In-Copy Fusion: Runtime Argument Fusion for Efficient OpenMP GPU Offloading

Dionysios-Odysseas Sotiropoulos
University of Modena and Reggio Emilia
Modena, Italy
dsotirop@unimore.it

Sara Royuela Alcázar
Barcelona Supercomputing Center
Barcelona, Spain
sara.royuela@bsc.es

Eduardo Quiñones
Barcelona Supercomputing Center
Barcelona, Spain
eduardo.quinones@bsc.es

Alessandro Capotondi
University of Modena and Reggio Emilia
Modena, Italy
alessandro.capotondi@unimore.it

Andrea Marongiu
University of Modena and Reggio Emilia
Modena, Italy
andrea.marongiu@unimore.it

Abstract

In modern GPU-based non-unified heterogeneous systems, CPU-GPU communication happens via the PCI bus. Data transfers are affected by startup overhead, which underutilizes the PCI channel bandwidth for small transfers. Modern programming models, such as CUDA and OpenMP, treat each input argument to a compute kernel independently, leading to data movement segmentation and execution slowdowns. This work presents In-Copy Fusion (ICF), a runtime optimization of OpenMP's accelerator model, implemented in LLVM, that gathers eligible `map` clauses into staging buffers, orchestrating an optimal transfer pipeline through fusion without modifying application or kernel code. The technique preserves OpenMP semantics with negligible overhead.

We evaluate ICF across a range of representative HPC benchmarks and configurations, including varying argument counts, data sizes, and argument-size disparities, as well as real-world benchmarks. The results show that ICF improves effective host-to-device (H2D) bandwidth and reduces end-to-end time relative to the baseline runtime with per-argument transfers across platforms and workloads, achieving an up to 4.8× speedup.

CCS Concepts

• **Computing methodologies** → **Parallel computing methodologies**.

Keywords

OpenMP, GPU offloading, host–device data movement, argument fusion, transfer aggregation, LLVM, libomptarget

ACM Reference Format:
Dionysios-Odysseas Sotiropoulos, Sara Royuela Alcázar, Eduardo Quiñones, Alessandro Capotondi, and Andrea Marongiu. 2026. In-Copy Fusion: Runtime Argument Fusion for Efficient OpenMP GPU Offloading. In *Proceedings of the 55th International Conference on Parallel Processing (ICPP '26), September 28–October 01, 2026, Singapore, Singapore.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3832810.3832841

This work is licensed under a Creative Commons Attribution 4.0 International License.
ICPP '26, Singapore, Singapore
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2657-6/26/09
https://doi.org/10.1145/3832810.3832841

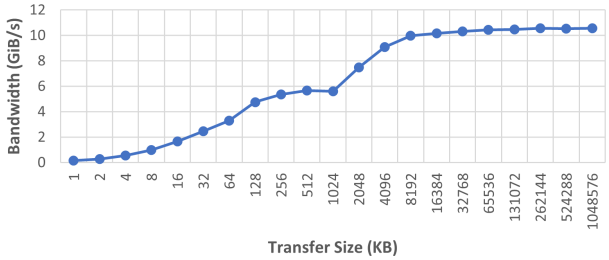


Figure 1: Measured end-to-end bandwidth in GiB/s for a sweep of total data sizes in a system with PCIe3.

1 Introduction

Modern high-performance computing systems increasingly rely on heterogeneous architectures that couple General-Purpose CPUs with discrete GPGPU accelerators. While GPGPUs offer substantial computational throughput, their integration into the system introduces a fundamental performance bottleneck: data movement between host and device memory. On systems without unified physical memory, host–device transfers utilize a system interconnect such as PCIe or NVLink, making data movement a first-order cost that can rival or exceed kernel execution time for many workloads [10, 17, 18].

A well-established characteristic of these interconnects is the presence of significant per-transfer overheads in addition to bandwidth limited payload costs [8, 19]. As a result, large, contiguous transfers reach peak interconnect bandwidth, while smaller transfers underutilize the channel and incur disproportionate overhead [26]. This behavior has been observed across platforms and workloads and remains a critical factor in end-to-end performance on discrete GPGPU systems. Fig. 1 illustrates these overheads by showing the measured bandwidth of a transfer of a single argument from

host pageable memory to the device, on a platform with PCIe3. The size of the argument increases in MiB over the x-axis, and it takes a transfer of 8MiB to reach the maximum, while transfers at KiB scale use *2 orders of magnitude less* bandwidth than the maximum.

At the same time, applications often offload kernels with multiple input data objects, each mapped independently according to the programming model's semantics. When these mappings are realized as separate host–device transfers, the resulting communication pattern can become fragmented, even when the aggregate data volume would otherwise be sufficient to efficiently utilize the interconnect. This fragmentation emerges not from application intent, but from the interaction between high-level offload abstractions and low-level runtime execution.

High-level programming models such as OpenMP [25] aim to simplify GPGPU offloading by allowing programmers to express data movement declaratively through mapping constructs. While this abstraction improves portability and programmer productivity, it also removes part of the data movement control from the programmer, thereby empowering the runtime with the capability of orchestrating host–device communication. This shift makes the runtime a natural focus for cross-argument optimizations that are difficult to express at the source level.

In this work, we explore an optimization routine for fused transfers implemented in the OpenMP runtime that opportunistically and dynamically fuses arguments while keeping the mechanism completely transparent to the programmer. Targeting OpenMP to expose our fusion optimization not only improves the host–device communication performance, but also does the work without requiring developer intervention.

Overall, this paper introduces the following contributions:

(i) Runtime analysis of OpenMP offloading behavior, *(ii)* ICF: a runtime-only argument fusion mechanism *(iii)* LLVM-based implementation with staged, pipelined transfers, *(iv)* Evaluation across synthetic and real workloads showing up to 4.8× speedup.

2 Motivation

OpenMP is widely used in high-performance computing because it abstracts away the complexity of controlling accelerator hardware (e.g., a GPU) while yielding competitive performance for parallel and heterogeneous applications [15, 21]. Its widespread adoption and standardized runtime make it a particularly impactful target for system-level optimizations. Accordingly, this work targets OpenMP rather than lower-level programming models.

The performance of OpenMP GPU offloading is, in many cases, determined by how data mappings expressed at the language level are realized as concrete host–device communication. Although OpenMP allows programmers to describe data movement declaratively using `map` clauses, the programming model does not prescribe how multiple mappings within a target region should be organized or issued by the runtime. As a result, the efficiency of data movement depends on runtime decisions that are opaque to the programmer and may differ across implementations and platforms.

A central observation motivating this work is that inefficiency in the OpenMP offloading device model and other lower-level programming models, such as CUDA, often stems from the decomposition of data movement across multiple mapped arguments. Applications frequently offload kernels with several input data objects that are logically independent but temporally co-located within a target region. OpenMP handles each input separately, leading to multiple transfer and allocation calls that impose stacking overhead, which inflates the communication process and reduces effective bandwidth. The most obvious way to mitigate this effect is to treat all data participating in the offload process as a single raw data blob.

Application-level packing is impractical due to invasive restructuring, interface changes, and poor generalization across kernels. Compiler-based approaches are similarly constrained, as mapping decisions depend on runtime information such as argument sizes, control flow, and dynamic data structures. Consequently, neither approach can reliably optimize transfer aggregation without violating abstraction boundaries or requiring global restructuring.

A runtime fusion approach that aims to dynamically optimize host–device communication is inherently challenging because the device driver transfer interface operates on a single base pointer and transfer size. Each transfer, therefore, describes a contiguous virtual address range, and the runtime cannot directly express a single transfer that spans multiple disjoint arguments. Moreover, host memory allocated by applications is typically pageable, and direct DMA from pageable memory is not possible. Interestingly, to ensure correctness, the device driver transparently performs an essential staging phase, in which pageable host data are first staged through page-locked (pinned) host memory before initiating DMA to the device. This staging is performed independently for each argument, supporting per-argument transfer granularity and preventing naive runtime-level fusion. While this behavior is typically hidden from the runtime, it exposes an implicit aggregation point.

This observation completes the motivation for *In-Copy Fusion (ICF)*. Rather than attempting to circumvent the mandatory staging phase performed by the device driver, ICF makes this phase explicit and controlled within the runtime, using it to aggregate multiple independently mapped arguments into a single staged transfer. By staging these arguments as a single larger transfer unit, the runtime can issue an effective fused host-to-device transfer that mitigates transfer fragmentation across the interconnect while preserving the semantics of the underlying offload model.

3 Host–Device Data Transfers in OpenMP

OpenMP offloading expresses data movement through high-level mapping constructs that are resolved entirely by the runtime at execution time. Although data dependencies are specified declaratively in the source code, the concrete implementation of device allocations and host–device transfers emerges only after the compilation, when the mapping metadata is processed by the offloading runtime and the device backend. The following section describes how OpenMP target constructs are lowered and how mapped variables are handled by the runtime, providing the necessary context for understanding the baseline transfer behavior and its limitations.

3.1 LLVM's Lowering of OpenMP Offloading Constructs

In OpenMP, both the `target` and `target data` constructs specify data movement declaratively via the `map` clause and the `to`, `from`, and `tofrom` map-type modifiers. Listing 1 illustrates the use of

this clause to perform a vector multiplication in a device. The `map(to: A[0:N])` clause implies device allocation and host-to-device transfer of the array section between A[0] and A[N-1], both elements included, at the start of the offloading block, and the `map(from: B[0:N])` implies device allocation and device-to-host transfer of the B[0:N] array section at the end of the offloading block. A `map(tofrom:...)` implies both `to` and `from` map-type modifiers. De-allocation of mapped argument device buffers is always implied at the end of the offloading block.

```
#pragma omp target map(to: A[0:N]) map(from: B[0:N])
    for(int i=0; i<N; ++i)
        B[i] = A[i] * A[i];
```

Listing 1: Syntax example of data offloading in OpenMP

During compilation with LLVM, these constructs are lowered to `__tgt_target_kernel()` and `__tgt_target_data_begin_mapper()` calls to the OpenMP offloading runtime, respectively, which manages device memory and executes data transfers. In LLVM's OpenMP implementation, this functionality is provided by the `libomptarget` runtime and device-specific plugins, such as the CUDA [24] plugin for NVIDIA GPUs. The top part of Fig. 2, namely *"LLVM IR Level Functions"*, shows the OpenMP *libomptarget* call graph from the offload directives to the `targetDataBegin()`.

Although `target` and `target data` serve different roles at the language level, they converge to a common runtime entry point for data movement. Before a target region executes, the compiler-generated code constructs a mapping list consisting of arrays of argument base addresses, sizes, and map-type flags. This list is passed to the runtime, which processes it to perform device allocations, host–device transfers, and present-table updates.

The `target enter data` construct follows the same data-entry runtime path, so eligible input mappings can be handled by the current ICF integration.

All OpenMP data-mapping operations ultimately converge on a single runtime routine that enters a mapped region. In the LLVM offloading runtime, this routine is `targetDataBegin()`. It is invoked both when entering a `target` region and when encountering a `target data` directive, making it the central point at which mapping semantics are enforced and data movement is realized.

At the point `targetDataBegin()` is called, the runtime has complete visibility of the mapping list for the region: the number of mapped variables, their host addresses, their sizes, and the associated map-type modifiers (e.g., `to`, `from`, `tofrom`). The runtime processes this list sequentially, applying a series of checks to determine whether each mapping requires a new device allocation, a data transfer, or can be satisfied by an existing present-table entry.

This design ensures correctness and simplicity, but it also means that data movement decisions are made on a per-entry basis. Each mapped variable is considered independently, without regard to the aggregate communication pattern induced by the region as a whole.

3.2 Baseline Allocation and Transfer Behavior

For mappings that require data to be transferred to the device, the *libomptarget* runtime typically performs the following steps for each mapping entry:

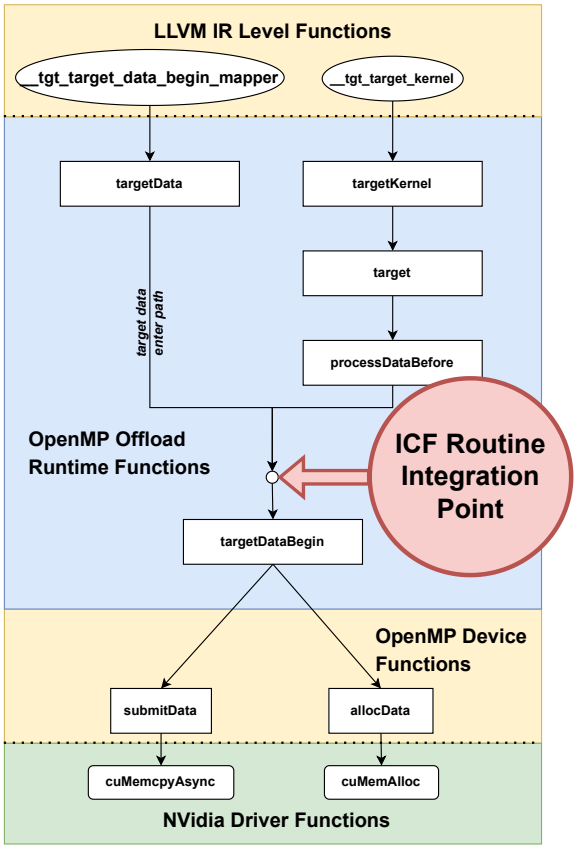


Figure 2: OpenMP runtime's offloading chart from the LLVM IR to CUDA driver function calls for host-to-device communication.

(1) Allocate a device buffer sized to the mapped variable.
(2) Initiate a host–device transfer for that buffer.
(3) Insert an entry into the present table to record the mapping.

When host buffers reside in pageable memory, which is the default for OpenMP applications, the CUDA driver internally stages each transfer through page-locked (pinned) memory before issuing a DMA operation to the device. This staging process takes the form of a host-pageable to host-pinned memory transfer (H2H), and it is completely transparent to the OpenMP runtime. The staging of data to internal pinned buffers is required for two reasons [6]: (1) the device is agnostic of the host's virtual memory address translation, and (2) the DMA of the device, which is responsible for the H2D communication over the interconnect, realizes the transfers through continuous data bursts. This means that for non-scalar arguments, all data for a particular argument must be present in contiguous addresses in the host memory, which is not always the case.

As a result, when a target region maps multiple input variables, the runtime issues multiple allocation and transfer operations back-to-back. Even if the total volume of data is large, the transfers are

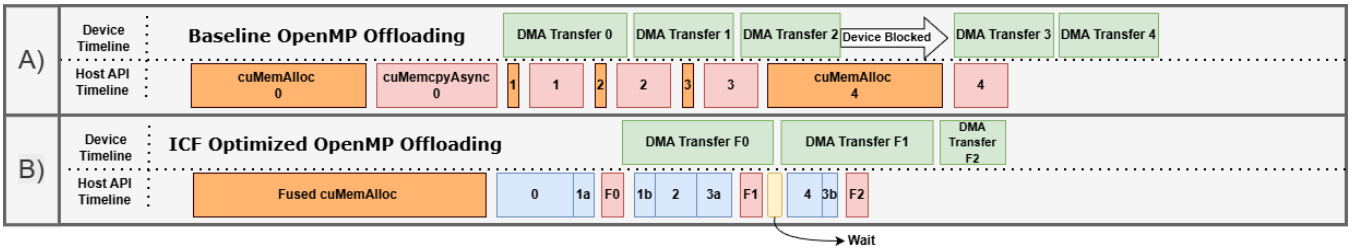


Figure 3: Conceptual trace of how transfers are realized in A) *Vanilla OpenMP* used as baseline, B) *In-Copy-Style Optimized Transfers* using the proposed ICF routine.

fragmented according to the mapping list's structure. This behavior directly reflects OpenMP's per-variable mapping semantics that lead to inefficient utilization of the interconnect when individual mappings are small or moderately sized.

Fig. 3 depicts an additional inefficiency of the OpenMP handling of multiple transfers. The figure shows the device allocation and transfer of five arguments under two different scenarios: (A) vanilla OpenMP and (B) the proposed ICF mechanism in OpenMP. The concept evolves rightwise over time, and for each, the figure shows the host and the device timelines. Both A and B implementations use an identical time scale. Under scenario *A*, the data are transferred in back-to-back allocations-transfers. Even though this scheme might hide some of the allocation time when multiple arguments are involved, because H2D transfers in OpenMP are asynchronous by default, it often results in worse performance than if all allocations are scheduled beforehand. This loss is due to allocation time irregularities caused by reuse of already allocated blocks [27, 28] and implicit synchronization barriers. Under scenario B, using the proposed ICF mechanism, allocations and transfers are batched and pipelined, approaching the optimal. This method is further explained in Section 4.

The key observation emerging from this analysis is that the runtime does not lack bandwidth, nor does it make suboptimal decisions for individual transfers. Instead, inefficiency arises because the runtime lacks a mechanism to reason about multiple mappings collectively. Even though all relevant metadata is available when data movement is initiated, the device's DMA burst transfer protocols and data fragmentation in host memory make fusion challenging in the baseline scenario. To resolve this problem, ICF exploits the staging phase to fragment H2H transfers into contiguous pinned buffers and to perform fused H2D transfers to the device.

4 In-Copy Fusion: Design and Implementation

Based on the limitations of modern systems presented in Sections 2 and 6 and the capabilities of the current LLVM OpenMP implementation presented in Section 3, this section describes the design of the In-Copy Fusion (ICF) mechanism and its integration into LLVM's OpenMP offloading runtime.

4.1 Extending the OpenMP Runtime

The bottom part of Fig. 2, namely *"OpenMP Device Functions"*, shows the LLVM call graph, proceeding from the previously described `libomptarget` implementation, with the low-level CUDA driver

Algorithm 1 In-Copy Fusion (ICF) in `targetDataBegin`.

```
1:  B ← GetPinnedBuffer                          ▷ Staging Buffer Hook
2:  F ← ∅                                        ▷ Fusable Argument List
3:  K ← 0                                        ▷ Total Fused Size
4:  for i ← 1 to ArgNum do
5:      if HasMaptypeTO(ArgTypes[i]) then
6:          F ← F ∪ {i}
7:          K ← K + ArgSizes[i]
8:      end if
9:  end for
10: S ← SelectOptimalChunkSize(K)                ▷
11: C ← ⌈K/S⌉
12: D ← AllocateDeviceBuffer(K)
13: for c ← 1 to C do
14:     PackHostToHost(B, c, F)
15:     TransferHostToDevice(D, c, B)
16: end for
17: for all i ∈ F do
18:     BindDeviceSlice(Args[i], D)
19:     ArgTypes[i] ← PRESENT
20: end for
21: return ContinueMapping(ArgTypes)
```

calls for allocation and H2D data transfers. The ICF routine is implemented in the OpenMP runtime function `targetDataBegin()`, which allocates device memory and transfers mapped data for `target` and `target data` regions. At the point where ICF executes, the runtime has already constructed the complete mapping list, including argument base pointers, sizes, and map-type flags.

ICF executes *before* the vanilla per-argument mapping loop of `targetDataBegin()`. Its role is to inspect the full mapping list once and opportunistically prepare a fused data transfer while leaving the remainder of the OpenMP runtime logic unchanged. After ICF completes, control returns to the baseline runtime, which proceeds as if each argument had been mapped independently. This mechanism integrates transparently into the OpenMP runtime, neither bypassing nor replacing the standard OpenMP mapping flow, therefore preserving all existing semantics.

Algorithm 1 outlines the ICF procedure, where B is a reusable pinned host buffer serving as an explicit staging area for data movement, F is a set used to record the indices of arguments selected for fusion, and K is a scalar used to accumulate the total size of fused

arguments. The algorithm first identifies input arguments eligible for fusion based solely on their map-type modifiers (lines 4-9). If no arguments satisfy this condition, the routine skips fusion and exits immediately, preserving the baseline runtime behavior. Once the total fused payload is known, ICF selects an appropriate chunk size (line 10) using a platform-specific calibration policy. The total number of chunks is then derived (line 11), and a single contiguous device buffer is allocated to hold the fused data (line 12). Then, the fused data is transferred, first packing the corresponding portions and then initiating the H2D transfer into the appropriate offset within the fused device buffer (lines 13-16). After all chunks have been transferred, ICF rebinds each original argument to a slice of the fused device buffer and sets as `present` all map-type modifiers, ensuring that subsequent runtime mapping logic does not trigger redundant allocation or transfers (lines 17–20). Finally, the routine returns control to the vanilla OpenMP mapping flow (line 21). From the perspective of the remainder of `targetDataBegin()` logic, all fused arguments now appear as already present in the device, allowing the unmodified part of the runtime to proceed normally while benefiting from the fused transfer performed by ICF.

The ICF routine is further populated with assertions to prevent fusion and default back to the baseline OpenMP flow before any allocations or transfers are realized. These assertions cover cases where pinned buffer allocation fails, when there are not enough arguments to fuse, and when the resulting chunking policy yields the exact same transfers as the baseline flow.

4.2 Serialization and Double-Buffered Transfer Pipeline

Fusing multiple pageable host buffers into a single transfer reduces per-transfer overhead but introduces a new challenge: serialization between host-side packing and device-side DMA. In a naive fused design, the host-to-host (H2H) packing phase must complete for the entire fused payload before the host-to-device (H2D) transfer can begin, eliminating overlap and reducing effective bandwidth. This behavior annuls one of the key performance advantages of the CUDA pageable transfer path.

To avoid this serialization, the underlying CUDA driver implementation uses a chunked double-buffered transfer pipeline. Since the staging phase of H2D communication is transparent to the runtime and is instead implemented in the closed-source CUDA driver, we design our own mechanism from scratch at the OpenMP runtime level. We pre-allocate two constant-sized pinned buffers on device startup and expose them to the runtime by modifying the OpenMP device plugin. The fused payload is divided into chunks of size S, and the two pinned staging buffers are used in a ping-pong fashion. For each chunk, ICF first packs the relevant portions of the original arguments into one staging buffer using H2H copies, and then issues an asynchronous H2D transfer from that buffer to the appropriate offset in the fused device allocation (Algorithm 1, lines 13–16).

At steady state, one staging buffer is used for packing while the other buffer's transfer is in flight, enabling overlap between H2H and H2D operations. This design restores the pipelined behavior of the CUDA pageable path while retaining the benefits of fusion and remaining entirely under runtime control.

The H2H packing step itself is implemented as a sequence of independent memory copies over the fused argument set. While this is sufficient to expose overlap and reduce end-to-end transfer time, it does not exploit potential compiler-level optimizations such as memcpy fusion or vectorization [14]. We deliberately leave such optimizations to future work, as they are orthogonal to the runtime mechanisms proposed in this work.

Fig. 3 *B* illustrates the offloading process, including the bulk device-allocation and double-buffered transfers. In the scenario depicted, the five arguments are heterogeneous and not properly sized to achieve perfect overlap. The chosen chunk size dictates partial packing of arguments (e.g., argument 0 and part of argument 1 denoted as 1*a* in the figure) so that data transfers are the same size and optimal.

4.3 Calibration of Chunking Policy

The performance of In-Copy Fusion (ICF) depends critically on the choice of chunk size used for double-buffered pageable transfers. While a single optimal chunk size per platform might be expected under idealized conditions, practical considerations such as startup overheads, stage serialization, and partial pipeline fill make the optimal choice dependent on the total transfer size. For this reason, ICF employs a runtime-based calibration-driven, empirically derived chunking policy for each platform.

Modeling pageable pipelined transfers. To motivate the calibration process, we introduce a simple analytical model of the pageable transfer pipeline implemented by ICF. Let K denote the aggregated data size of all arguments to be transferred and S the chunk size, yielding

$$n = \left\lceil \frac{K}{S} \right\rceil$$

chunks. Each chunk undergoes two stages: a host-to-host (H2H) copy from pageable memory into a pinned staging buffer, followed by a host-to-device (H2D) DMA transfer.

We model the time of these stages as affine functions:

$$T_{\mathrm{H2H}}(S) = \alpha_s + \beta_s S, \qquad T_{\mathrm{H2D}}(S) = \alpha_d + \beta_d S,$$

where α_s, α_d represent fixed per-transfer overheads and β_s, β_d capture bandwidth-limited costs.

Under a double-buffered pipeline, where H2H packing of chunk $i+1$ overlaps with H2D transfer of chunk i, the total transfer time can be approximated as

$$\begin{aligned} T_{\mathrm{pipe}}(K,S) \approx\ & T_{\mathrm{H2H}}(S) \\ & + (n-1)\cdot \max\big(T_{\mathrm{H2H}}(S), T_{\mathrm{H2D}}(S)\big) \\ & + T_{\mathrm{H2D}}(S). \end{aligned} \tag{1}$$

This expression highlights that, after the initial pipeline fill, throughput is bounded by the slower of the two stages. Optimal performance is therefore achieved when the pipeline is balanced, i.e., when H2H and H2D times are comparable, while ensuring that the non-overlapping startup and drain phases do not dominate total execution time.

Need for empirical calibration. Although Equation (1) provides insight into the behavior of the pipeline, it does not yield a closed-form optimal chunk size in practice. The parameters α and β are

Table 1: Evaluation platform specifications.

System	CPU	PCIe	GPU
A	Intel Core i7-8700K @ 3.70 GHz	Gen3	Quadro P400
B	Intel Xeon Platinum 8460Y+	Gen5	H100

platform-specific, and additional effects such as random memory access times, runtime scheduling, and PCIe behavior further complicate analytical optimization. As a result, ICF determines the chunking policy empirically via calibration using a lightweight profiling benchmark executed once per platform.

5 Experimental Evaluation and Results

This section evaluates the effectiveness and practical impact of the In-Copy Fusion (ICF) optimization across platforms, workloads, and transfer schemes. We first characterize the platform-specific calibration required by ICF, then quantify its benefits using synthetic benchmarks that isolate data-movement behavior, and finally assess its impact on representative real-world applications. Together, these experiments show when and why runtime argument fusion improves H2D communications.

5.1 Setup

We evaluated ICF on two systems, anonymized as *System A* and *System B*, each representative of a distinct PCIe generation and GPU class. System A represents a low-end personal computer system, while System B represents a modern, highly capable HPC supercomputer node. This contrast allows us to study how transfer fusion behaves across dissimilar interconnect bandwidths and host–device balances. The system specifications are summarized in Table 1.

All experiments measure offload time from the launch of an OpenMP offloaded region to its completion. This includes device allocation, data transfers, device computation, writeback of results, and device de-allocation, herein referred to as *end-to-end time*. Experiments that also measure *copy-in performance* capture only the execution time of device allocation and host-to-device transfers.

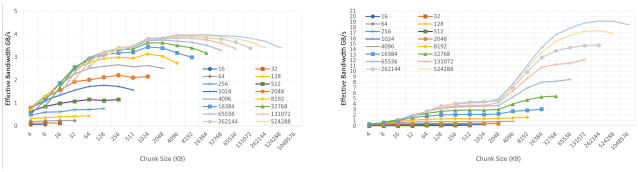
The reported *copy-out performance* includes device-to-host writeback and device-buffer de-allocation [1].

Each parameter configuration (e.g., total data size, number of mapped arguments) is executed for 51 iterations; we report the median value measured to reduce sensitivity to system noise and runtime variability.

5.2 ICF Chunking Policy Calibration

Calibration is performed by overriding the ICF runtime's default chunking policy and executing a microbenchmark that exhaustively explores the (K, S) parameter space. The benchmark offloads an empty multi-argument kernel with aggregated data size K across all arguments, fuses them with ICF using chunk size S, and measures the end-to-end execution time, including device allocation, transfer, and deallocation. This benchmark isolates the transfer pipeline from kernel execution effects.

[1]Note that ICF does not fuse device–to–host transfers in the current implementation; copy out gains therefore arise from de-allocating the single fused device allocation rather than multiple per-argument allocations



(a) System A. (b) System B.

Figure 4: Optimal transfer microbenchmark. Measured end-to-end bandwidth in GiB/s (y-axis) for a sweep of total data sizes K (lines) and chunk sizes (x-axis).

Table 2: Chunking rules for Systems A and B. For a given total transfer size K, the runtime selects the corresponding chunk size $S^\star(K)$.

Total size range K	System A: $S^\star(K)$	System B: $S^\star(K)$
$K \le 256$ KiB	K KiB	16 KiB
$K \le 4$ MiB	$K/16$ KiB	$K/32$ KiB
$K \le 128$ MiB	$K/8$ MiB	$K/4$ MiB
$K \le 256$ MiB	4 MiB	$K/4$ MiB
$K \le 1$ GiB	4 MiB	$K/16$ MiB
$K > 1$ GiB	4 MiB	512 MiB

The experiment sweeps:

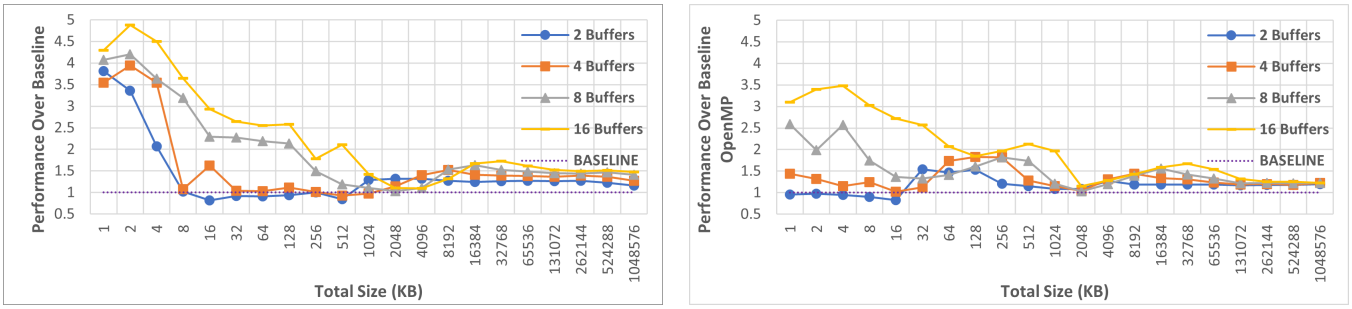
- Total transfer sizes $K \in [16\,\text{KiB}, 1\,\text{GiB}]$, in powers of two.
- For each K, chunk sizes $S \in \{4\,\text{KiB}, 8\,\text{KiB}, \ldots, K\}$.

For each (K, S) configuration, the median effective bandwidth is recorded. The optimal chunk size $S^\star(K)$ is selected as the value maximizing median bandwidth, with ties broken in favor of smaller chunks. The resulting mapping $K \mapsto S^\star(K)$ constitutes the platform-specific chunking policy used by ICF at runtime.

The measured bandwidth envelopes and the derived chunk-size rules for Systems A and B are shown in Fig. 4 and Table 2, respectively. The policy described in Table 2 is used for all experiments shown in section 5. Fig. 4 clearly demonstrates the transfer segmentation effect that leads to under-utilization of the interconnect and low effective bandwidth when S is small, while at the other end, we observe the serialization effect where effective bandwidth drops when S approaches K. The latter effect is mostly visible in Figure 4(a), where host memory speed is comparable to the effective maximum bandwidth of the interconnect, while in Figure 4(b), the serialization effect is only observed for very large K because the interconnect bandwidth is an order of magnitude larger than that of the host memory. As a result, the pipeline behavior described in Equation 1 is dominated by host-side packing (i.e., the term T_{H2H}), leading to larger optimal chunk sizes.

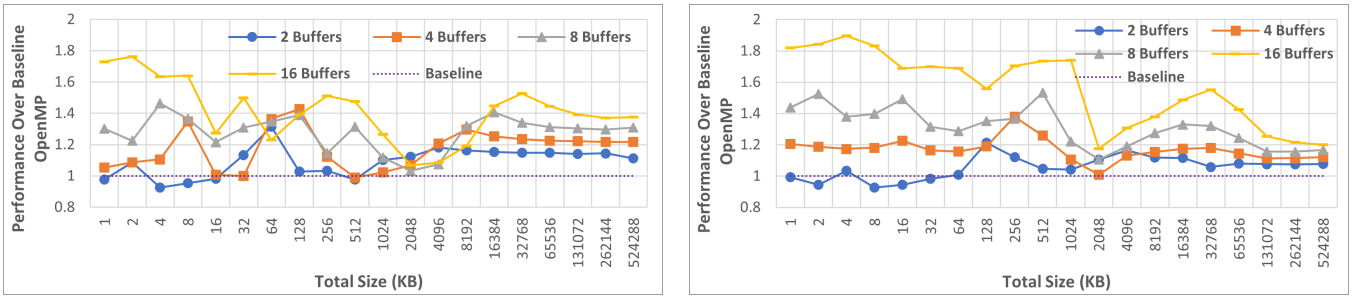
5.3 Experimental Results: Synthetic Benchmarks

We evaluate ICF's performance across two synthetic benchmarks designed to isolate and stress specific aspects of the OpenMP offloading mechanism that directly affect host–device data movement. By construction, these benchmarks decouple transfer behavior from



(a) System A. (b) System B.

Figure 5: In-Copy Fusion synthetic benchmark (empty kernel) on Systems A and B.



(a) System A. (b) System B.

Figure 6: Multi-vector add (x-vadd) synthetic benchmark on Systems A and B.

application-specific effects, enabling us to quantify the impact of argument fusion under controlled, repeatable conditions.

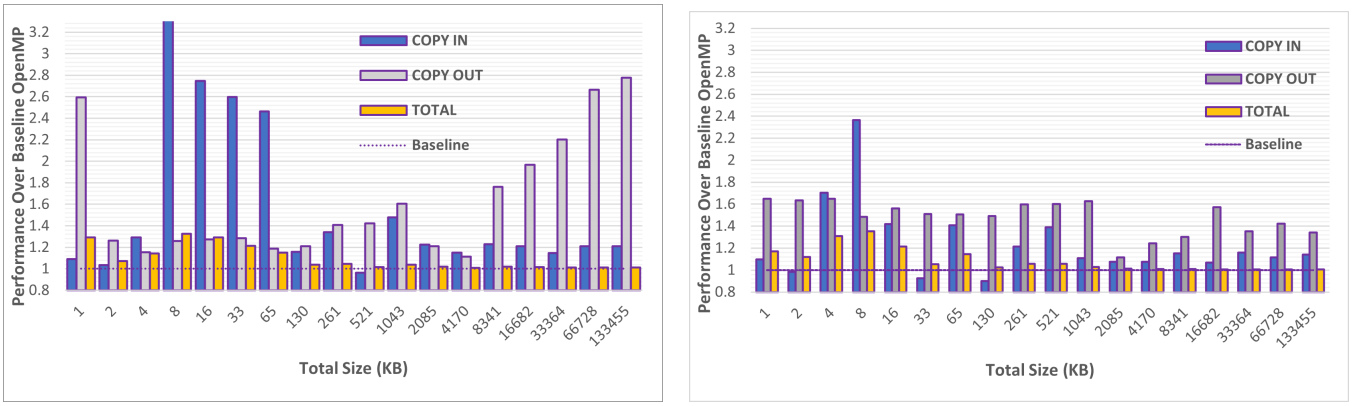
5.3.1 Empty kernel (best-case). We measure the ceiling improvement attributable to transfer coalescing by using an OpenMP target data region with no device computation and x fusable, same-sized arguments ($x \in \{1, 2, 4, 8, 16\}$). For each point, argument sizes sum to K (powers of two). Fig. 5 reports the speedup of ICF over the baseline pageable path as a function of K. For both systems, ICF achieves its largest speedups when K is distributed across many small arguments. In this region, the baseline runtime performs many small transfers, whereas ICF consolidates them into fewer, larger transfers, resulting in speedups of up to 4.8× on System A and 3.5× on System B. As K increases, performance gradually converges toward the baseline. When individual argument sizes become sufficiently large, both runtimes issue transfers that already reach high bandwidth, and the relative benefit of fusion diminishes. In this regime, ICF still provides a modest but stable improvement, plateauing at approximately 50% on System A and 30% on System B. This persistent speedup is attributed to the fused allocation and de-allocation times.

5.3.2 Multi-vector add (x-vadd). We offload $x \in \{2, 4, 8, 16\}$ single-precision vectors, compute elementwise sums on the device, and retrieve the resulting vector to the host. Argument sizes sum to K. This benchmark introduces lightweight computation while preserving fragmented input mappings, reflecting a common pattern in bandwidth-sensitive GPU kernels. Fig. 6 reports the end-to-end speedup of ICF over baseline. We observe 75% and 90% speedups for System A and System B, respectively, at low data sizes. For larger data sizes, ICF still continues to provide benefits from fused allocations and de-allocations, below 40% and 20%. The x-vadd benchmark proves that optimizing the data transfers with ICF is crucial even when computation is involved. Overall, the synthetic benchmarks show that ICF is most effective in regimes dominated by fragmented transfers and that it does not degrade performance when transfers are already bandwidth-efficient.

5.4 Experimental Results: Real-World Benchmarks

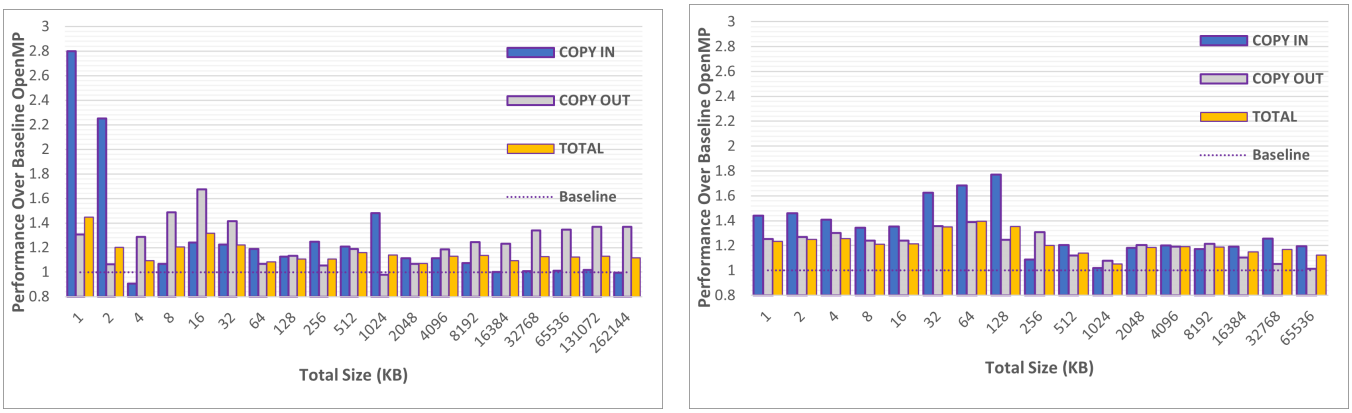
We evaluate calibrated ICF on three representative real-world workloads: Breadth-First Search (BFS) [5], Molecular Dynamics in a 2-dimensional space (MD 2D) [29], and B+Tree [5, 9]. These applications were selected to exercise diverse argument structures, ranging from balanced to highly irregular and dynamic mappings.

Breadth-First Search (BFS). represents a non-uniform transfer pattern with 6 mapped arguments of different sizes. Multiple auxiliary arrays and metadata structures are transferred together with frontier and graph representation data, resulting in a heterogeneous argument set. This configuration stresses ICF's ability to fuse unevenly sized mappings into a single payload and to pack only the overlapping portions of each argument for every chunk while still maintaining performance gains. Fig. 7 shows speedup of ICF *Copy In* – device buffer allocation and host-to-device transfers–,



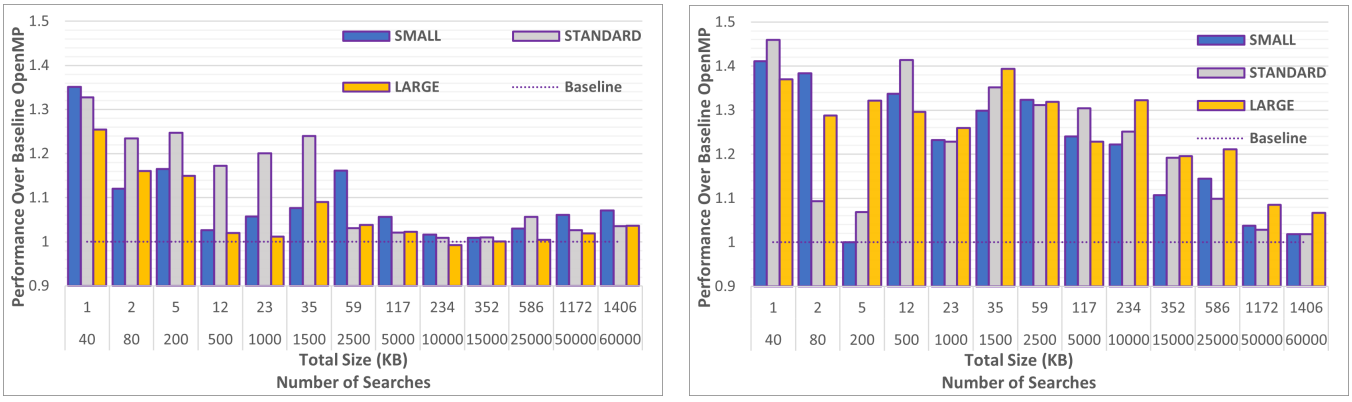
(a) System A. (b) System B.

Figure 7: Performance gains of ICF Optimization on Breadth-First Search (BFS).



(a) System A. (b) System B.

Figure 8: Performance gains of ICF Optimization on Molecular Dynamics in 2D (MD-2D).



(a) System A. (b) System B.

Figure 9: Performance gains of ICF Optimization on B+Tree Search.

Copy Out –device-to-host transfers and device de-allocation, with any ICF gain attributable to fused de-allocation– and the end-to-end speedup of the full offloading procedure with respect to the

OpenMP baseline. We report a *Copy In* maximum speedup of 3.2× and 2.4× for system A and B respectively, a maximum *Copy Out* performance of 2.8× and 1.6× for each system respectively and 33% and 35% speedups on end-to-end execution for System A and System B, respectively.

Molecular Dynamics in 2-dimensions (MD 2D). consists of a compute intensive workload followed by a memory-bound update kernel. The evaluated kernel corresponds to the simulation's update phase and constitutes a realistic kernel with multiple input arguments of equal size. This benchmark demonstrates that size-balanced, multi-input kernels, similar to the synthetic workloads evaluated in this work, occur naturally in real applications. Moreover, the update kernel is not executed in isolation but follows a preceding offloaded kernel. As a result, it executes in the presence of an already initialized device context and previously allocated device buffers, reflecting realistic execution conditions. The performance gains from ICF in these benchmarks are depicted in Fig.8. We report a *Copy In* maximum speedup of 2.8× and 78% for Systems A and B, 68% and 40% speedup for the *Copy Out* and an overall 43% and 40% maximum speedup in end-to-end execution for each platform respectively.

B+Tree Search. provides a third scenario that involves dynamic, on-demand offloading during execution. This application is a modification of b+tree [9] tailored for OpenMP and shaped to better fit a practical scenario: The handling of incoming search requests on the tree via GPGPU acceleration. This kernel evaluates how ICF handles continuous fused allocations and transfers of 4 similarly-sized arguments that evolve during execution. Since the tree is generated randomly, search times across various instances of the experiments vary. As such, the results presented in Fig. 9 feature experiments on 3 different tree sizes to affect search time (small with 100 nodes, average with 1,000, and large with 1,000,000), and the average of medians from 7 instances of the experiment is reported. Each execution handles batches of an increasing number of incoming query requests, denoted by the upper x-axis. With each batch invoking a separate offload kernel. The experiment reports the performance gain of ICF relative to the baseline of each batch served. For comparison with the other experiments presented, the lower x-axis denotes the approximate offload size. The experiment measures the execution time of each batch of search requests. In this scenario, ICF achieves a 34% and 46% maximum speedup for systems A and B, respectively. This demonstrates that ICF remains effective even under dynamic, irregular offloading patterns.

Across all three applications, ICF reduces copy-in overhead whenever data movement is distributed across multiple mapped arguments. Gains are consistently larger on System B, indicating that the benefits of consolidation increase with higher interconnect throughput and increased sensitivity to transfer granularity. In configurations where fusion is not beneficial, performance matches the baseline runtime, confirming that ICF behaves as an opportunistic optimization rather than a mandatory transformation.

6 Related Work

Host–device transfer costs over PCIe have been analyzed extensively. Both analytical and empirical models characterize transfer time as a fixed per-transfer startup cost plus a size-proportional component, explaining why batching many small copies into fewer large bursts can improve effective throughput and reduce end-to-end time [8, 12, 19, 30, 31]. We adopt this premise in our cost model and use it to decide when fusion is profitable.

Within the OpenMP offloading ecosystem, much of the literature focuses on generating efficient data mappings and *eliminating unnecessary transfers* via static or hybrid analysis [20, 22]. Recent work (e.g., OMPDart) models host/device data dependencies and rewrites code to introduce more effective mapping constructs, reducing redundant transfers across regions when lifetimes and reuse can be inferred [20]. These approaches are complementary to our goal: they reduce transfer volume and API calls by compile-time reasoning, whereas our mechanism targets runtime fusion of distinct map entries in the standardized OpenMP runtime.

Beyond fusion, classic techniques such as communication–computation overlap (pipelining, concurrent execution, and double buffering) remain standard baselines for mitigating transfer overhead. We treat these as contextual points of comparison rather than competing mechanisms for argument-level fusion [2, 3, 13, 31].

A line of work directly explores the fusion of transfers and allocations [14, 16, 23]. Muthukrishnan et al. propose *FinePack* [23], a set of GPU/interconnect hardware enhancements that transparently pack multiple fine-grained peer-to-peer remote stores into fewer, larger interconnect messages. Kessler et al. propose a global optimization that reorders and merges operand transfers and allocations to maximize coalescing opportunities across kernels [14]; evaluation on CUDA demonstrates substantial gains, particularly for small and medium-sized operands. Complementing this, Li and Kessler introduce *lazy allocation* [16], which defers allocations and exploits runtime operand information to reduce PCIe traffic. These works validate the benefits of aggregating data movement, but they typically assume source/IR control, custom runtime mechanisms, or hardware support; consequently, they do not address execution-time fusion of `map(...)` entries within the standardized OpenMP offloading runtime.

Several additional efforts are orthogonal but relevant for positioning our contribution. CuMAS captures and reorders CUDA calls from multiple applications to improve PCIe utilization and transfer phases in shared-GPU settings [4]. Compiler-driven approaches based on polyhedral reasoning demonstrate how program transformations can automatically generate heterogeneous executables [11]. While effectiveness depends on static analyzability, such transformations could complement runtime fusion by exposing larger or cleaner mapping regions. Finally, works on shared/unified virtual memory investigate reducing explicit transfers through implicit migration and paging policies, while also documenting inefficiencies under reuse-heavy or high-footprint workloads [1, 7]. These mechanisms address a different layer of the stack but could be combined with fusion to reduce the number of remaining explicit transfers.

7 Conclusion

We presented In-Copy Fusion (ICF), a runtime-only optimization for OpenMP GPU offloading that mitigates transfer fragmentation by explicitly exploiting the mandatory staging phase required for pageable host memory to aggregate multiple mapped arguments

into optimally tuned, double-buffered transfer bursts. By operating entirely within the OpenMP runtime and preserving the standard offloading semantics, ICF addresses long-standing inefficiency in directive-based GPU programming without requiring compiler changes or programmer intervention, while remaining orthogonal to further static optimizations.

Experimental results on two representative platforms show that ICF consistently improves communication efficiency when transfers are fragmented. In synthetic benchmarks, ICF achieves maximum speedups of 4.8× on the PCIe Gen3 system and 3.5× on the PCIe Gen5 system over the baseline OpenMP runtime. Across the real workloads, ICF delivers copy-in speedups of up to 3.2× on System A and 2.4× on System B, together with end-to-end speedups of up to 43% and 46%, respectively. These gains are achieved transparently at runtime and scale with interconnect capability, demonstrating that argument fusion becomes increasingly valuable as system bandwidth grows. Overall, these results show that runtime-managed staging is a practical, effective, and portable mechanism for improving OpenMP offloading performance by addressing transfer fragmentation at execution time.

The technique presented in this work is not extended to support the `target update` directive or device–to–host fusion as they follow separate runtime paths. Nevertheless, ICF can be implemented separately in future work.

Finally, since the ICF mechanism is implemented at runtime with only minimal modifications to device plugins for initializing pinned staging buffers, it remains largely independent of the underlying accelerator architecture. In future work, we aim to evaluate the technique in systems from other vendors, such as AMD, and to explore tighter integration with runtime profiling and adaptive decision-making to further generalize argument fusion across heterogeneous platforms.

Acknowledgments

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

References

[1] 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*. 1–15.

[2] Petros Anastasiadis, Nikela Papadopoulou, Georgios Goumas, Nectarios Koziris, Dennis Hoppe, and Li Zhong. 2023. Paralia: A performance aware runtime for auto-tuning linear algebra on heterogeneous systems. *ACM Transactions on Architecture and Code Optimization* 20, 4 (2023), 1–25.

[3] Burak Bastem, Didem Unat, Weiqun Zhang, Ann Almgren, and John Shalf. 2017. Overlapping data transfers with computation on GPU with tiles. In *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, 171–180.

[4] Mehmet E Belviranli, Farzad Khorasani, Laxmi N Bhuyan, and Rajiv Gupta. 2016. Cumas: Data transfer aware multi-application scheduling for shared gpus. In *Proceedings of the 2016 International Conference on Supercomputing*. 1–12.

[5] 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 *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[6] Jake Choi, Hojun You, Chongam Kim, Heon Young Yeom, and Yoonhee Kim. 2021. Comparing unified, pinned, and host/device memory allocations for memory-intensive workloads on Tegra SoC. *Concurrency and Computation: Practice and Experience* 33, 4 (2021), e6018.

[7] Bennett Cooper, Thomas RW Scogland, and Rong Ge. 2024. Shared Virtual Memory: Its Design and Performance Implications for Diverse Applications. In *Proceedings of the 38th ACM International Conference on Supercomputing*. 26–37.

[8] Jose Monsalve Diaz, Kyle Friedline, Swaroop Pophale, Oscar Hernandez, David E Bernholdt, and Sunita Chandrasekaran. 2019. Analysis of OpenMP 4.5 offloading in implementations: correctness and overhead. *Parallel Comput.* 89 (2019), 102546.

[9] Jordan Fix, Andrew Wilkes, and Kevin Skadron. 2011. Accelerating braided b+ tree searches on a gpu with cuda. In *2nd Workshop on Applications for Multi and Many Core Processors: Analysis, Implementation, and Performance (A4MMC), in conjunction with ISCA*.

[10] 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.

[11] Tobias Grosser and Torsten Hoefler. 2016. Polly-ACC transparent compilation to heterogeneous hardware. In *Proceedings of the 2016 International Conference on Supercomputing*. 1–13.

[12] Francisco Daniel Igual Peña et al. 2011. *Matrix computations on graphics processors and clusters of gpus*. Universitat Jaume I.

[13] Jaehoon Jung, Daeyoung Park, Youngdong Do, Jungho Park, and Jaejin Lee. 2020. Overlapping host-to-device copy and computation using hidden unified memory. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 321–335.

[14] 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*. 49–58.

[15] Mikhail Khalilov and Alexey Timoveev. 2021. Performance analysis of CUDA, OpenACC and OpenMP programming models on TESLA V100 GPU. *Journal of Physics: Conference Series* 1740, 1 (2021), 012056.

[16] 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)*. IEEE, 311–315.

[17] 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).

[18] 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.

[19] Daniel Lustig and Margaret Martonosi. 2013. Reducing GPU offload latency via fine-grained CPU-GPU synchronization. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 354–365.

[20] Luke Marzen, Akash Dutta, and Ali Jannesari. 2024. Static generation of efficient openmp offload data mappings. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[21] Suejb Memeti, Lu Li, Sabri Pllana, Joanna Kołodziej, and Christoph Kessler. 2017. Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: programming productivity, performance, and energy consumption. In *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*. 1–6.

[22] Alok Mishra, Abid M Malik, and Barbara Chapman. 2020. Data transfer and reuse analysis tool for gpu-offloading using openmp. In *International Workshop on OpenMP*. Springer, 280–294.

[23] Harini Muthukrishnan, Daniel Lustig, Oreste Villa, Thomas Wenisch, and David Nellans. 2023. Finepack: Transparently improving the efficiency of fine-grained transfers in multi-gpu systems. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 516–529.

[24] 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.

[25] OpenMP Architecture Review Board. 2008. OpenMP Application Program Interface Version 3.0. http://www.openmp.org/mp-documents/spec30.pdf

[26] 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.

[27] 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.

[28] Marcos Rodriguez, Irune Yarza, Leonidas Kosmidis, and Alejandro J Calderón. 2025. Analysis of GPU Memory Allocation Characteristics. In *16th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 14th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2025)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 1–1.

[29] SOLLVE Project. 2019. *SOLLVE Benchmarks: Performance Benchmarks for OpenMP Features*. https://github.com/SOLLVE/benchmarks GitHub repository (release tag sc19uvm), accessed 2026-02-08.

[30] Dionisis-Odysseas 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: Workshops and Special Sessions*. 82–87.

[31] Ben Van Werkhoven, Jason Maassen, Frank J Seinstra, and Henri E Bal. 2014. Performance models for CPU-GPU data transfers. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 11–20.