

# Evaluating the benefits of Argument Fusion: Optimizing host-device communication

Dionisis-Odyseas Sotiropoulos

Dipartimento di Scienze Fisiche,  
Informatiche e Matematiche  
University of Modena and Reggio  
Emilia  
Modena, Modena, Italy  
dsotirop@unimore.it

Alessandro Capotondi

Dipartimento di Scienze Fisiche,  
Informatiche e Matematiche  
University of Modena and Reggio  
Emilia  
Modena, Modena, Italy  
a.capotondi@unimore.it

Andrea Marongiu

Dipartimento di Scienze Fisiche,  
Informatiche e Matematiche  
University of Modena and Reggio  
Emilia  
Modena, Modena, Italy  
andrea.marongiu@unimore.it

## Abstract

Data movement remains a major bottleneck in heterogeneous systems with discrete GPUs, particularly when kernels expose many arguments that are transferred independently, leading to fragmented allocations and inefficient interconnect utilization. In this work, we evaluate *Argument Fusion*, a technique that merges multiple kernel arguments into a single contiguous buffer, reducing the number of allocation, transfer, and deallocation operations. We further study *Argument Fusion+*, which combines fusion with pinned memory and asynchronous CUDA transfers. Using both a parametric synthetic benchmark and real benchmarks from the Polybench suite, we show that *Argument Fusion* achieves up to 2.1× speedup, while *Argument Fusion+* reaches up to 2.6×. On real benchmarks, the combined approach improves end-to-end execution time by up to 68%.

## Keywords

GPU, Heterogeneous Systems, Offloading, Memory

### ACM Reference Format:

Dionisis-Odyseas Sotiropoulos, Alessandro Capotondi, and Andrea Marongiu. 2026. Evaluating the benefits of Argument Fusion: Optimizing host-device communication. In *Proceedings of the 23rd ACM International Conference on Computing Frontiers (CF Companion '26)*, May 19–21, 2026, Catania, Italy. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3801488.3806378>

## 1 Introduction

With the advent of increasingly powerful, highly parallel, and specialized processing units, the performance of compute-intensive parts of modern applications has scaled well over the past decade. Data transfers, however, remain a dominant performance bottleneck for many systems and workloads [4, 7, 9]. This is particularly evident in modern heterogeneous systems coupling a general-purpose multicore CPU (the *host*) with one or more massively data-parallel general-purpose graphics processing units (GPUs; the *device*). Programming models such as NVIDIA CUDA [10], Khronos OpenCL [14], and the OpenMP Acceleration Model [11] follow an execution

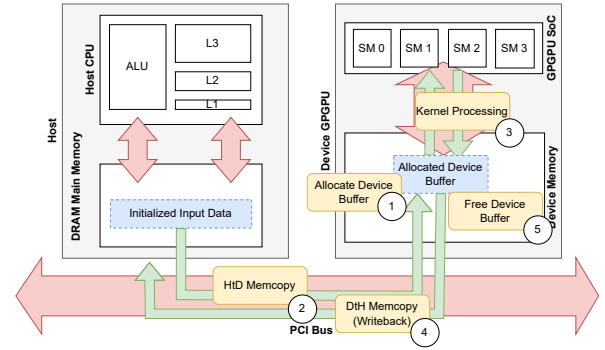


Figure 1: Standard Offloading Paradigm on a System with a discrete GPU.

paradigm in which control originates on the host while compute-intensive code regions (typically referred to as *compute kernels*) are offloaded for execution on the device.

The process of transferring execution from the *host* to the *device*, commonly referred to as *kernel offloading*, typically entails the following steps (depicted in Figure 1): (i) allocation of memory buffers in device memory for input and output data; (ii) *host-to-device* transfers of input data; (iii) control transfer and kernel execution on the device; (iv) *device-to-host* transfers of output data; and (v) deallocation of the device memory buffers.

Both control transfer and data movement introduce overhead that can significantly affect application performance. In systems with discrete GPUs, where the CPU and GPU reside on separate chips, communication occurs over an off-chip interconnect, further amplifying the cost of these operations.

CUDA, the primary programming model for NVIDIA GPUs, provides several features aimed at mitigating these inefficiencies. Asynchronous memory operations and *streams* allow partial overlap between communication and computation, while *pinned memory* can reduce the number of intermediate data copies, albeit at the cost of increased programming complexity. Despite these optimizations, CUDA manages each kernel argument independently, initiating separate allocation, transfer, and deallocation operations for each input.

As a result, the total amount of data exchanged between the *host* and the *device* is often transferred through many short bursts rather



This work is licensed under a Creative Commons Attribution 4.0 International License. *CF Companion '26*, Catania, Italy

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2569-2/26/05

<https://doi.org/10.1145/3801488.3806378>

than a few large transfers. This behavior increases runtime overhead and may lead to inefficient utilization of the communication channel.

These inefficiencies are particularly pronounced when kernels expose many arguments of small size. High-level programming models such as *OpenMP* [11] and *OpenCL* [14] follow similar argument management strategies, leading to comparable performance limitations.

In this paper we explore *Argument Fusion*, the merging of multiple kernel arguments into a single contiguous buffer that is allocated, transferred, and deallocated only once. While argument fusion has been proposed previously, to the best of our knowledge, no extensive evaluation of its benefits and limitations has been conducted, particularly in combination with advanced CUDA features such as asynchronous memory operations and pinned memory allocators.

In this work we present an experimental evaluation of argument fusion and its interaction with advanced CUDA features.

Our experimental results on two heterogeneous platforms (Section 4.3) show that *Argument Fusion* often outperforms both *pinned memory* and *asynchronous CUDA API* techniques when kernels expose multiple arguments, achieving speedups on the total execution of up to 2.1× over the standard offloading paradigm. Furthermore, we show that these optimizations are largely orthogonal, and that combining them can improve the total execution time of memory-bound kernels by up to 2.6×.

## 2 Related Work

The communication bottleneck in systems with discrete GPUs has been extensively studied. A large body of work focuses on *computation and communication overlap* [1, 2, 8, 13, 15]. By exploiting asynchronous transfers provided by programming models (e.g., *cudaMemcpyAsync* in CUDA), these approaches aim to hide communication latency by overlapping Host-to-Device transfers with kernel execution. Typical strategies include tiling kernel inputs so that execution can begin before the full data set has been transferred, or overlapping communication and execution across multiple workloads.

Other approaches attempt to improve memory management or reduce unnecessary transfers [3, 5, 6, 16]. For example, work [3] proposes compiler optimizations that eliminate redundant transfers through code migration, while *UVM: Discard* [16] introduces a mechanism that removes low-reuse data from device memory to allow data with higher temporal locality to remain resident.

Only a small number of works directly address the problem of merging kernel arguments into larger transfers. In *Lazy Allocation and Transfer Fusion* [6], the authors propose combining multiple input arguments into contiguous buffers to reduce allocation and transfer overheads, focusing primarily on host-side preparation costs and evaluating the technique on a vector-add workload. Work [5] instead considers fusion in the presence of dependencies between data produced and transferred at different stages of execution, constructing dependency and affinity graphs and formulating the optimal argument arrangement as a traveling salesman problem.

While these works demonstrate the feasibility of fusion-based approaches, they do not provide an extensive characterization of

the performance benefits that fusion alone can offer across different workloads and system configurations. In contrast, this work performs a detailed empirical evaluation of argument fusion, comparing it with other orthogonal optimizations and exploring its performance across a wide parameter space using both synthetic workloads and real-world benchmarks.

## 3 Motivation and Background

When GPU kernels are offloaded with multiple mapped variables, the runtime commonly processes each argument independently, issuing separate device allocations and host-to-device transfers. As the number of arguments increases, data movement becomes fragmented into many small operations rather than a few large transfers. This behavior increases transfer overhead and can reduce effective PCIe utilization, especially when the amount of data per argument is small.

This fragmentation introduces four main inefficiencies. First, each argument may incur its own device allocation and deallocation overhead. Second, small individual transfers tend to underutilize the available interconnect bandwidth. Third, independent transfers limit opportunities for overlap between communication and execution. Fourth, the overall offloading path may include additional synchronization and runtime overheads associated with handling many separate memory operations.

To study these effects, we consider five transfer configurations. *Baseline* uses standard pageable-memory transfers without fusion. *Pinned* replaces pageable host buffers with pinned memory to improve transfer efficiency. *Async* uses asynchronous transfers to enable communication overlap. *Fusion* packs multiple input arguments into a single contiguous buffer and performs a fused transfer. *Argument Fusion+* combines fusion with pinned memory and asynchronous transfer support. These configurations define the evaluation space used in the next section. Figure 2 summarizes the transfer configurations evaluated in this work.

## 4 Argument Fusion Evaluation

To evaluate the performance improvements enabled by *Argument Fusion*, we use two complementary scenarios. First, we study the technique using a parametric synthetic benchmark. We then evaluate it on real benchmarks from the Polybench suite<sup>1</sup>. For the purposes of this work, only kernel *inputs* are fused.

### 4.1 Argument Fusion Implementation

Before the presentation of the experimental results, we elaborate on the methodology of manually applying *Argument Fusion* to unoptimized code for a kernel with multiple arguments.

For illustration purposes, we describe the optimization process using the baseline CUDA code of a generic application with 3 inputs. The unoptimized device code is shown in Listing 1. The code features 3 host memory allocations on line 2, 3 device memory allocations on lines 4-6 followed by 3 transfers *HtD* on lines 8-10, the kernel launch on line 12, a copy-back of the result on line 14 and finally the deallocations of the 3 device buffers on lines 16-18.

<sup>1</sup>Polybench: The polyhedral benchmark suite. Link: <http://www.cs.ucla.edu/pouchet/software/polybench>

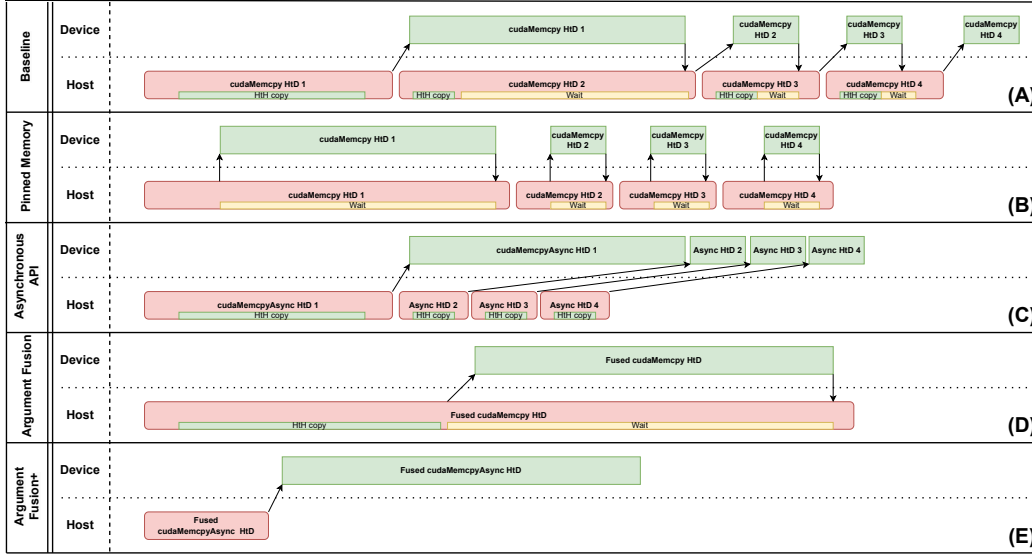


**Figure 2: Host and Device execution trace for A) Baseline: Separate non-pinned synchronous transfers, B) Pinned Memory: Separate pinned synchronous transfers, C) Asynchronous API: Separate non-pinned asynchronous transfers, D) Argument Fusion: Single fused non-pinned synchronous transfer, E) Argument Fusion+: Single Fused pinned asynchronous transfer.**

**Listing 1: Standard offloading paradigm code for a generic kernel with 3 input arguments.**

```

1 // Host Allocation
2 A = malloc(n) B = malloc(m); C = malloc(z);
3 // 1) Device Allocations
4 cudaMalloc((void **)&A_gpu, n);
5 cudaMalloc((void **)&B_gpu, m);
6 cudaMalloc((void **)&C_gpu, z);
7 // 2) 3 Host-to-Device Transfers
8 cudaMemcpy(A_gpu, A, n, HostToDevice);
9 cudaMemcpy(B_gpu, B, m, HostToDevice);
10 cudaMemcpy(C_gpu, C, z, HostToDevice);
11 // 3) Kernel Launch
12 kernel<<< grid1, block1 >>>(A_gpu, B_gpu, C_gpu);
13 // 4) Writeback of Results
14 cudaMemcpy(result, A_gpu, n, DeviceToHost);
15 // 5) 3 Device De-Allocations
16 cudaFree(A_gpu);
17 cudaFree(B_gpu);
18 cudaFree(C_gpu);

```

**Listing 2: Optimized offloading with Argument Fusion for a generic kernel with 3 argument inputs.**

```

1 //Single host allocation
2 A = malloc(n+m+z); B = A+n; C = B+m;
3 // 1)Single Device Allocation
4 cudaMalloc((void **)&A_gpu, total_size=n+m+z);
5 B_gpu = A_gpu+n; //Argument Assignment device
6 C_gpu = B_gpu+m;
7 // 2) 1 H2D Transfer
8 cudaMemcpy(A_gpu, A, total_size, HostToDevice);
9 // 3) Kernel Launch, not modified
10 kernel<<< grid1, block1 >>>(A_gpu, B_gpu, C_gpu);
11 // 4) Writeback of Results
12 cudaMemcpy(result, A_gpu, n, DeviceToHost);
13 // 5) Single Device De-Allocation
14 cudaFree(A_gpu);

```

**Table 1: System Platforms Specification Used for Evaluation.**

| Specification    | System A    | System B         |
|------------------|-------------|------------------|
| GPU Arch.        | Pascal      | Ada Lovelace     |
| GPU Model        | Quadro P400 | GeForce RTX 4090 |
| GPU Memory       | 2GB GDDR5   | 24GB GDDR6X      |
| System Interface | PCIe 3.0    | PCIe 4.0         |
| CUDA Cores       | 256         | 16384            |
| Cores/SM         | 64          | 128              |

In order to perform a single fused transfer of all 3 buffers we need to modify: (i) Host Allocations (malloc), (ii) Device Allocations (cudaMalloc), (iii) Host-To-Device Transfers (cudaMemcpy), (iv) Device Deallocations (cudaFree) and (v) Host Deallocations (Free).

The optimized code for *Argument Fusion* is shown in Listing 2. There, we have a single *host* allocation (line 2) and a single *device* allocation (line 4). Pointers to the sections of the unified device buffer are calculated for host and device on lines 2 and 5-6 respectively before the single, fused transfer is done (line 8). The kernel is launched without any additional modification on line 10. On line 12 we have the copyback of the result followed by a single deallocation of the unified buffer on line 14.

The *pinned buffer* configuration (Figure 2.B) is implemented by replacing malloc-based host allocations with `cudaMallocHost`, while the asynchronous-transfer configuration (Figure 2.C) is implemented by replacing `cudaMemcpy` with `cudaMemcpyAsync`.

## 4.2 Experimental Setup

Experiments were conducted on two heterogeneous systems, one equipped with an NVIDIA Pascal P400 GPU and one with an

NVIDIA Ada Lovelace RTX 4090 GPU. We refer to the systems by GPU model, since our study focuses on GPU-side allocation and transfer operations. The device specifications are summarized in Table 1. Both systems run CUDA version 12.2 on Ubuntu 22.04 LTS.

Unless otherwise noted, results report performance improvements over the *baseline* configuration (Figure 2.A). End-to-end execution time is measured from device memory allocation to device memory deallocation, excluding host allocation and initialization time. Each experiment is repeated 100 times and the reported speedup corresponds to the average after removing outliers using the Interquartile Range (IQR) method<sup>2</sup>. All comparisons are performed using identical kernels, data sizes, and system configurations.

### 4.3 Evaluation of *Argument Fusion* in a synthetic benchmark

We use a parametric synthetic benchmark featuring a lightweight kernel with minimal computation to systematically study the benefits of *Argument Fusion*. The benchmark allows control over two parameters:

- *Num Args*: number of kernel arguments
- *Data Size*: total data transferred to the device

The explored parameter space ranges from 1KB to 1GB of transferred data. The number of kernel arguments is set to 16, which corresponds to the maximum number of inputs observed in our target benchmark suite.

**4.3.1 *Argument Fusion Component Analysis.*** Using the synthetic benchmark, we evaluate the benefits of *Argument Fusion* for kernels with 16 equally sized arguments as shown in Figure 3. In this configuration denoted by the purple line, the baseline runtime performs 16 `cudaMalloc`, 16 `cudaMemcpy`, and 16 `cudaFree` operations, whereas *Argument Fusion* replaces them with a single allocation, transfer, and deallocation.

Across the evaluated data sizes, *Argument Fusion* consistently improves performance by reducing the number of allocation and transfer operations. Peak speedups reach 2.1× on the P400 (at 16MB) and 1.85× on the RTX 4090 (at 4KB), with average improvements of 1.4× and 1.3× respectively. A discontinuity around the 2MB data size threshold is attributed to the CUDA allocator behavior, which allocates memory in 2MB blocks [12].

The contribution of each component is also shown in Figure 3 where each line corresponds to the speedup on the total execution obtained by replacing 16 operations of the baseline (e.g., `cudaMalloc`) with a single fused operation. The dominant contributor to the speedup is the reduction in transfer overhead (`cudaMemcpy`).

**4.3.2 *Optimization Scenario Comparison.*** In this subsection we evaluate the fully optimized *Argument Fusion+* configuration *only* for data transfer performance (`cudaMemcpy`), which combines (i) pinned host memory allocations, (ii) asynchronous transfers, and (iii) *Argument Fusion*. To enable a clear analysis of the individual contributions of these optimizations, we present results for a representative configuration consisting of 16 equally sized arguments while sweeping total transferred data size as shown in Figure 4.

<sup>2</sup>Small transfer sizes are particularly sensitive to measurement noise and interference from other processes during runtime.

The fully optimized configuration *Argument Fusion+* (purple line) achieves peak speedups of 8.8× and 9.2× on HtD transfers alone for the P400 and RTX 4090 respectively, and 2.6× and 2.55× on the total benchmark execution, exceeding standalone *Argument Fusion*. Across the explored parameter space we observe an average speedup of approximately 1.7×. Figure 4 further shows the speedup of HtD transfers over the baseline of the rest of the discussed optimizations separately.

### 4.4 Polybench Benchmarks

Table 2: Polybench Benchmarks Arguments.

| Benchmark     | Input Arguments |            |       |
|---------------|-----------------|------------|-------|
|               | 2D Arrays       | 1D Vectors | Total |
| <b>Gemver</b> | 1               | 8          | 9     |
| <b>Mvt</b>    | 1               | 4          | 5     |
| <b>Bicg</b>   | 1               | 4          | 5     |
| <b>Atax</b>   | 1               | 3          | 4     |
| <b>3Vadd</b>  | 0               | 3          | 3     |

Table 3: Polybench Data Sizes.

| Bench.        | Datasets |       |       |       |            |
|---------------|----------|-------|-------|-------|------------|
|               | (KB)     |       |       | (MB)  |            |
|               | Mini     | Small | St/rd | Large | Xtra Large |
| <b>Gemver</b> | 8        | 60    | 638   | 15    | 61         |
| <b>Mvt</b>    | 7        | 58    | 631   | 15    | 61         |
| <b>Bicg</b>   | 7        | 58    | 631   | 15    | 61         |
| <b>Atax</b>   | 7        | 58    | 629   | 15    | 61         |
| <b>3Vadd</b>  | 8        | 60    | 631   | 15    | 61         |

*Argument Fusion* and *Argument Fusion+* have been evaluated on a selection of 4 multi-argument benchmarks drawn from the Polybench benchmark suite and a simple 3-vector add kernel. The argument counts and dataset sizes for these benchmarks are reported in Tables 2 and 3 respectively.

Figure 5 shows the performance gain on the total execution of these benchmarks when *Argument Fusion* is applied (blue bars) and when *Argument Fusion+* is applied. We report maximum speedups of 38% and 67% for systems A and B respectively.

## 5 Conclusion & Future Work

In this work, we have provided a detailed evaluation of the benefits introduced by *Argument Fusion* as a technique to optimize host-device data movements in discrete GPU systems. We compared the technique against modern CUDA features aimed at mitigating communication inefficiencies, and evaluated it in conjunction with these techniques as well, providing a thorough exploration that was missing in all prior work.

In future work, we plan to evaluate the benefits of *Argument Fusion* in other interconnects (e.g., NVLink) and a cost model to show when fusion is impactful. Furthermore, we aim to develop our own fusion algorithm in the runtime, by gathering arguments in main memory into a contiguous fused staging buffer.

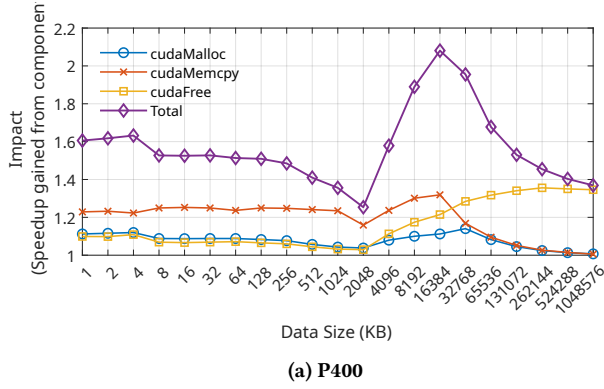
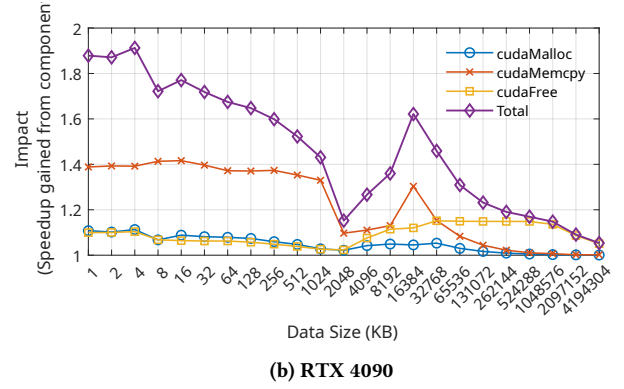


Figure 3: Impact of components on Argument Fusion Speedup for the Synthetic Benchmark with 16 arguments.

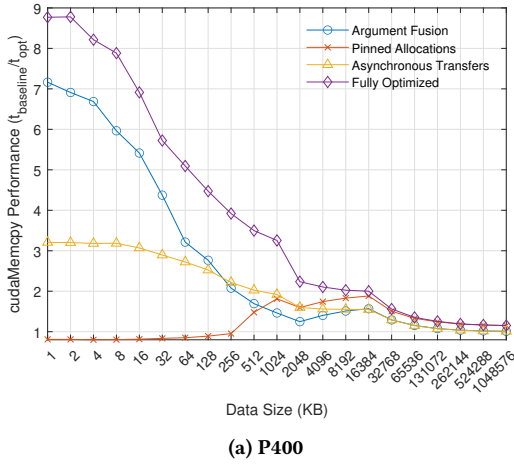
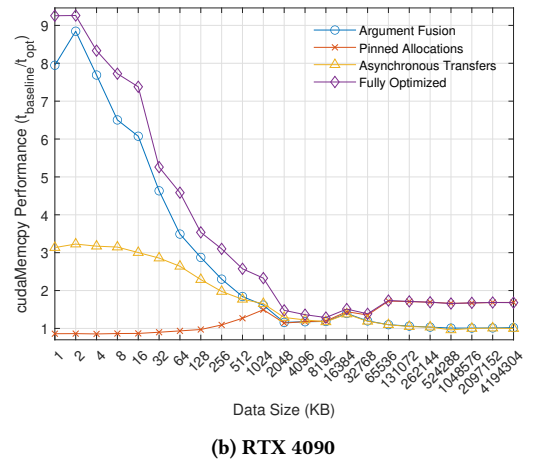


Figure 4: Synthetic benchmark results on the speedup of transfers from individual optimizations and *Argument Fusion+*.

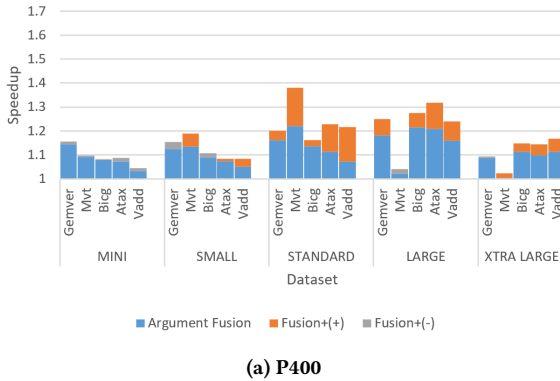
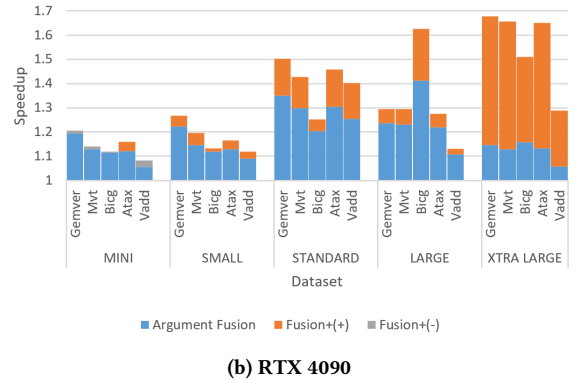


Figure 5: Speedup of Polybench Benchmarks optimized by Argument Fusion (blue) and Argument Fusion+ (-: gray negative performance compared to Argument Fusion due to pinned memory, +: orange positive performance).

## Acknowledgments

This work was funded by the European Network of Excellence dAIEDGE under Grant Agreement No 101120726.

## References

- [1] Petros Anastasiadis, Nikela Papadopoulou, Georgios I. Goumas, and Nectarios Koziris. 2021. CoCoPeLia: Communication-Computation Overlap Prediction for Efficient Linear Algebra on GPUs. *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2021), 36–47. <https://api.semanticscholar.org/CorpusID:233434029>
- [2] Michael Bauer, Henry Cook, and Bruce Khailany. 2011. CudaDMA: optimizing GPU memory bandwidth via warp specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (Seattle, Washington) (SC '11). Association for Computing Machinery, New York, NY, USA, Article 12, 11 pages. doi:10.1145/2063384.2063400
- [3] Cong Fu, Zhenhua Wang, and Yanlong Zhai. 2017. A CPU-GPU Data Transfer Optimization Approach Based on Code Migration and Merging. In *2017 16th International Symposium on Distributed Computing and Applications to Business, Engineering and Science (DCABES)*. 23–26. doi:10.1109/DCABES.2017.13
- [4] Chris Gregg and Kim Hazelwood. 2011. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 134–144.
- [5] Christoph Kessler. 2019. Global optimization of operand transfer fusion in heterogeneous computing. In *Proceedings of the 22nd International Workshop on Software and Compilers for Embedded Systems* (Sankt Goar, Germany) (SCOPES '19). Association for Computing Machinery, New York, NY, USA, 49–58. doi:10.1145/3323439.3323981
- [6] Lu Li and Christoph Kessler. 2018. Lazy Allocation and Transfer Fusion Optimization for GPU-Based Heterogeneous Systems. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. 311–315. doi:10.1109/PDP2018.2018.00054
- [7] 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 Transactions on Architecture and Code Optimization* (2025).
- [8] 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. Archit. Code Optim.* 22, 3, Article 93 (Sept. 2025), 26 pages. doi:10.1145/3746231
- [9] Yinan Li, Bailu Ding, Ziyun Wei, Lukas M. Maas, Momin Al-Ghosien, Spyros Blanas, Nicolas Bruno, Carlo Curino, Matteo Interlandi, Craig Peeper, et al. 2025. Scaling GPU-Accelerated Databases beyond GPU Memory Size. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4518–4531.
- [10] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for? *Queue* 6, 2 (2008), 40–53.
- [11] OpenMP Architecture Review Board. 2008. OpenMP Application Program Interface Version 3.0. <http://www.openmp.org/mp-documents/spec30.pdf>
- [12] Ramya Prabhu, Ajay Nayak, Jayashree Mohan, Ramachandran Ramjee, and Ashish Panwar. 2025. vattention: Dynamic memory management for serving llms without pagedattention. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 1133–1150.
- [13] K. Raju and Niranjana N. Chiplunkar. 2021. Performance enhancement of CUDA applications by overlapping data transfer and Kernel execution. *Applied Computer Science* 17, 3 (2021).
- [14] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering* 12, 3 (2010), 66–73. doi:10.1109/MCSE.2010.69
- [15] N. V. Sunitha, K. Raju, and Niranjana N. Chiplunkar. 2017. Performance improvement of CUDA applications by reducing CPU-GPU data transfer overhead. In *2017 International Conference on Inventive Communication and Computational Technologies (ICICCT)*. 211–215. doi:10.1109/ICICCT.2017.7975190
- [16] Weixi Zhu, Guilherme Cox, Jan Vesely, Mark Hairgrove, Alan L. Cox, and Scott Rixner. 2022. UVM Discard: Eliminating Redundant Memory Transfers for Accelerators. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. 27–38. doi:10.1109/IISWC55918.2022.00013