellor-Crummey, "Low overhead and context sensitive profiling of gpu-accelerated applications," in *Proceedings of the 36th ACM International Conference on Supercomputing*, 2022, pp. 1–13.

[35] W. Zhu, G. Cox, J. Vesely, M. Hairgrove, A. L. Cox, and S. Rixner, "Uvm discard: Eliminating redundant memory transfers for accelerators," in *2022 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2022, pp. 27–38.

Appendix

1. Abstract

This artifact appendix provides guidelines for reproducing the profiling results on the 5 different UVM and Async Memcpy setups, which are shown in Figure 4, 5, 6, 7, 8, 9, 10, 11, 12, 13.

2. Artifact check-list (meta-information)

- **Algorithm:** workloads list in Table 2.
- **Program:** Python, Nvidia CUDA, CUPTI, and Nsight Compute.
- **Run-time environment:** Linux x86_64 5.4.0-153-generic.
- **Data set:** Synthetic data (scripts for generating datasets included in the artifacts).
- **Hardware:** CPU: AMD EPYC 7742 64-Core Processor @2.90GHZ; GPU: NVIDIA A100.
- **Disk space required:** 100GB.
- **Time to prepare workflow (approximately):** About one hour to install related Python packages and NVIDIA Nsight Systems.
- **Time to profile (approximately):** 12 hours to profile all workloads for 30 iterations. 10 minutes for data post-processing.
- **Publicly available:** Yes.

3. Description

3.1. How to access. Our profiling results and benchmark source codes are available at Zenodo (https://zenodo.org/record/8222694) and GitHub (https://github.com/UT-LCA/UVMAsyncBench)

3.2. Hardware dependencies. The experiments are expected to run on machines with Nvidia GPUs no eariler than the Ampere architecture, with at least 64GB CPU memory and 32GB GPU memory. Our profiling results were collected on an a machine with AMD EPYC 7742 (1TB DRAM) and Nvidia A100 (40GB DRAM). Please expect difference when running on different hardware platforms.

3.3. Software dependencies. The experiments are expected to run on Linux machines, with GCC, CUDA, and Nsight Systems support. Our profiling results were collected on an a machine with GCC 9.4.0, CUDA 11.4, and NVIDIA Nsight Systems 2021.2.4.12. Please expect difference when using different software versions.

4. Installation

Please obtain the workloads using the Zenodo or Github link. Please install Python3 and GCC on your machine first. We recommend Python 3.8.10 and GCC 9.4.0. Our data parsing and visualization scripts do not require any Nvidia GPU related environment. To reproduce the profiling results, please get CUDA 11.4 from https://developer.nvidia.com/cuda-11-4-0-download-archive.

5. Experiment workflow

To reproduce all profiling results, please follow the three steps (you may skip this if you are only interested in parsing and visualizing the prepared profiling results).

Step 1: setup environment variable.

```
source env.sh
```

Step 2: profile microbenchmarks & collect performance counters (30 iterations).

```
cd workloads/micro/
python3 run_micro_all.py -i 30 --profiling
python3 run_micro_perf.py -i 30 --profiling
```

Step 3: profile real-world applications & collect performance counters (30 iterations).

```
cd workloads/realworld/
python3 run_real_all.py -i 30 --profiling
python3 run_real_perf.py -i 30 --profiling
```

6. Evaluation and expected results

Reproduce Figure 4, Figure 5, Figure 6, and Figure 7.

```
cd workloads/micro/
python3 run_micro_all.py -i 30
```

Reproduce Figure 8.

```
cd workloads/realworld/
python3 run_real_all.py -i 30
```

Reproduce Figure 9 and Figure 10.

```
cd workloads/
python3 process_perf.py -i 30
```

Reproduce Figure 11 and Figure 12.

```
cd workloads/micro/
python3 run_micro_sensitivity.py -i 30
```

Reproduce Figure 13.

```
cd workloads/micro/
python3 run_micro_shared.py -i 30
```

7. Experiment customization

You may use the *–i* input argument to change the number of profiling iterations. You may change the *run_∗.sh* script under each workload folder for different input sizes.