# Performance Implications of Pipelining the Data Transfer in CPU-GPU Heterogeneous Systems

RUIHAO LI, Electrical and Computer Engineering, The University of Texas at Austin, Austin, United States
BAGUS HANINDHITO, The University of Texas at Austin, Austin, United States
SANJANA YADAV, The University of Texas at Austin, Austin, United States
QINZHE WU, The University of Texas at Austin, Austin, United States
KRISHNA KAVI, University of North Texas, Denton, United States
GAYATRI MEHTA, University of North Texas, Denton, United States
NEERAJA J. YADWADKAR, The University of Texas at Austin, Austin, United States
LIZY K. JOHN, The University of Texas at Austin, Austin, United States

Driven by the increasing demands of machine learning, heterogeneous systems combining CPUs and GPUs have emerged as the dominant architecture for parallel computing in recent years. To optimize memory management and data transfer between CPUs and GPUs, Nvidia GPUs have introduced unified virtual memory (*UVM*) and pinned memory (*PM*) over the last decade. *UVM* can avoid explicit memory copies and potentially overlap GPU kernel computations with CPU-GPU data transfer. *PM* ensures that data with high locality remains in the main memory, preventing it from being paged out. In addition to these two techniques, asynchronous memory copy (*Async Memcpy*) was introduced recently in Nvidia GPUs to improve the CPU-GPU pipeline further. By utilizing *Async Memcpy*, the data transfer from GPU global memory to shared memory can be overlapped with GPU computations, adding an additional stage to the CPU-GPU data transfer pipeline. A thorough performance analysis of how *Async Memcpy* affects the current *UVM* and *PM* CPU-GPU data transfer scheme is desired.

Extension of Conference Paper [26]. We make the following major extensions: (1) In addition to unified virtual memory, we also evaluate how asynchronous memory copy affects the CPU-GPU pipeline when using pinned memory; (2) We include multiple machine system configurations and analyze their effect on the CPU-GPU heterogeneous system performance; (3) In addition to Nvidia A100, we include Nvidia H100 in our experiments to evaluate whether our findings are consistent with different GPU generations; (4) We include more machine learning (including training) workloads in our benchmark suite; (5) We present an in-depth quantitative analysis of how asynchronous memory copy impacts performance; (6) We prototype and evaluate the inter-job pipeline proposed in the conference version.
This research was supported in part by NSF grant numbers #2326894 and #2425655, the Texas ECE junior faculty start-up fund, UT iMAGiNE consortium, an award from the UT Machine Learning Lab (MLL), the AMD Chair Endowment, the Cisco Research Award, and the Amazon Research Award.
Authors' Contact Information: Ruihao Li, Electrical and Computer Engineering, The University of Texas at Austin, Austin, Texas, United States; e-mail: liruihao@utexas.edu; Bagus Hanindhito, The University of Texas at Austin, Austin, Texas, United States; e-mail: hanindhito@bagus.my.id; Sanjana Yadav, The University of Texas at Austin, Austin, Texas, United States; e-mail: sanjanayadav@utexas.edu; Qinzhe Wu, The University of Texas at Austin, Austin, Texas, United States; e-mail: qw2699@utexas.edu; Krishna Kavi, University of North Texas, Denton, Texas, United States; e-mail: Krishna.Kavi@unt.edu; Gayatri Mehta, University of North Texas, Denton, Texas, United States; e-mail: Gayatri.Mehta@unt.edu; Neeraja J. Yadwadkar, The University of Texas at Austin, Austin, Texas, United States; e-mail: neeraja@austin.utexas.edu; Lizy K. John, The University of Texas at Austin, Austin, Texas, United States; e-mail: ljohn@ece.utexas.edu.


ACM 1544-3973/2025/09-ART93
https://doi.org/10.1145/3746231

In this article, we provide performance implications of the combined effect of *UVM*, *PM*, and *Async Memcpy*, exploring which applications benefit from which combination of these features. We implement all these features on a suite of 25 workloads, including microbenchmarks and realworld applications. We observe an average performance gain of 24% when utilizing *UVM* and a 34% gain when employing *PM* on realworld applications, compared to not applying any data transfer optimization techniques. The performance benefits of *Async Memcpy* vary across different workloads. For workloads featuring extensive shared memory usage and high compute density (e.g., *kmeans* and *lud*), *Async Memcpy* delivers around a 20% performance improvement over using *UVM* or *PM* alone. In other workloads like *knn*, we note a 20% performance degradation when using *Async Memcpy*. Furthermore, we conduct an in-depth investigation of the GPU kernel using performance counters to uncover the root causes of performance differences among various data transfer models. We also perform sensitivity analyses to examine how the number of blocks and threads, as well as the L1-cache/shared memory partitioning, impact performance. We explore future research directions aimed at enhancing the data transfer pipeline by overlapping memory allocation with data transfer and computation across GPU kernels.

CCS Concepts: • **Computer systems organization → Parallel architectures**; **Heterogeneous (hybrid) systems;**

Additional Key Words and Phrases: GPU, async memcpy, unified virtual memory, pinned memory

**ACM Reference Format:**
Ruihao Li, Bagus Hanindhito, Sanjana Yadav, Qinzhe Wu, Krishna Kavi, Gayatri Mehta, Neeraja J. Yadwadkar, and Lizy K. John. 2025. Performance Implications of Pipelining the Data Transfer in CPU-GPU Heterogeneous Systems. *ACM Trans. Arch. Code Optim.* 22, 3, Article 93 (September 2025), 26 pages. https://doi.org/10.1145/3746231

## 1 Introduction

GPUs have become a key component in boosting the throughput of machine learning (ML) and big data workloads due to the massive parallelism they offer. With the exponential growth in data sizes, significant efforts over the past decade have focused on optimizing GPU architectures to enhance computational capabilities. For example, Nvidia introduced tensor cores in the Volta [39] architecture to exploit parallelism in ML tasks, and later incorporated fine-grained structured sparsity in the Ampere [36] architecture to enable efficient sparse matrix multiplication. Despite these advancements, GPUs still depend on CPUs to act as central controllers for workload distribution and data transfer. Thus, optimizing both CPUs and GPUs together as a unified heterogeneous system is essential for maximizing performance.

Recent studies have focused on reducing performance losses caused by data transfers between CPU and GPU systems. In memory-bound applications, data transfer times between the CPU and GPU can significantly surpass the time required for GPU kernel execution, by as much as 50 times [16]. With the advent of large language models, which have scaled from millions to billions of parameters in recent years [15, 52, 53], while GPU memory capacity has only grown incrementally, developing efficient CPU-GPU data transfer solutions has become increasingly critical. One approach to enhancing the performance of heterogeneous systems is to overlap CPU-GPU data transfer with GPU kernel processing, which has already been applied in different application domains, including graph processing [47], ML [44], and database systems [28]. Nvidia GPUs have introduced several features to mitigate data transfer overhead, including unified virtual memory (*UVM*), pinned memory (*PM*), and asynchronous memory copy (*Async Memcpy*) [36, 38]. In this work, we emphasize that the effectiveness of these optimizations can vary depending on the characteristics of the application, such as the computation density, which may not be immediately evident to users. We provide an in-depth analysis of the performance implications of *PM*, *UVM*, and *Async Memcpy* in modern Nvidia GPUs, exploring their impact across a wide range of workloads.

*PM* ensures that the memory allocated on the (CPU) host is mapped into the GPU address space, preventing it from being swapped out of main memory to storage devices [14, 30, 41]. *UVM* [31] creates a shared memory address space between CPU hosts and GPU devices, allowing both to access the same virtual memory. With the support of *PM* and *UVM*, GPUs can initiate data transfer precisely when the data is required for computation, rather than transferring all data before launching the GPU kernel. In addition to *PM* and *UVM*, Nvidia recently introduced an architectural feature in the Ampere architecture (CUDA 11) called *Async Memcpy* [36]. This feature enables data to be loaded directly from global memory into the shared memory of streaming multiprocessors (SMs). As its name implies, *Async Memcpy* operates in the background, allowing the SM to execute computation tasks concurrently.

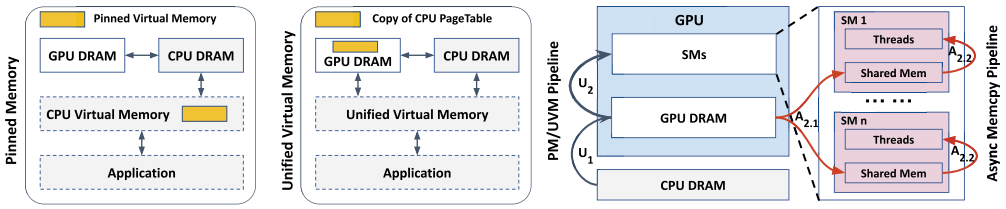
While there is a wealth of prior studies of *PM* [14, 30, 41], *UVM* [5, 6, 22, 59], and *Async Memcpy* [51] separately, the performance implications of combined effects of these mechanisms for a given workload remain unclear. Programmers can use *PM* to improve the CPU-GPU data transfer for frequently used data [41]. *UVM* has been explored deeply with studies focusing on various aspects including analyzing prefetchers and over-subscriptions, developing efficient page fault handlers, and reducing data movement [4, 6]. While many prior works have examined the sparse units and power consumption of the Ampere architecture [8, 56], few of them discussed *Async Memcpy*. Moreover, no previous work has investigated the intersection between *PM*, *UVM*, and *Async Memcpy*. It is important to analyze the performance of these three hardware features together, as all architectural enhancements come with their overheads.

The overall system performance cannot benefit from the three architectural enhancements if the overhead is not well handled. In *PM*, additional system calls are required to guarantee pinned pages physically reside in the main memory of the system [14, 30]. 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 [5, 6]. *Async Memcpy* complicates the data transfer pipeline because additional GPU resources are needed to coordinate the transfer from global memory to shared memory along with SM computation [51].

Despite the performance overhead, programmers also need to make a choice when writing their CUDA programs, i.e., whether to write a *PM*, *UVM*, or *Async Memcpy* version. In addition, there are no automatic tools such as compilers available for converting programs to *PM*, *UVM*, or *Async Memcpy* versions. Software developers need to hand-tune the CUDA programs for better performance, making a design guideline for these three architectural features more desirable. Answers to questions such as the following can help programmers in tuning their applications. (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 *PM* or *UVM*, i.e., the bottlenecked by the CPU DRAM to GPU global memory transfer? (b) What are the performance implications for the choice between *PM*, *UVM*, and *Async Memcpy*? Are these implications workload-agnostic? How should programmers make these choices? (c) Would the overall performance improve further if we use both *PM*/*UVM* and *Async Memcpy*?[1] Can programmers make the decision with limited profiling or intensive profiling?

To answer these questions, in this article, we make in-depth analysis of how *PM*, *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 three architectural features together. The following are the key contributions of our work.

[1] *Async Memcpy* can be used together with either *PM* or *UVM*. However, *PM* and *UVM* cannot be used together.




(a) PM and UVM data transfer. (b) CPU-GPU system data transfer pipeline.

Fig. 1. With *PM*/*UVM*, the pipeline contains $U_1$ and $U_2$ stages. Adding *Async Memcpy* atop of *PM*/*UVM*, stage $U_2$ can be pipelined into $A_{2.1}$ and $A_{2.2}$.

— We explore the performance implications of CUDA programming choices for data transfer (*PM*, *UVM*, and/or *Async Memcpy*). We break down the execution time into GPU kernel time, data transfer time, and data allocation time on 25 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 *PM*, *UVM*, and *Async Memcpy*.
— Our analysis on *PM*, *UVM*, and *Async Memcpy* can help CUDA programmers understand these three hardware features better and develop more efficient GPU codes. A good usage of these features can help programs achieve up to 6× speedups, while an inappropriate usage may lead to up to 3× slow downs on certain workloads.
— We create and make available a benchmark suite for the three architectural features and different combinations, including 7 microbenchmarks and 18 realworld applications, which cover multiple domains. We implement the *PM*, *UVM*, and *Async Memcpy* versions of each workload that were not available already. We believe that releasing this benchmark suite publicly will enable further research in this domain.[2]

## 2 Background and Related Work

Figure 1 summarizes state-of-the-art CPU-GPU heterogeneous system memory architectures with *PM*, *UVM*, and *Async Memcpy*. We describe details about *PM* and *UVM* in Section 2.1, *Async Memcpy* in Section 2.2, and performance implications of combing these architectural features in Section 2.3. We also discuss prior performance characterization studies on *PM*, *UVM*, and *Async Memcpy* in this section.

### 2.1 Pinned Memory and Unified Virtual Memory

We describe details about *PM* in Section 2.1.1, *UVM* in Section 2.1.2, and how these two architectural features affect CPU-GPU data transfer pipeline in Section 2.1.3.

*2.1.1 Pinned Memory.* *PM* is allocated in a dedicated memory space that physically resides in the main memory of the system. PM is allocated using *cudaMallocHost* interface, as illustrated in Figure 2(a).

Nvidia *PM* ensures that the "pinned" memory regions allocated on the (CPU) host are mapped into the GPU address space, preventing it from being swapped out of the main memory [14, 30, 41]. This is important for achieving high-performance data transfer between CPUs and GPUs, as PM eliminates the overhead associated with paging memory in and out of the virtual memory space of the system. However, "pinned" memory can introduce additional overhead, such as size constraints

[2]Codes are available at https://github.com/UT-LCA/UVMAsyncBench

**Without PM/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);
```

**With PM**

```
DataType *h_a, *d_a;
cudaMallocHost(&h_a, size);
cudaMalloc(&d_a, size);
cudaMemcpyAsync(d_a, host_a, size, cudaMemcpyHostToDevice);
cudaKernel<<<...>>>(d_a);
cudaMemcpyAsync(host_a, host_a, size, cudaMemcpyDeviceToHost);
```

**With UVM**

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

(a) Without/with PM/UVM (Pseudo-code).

**Without Async Memcpy**

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

**With Async Memcpy**

```
__shared__ DataType data[size * 2];
for (; tile < end; tile++) {
     f = (tile + 1) % 2;
     c = tile % 2;
     memcpy_async(data[f*size:(f+1)*size], input[tile]);
     compute on data[c*size:(c+1)*size];
}
```

(b) Without/with Async Memcpy (Pseudo-code).

Fig. 2. CUDA programming Pseudo-code of PM, UVM, and Async Memcpy.

and allocation costs, which may reduce performance [3, 14]. Considering the trade-offs of using *PM*, state-of-the-art research falls into two major directions:

(1) **Using *PM* Efficiently**. Prior works [9, 20, 54] applied PM to store frequently used data and pipelines the CPU to GPU data transfer with GPU processing, which helps improve the performance of scientific computing, network processing, and ML systems.
(2) **Reducing the PM Capacity Requirement**. PM is non-pageable and hence it cannot be allocated by the CPU to other processes. Prior work [49] showed the feasibility of compressing data using CPU cores, which helps reduce PM consumption.

*2.1.2 Unified Virtual Memory.* *UVM* is a powerful technology introduced with the Kepler architecture [31], which provides a unified memory space and automates memory management and data migration between the physical memory modules of the CPU host and GPU device.

*UVM* is designed to be transparent to applications, making CUDA programming simpler and more intuitive (see Figure 2(a)). By offering a unified virtual memory space, UVM enables applications to effectively leverage the combined memory resources of multiple GPUs for data-intensive tasks, such as ML and high-performance computing [4]. However, *UVM* introduces performance overhead because GPUs must maintain a copy of the CPU page table for address translation and synchronize GPU page faults with the CPU. These trade-offs have attracted significant attention in the research community, leading to two main areas of focus:

(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 [22], improving GPU cache utilization [23], and dynamically managing variable-sized pages [25]. 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. [58] compared *UVM* and traditional memory management methodology and found the possibility of using *UVM* with minimal overhead. Allen et al. [4–6] dived into the software and hardware-based root causes of the internal behaviors of page fault generation and servicing. Shao et al. [48] revealed the reasons behind the diverse sensitivities to oversubscription among different workloads.

*2.1.3 How PM and UVM Affect Data Transfer.* In addition to improving data locality and programmability, *PM* and *UVM* also affect the CPU-GPU data transfer pipeline. As shown in Figure 1,

when using *PM* or *UVM*, the CPU DRAM to GPU global memory data transfer ($U_1$) can run in parallel with global memory to shared memory data transfer and SM execution ($U_2$).[3] Such CPU-GPU data transfer pipelines are also compatible with other intra-GPU data transfer pipelines, e.g., *Async Memcpy*, with more details in the following sections.

### 2.2 Asynchronous Memcpy

Pipelining computation and data transfer can improve the CPU-GPU heterogeneous system performance. This technique has already been used in the *PM* and *UVM* systems, 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 *Async Memcpy* of data from global memory to shared memory. In Figure 1, for *Async Memcpy*[4], copying data from global memory to shared memory is marked as $A_{2.1}$, and fetching data from shared memory during processing is marked $A_{2.2}$. *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 2(b)). Additional primitives are provided to enable waiting for the completion of Async Memcpy operations.

Though many works studied the Ampere architecture before, the majority of the works focused on the sparse units [12, 13] and power efficiency [56], 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 (DL) compilers recently to optimize the pipeline of tensor programs [21, 55].
(2) **Characterization and Analysis** that study the performance of *Async Memcpy* and compare it against its predecessor architecture. For example, Svedin et al. [51] 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 PM/UVM vs Async Memcpy

As shown in Figure 1, *Async Memcpy* can be used in conjunction with either *PM* or *UVM*. These architectural features work together to establish a three-stage data transfer pipeline in CPU-GPU heterogeneous systems: (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 ($A_{2.2}$).

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 *PM* or *UVM* can boost GPU system performance more is worth exploring, considering the sophisticated three-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 25 workloads in the benchmark suite we created. Lastly, we discuss how to

[3] An enhanced *UVM* version supports *prefetch* data from global memory to L2 cache [5, 22], reducing the global memory to shared memory time.
[4] In this article *Async Memcpy* only refers to asynchronous copying of data from global memory to shared memory.

Table 1. Machine Configurations of the four Systems Used in the Study

| | *Machine-A* | *Machine-B* | *Machine-C* | *Machine-D* |
|---|---|---|---|---|
| **CPU** | 20× Intel Xeon Gold 6138 @ 2.0 GHz<br>32 KB L1-dcache, 32 KB L1-icache<br>1 MB L2-cache, 1.375 MB L3-cache/core | 64× AMD EPYC 7763 @ 2.45GHz<br>32 KB L1-dcache, 32 KB L1-icache<br>512 KB L2-cache, 4 MB L3-cache/core | 52× Intel Xeon Platinum 8470 @ 2.0 GHz<br>48 KB L1-dcache, 32 KB L1-icache<br>2 MB L2-cache, 1.875 MB L3-cache/core | 48× AMD EPYC 9454 @ 2.75 GHz<br>32 KB L1-dcache, 32 KB L1-icache<br>1 MB L2-cache, 4 MB L3-cache/core |
| **DRAM** | 12× 16 GB DDR4 @ 2666 MT/s | 16× 16 GB DDR4 @ 3200 MT/s | 16× 64 GB DDR5 @ 4400 MT/s | 24× 16 GB DDR5 @ 4800 MT/s |
| **PCIe** | System PCIe Gen 3.0<br>PCIe to GPU Gen 3.0 | System PCIe Gen 4.0<br>PCIe to GPU Gen 4.0 | System PCIe Gen 5.0<br>PCIe to GPU Gen 4.0 | System PCIe Gen 5.0<br>PCIe to GPU Gen 5.0 |
| **GPU** | Nvidia Tesla **A100** @ 1410 MHz<br>80 GB HBM2e @ 1512 MHz | Nvidia Tesla **A100** @ 1410 MHz<br>40 GB HBM2 @ 1215 MHz | Nvidia Tesla **A100** @ 1410 MHz<br>80 GB HBM2e @ 1593 MHz | Nvidia Tesla **H100** @ 1620 MHz<br>80 GB HBM2e @ 1593 MHz |
| **Software** | Ubuntu 22.04, Linux kernel 5.15<br>GCC 11, CUDA 12 | Rocky 8.6, Linux kernel 4.18<br>GCC 11, CUDA 12 | Ubuntu 22.04, Linux kernel 5.15<br>GCC 11, CUDA 12 | Rocky 8.6, Linux kernel 4.18<br>GCC 11, CUDA 12 |

determine the configuration of each workload since the performance can be affected when using different input sizes and machine system configurations.

### 3.1 Experimental Setup

*3.1.1 Machine Configurations.* We conduct our characterization study on multiple Nvidia A100 servers with varied machine system configurations (*Machine-A*, *Machine-B*, and *Machine-C*), and on an Nvidia H100 server (*Machine-D*). The hardware and software configurations for all four machines are listed in Table 1.

*3.1.2 Software Configurations.* We use Nvidia Nsight Compute [34], Nsight Systems [35], and CUPTI [32] for performance counter collections. We use the CUDA *Pipeline* API for the *Async Memcpy* implementation since it showed better performance than *Arrive/Wait Barriers* [51].

*3.1.3 PM, UVM and Async Memcpy Configurations.* We use the following six configurations in our experiments:

(a) **baseline** – without *PM*, *UVM*, or *Async Memcpy*,
(b) **async** – using *Async Memcpy* only,
(c) **uvm** – using *UVM* (with *prefetch*[5]) only,
(d) **uvm_async** – using *UVM* and *Async Memcpy*,
(e) **pm** – using *PM* only, and
(f) **pm_async** – using *PM* and *Async Memcpy*.

### 3.2 Overview of Benchmarks

In our performance studies, we use 18 realworld applications and 7 microbenchmarks, which are categorized into three groups as shown in Table 2. These 25 workloads cover the domain of linear algebra, physics simulation, data mining, image processing, and DL. 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 *PM*, *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 [51]. In addition to Vector-to-Constant, we include five additional microbenchmarks from Polybench [43].[6] Vector-to-Vector (*saxpy*), Matrix-to-Vector (*gemv*), and Matrix-to-Matrix (*gemm*) are considered as extensions to the two Vector-to-Constant workloads,

[5]Our prior analysis proofed that prefetcher can improve the *UVM* performance[26]. If not specified, we assume *UVM* is used with prefetcher in this article.
[6]We adjusted the Polybench codes to make them scalable for large input sizes. We also compared the performance of our own implementation with cutlass [33] to guarantee the efficacy of our kernel implementations.

Table 2. Benchmark Programs with Input Information (Contains 25 Representative Workloads from five Diverse Suites)

| Suites | Source | Program Name | Description | Input Dimension | Super Input | Mega Input | Other Input Info |
|---|---|---|---|---|---|---|---|
| Micro | Svedin et al. [51] | vector_seq | Vector-to-Constant, element-wise arithmetic operations on vector (sequential access) | Vector (1D) | 1024M | 8192M | 1 vector |
| | | vector_rand | Vector-to-Constant, element-wise arithmetic operations on vector (random access) | Vector (1D) | 1024M | 8192M | 1 vector |
| | PolyBench [43] | saxpy | Vector-to-Vector multiplication and addition | Vector (1D) | 1024M | 4096M | 2 vectors |
| | | gemv | general Matrix-to-Vector multiplication | Matrix (2D) | 32K * 32K | 64K * 64K | 1 vector + 1 matrix |
| | | gemm | general Matrix-to-Matrix multiplication | Matrix (2D) | 16K * 16K | 32K * 32K | 2 matrices |
| | | 2DCONV | general 2D convolution | Grid (2D) | 32K * 32K | 64K * 64K | 3*3 Kernel + 1 2D-matrix |
| | | 3DCONV | general 3D convolution | Grid (3D) | 0.75K * 0.75K * 0.75K | 1.5K * 1.5K * 1.5K | 3*3*3 Kernel + 1 3D-matrix |
| Apps (non-DL) | Rodinia [11] | LavaMD | Calculates particle potential and relocation due to mutual forces between particles within a 3D space | Box (3D) | 64 * 64 * 64 | 128 * 128 * 128 | 32 blocks, 100 elements per block |
| | | NW | Needleman-Wunsch, a nonlinear global optimization method for DNA sequence alignments | Sequence (2D) | 20K * 20K | 40K * 40K | 2 matrices |
| | | Kmeans | An unsupervised ML algorithm used for clustering data | Point Vector (1D) | 8192K | 16384K | 2 point vectors, each point has 64 features |
| | | Srad | Speckle Reducing Anisotropic Diffusion is a method for ultrasonic and radar imaging applications | Grid (2D) | 20K * 20K | 40K * 40K | 6 matrices |
| | | Backprop | An ML algorithm that trains the weights of connecting nodes on a layered neural network | Node Vector (1D) | 32M | 64M | 1 node vector, each node has 16 neurons |
| | | Pathfinder | Solves the shortest or optimal path problem through a 2D grid | Grid (2D) | 5M * 100 | 20M * 100 | 1 2D-grid + 2 1D-grid |
| | | HotSpot | Simulates the thermal behavior of integrated circuits | Grid (2D) | 8K * 8K | 12K * 12K | 3 2D-grids |
| | | LUD | LU Decomposition is an algorithm that calculates the solutions of a set of linear equations | Grid (2D) | 16K * 16K | 64K * 64K | 1 2D-grid |
| | UVMBench [17] | bayesian | Bayesian network learning algorithm | Node Matrix (2D) | 50*250K | 125*10M | 1 matrix |
| | | KNN | K-Nearest neighbors algorithm | Distance Grid (2D), Point Grid (2D) | 16K * 16K, 16K * 128 | 32K * 32K, 32K * 512 | 1 distance 2D-grid + 2 point 2D-grid |
| Apps (DL) | Darknet [45] | Resnet18 | Residual Network with 18 convolution layers | Image | Batch Size 1 | Batch Size 32 | Image Size 224 * 224 * 3 |
| | | Resnet50 | Residual Network with 50 convolution layers | Image | Batch Size 1 | Batch Size 32 | Image Size 224 * 224 * 3 |
| | | Yolov3-tiny | Yolov3-tiny | Image | Batch Size 1 | Batch Size 4 | Image Size 416 * 416 * 3 |
| | | Yolov3 | Yolov3 | Image | Batch Size 1 | Batch Size 4 | Image Size 416 * 416 * 3 |

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 realworld application suite includes 10 non-deep learning (non-DL) workloads and 8 DL workloads (we consider training and inference as separate workloads).

**Non-DL workloads.** Our benchmark suite includes the widely used *Rodinia* [11] 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, eight diverse benchmarks are selected. We select *lavaMD*, *NW*, *Kmeans*, *Srad*, *Backprop*, and *Pathfinder* based on the representativeness of the six workloads. They were classified into different groups based on prior performance characterization studies [46]. We also include *HostSpot* and *LUD*, since they were used in prior *Async Memcpy* studies [51].

We use workloads *bayesian* and *KNN* from *Uvmbench* [17], 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.

**DL workloads.** We also study DL workloads since they are widely used in CPU-GPU heterogeneous systems, especially in the last decade. Instead of single kernels, we choose end-to-end DL network profiling. We consider both the inference and training phases of the four ML models. We

Table 3. Parameter Configurations

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

We focus on *Super* and *Mega* inputs in this article.

choose *darknet*[7] as the ML framework since it is implemented in C rather than Python, making it easier to customize CUDA kernels with the three architectural enhancements. We implement the *PM*, *UVM*, and *Async Memcpy* versions of *darknet*.

### 3.3 Benchmark and Machine Configurations

Benchmark performance highly depends on the chosen machine and workload configurations. In this section, we examine how the workload input size and system configuration affect benchmarking outcomes, and we describe our approach to selecting the appropriate configuration and input size. Specifically, Table 1 details the machine configurations, while Table 3 defines six input sizes ranging from 1 MB to 32 GB. Furthermore, Table 2 provides detailed information on the *super* and *mega* input sizes, which are the primary focus of the subsequent sections.

To ensure sufficient execution time to amortize system overhead, the program input size must be large enough. While large inputs are commonly used in benchmarks to reduce measurement noise, prior work [26] shows that bigger inputs do not always yield stable performance in CPU-GPU heterogeneous systems. In this article, we extend our previous investigation by deploying the same set of microbenchmarks on four additional machines (Table 1). We use *Machine-B*, featuring AMD's Milan (Zen 3) chiplet architecture [29], and *Machine-D*, which adopts AMD's Genoa (Zen 4) chiplet architecture [10]. While AMD's Infinity Fabric technology remains unchanged from Rome to Milan [29], Genoa leverages Infinity Fabric 3.0 with GMI3 interface support for more stable and predictable cross-chiplet data transfers [10]. To cross-validate our findings, we also include *Machine-A* with an Intel Skylake CPU (monolithic) and *Machine-C* with an Intel Sapphire Rapids CPU (chiplet).

We ran each workload 30 times and plotted the distribution of each run.[8] Figure 3 presents the overall execution time distribution for the seven workloads in our microbenchmark suite under three different input sizes. On *Machine-C* and *Machine-D*, performance remains more stable across different input sizes because these newer systems incorporate updated CPUs, DRAM, and PCIe generations, yielding higher CPU-DRAM and CPU-GPU PCIe bandwidth. This reduces congestion and delivers more consistent performance overall. As a result, we choose to mainly focus on *Machine-C* and *Machine-D* with input sizes *Super* and *Mega* for the rest of our experiments.

**Takeaway 1:** To ensure that benchmarking results are broadly applicable to CPU-GPU heterogeneous systems, it is important to select both input sizes and machine configurations carefully.

[7]Instead of using the *CNN* and other ML workloads in *Uvmbench*, we choose workloads (networks) in *darknet* [45], since *darknet* implemented both inference and training for end-to-end network.

[8]We verified our findings under both specified and unspecified NUMA node configurations. In each case, results remained consistent because we focus on a single-GPU scenario, and all program memory footprints lie well within the capacity of a single NUMA node.

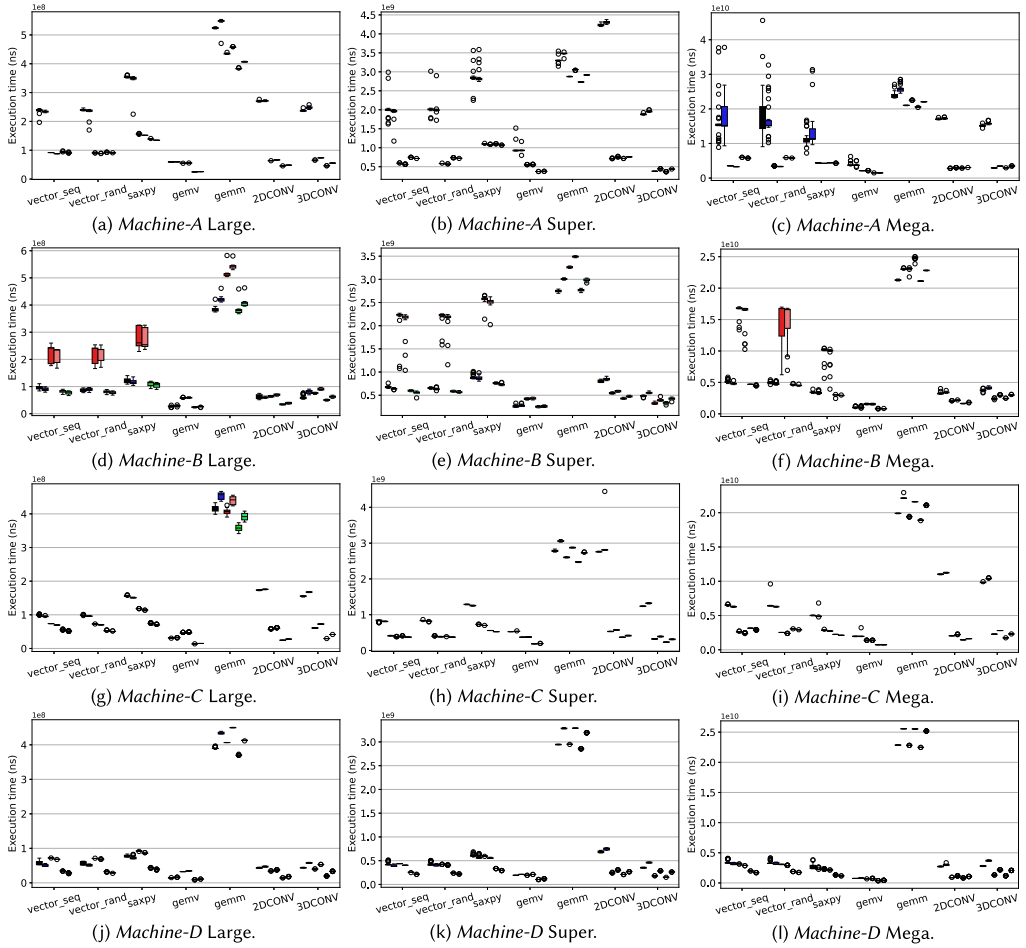


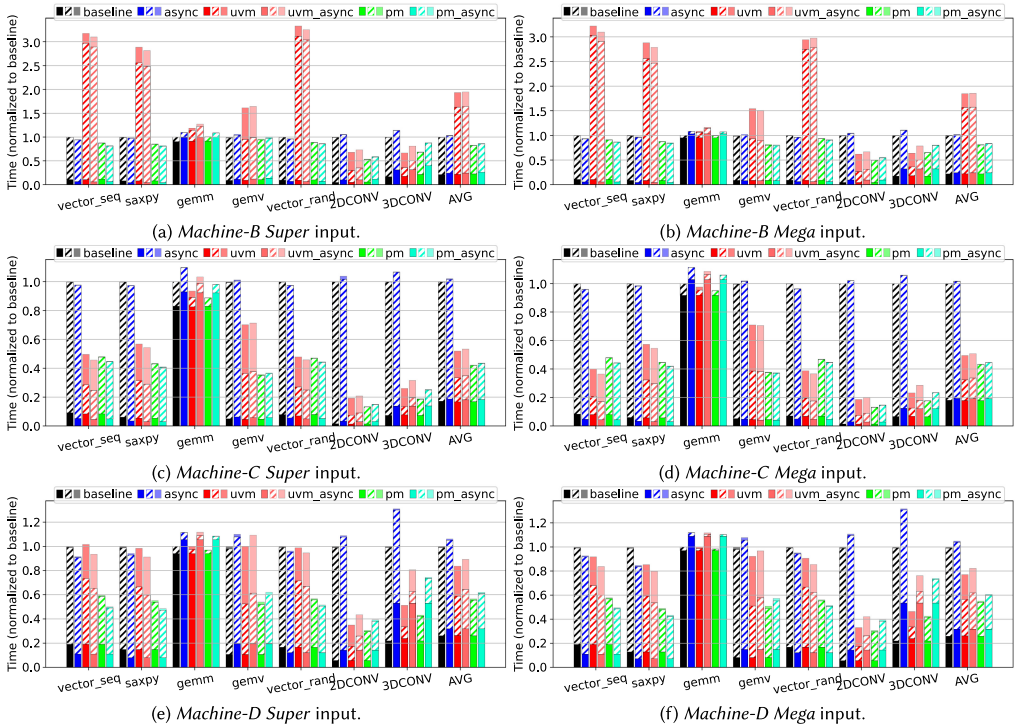

Fig. 3. Execution time (distribution of 30 runs) on microbenchmarks with different input sizes. For each workload, from left to right, the six bars indicate the six data transfer configurations (in the same order as Section 3.1.3). *Machine-C* and *Machine-D* are more stable than *Machine-A* and *Machine-B* for all input sizes.

Systems featuring newer CPU, DRAM, and PCIe generations with larger (GB-level) input sizes typically yield less noisy results.

## 4 Performance Results

In this section, we provide a side-by-side comparison of the six setups across all benchmarks, breaking down the overall execution time into GPU kernel execution, CPU-GPU data transfer, and data allocation time.[9] We analyze the performance of microbenchmarks (Section 4.1) and realworld applications (Section 4.2 and Section 4.3) for various configurations of *Async Memcpy*, *UVM*, and *PM*. This comparison helps determine whether a workload is limited by CPU DRAM to GPU global memory data transfers or by internal GPU data transfers (from global memory to shared memory), and what architectural configuration could help improve the performance.

[9]If CPU-GPU data transfers overlap with GPU kernel execution, e.g., in *UVM*, we determine the GPU kernel time by subtracting the CPU-GPU data transfer time from the overall kernel time.




(a) *Machine-B Super* input. (b) *Machine-B Mega* input.

(c) *Machine-C Super* input. (d) *Machine-C Mega* input.

(e) *Machine-D Super* input. (f) *Machine-D Mega* input.

Fig. 4. Comparisons on Microbenchmarks (using latency as the metric, the smaller the better). From bottom (darkest) to top (lightest), each shade shows gpu_kernel, memcpy, and allocation. The performance of a workload can benefit from using a combination of *Async Memcpy* and *UVM/PM* if it benefits from using each technique individually.

### 4.1 Microbenchmarks

In this section, we present a performance comparison across six different configurations for seven microbenchmarks. We use both *Super* and *Mega* input sizes, as they exhibit relatively stable performance across multiple runs (on *Machine-C* and *Machine-D*). Additionally, we include *Machine-B*[10] to analyze how the machine system configurations impact the performance of *UVM* and *PM*. We analyze the average performance from 30 runs and provide a side-by-side comparison of the seven microbenchmarks, as shown in Figure 4. We first examine the individual effect of *Async Memcpy*, *UVM*, and *PM*, separately. Following that, we explore the combined effects of *Async Memcpy* with *UVM* and *PM*.

***Async Memcpy:*** When considering the overall execution time, there is almost no performance difference between *baseline* and *async*. Considering the average of the seven workloads, by using *async* only, there is a 2.05% and 1.72% slow down compared with *baseline* on *Super* and *Mega* when using *Machine-C*. When using *Machine-D*, similarly, there is a 6.12% and 4.91% slow down compared with *baseline* on *Super* and *Mega* input sizes. However, the performance difference (between *async* and *baseline*) is much more noticeable when comparing the pure GPU kernel time. For example, *async* achieves 42.42% GPU kernel time reduction over *baseline* in the *vector_seq* workload

[10] Although Section 3.3 highlights some outlier data points on *Machine-B*, the performance impact of *UVM* and *PM* clearly exceeds the noise.

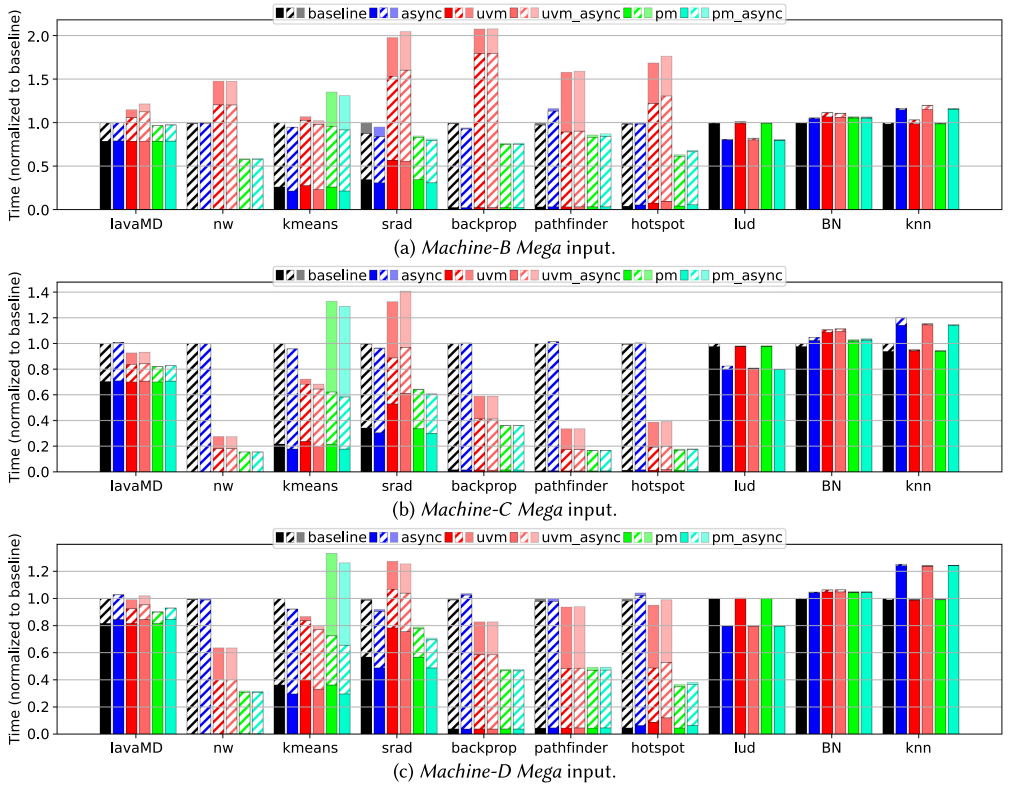
while 143.37% increment over *baseline* in the 3*DCONV* workload (both with *Mega* input sizes on *Machine-D*). For CPU-GPU data transfer bounded workloads (like *vector_seq*), the 42.42% kernel time reduction only results in 7.37% overall performance improvement. For GPU kernel bounded workloads (like 3*DCONV*), the overhead due to the additional pipeline stages in *async* leads to significant performance degradation.

***UVM/PM:*** *UVM* is designed to provide a simplified programming interface for automating the CPU-GPU data transfer. With an effective prefetcher, the GPU computation can be pipelined well with CPU-GPU data transfer. Overall, When using *UVM* (with *prefetch*), there is 48.13% and 50.53% performance improvement over *baseline* on *Machine-C* for *Super* and *Mega* input sizes, respectively. For GPU kernel and CPU-GPU data transfer time, *UVM* achieves 79.90% and 82.24% time savings over *baseline* when using *Super* and *Mega* input sizes on *Machine-C*. However, the ease of programming provided by *UVM* does not always guarantee a performance improvement [4, 6, 48]. An interesting observation is that the performance of *UVM* is extremely sensitive to the machine system configuration. We conduct an A/B test by comparing the *UVM* performance between *Machine-C* and *Machine-B*, as they have the same GPU model but different machine system configurations. There is a 48.13% and 51.53% performance improvement on *Machine-C* compared with using *UVM* with *baseline* for *Super* and *Mega* input sizes, respectively. However, on *Machine-B*, there is a 93.43% and 84.67% slowdown when comparing *UVM* with *stanard* for *Super* and *Mega* input sizes.

*PM* makes the operating system ensure that the corresponding physical pages remain in the main memory rather than being paged to storage devices. If the allocation *PM* regions with high locality, the CPU-GPU data transfer time can be reduced significantly by utilizing DMA data transfer [19]. Overall, When using *PM*, there is 57.82% and 56.63% performance improvement over *baseline* on *Machine-C* for *Super* and *Mega* input sizes, respectively. However, the performance improvement of *PM* (over *baseline*) is mostly affected by the machine system configuration. On *Machine-B*, the performance improvement of using *PM* (over *baseline*) is only 16.99% and 18.56%. *Machine-D* sits in the middle between *Machine-B* and *Machine-C*, the performance improvement of using *PM* (over *baseline*) is 43.43% and 45.02% for *Super* and *Mega* input sizes, respectively.

***Async Memcpy+UVM/PM:*** It is also worth investigating the potential for utilizing *Async Memcpy* in conjunction with *UVM* or *PM*. We introduce our final two configurations, labeled *uvm_async* and *pm_async*, which combine two architectural features. For workloads whose performance cannot benefit from additional pipeline stages due to inefficient shared memory usage, such as *2DCONV*, *3DCONV*, and *gemm*, the extra control logic overhead of *Async Memcpy* can degrade performance (examined further in Section 5.1). Such overhead results in the average performance improvement of *uvm_async* over *baseline* is 49.21% for the *Mega* input size on *Machine-C*, which is slightly lower than *uvm* over *baseline* (50.53%). Similarly, the performance improvement of *pm_async* is also slightly worse than *async* only (55.28% versus 56.63% for the *Mega* input size on *Machine-C*). However, that does not mean *Async Memcpy* is not compatible with *PM/UVM* in all scenarios. For workloads *vector_seq* and *vector_rand*, *uvm_async* can make the time savings over *baseline* of the two workloads to 63.50% and 63.37%, which is better than the 59.98% and 61.25% time savings over *baseline* when using *uvm* (for the *Mega* input size on *Machine-C*).

**Takeaway 2:** (1) The performance of *UVM* and *PM* is sensitive to machine system configurations. Generally, *PM* can reduce CPU-GPU data transfer time, whereas *UVM* may not offer the same advantage. (2) If *Async Memcpy* brings performance overhead to a workload, such overhead still exists when using *Async Memcpy* atop *UVM/PM*, e.g., *3DCONV*. (3) The performance of a workload can benefit from using a combination of *Async Memcpy* and *UVM/PM* if it benefits from using each technique individually, e.g., *vector_seq* and *saxpy*.




(a) *Machine-B Mega* input.

(b) *Machine-C Mega* input.

(c) *Machine-D Mega* input.

Fig. 5. Comparisons on realworld (non-DL) applications (using time as the metric, the smaller the better). From bottom (darkest) to top (lightest), each shade shows gpu_kernel, memcpy, and allocation. The performance of *UVM* and *PM* is also sensitive to the type of machine system configurations (same as the case of microbenmarks). The performance of *Async Memcpy* is more sensitive to the type of workload, e.g., benefits the performance of *lud* but not *knn*.

### 4.2 Realworld non Deep Learning Applications

In addition to microbenchmarks, we show the execution time breakdown of the 10 realworld applications as well. We use the average of 30 runs and *Mega* input sizes, shown in Figure 5.

***UVM/PM:*** The effect of machine system configurations still exists in realworld applications. When using *Machine-B*, there is a 41.68% slow down (considering the geo-mean of the 10 workloads) when using *UVM* compared with *baseline* for *Mega* input size. When using machines with other system configurations, *UVM* can improve performance. There is 23.99% and 4.61% performance improvement achieved by using *UVM* compared with using *baseline* on *Machine-C* and *Machine-D*, respectively. *PM* also achieves 33.90% and 22.80% performance improvement over *baseline* on *Machine-C* and *Machine-D*, respectively. Most of the speedups come from a reduction of CPU-GPU data transfer time. *UVM* achieves 67.35% and 48.03% memcpy time savings compared with *baseline* on *Machine-C* and *Machine-D*. Similarly, *PM* achieves 71.39% and 58.04% memcpy time savings compared with *baseline* on the two machines.

***Async Memcpy:*** Similar to microbenchmarks, using *Async Memcpy* alone does not significantly impact overall performance. On average, for the ten workloads tested, *Async Memcpy* results in

only a 0.35% slowdown and a 0.43% speedup compared to the *baseline* configuration for the Mega input size on *Machine-C* and *Machine-D*, respectively.

Among the ten workloads, the performance of *lud* and *knn* is most affected by *Async Memcpy*. There is a 20.03% performance improvement over *baseline* when using *Async Memcpy* on *lud*, while a 25.22% slowdown on *knn* (both results on *Machine-D*). This is because *lud* has a higher computation density than *knn*, providing more opportunity for an additional data transfer pipeline stage. Similar to *knn*, the microbenchmark *gemm* also has similar characteristics, we will make further sensitive analysis in Section 5.1.

***Async Memcpy+UVM/PM:*** In realworld applications, *Async Memcpy* together with *UVM/PM* can bring additional performance benefits if the performance can benefit from using *Async Memcpy*. *Async Memcpy* pipelines the computation and global memory to shared memory data transfer, so *uvm_async* can save GPU kernel time compared with *uvm*. For instance, the *uvm_async* saves 8.38% GPU kernel time over *baseline* for workload *kmeans* on *Machine-D*, while *uvm* spends 9.94% more GPU kernel time over *baseline*.

Although *uvm_async* generally outperforms *uvm* when standalone *async* usage yields performance gains, there are exceptions, such as *srad* on *Machine-C*. The reason is that the performance of *Async Memcpy* is highly sensitive to the amount of shared memory access (details in Section 5.1). Enabling the prefetcher (with *UVM*) can incur additional shared memory traffic, making potential performance degradation when using *Async Memcpy*. In addition, *Async Memcpy* also requires additional control instructions, making the performance even worse (on *Machine-B* and *Machine-C*). There is another interesting data point *lud*. For *lud*, the performance only benefits from *Async Memcpy* but not *UVM* (with *prefetch*). Because CPU–GPU data transfers for *lud* are negligible, the dominant time instead comes from moving data between GPU global memory and shared memory. When combining these two techniques, *lud* maintains the same speedup as *Async Memcpy* only; it is not affected by *UVM* overhead.

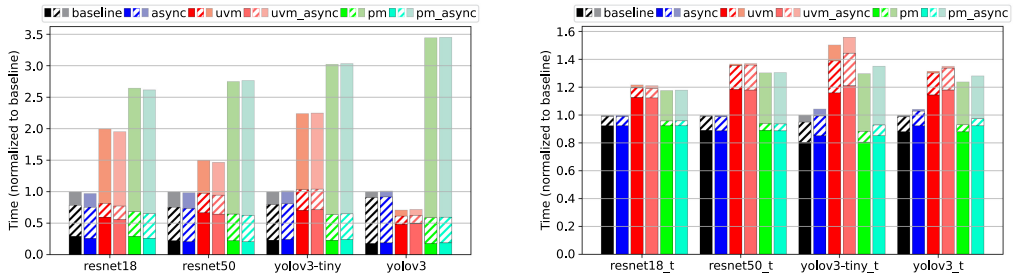
**Takeaway 3:** (1) *UVM* and *PM* are affected by both types of machine configurations and workload, while *Async Memcpy* is mostly affected by workload characteristics. (2) Workloads with predictable data access patterns and bounded by CPU-GPU data transfer, e.g., *2DCONV*, can benefit more from *UVM* and *PM* (up to 4.96× speedups over *baseline*).

### 4.3 RealWorld Deep Learning Applications

We also assess the impact of *UVM*, *PM*, and *Async Memcpy* on DL applications. The results using *Mega* input sizes on *Machine-D* are presented in Figure 6 (similar patterns were observed with *Super* input sizes and on the other two machines). Overall, *Async Memcpy* has minimal effect on the performance of these DL workloads, while *UVM* and *PM* significantly influence performance.

*PM* cannot improve the performance of either inference or training workloads. On average, there is a 196.43% slowdown for inference and a 25.30% slowdown for training compared to *baseline*. The reason is that DL frameworks typically allocate PM on a per-tensor, per-layer basis, resulting in a large number of allocation function calls. Consequently, the primary overhead stems from PM allocation [1, 7, 45], which distinguishes these applications from microbenchmarks and other non-DL real-world applications. However, in an online inference serving system with a warm start [2, 24, 27, 57], the data allocation overhead can be amortized. In addition, using *PM* helps reduce CPU-GPU data transfer time by 29.75% for inference and 49.14% for training.

Similar to *PM*, *UVM* typically does not improve performance in most scenarios. During the training phase, convolutional deep neural networks depend heavily on *gemm* (general matrix-to-matrix multiplication) kernels. Introducing additional stages to the data transfer pipeline adds extra control logic to the GPU kernels, potentially reducing efficiency. However, in the inference phase,




(a) Inference. (b) Training.

Fig. 6. Comparisons on realworld (DL) applications for *Mega* input sizes on *Machine-D* (using time as the metric, the smaller the better). From bottom (darkest) to top (lightest), each shade shows gpu_kernel, memcpy, and allocation. Performance is affected more by *UVM* and *PM* than *Async Memcpy*. *PM* has more performance overhead on inference due to the frequent memory allocation.

*yolov3* benefits from *UVM*. Given that *yolov3* is the largest of the four networks, with CPU-GPU data transfer accounting for over 70% of the total execution time, the CPU overhead of allocating managed memory is amortized.

**Takeaway 4:** (1) *UVM* and *PM* affect the performance of *gemm* intensive DL applications more than *Async Memcpy*. (2) Frequently allocating/deallocating PM may introduce additional overhead when using *PM* for DL applications.

## 5 In-Depth Analysis

We conduct an in-depth analysis of workloads by measuring GPU performance counters. We analyze performance counters that contribute to performance variance when using *Async Memory* (Section 5.1) and examine how different machine system configurations impact the performance of *UVM* (Section 5.2).

### 5.1 In-Depth Analysis for *Async Memcpy*

Analyzing performance counters is crucial for optimizing system performance and identifying bottlenecks, as demonstrated in numerous prior studies [18, 40]. Based on the results on *vector_seq*, *3DCONV*, *lud*, and *knn*, it is not clear whether a workload whose execution time is dominated by GPU kernels can benefit from *Async Memcpy*. In this section, we will dive deep into the GPU kernel to reveal the causes affecting *Async Memcpy* performance.

*5.1.1 FLOPs Per Shared Memory Access.* *Async Memcpy* can potentially improve the program performance by overlapping the data transfer from global memory to shared memory and SM computations. However, it will only improve the overall performance if: (a) the program has intensive shared memory usage; and (b) the shared memory maintains high locality to amortize the overhead of data transfer and additional control logic.

We profile the FLOPs (floating point operations) per shared memory access and shared memory accesses per GPU cycle for both microbenchmarks and realworld applications, as shown in Figure 7 (DL workloads using the *gemm* kernel exhibit characteristics similar to those observed in the microbenchmark *gemm*). As shown in previous results (Figure 4 and Figure 5), *vector_seq*, *vector_rand*, *saxpy*, and *lud* benefit the most from using *Async Memcpy*. These workloads show significantly higher FLOPs per shared memory access and higher shared memory access per GPU cycle compared to others (*BN* has high FLOPs per shared memory access but low shared memory

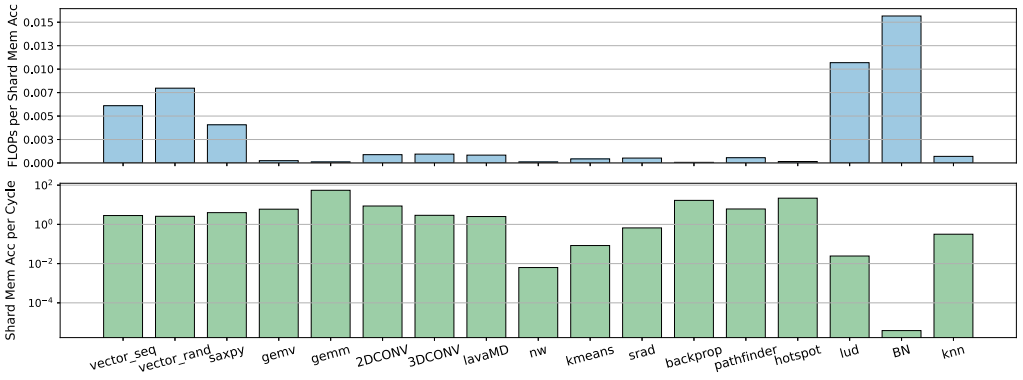



Fig. 7. FLOPs per shared memory access (top) and shared memory access per cycle (bottom) for *mega* input on *Machine-D*. Workloads only with high FLOPs per shared memory access and shared memory access per cycle can benefit from using *Async Memcpy*.

utilization, so cannot benefit from using *Async Memcpy*), highlighting the performance advantage of using *Async Memcpy*.

**Takeaway 5:** To benefit from *Async Memcpy*, a program must achieve high FLOPs to shared memory access ratio and sustain substantial shared memory utilization.

*5.1.2 Other Kernel Performance Counters.* The ratio of FLOPs to shared memory accesses and shared memory utilization are the two metrics that determine whether a program can benefit from *Async Memcpy*. Additionally, we examine other performance counters affected by *Async Memcpy* to gain a deeper understanding of its impact. Specifically, we analyze instruction mix, global memory throughput, and pipeline stall cycle breakdowns, as these metrics are affected the most by *Async Memcpy* due to its added computation overhead on GPU kernels for managing data transfer pipelines.

**Instruction Mix.** We first use the GPU instruction mix to assess the potential cost of using *Async Memcpy*. We compare the total number of memory accesses, arithmetic, and control instructions, and observe a significant difference across the six configurations. For example, as shown in Figure 8(a), *Async Memcpy* results in a 66.62% increase in control instructions compared to the *baseline* setup for the *knn* workload. Figure 8(b) shows that *Async Memcpy* reduces memory instructions by 15.62% for the *vector_seq* workload compared to *baseline*. However, as illustrated in Figure 8(c), *Async Memcpy* introduces 107.82%, 17.49%, and 43.40% more memory instructions for the *3DCONV*, *lud*, and *knn* workloads, respectively. In addition to workloads that are sensitive to *Async Memcpy*, we also present cases where the performance impact of *Async Memcpy* is minimal, such as *Yolov3*. While the performance counters of *Yolov3* are affected by *Async Memcpy*, the changes are not significant enough to affect overall performance. In summary, while *UVM* and *PM* generally have little effect on the instruction mix, *Async Memcpy* does alter it, potentially impacting overall program performance.

**Global Memory Throughput and Pipeline Stall Cycles.** We also compare global memory throughput across the six configurations. As shown in Figure 8(d), *Async Memcpy* increases global memory throughput by 73.55% and 26.58% compared to the *baseline* configuration for the *vector_seq* and *lud* workloads, respectively. However, for workloads like *3DCONV* and *knn*, which suffer performance degradation with *Async Memcpy*, global memory throughput decreases by 59.44% and 11.07%, respectively, compared with the *baseline* setup.

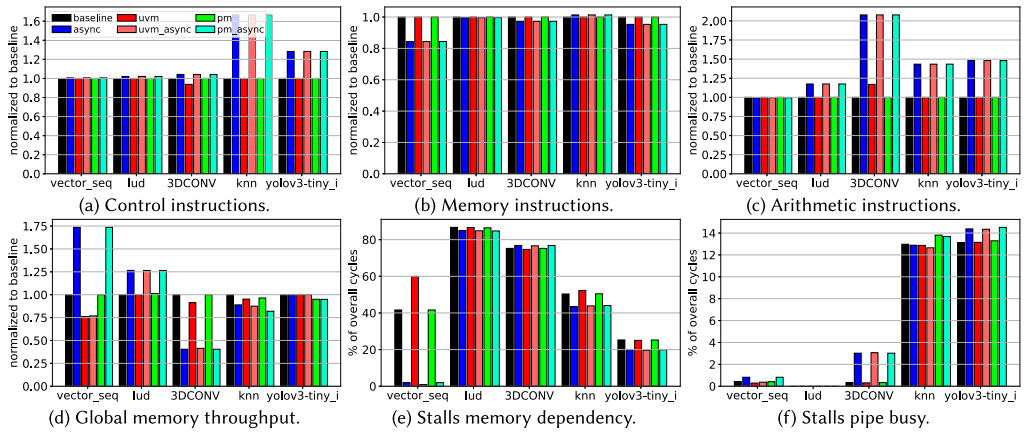



Fig. 8. Performance counter comparison (for *Mega* input on *Machine-D*) across different configurations. For each workload, from left to right, the six bars indicate the six data transfer configurations (all in the same order). GPU kernel performance counters are affected more when using *Async Memcpy* than using *UVM*/*PM*.

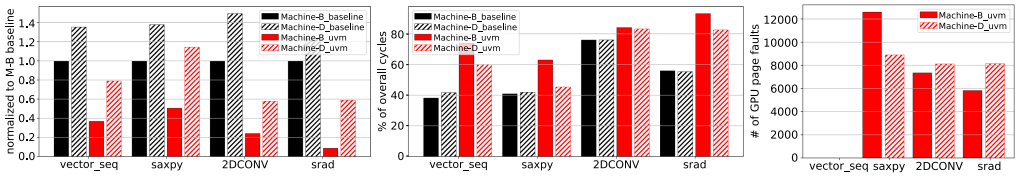
Since *Async Memcpy* adds an extra stage to the data transfer pipeline, we also profile pipeline stall cycles. Among the factors contributing to stalls, memory dependency and pipeline busy are the two counters that exhibit significant changes with *Async Memcpy*. As illustrated in Figure 8(e), *Async Memcpy* reduces stall cycles due to memory dependency by 39.71% for the *vector_seq* workload and by 6.90% for the *knn* workload, compared to the *baseline* configuration. However, as shown in Figure 8(f), for the *3DCONV* workload where performance is negatively impacted by *Async Memcpy*, the stall cycles due to pipeline busy increase by 2.69% relative to the *baseline*, ultimately slowing overall performance.

**Takeaway 6:** (1) The overhead of *Async Memcpy* comes from an increased instruction footprint. (2) The performance improvement of *Async Memcpy* comes from improved global memory throughput and reduced stall cycles due to memory dependency and pipeline busy.

### 5.2 In-Depth Analysis for *UVM* and *PM*

The performance results from both microbenchmarks (Figure 4) and realworld applications (Figure 5) reveal that *UVM* offers more promising improvements over the *baseline* configuration on *Machine-C* and *Machine-D*, but not on *Machine-B*. This trend is especially clear for the *vector_seq*, *saxpy*, *2DCONV*, and *srad* workloads. To better understand the differences across machine configurations, we profile both the *baseline* and *UVM* setups and perform a side-by-side comparison of *Machine-B* and *Machine-D* for these four workloads.

As shown in Figure 9, using *UVM* significantly reduces global memory throughput compared to the *baseline* configuration (Figure 9(a)). On average, global memory throughput dropped by 68.28% on *Machine-B* but only by 38.90% on *Machine-D*. This drop in global memory throughput also leads to an increase in kernel stall cycles due to memory dependencies (Figure 9(b)). Interestingly, the performance degradation associated with using *UVM* is not due to the extra page faults, as *Machine-D* exhibits more GPU page faults (Figure 9(c)) in most applications (for the workload *vector_seq*, both *Machine-B* and *Machine-D* incur fewer than 50 page faults, making them unobservable in Figure 9(c).). Instead, the high data transfer bandwidth on *Machine-D* results in less performance overhead from *UVM* compared to *Machine-B*.




(a) Global memory throughput. (b) Stalls memory dependency. (c) # of GPU page faults.

Fig. 9. Performance counter comparison (for *Mega* input) between different machine system configuration when using *UVM*. *Machine-D* has higher global memory throughput and fewer kernel memory dependency stall cycles. The performance effect of *UVM* on different machines does not have strong correlations with the number of page faults.

**Takeaway 7:** On machines featuring newer CPU, DRAM, and PCIe generations, *UVM* and *PM* can deliver larger performance gains. These improvements arise not from fewer GPU page faults, but from higher data transfer bandwidth.

## 6 Guidelines for Programmers

Based on the performance implications explored in Section 4 and Section 5, we now offer guidance for programmers on applying these insights.

***Async Memcpy*:** Programs with high compute density and heavy shared memory usage can see performance benefits from using *Async Memcpy* (Figure 7). Although we obtain these metrics through runtime profiling, they are also accessible via static program analysis tools. As a result, programmers can gain valuable insights before implementing the *Async Memcpy* version and make more informed decisions.

***UVM*:** Our performance analysis indicates that the benefits of *UVM* are highly dependent on machine system configurations, which are best assessed through runtime profiling rather than static analysis. From a programmer's perspective, converting a program to use *UVM* demands considerably less effort than adapting it for *Async Memcpy*. Moreover, as programmers develop a deeper understanding of workload memory access patterns and data transfer bandwidth continues to improve, leveraging *UVM* (with prefetch) for workloads with predictable memory access patterns becomes an increasingly attractive option.

***PM*:** *PM* can reduce CPU-GPU data transfer time if there is sufficient physical CPU memory. However, if an application frequently allocates small objects, the overhead of PM allocation may outweigh its benefits.

***Async Memcpy+UVM/PM*:** Our analysis indicates that when a program benefits from *Async Memcpy*, adding *PM* is generally safe since it leaves the shared memory access pattern unchanged. However, when using *UVM*, especially with prefetchers, programmers should be cautious, as prefetching can result in increased runtime data transfers between shared and global memory.

## 7 Sensitivity Studies

Different workloads exhibit unique characteristics, such as variations in hardware performance counters (discussed in Section 5), leading to different performance outcomes when using data transfer techniques like *PM*, *UVM*, and *Async Memcpy*. Moreover, even within a single workload, overall performance can be influenced 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/PM* are sensitive to these configurations.

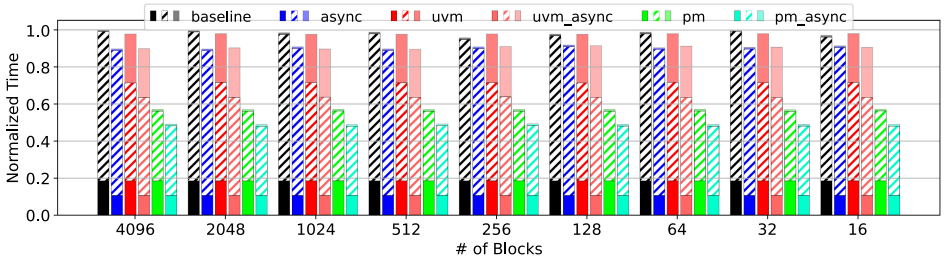



Fig. 10. Sensitivity of *vector_seq* with respect to # of blocks. From bottom (darkest) to top (lightest), each shade shows gpu_kernel, memcpy, and allocation. Performance remains stable when # of blocks changes.

### 7.1 CUDA Block and Thread

Programmers define the parallelism of each CUDA program according to the GPU resource hierarchy (Grid, Block, and Thread) as a guideline. Since there are virtually no restrictions[11] on the number of blocks in the entire GPU grid, this transparency allows large workloads to be easily programmed on GPUs without considering the actual hardware resource limitations. In this section, we explore how the performance of *PM*, *UVM*, and *Async Memcpy* is affected by the number of blocks and threads in the CUDA program.

In Nvidia A100 and H100 GPUs, at most 2,048 threads can be mapped to one SM unit (H100 has 132 SM units, each of which contains 128 CUDA cores [37]). How *Async Memcpy* and/or *UVM/PM* 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 (224 KB per SM on H100).

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 4,096 to 16. We use 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* is affected by multiple factors as shown in Section 5.1 (Figure 8). We plot the execution time breakdown of *vector_seq* in Figure 10. Interestingly, there is no obvious performance change (~2%) on all six configurations when using a different number of blocks. On average, *async*, *uvm*, *uvm_async*, *pm*, and *pm_async* achieves 8.06%, 0.66%, 8.12%, 42.06%, and 50.19% performance improvement over *baseline*, respectively.

In addition to varying the number of CUDA blocks, we also investigate how changing the number of threads per block affects performance while keeping the total number of CUDA blocks fixed.[12] As shown in Figure 11, the program performance is more sensitive to the number of threads per block (more than 50% variance when using different numbers of threads). The reason is that when there are fewer than 256 threads, GPU resources remain underutilized (H100 has 16,896 CUDA cores). The execution time breakdown also confirms this. The GPU kernel execution time of using 32 threads is 17.79× more compared with using 1,024 threads. Though the performance downgrades when using fewer threads, *async* performs much better than *baseline* (4.67% speedups on 1,024 threads, but 28.58% speedups on 32 threads). The reason is that with the same per-block

[11]CUDA uses 16-bit integers for block indices in the 3D grid. As long as the block index does not overflow, there will be no compilation errors.

[12]We set the number of blocks as 256, which is larger than in our previous analysis [26] because the Nvidia H100 features 2.4× as many CUDA cores as the Nvidia A100.

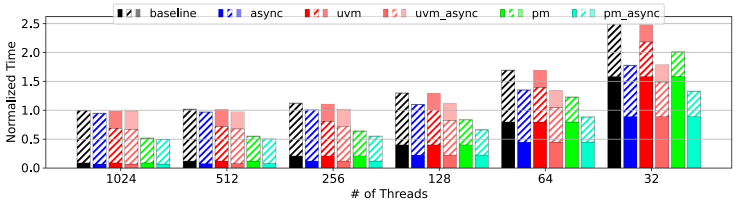



Fig. 11. Sensitivity of *vector_seq* with respect to # of threads per block. From bottom (darkest) to top (lightest), each shade shows gpu_kernel, memcpy, and allocation. Performance varies when # of threads changes.

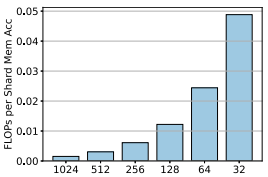



Fig. 12. FLOPS per shared memory access increases as the number of threads reduces.

shared memory capacity, fewer threads in one block can lead to a deeper buffer for each thread and larger FLOPs per shared memory access (Figure 12).

**Takeaway 8:** The performance of *UVM*, *PM*, and *Async Memcpy* is not sensitive to the number of CUDA blocks, but very sensitive to the number of threads per block. The performance improvement of *Async Memcpy* is more prominent when using less number of threads per CUDA block.

### 7.2 L1-Cache/Shared Memory Partition

Nvidia Ampere and Hopper architecture features a unified L1 cache, texture cache, and shared memory for a total of 256KB per SM. The shared memory can be configured to use up to 224 KB of the unified memory while the rest is used for both the L1 and texture cache [37]. 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 important to investigate whether *Async Memcpy*, *UVM*, and *PM* are sensitive to the partitioning of the L1 cache and shared memory. With *UVM*, fetching data into shared memory can trigger page faults, while *Async Memcpy* mitigates this by pipelining computation and data transfer through double buffering, making an efficient partition even more critical. Additionally, the L1 cache/shared memory partition directly influences the maximum number of threads that can be scheduled on a single SM [50]. For example, although the Nvidia driver might theoretically map up to eight CUDA blocks (with 256 threads each) to one SM, if each block is allocated 128 KB of shared memory, then only one block can be scheduled on that SM. Thus, it is valuable to study whether reducing the number of threads per SM affects overall performance.

We increased the shared memory capacity from 2 KB to 128 KB and evaluated performance across six different configurations, as shown in Figure 13. Notably, increasing shared memory from 2 KB to 64 KB consistently boosts performance, with improvements of 1.51%, 2.62%, 1.40%, 1.91%, 2.73%, and 3.53% across the configurations, respectively. Beyond 64 KB, the per-thread buffer depth becomes sufficient to effectively pipeline kernel computation with data transfers from global to shared memory. However, when the shared memory is set to 128 KB, overall performance can decline because the L1 cache becomes a bottleneck. For example, while *Async Memcpy* shows a 9.13% improvement over the baseline (compared to a 4.84% improvement with only 2 KB of shared memory), the total performance remains lower than that achieved with a smaller shared memory allocation. This finding highlights the importance of reserving enough L1 cache to avoid GPU kernel computation pipeline stalls caused by cache misses.

**Takeaway 9:** While increasing shared memory can boost the effectiveness of *Async Memcpy*, it remains essential to reserve enough memory for the L1 cache.

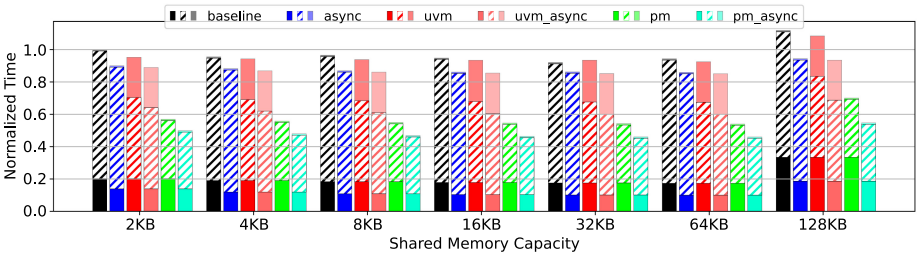



Fig. 13. Sensitivity of *vector_seq* with respect to L1-Cache/Shared Mem partition (4,096 blocks, 256 threads per block). From bottom (darkest) to top (lightest), each shade shows gpu_kernel, memcpy, and allocation. Excessive allocation of shared memory degrades the performance of *UVM*, *PM*, and *Async Memcpy*.

## 8 A New Data Transfer Model

Although *Async Memcpy* and *UVM/PM* can improve performance in CPU-GPU heterogeneous systems, current data transfer pipelines still have limitations. In this section, we introduce a new data transfer model that leverages inter-job pipelining. Using the profiling results presented earlier, we derive preliminary estimates for the performance gains of this approach. We also prototyped the new data transfer model on real hardware to demonstrate the trade-offs and outline future research directions.

### 8.1 Limitations of *PM*, *UVM*, and *Async Memcpy*

*UVM* improves the system performance by overlapping data transfer and computation. *PM* improves the system performance by avoiding swapping out OS papers with high locality. Based on the execution time breakdown (average of the 18 realworld applications on *Machine-C* using *mega* input size), the CPU-GPU data transfer time decreases from 57.96% to 24.90% and 25.09% of the overall execution time, when compared *baseline* with *UVM* and *PM*. Because data transfers now take less time, a larger proportion of the execution is devoted to GPU kernel processing (from 41.84% to 59.25% and 63.80% for *UVM* and *PM*, respectively).

Although *Async Memcpy* is compatible with *UVM/PM*, the time devoted to GPU kernel execution remains relatively low, leaving GPUs idle for over 60% of the total execution time. Additionally, because *Async Memcpy* and *UVM* do not enhance CPU task performance, overall system time becomes constrained by data allocation operations (e.g., *cudaMalloc()* and *cudaFree()* executed on CPUs). While data allocation accounts for only 18.99% of total execution time under the *baseline* configuration, it increases to 37.66% when using *Async Memcpy* and *UVM*.

### 8.2 Directions for New Data Transfer Models

Overlapping data allocation with other tasks, such as data transfer and GPU kernel execution, is a promising direction for improving the data transfer pipeline. However, data allocation typically needs to be completed before data transfer and kernel execution for each workload, making it impractical to overlap these tasks in most scenarios. An exception exists when GPU jobs (kernels) are processed in batches. In this case, the data allocation for the second job (kernel) can occur while the first job (kernel) is being processed, which could be leveraged in Kernel-as-a-Service (Kaas) systems [42]. Overlapping data allocation across kernels represents a new data transfer model that could be explored.

To better illustrate how this new data transfer model operates alongside *Async Memcpy* and *UVM/PM*, we present a visual comparison in Figure 14, showing the current pipeline (top half) and the new data transfer model (bottom half). By integrating *UVM* and *Async Memcpy*, data

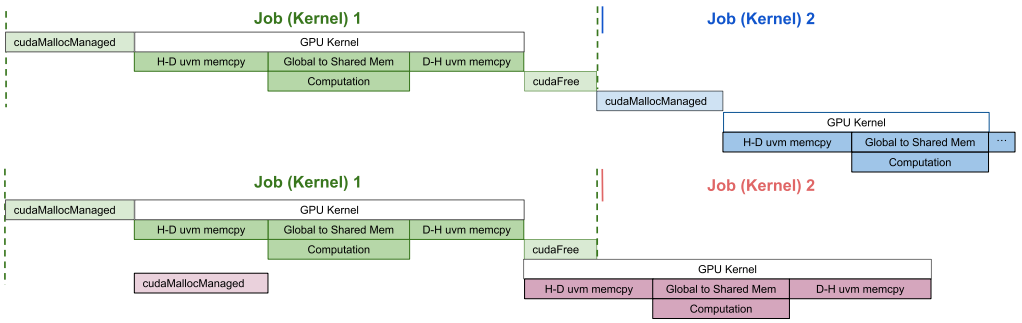



Fig. 14. Without/with the inter-job pipeline. H-D and D-H are the abbreviations of Host-to-Device and Device-to-Host.

**Without Overlapping Allocations with Kernel Execution**

```
host_A[16], host_B[16], host_C[16];
device_A[16], device_B[16], device_C[16];
for (i = 0; i < 16; i++) {
     cudaMalloc(device_A[i]);
     cudaMalloc(device_B[i]);
     cudaMalloc(device_C[i]);
     cudaMemcpy(device_A[i], host_A[i]);
     cudaMemcpy(device_B[i], host_B[i]);
     gemv(host_A[i], host_B[i], host_C[i]);
     cudaMemcpy(host_C[i], device_C[i]);
     cudaFree(device_A[i]);
     cudaFree(device_B[i]);
     cudaFree(device_C[i]);
}
```

(a) Without the inter-job pipeline.

**Overlapping Allocations with Kernel Execution**

```
host_A[16], host_B[16], host_C[16];
device_A[16], device_B[16], device_C[16];
cudaMalloc(device_A[0]);
cudaMalloc(device_B[0]);
cudaMalloc(device_C[0]);
for (i = 0; i < 16; i++) {
     sub-stream {
          cudaMalLocAsync(device_A[i + 1]);
          cudaMalLocAsync(device_B[i + 1]);
          cudaMalLocAsync(device_C[i + 1]);
     }
     cudaMemcpy(device_A[i], host_A[i]);
     cudaMemcpy(device_B[i], host_B[i]);
     gemv(host_A[i], host_B[i], host_C[i]);
     cudaMemcpy(host_C[i], device_C[i]);
     sub-stream {
          cudaFreeAsync(device_A[i]);
          cudaFreeAsync(device_B[i]);
          cudaFreeAsync(device_C[i]);
     }
}
```

(b) With the inter-job pipeline.

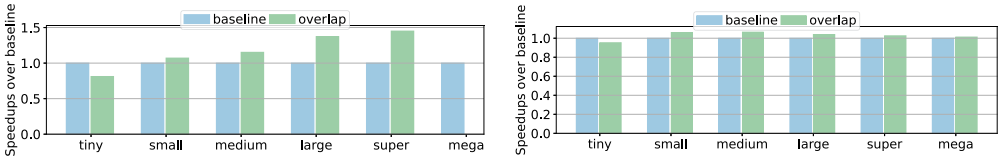
Fig. 15. CUDA programming Pseudo-code without/with the inter-job pipeline.

transfers from CPU memory to GPU global memory, and from global memory to shared memory, can overlap with GPU kernel execution The new model aims to reduce overall execution time by overlapping CPU and GPU execution across different jobs (kernels). Once the GPU kernel for job 1 begins, job 2 can initiate data allocation (*cudaMallocManaged*()) using the idle CPU. After the GPU kernel of job 1 completes, job 2 can start its own kernel execution, while job 1 handles data deallocation (*cudaFree*()) on the CPU. Ideally, the 37.66% of time spent on data allocation (and deallocation) could overlap with the 37.79% of GPU kernel time, potentially resulting in more than 30% overall performance improvement. This approach offers a promising direction to improve the CPU-GPU data transfer pipeline.

### 8.3 Prototyping the New Data Transfer Model

We prototyped the new data transfer model that overlaps data allocation/deallocation with kernel execution. We evaluated it using a synthetic workload comprising 16 *gemv* kernels, each with a 4K×4K (which is the *large* input size used in Figure 3) input size (Pseudo-code in Figure 15(a)). By overlapping GPU kernel execution with data allocation, we initiated allocation for the next kernel in a sub-stream while executing the current kernel on the main stream (Pseudo-code in Figure 15(b)).

We implemented overlapping data allocation and deallocation with GPU kernel execution for *gemv* and *gemm* on *Machine-D*, as these two kernels are widely used in modern ML frameworks




(a) Performance on *gemv* (out-of-memory for *mega* using *overlap*).

(b) Performance on *gemm*.

Fig. 16. Prototyping overlapped data allocation with GPU kernel execution of *gemv* and *gemm* on *Machine-D*.

[1, 7, 24, 45]. Figure 16 illustrates that overlapping data allocation with GPU kernel execution can yield up to 1.44× speedup over the baseline approach of serializing data allocation and kernel execution. Overlapping data allocation with GPU kernel execution yields greater performance benefits for *gemv*, because *gemm* is more compute-bound and devotes most of its runtime to computation (Figure 4). However, for *tiny* input sizes, the overhead of initializing a separate CUDA stream for data allocation exceeds the cost of the CUDA kernel, leading to reduced performance. Additionally, using *mega* input sizes can lead to out-of-memory errors because the memory allocated by the running kernel has not yet been released before allocating memory for the next kernel in the background, which doubles the GPU memory capacity requirement.

**Future Research Directions:** (1) A GPU program analysis framework that determines whether a program can benefit from overlapping data allocation and GPU kernel execution. (2) A fine-grained GPU memory allocator capable of allocating and deallocating memory in chunks to mitigate out-of-memory issues during overlapped data allocation and kernel execution.

## 9 Conclusion

In this article, we explored the performance implications of *Async Memcpy*, *UVM*, and *PM*. We conducted an in-depth characterization study and developed a benchmark suite comprising 7 microbenchmarks and 18 realworld applications. We believe this benchmark suite has the potential to facilitate further research in this area.

We observed performance gains of 24% and 34% with *UVM* and *PM*, respectively, on realworld applications. By pipelining GPU computation with global-to-shared memory data transfers, *Async Memcpy* yields roughly 20% improvements for programs with high compute density and intensive shared memory usage (e.g., *kmeans* and *lud* running on top of *UVM/PM*). Breaking down the execution time provides developers with practical guidelines: workloads with both high compute density and heavy shared memory demands can benefit from *Async Memcpy*, *UVM* is advantageous for workloads with regular data access patterns, and *PM* offers benefits for workloads with less intensive data allocation.

Additionally, we conducted a sensitivity study to highlight how the performance of *Async Memcpy* and *UVM/PM* can be affected by factors such as the number of threads per block and the L1-cache/shared memory partition strategy. These parameters should be taken into account in compiler optimizations and resource mapping designs. We also outlined future research directions, emphasizing the importance of collaborative efforts between system software and hardware architecture communities to advance the optimization of data transfer pipelines.

## Acknowledgement

We thank the reviewers for their helpful feedback. The authors would also like to acknowledge the computing servers donated by Nvidia, and the computing resources provided by the Texas Advanced Computing Center (TACC).

## References


[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. {TensorFlow}: A system for {Large-Scale} machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283.

[2] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming {throughput-latency} tradeoff in {LLM} inference with {sarathi-serve}. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 117–134.

[3] Veerendra Allada, Troy Benjegerdes, and Brett Bode. 2009. Performance analysis of memory transfers and GEMM subroutines on NVIDIA tesla GPU cluster. In *Proceedings of the 2009 IEEE International Conference on Cluster Computing and Workshops*. IEEE, 1–9.

[4] Tyler Allen, Bennett Cooper, and Rong Ge. 2024. Fine-grain quantitative analysis of demand paging in unified virtual memory. *ACM Trans. Archit. Code Optim.* 21, 1, Article 14 (2024), 24 pages. https://doi.org/10.1145/3632953

[5] Tyler Allen and Rong Ge. 2021. Demystifying gpu uvm cost with deep runtime and workload analysis. In *Proceedings of the 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 141–150.

[6] Tyler Allen and Rong Ge. 2021. 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, 1–15.

[7] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. 2024. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 929–947.

[8] Hartwig Anzt, Yuhsiang M Tsai, Ahmad Abdelfattah, Terry Cojean, and Jack Dongarra. 2020. Evaluating the performance of NVIDIA's A100 ampere GPU for sparse and batched computations. In *Proceedings of the 2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 26–38.

[9] Herwin Alayn Huillcen Baca and Flor de Luz Palomino Valdivia. 2019. Efficient sparse matrix-vector multiplication on GPUs using the CSR format, pinned memory and overlap data transfer. In *Proceedings of the 2019 IEEE XXVI International Conference on Electronics, Electrical Engineering and Computing (INTERCON)*. IEEE, 1–4.

[10] R. Bhargava and K. Troester. 2024. AMD next-generation "Zen 4" core and 4th gen AMD EPYC Server CPUs. In *IEEE Micro* 44, 3 (2024), 8–17. DOI: 10.1109/MM.2024.3375070

[11] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*. Ieee, 44–54.

[12] Jack Choquette and Wish Gandhi. 2020. Nvidia a100 gpu: Performance & innovation for gpu computing. In *Proceedings of the 2020 IEEE Hot Chips 32 Symposium (HCS)*. IEEE Computer Society, 1–43.

[13] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro* 41, 2 (2021), 29–35.

[14] Mohammad Dashti and Alexandra Fedorova. 2017. Analyzing memory management methods on integrated CPU-GPU systems. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*. 59–69.

[15] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. arXiv:2407.21783. Retrieved from https://arxiv.org/abs/2407.21783 (2024).

[16] Chris Gregg and Kim Hazelwood. 2011. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *Proceedings of the (IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 134–144.

[17] Yongbin Gu, Wenxuan Wu, Yunfan Li, and Lizhong Chen. 2020. Uvmbench: A comprehensive benchmark suite for researching unified virtual memory in gpus. arXiv:2007.09822. Retrieved from https://arxiv.org/abs/2007.09822 (2020).

[18] Yueming Hao, Nikhil Jain, Rob Van der Wijngaart, Nirmal Saxena, Yuanbo Fan, and Xu Liu. 2023. DrGPU: A top-down profiler for GPU applications. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*. 43–53.

[19] Mark Harris. 2012. How to Optimize Data Transfers in CUDA C/C++. Retrieved from https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/

[20] Tayler H Hetherington, Mike O'Connor, and Tor M Aamodt. 2015. Memcachedgpu: Scaling-up scale-out key-value stores. In *Proceedings of the 6th ACM Symposium on Cloud Computing*. 43–57.

[21] Guyue Huang, Yang Bai, Liu Liu, Yuke Wang, Bei Yu, Yufei Ding, and Yuan Xie. 2023. Alcop: Automatic load-compute pipelining in deep learning compiler for ai-gpus. *Proceedings of Machine Learning and Systems* 5 (2023), 680–694.

[22] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. 2020. Batch-aware unified memory management in GPUs for irregular workloads. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. 1357–1370.

[23] Jagadish B Kotra, Michael LeBeane, Mahmut T Kandemir, and Gabriel H Loh. 2021. Increasing gpu translation reach by leveraging under-utilized on-chip resources. In *Proceedings of the MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1169–1181.

[24] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.

[25] Jiwon Lee, Ju Min Lee, Yunho Oh, William J Song, and Won Woo Ro. 2023. SnakeByte: A TLB design with adaptive and recursive page merging in GPUs. In *Proceedings of the 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1195–1207.

[26] Ruihao Li, Sanjana Yadav, Qinzhe Wu, Krishna Kavi, Gayatri Mehta, Neeraja J Yadwadkar, and Lizy K John. 2023. Performance implications of async memcpy and UVM: A tale of two data transfer modes. In *Proceedings of the 2023 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 115–127.

[27] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, and Ion Stoica. 2023. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 663–679.

[28] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. 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*. 1633–1649.

[29] Michael Mattioli. 2021. Rome to milan, AMD continues its tour of Italy. *IEEE Micro* 41, 4 (2021), 78–83.

[30] Dan Negrut, Radu Serban, Ang Li, and Andrew Seidl. 2014. Unified memory in cuda 6.0. a brief overview of related data access and transfer issues. *SBEL, Madison, WI, USA, Technical Reports TR-2014-09* (2014).

[31] Nvidia. 2013. Nvidia Tesla K40. Retrieved from https://www.nvidia.com/content/PDF/kepler/nvidia-tesla-k40.pdf

[32] Nvidia. 2024. CUPTI. Retrieved from https://docs.nvidia.com/cuda/cupti/

[33] Nvidia. 2024. CUTLASS 3.0. Retrieved from https://github.com/NVIDIA/cutlass/

[34] Nvidia. 2024. Nsight Compute. Retrieved from https://developer.nvidia.com/nsight-compute/

[35] Nvidia. 2024. Nsight Systems. Retrieved from https://developer.nvidia.com/nsight-systems/

[36] Nvidia. 2024. Nvidia A100 GPU Architecture. Retrieved from https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[37] Nvidia. 2024. Nvidia H100 GPU Architecture. Retrieved from https://resources.nvidia.com/en-us-tensor-core/

[38] Nvidia. 2024. Nvidia P100 GPU Architecture. Retrieved from https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

[39] Nvidia. 2024. Nvidia V100 GPU Architecture. Retrieved from https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[40] Reena Panda, Shuang Song, Joseph Dean, and Lizy K John. 2018. Wait of a decade: Did SPEC CPU 2017 broaden the performance horizon?. In *Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 271–282.

[41] Carl Pearson, Abdul Dakkak, Sarah Hashash, Cheng Li, I-Hsin Chung, Jinjun Xiong, and Wen-Mei Hwu. 2019. Evaluating characteristics of CUDA communication primitives on high-bandwidth interconnects. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. 209–218.

[42] Nathan Pemberton, Anton Zabreyko, Zhoujie Ding, Randy Katz, and Joseph Gonzalez. 2022. Kernel-as-a-service: A serverless interface to GPUs. arXiv:2212.08146. Retrieved from https://arxiv.org/abs/2212.08146 (2022).

[43] L.-N. Pouchet. 2012. Polybench: The polyhedral benchmark suite. Retrieved from http://www.cs.ucla.edu/%7Epouchet/software/polybench

[44] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. 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*. 1–14.

[45] Joseph Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. Retrieved from http://pjreddie.com/darknet/.

[46] Jee Ho Ryoo, Saddam J Quirem, Michael Lebeane, Reena Panda, Shuang Song, and Lizy K John. 2015. Gpgpu benchmark suites: How well do they sample the performance spectrum?. In *Proceedings of the 2015 44th International Conference on Parallel Processing*. IEEE, 320–329.

[47] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. 2020. Subway: Minimizing data transfer during out-of-GPU-memory graph processing. In *Proceedings of the 15th European Conference on Computer Systems*. 1–16.

[48] Chuanming Shao, Jinyang Guo, Pengyu Wang, Jing Wang, Chao Li, and Minyi Guo. 2022. Oversubscribing gpu unified virtual memory: Implications and suggestions. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*. 67–75.

[49] SB Shriram, Anshuj Garg, and Purushottam Kulkarni. 2019. Dynamic memory management for gpu-based training of deep neural networks. In *Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 200–209.

[50] Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. Orion: Interference-aware, fine-grained GPU sharing for ML applications. In *Proceedings of the 19th European Conference on Computer Systems*. 1075–1092.

[51] Martin Svedin, Steven WD Chien, Gibson Chikafa, Niclas Jansson, and Artur Podobas. 2021. 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*. 1–6.

[52] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. arXiv:2302.13971. Retrieved from https://arxiv.org/abs/2302.13971 (2023).

[53] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. arXiv:2307.09288. Retrieved from https://arxiv.org/abs/2307.09288 (2023).

[54] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 41–53.

[55] Yuke Wang, Boyuan Feng, Zheng Wang, Tong Geng, Kevin Barker, Ang Li, and Yufei Ding. 2023. MGG: Accelerating graph neural networks with fine-grained intra-kernel communication-computation pipelining on multi-GPU platforms. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 779–795.

[56] Kohei Yoshida, Rio Sageyama, Shinobu Miwa, Hayato Yamaki, and Hiroki Honda. 2022. Analyzing performance and power-efficiency variations among NVIDIA GPUs. In *Proceedings of the 51st International Conference on Parallel Processing*. 1–12.

[57] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for {transformer-based} generative models. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 521–538.

[58] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W Keckler. 2016. Towards high performance paged memory for GPUs. In *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 345–357.

[59] Weixi Zhu, Guilherme Cox, Jan Vesely, Mark Hairgrove, Alan L Cox, and Scott Rixner. 2022. UVM discard: Eliminating redundant memory transfers for accelerators. In *Proceedings of the 2022 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 27–38.

Received 27 September 2024; revised 28 February 2025; accepted 19 June 2025