

UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche, Informatiche e Matematiche

Dottorato di ricerca in

Computer and Data Science for Technological and Social Innovation

Ciclo XXXVIII

Tesi di Dottorato

Improving the Efficiency of Data Transfers in Heterogeneous Computing: A Unified View of Static and Dynamic Data Fusion and Splitting Techniques

Relatore / Tutor:
Prof. Andrea Marongiu

Candidato:
Dionysios-Odysseas Sotiropoulos

Correlatore / Co-Tutor:
Prof. Alessandro Capotondi

Coordinatore del Corso di Dottorato:
Prof. Andrea Marongiu

Anno Accademico 2024/2025

Abstract

Modern heterogeneous systems commonly combine general-purpose CPUs with discrete accelerators such as GPUs. While these accelerators provide high computational throughput, their effective use often depends on the efficiency of data movement between physically distinct host and device memories. In loosely coupled systems, offloaded execution requires allocation, host-to-device transfer, synchronization, device execution, writeback, and deallocation. This thesis studies a specific inefficiency in this execution model: communication fragmentation. When kernels or OpenMP target regions use multiple input arrays or mapped variables, the runtime may materialize each object as an independent allocation and transfer, even when the aggregate data volume could be communicated more efficiently.

The thesis frames this problem as one of communication granularity. It first evaluates manual Argument Fusion as a controlled application-level transformation in which multiple logical kernel arguments are packed into a single contiguous buffer, transferred as one physical payload, and accessed through offset-based addressing. Argument Fusion is used to characterize the performance effects of transfer aggregation, allocation and deallocation consolidation, pageable versus pinned host memory, and synchronous versus asynchronous transfer submission. The evaluation shows that fusion is most effective when communication is fragmented across several small or medium-sized arguments, and that its benefits arise not only from larger host-to-device transfers but also from reduced memory-management multiplicity.

Building on this characterization, the thesis presents In-Copy Fusion (ICF), a runtime-level fusion mechanism implemented inside LLVM's OpenMP offloading runtime. ICF moves the fusion principle from manually rewritten application code into the OpenMP data-mapping path. At runtime, it identifies eligible input mappings, packs their data into runtime-managed pinned staging buffers, transfers the data into a fused device allocation, and rebinds the original OpenMP mappings to slices of that allocation. This preserves source-level code, kernel interfaces, and OpenMP mapping semantics while changing the physical organization of communication. To avoid the serialization of naive monolithic staging, ICF splits the fused payload into calibrated chunks, allowing host-side packing and device-side transfer to proceed as a staged pipeline.

The proposed mechanisms are evaluated using controlled synthetic benchmarks and application-level workloads, including regular fragmented inputs, vector addition, molecular-dynamics update kernels, breadth-first search, Service B+Tree query processing, and a static operand-layout stress case inspired by prior transfer-fusion work. The results show that communication-granularity optimization can improve copy-in and end-to-end execution when offloaded data movement is fragmented and runtime communication overheads are significant. They also identify the limits of the approach: benefits diminish when transfers are already bandwidth-efficient, when computation or writeback dominates total time, when baseline runtime behavior incidentally overlaps allocation and transfer costs, or when mappings are not safely eligible for fusion.

Overall, this thesis demonstrates that the physical granularity of host–device data movement is a first-order performance factor in discrete heterogeneous systems. Manual Argument Fusion exposes and quantifies this effect, while In-Copy Fusion shows that part of the same opportunity can be recovered transparently inside an OpenMP offloading runtime through bounded, semantics-preserving runtime optimization.

Acknowledgements

I would like to express my gratitude to my supervisors, Andrea Marongiu and Alessandro Capotondi, for guiding me through this research journey, and to my colleagues in the small lab at Unimore, especially Gianluca Bellocchi, for his help and friendship.

Finally, I would like to thank my friends and family, especially Tita-Konstantina Nikolopoulou, for supporting me through these turbulent years with her love and sympathy.

Contents

Abstract **i**

1 Introduction **1**
1.1 Problem Statement . . . 3
1.2 Thesis Statement . . . 4
1.3 Terminology . . . 4
1.4 Research Questions . . . 5
1.5 Contributions . . . 6
1.6 Scope and Non-Goals . . . 8
1.7 Thesis Organization . . . 8

2 Background **10**
2.1 From Scaling Limits to Heterogeneous Computing . . . 10
2.1.1 From Frequency Scaling to Specialization . . . 10
2.1.2 The Memory Wall in Heterogeneous Systems . . . 11
2.2 Programming Models for Heterogeneous Offloading . . . 11
2.3 Standard Offloading Paradigm in Loosely Coupled Systems . . . 13
2.4 Transfer Segmentation and Communication Granularity . . . 15
2.5 Argument Fusion as a Communication-Granularity Optimization . . . 17
2.6 From Manual Fusion to Runtime-Level Fusion . . . 18
2.7 Chapter Summary . . . 18

3 Related Work **20**
3.1 Host–Device Communication Characterization . . . 20
3.2 Programming Models and Offloading Abstractions . . . 21
3.3 OpenMP Offloading and Data-Mapping Optimization . . . 22
3.4 Transfer Scheduling, Pipelining, and Runtime Overlap . . . 23
3.5 Transfer Aggregation, Operand Fusion, and Static Layout Optimization . . . 24
3.6 Runtime Memory Management and Allocation Effects . . . 26
3.7 Unified, Shared, and Managed Memory . . . 26
3.8 Positioning of This Thesis . . . 27

4 Evaluation of Argument Fusion **30**
4.1 Chapter Overview . . . 30
4.2 Motivation and Communication Effects . . . 30
4.2.1 Execution Actors and Communication Pipeline . . . 31
4.2.2 Runtime Manifestation of Transfer Inefficiencies . . . 32

4.2.3 Measurement Methodology for Runtime Analysis . . . 34
4.2.4 Quantifying Device Inactivity . . . 34
4.2.5 Quantifying Fragmentation of Host-to-Device Transfers . . . 35
4.2.6 Unified Interpretation and Optimization Implications . . . 36
4.3 Argument Fusion Methodology . . . 37
4.3.1 Baseline Multi-Argument Offloading . . . 37
4.3.2 Manual Argument Fusion Transformation . . . 37
4.3.3 Optimized Configurations . . . 40
4.4 Experimental Setup . . . 42
4.4.1 Platforms . . . 42
4.4.2 Measurement Methodology . . . 43
4.4.3 Synthetic Benchmark Parameters . . . 44
4.4.4 Application Benchmarks . . . 45
4.5 Synthetic Benchmark Evaluation . . . 47
4.5.1 Communication-Path Impact of Argument Fusion . . . 47
4.5.2 Runtime Component Analysis of Fusion . . . 49
4.5.3 Impact of Pinned Host Memory . . . 55
4.5.4 Impact of Asynchronous Transfers . . . 56
4.5.5 Fusion+ and Combined Communication Optimizations . . . 59
4.5.6 Synthetic Evaluation Synthesis . . . 61
4.6 Application-Level Evaluation on PolyBench . . . 62
4.6.1 Benchmark Selection . . . 63
4.6.2 Argument Counts and Dataset Sizes . . . 63
4.6.3 Performance Results . . . 64
4.6.4 Interpretation . . . 67
4.7 Discussion . . . 68
4.7.1 Conditions for Effective Argument Fusion . . . 68
4.7.2 Transfer Aggregation versus Allocator Consolidation . . . 69
4.7.3 Interaction with Pinned Memory and Asynchronous Transfers . . . 69
4.7.4 Limits of Manual Fusion and Runtime Implications . . . 70
4.8 Summary . . . 71

5 In-Copy Fusion: Runtime Argument Fusion for OpenMP GPU Offloading 73
5.1 From Manual Fusion to Runtime Fusion . . . 73
5.1.1 Research-Question Scope . . . 73
5.1.2 Contributions . . . 73
5.1.3 Chapter Organization . . . 74
5.2 The Runtime Fusion Opportunity in LLVM OpenMP . . . 74
5.2.1 OpenMP Target Mapping Syntax . . . 74

5.3 Why Runtime Fusion Is Difficult . . . 77
5.3.1 Disjoint Host Allocations and OpenMP Mapping Semantics . . . 77
5.3.2 Pageable-Memory Staging as the Enabling Layer . . . 78
5.3.3 Observed Pageable Transfer Behavior . . . 79
5.3.4 Baseline Interleaving and Incidental Allocation–Transfer Overlap . . . 80
5.4 In-Copy Fusion Design . . . 80
5.4.1 From Monolithic Staging to Chunked Staged Fusion . . . 81
5.4.2 Chunked Transfer Pipeline . . . 82
5.4.3 Directive-Dependent Synchronization Effects . . . 82
5.4.4 Calibration-Driven Chunk Selection . . . 83
5.5 Runtime Implementation in `libomptarget` . . . 83
5.5.1 Integration Point and High-Level Flow . . . 84
5.5.2 Fusion Eligibility Analysis . . . 85
5.5.3 Fused Device Allocation and Slice Layout . . . 87
5.5.4 Pinned Staging Buffer Management . . . 87
5.5.5 Chunk Construction and Host-Side Packing . . . 87
5.5.6 Asynchronous H2D Submission . . . 88
5.5.7 Present-Table Integration and Device Pointer Rebinding . . . 89
5.5.8 Fallback Conditions and Runtime Safety . . . 89
5.5.9 Cleanup and Mapping Lifetime Management . . . 89
5.6 Experimental Methodology . . . 90
5.6.1 Experimental Platforms . . . 90
5.6.2 Runtime Configurations . . . 91
5.6.3 OpenMP Execution Forms . . . 91
5.6.4 Benchmark Design . . . 92
5.6.5 Timing Methodology . . . 95
5.6.6 Effective Bandwidth . . . 95
5.7 Experimental Results . . . 96
5.7.1 Chunking Policy Calibration . . . 96
5.7.2 Synthetic Benchmark Evaluation . . . 97
5.7.3 Application-Level Evaluation . . . 103
5.7.4 Cross-Benchmark Interpretation . . . 106
5.8 Discussion . . . 108
5.8.1 Scope of the Optimization . . . 108
5.8.2 When ICF Helps and When It Saturates . . . 108
5.8.3 Runtime Effects: Synchronization, Allocation, and Chunking . . . 108
5.8.4 Backend Dependence and Portability . . . 109
5.8.5 Relationship to Manual Fusion and Current Limitations . . . 109
5.9 Summary . . . 110

6 Conclusions and Future Work **111**
6.1 Thesis Conclusions 111
6.2 Answers to the Research Questions 112
6.3 Limitations 113
6.4 Future Work 114

References **116**

List of Figures

2.1 Standard offloading execution model in loosely coupled heterogeneous systems. 14
2.2 Example host–device bandwidth curve as a function of transfer size. Small transfers underutilize the interconnect, while larger transfers approach the sustained bandwidth limit. 16

4.1 Conceptual communication pipeline for host–device transfers in CUDA-based systems. 31
4.2 Nsight Systems trace of consecutive host–to–device transfers showing gaps in DMA activity. 35
4.3 Transfer inefficiencies: (a) device inactivity and (b) transfer fragmentation. . . . 36
4.4 Conceptual host–device activity traces for evaluated configurations 41
4.5 Communication-path speedup of Argument Fusion over the baseline pageable synchronous implementation. 47
4.6 Device allocation speedup of Argument Fusion over the baseline pageable synchronous implementation. 49
4.7 Nsight Systems trace of consecutive cudaMalloc calls 49
4.8 Host-to-device transfer speedup of Argument Fusion over the baseline pageable synchronous implementation. 51
4.9 Device deallocation speedup of Argument Fusion over the baseline pageable synchronous implementation. 53
4.10 Individual cudaFree timings in the fragmented and fused cases 55
4.11 Host-to-device transfer speedup of pinned memory over pageable memory for varying transfer sizes and argument counts. 56
4.12 Host-to-device transfer speedup of pageable asynchronous transfers over pageable synchronous transfers. 57
4.13 Host-to-device transfer speedup of pinned asynchronous transfers over the baseline pageable synchronous implementation. 58
4.14 Host-to-device transfer speedup of Fusion+ over the baseline pageable synchronous implementation. 59
4.15 Communication-path speedup of Fusion+ over the baseline pageable synchronous implementation. 60
4.16 Measured end-to-end speedup of Argument Fusion relative to the baseline implementation across PolyBench applications. 64
4.17 Measured end-to-end speedup of Fusion+ relative to the baseline implementation across PolyBench applications. 64

4.18 Measured host-to-device transfer speedup of Argument Fusion relative to the baseline implementation across PolyBench applications. 65
4.19 Measured host-to-device transfer speedup of Fusion+ relative to the baseline implementation across PolyBench applications. 66
4.20 Normalized execution breakdown of the baseline pageable synchronous implementation for `gemver`. 67
4.21 Normalized execution breakdown of the fused pageable synchronous implementation for `gemver`. 67

5.1 LLVM OpenMP offloading communication flow after source-level OpenMP directives and map clauses have been lowered to runtime calls. The ICF integration point is inside the runtime mapping path, where `libomptarget` has access to the mapping list before backend allocation and host-to-device transfer operations are fully materialized. 76
5.2 Nsight Systems traces of staged pageable CUDA transfers 79
5.3 Conceptual comparison between baseline OpenMP offloading and ICF 81
5.4 Baseline pageable CUDA communication versus monolithic runtime fusion . . 81
5.5 Benchmark taxonomy used in the ICF evaluation. Kernel-style benchmarks are controlled OpenMP offload patterns used to expose runtime mapping behavior before moving to application-level workloads. 93
5.6 Effective copy-in bandwidth obtained by ICF under different staged chunk sizes and total fused communication volumes. 96
5.7 EMPTY benchmark total speedup under explicit data-management execution. . 97
5.8 EMPTY benchmark component-level speedup on System A under explicit data-management execution. 98
5.9 EMPTY benchmark speedup on System A under ordinary `target` execution. . 99
5.10 XVADD benchmark speedup under explicit data-management execution. 100
5.11 Kessler-Interleaved operand-layout conflict. A single static operand order can make several kernel operand sets contiguous, but the conflicting operand set K_3 remains split. Runtime ICF differs by constructing a temporary fused communication representation for the concrete mapping set of each offload event. 101
5.12 Kessler-Interleaved total speedup on System A 102
5.13 MD-2D copy-in, copy-out, and total speedup under explicit data-management execution. 103
5.14 MD-2D speedup on System A under ordinary `target` execution. 104
5.15 BFS copy-in, copy-out, and total speedup under explicit data-management execution. 105
5.16 Service B+Tree speedup for dynamic query batches 105

List of Tables

1.1 Terminology used for host–device communication and communication-granularity optimization. . . . 5
1.2 Mapping between the high-level research questions, thesis contributions, operational objectives, and supporting chapters. . . . 6

2.1 High-level comparison of heterogeneous programming models from the data-movement perspective considered in this thesis. . . . 13

3.1 Positioning of this thesis relative to related work. . . . 28

4.1 Evaluated communication configurations. . . . 42
4.2 Experimental platforms. . . . 42
4.3 Communication characteristics of evaluated benchmarks. . . . 46
4.4 Summary of the runtime factors influencing the synthetic communication-path results. . . . 62
4.5 Practical guidelines derived from the Argument Fusion evaluation. . . . 71

5.1 ICF implementation mechanisms and runtime roles. . . . 84
5.2 ICF fusion eligibility policy. . . . 86
5.3 Experimental platforms used in the ICF evaluation. . . . 91
5.4 Benchmarks used throughout the ICF evaluation. . . . 93
5.5 Application-level mapping structure used in the ICF evaluation. . . . 94
5.6 Cross-benchmark interpretation of ICF behavior. . . . 107

List of Code

4.1 Standard offloading paradigm code for a generic kernel with 3 input arguments. 38
4.2 Optimized offloading with Argument Fusion for a generic kernel with 3 argument inputs. 39
4.3 Pinned memory transformation. 39
4.4 Asynchronous transfer transformation using the default stream. 40
4.5 CUDA warmup routine used prior to benchmark execution. 44
5.1 Simplified OpenMP target data mapping example. 74

1. Introduction

The evolution of high-performance computing has increasingly relied on heterogeneous architectures. As improvements from frequency scaling and general-purpose microarchitectural techniques became harder to sustain, system designers progressively turned toward parallel and specialized processing units to continue improving performance and energy efficiency [13, 14, 19]. Modern platforms therefore commonly combine general-purpose CPUs with accelerators such as graphics processing units (GPUs), field-programmable gate arrays (FPGAs), and domain-specific processing units. These accelerators can provide substantial throughput for suitable workloads, but their effective use depends on more than raw computational capability.

A central cost of heterogeneous execution is data movement. In many accelerator-based systems, the main processor (*host*) and the accelerator (*device*) do not operate over a single uniform memory space. Instead, data must be allocated, transferred, and eventually released across distinct host and device memory domains. In loosely coupled systems, such transfers cross an interconnect such as PCIe, NVLink, or a platform-specific host–accelerator communication fabric. The resulting overhead can be large enough to affect, or even dominate, the end-to-end performance of offloaded computation [17, 40].

This concern is not specific to one accelerator technology. Early work in the research context of this thesis involved profiling in HERO, a RISC-V manycore accelerator research platform implemented on FPGA [26]. Preliminary profiling of data movement between the processing system and programmable logic through DMA exposed the same qualitative behavior that motivates this thesis: effective transfer rate depends strongly on transfer size, and small transfers can achieve much lower throughput than larger transfers.

This thesis focuses on that organization. The key observation is that host–device data movement cost depends not only on the total number of bytes transferred, but also on how those bytes are decomposed into concrete communication operations. A large transfer can amortize startup, runtime, synchronization, and interconnect-level costs over a larger payload. A sequence of smaller transfers repeatedly pays part of these costs and may operate in a low-efficiency region of the communication path [56, 46]. Consequently, two executions that move the same total amount of data may have different performance if one communicates the data as a few large operations and the other communicates them as many small operations.

The thesis studies this issue through the lens of *communication granularity*. Communication granularity denotes the physical organization of data movement and associated memory-management operations: how many allocations are issued, how many transfers are submitted, how large each transfer is, how many synchronization points are introduced, and how the runtime represents logically distinct data objects in device memory. From this perspective, data-movement inefficiency is not only a matter of insufficient interconnect bandwidth. It is also a matter of how software layers expose, preserve, or transform the structure of application data when preparing

accelerator execution.

This issue is closely connected to heterogeneous programming models. Low-level models such as CUDA expose explicit control over device allocation, transfers, streams, and synchronization [43, 40]. Other models, including OpenCL, HSA-oriented runtimes, SYCL, and directive-based systems such as OpenMP, raise the level of abstraction and aim to improve portability and programmer productivity [53, 20, 25, 45]. These programming models differ in syntax, execution model, and runtime organization, but they share a common systems problem: logical program objects must eventually be materialized as physical memory-management and data-transfer operations.

In typical offloaded execution, a kernel or OpenMP `target` region may operate on multiple input arrays, output arrays, or mapped objects. High-level runtime systems commonly materialize these objects according to the logical mapping structure exposed by the programming model. In OpenMP, for example, data-mapping clauses describe list items that are mapped to a device data environment [45]. The resulting execution may therefore contain multiple device allocations, multiple host-to-device copy-in operations, multiple device-to-host copy-out operations, and multiple deallocations, even when the aggregate data volume could in principle be communicated more efficiently. This creates a mismatch between the logical structure of the program and the physical structure of the communication path.

This thesis investigates two related ways of changing communication granularity. The first is *Argument Fusion*, an application-level transformation in which multiple logical kernel arguments are packed into a single contiguous buffer, transferred as one physical payload, and accessed through offset-based addressing. Argument Fusion is not introduced in this thesis as a new optimization principle. Related forms of transfer aggregation, lazy allocation, and operand transfer fusion have already been proposed in prior work [27, 24], and a preliminary version of the Argument Fusion evaluation has been published separately [51]. Instead, this thesis uses Argument Fusion as a controlled transformation for characterizing the performance effects of communication granularity, transfer aggregation, and allocation/deallocation multiplicity.

The second mechanism is *In-Copy Fusion* (ICF), a runtime-level implementation of the same general principle inside LLVM's OpenMP offloading runtime [52]. The choice of OpenMP is deliberate. Lower-level models such as CUDA expose the allocation and transfer path directly to the programmer, making manual transformations such as Argument Fusion possible but application-specific. Portable heterogeneous models such as OpenCL, SYCL, and HSA-oriented runtimes expose different abstraction and backend choices, but still ultimately require logical program objects to be materialized as concrete memory-management and transfer operations. OpenMP is used here as a representative directive-based offloading model because its `map` clauses expose logical data-mapping intent while delegating allocation, transfer, synchronization, and mapping-table management to the runtime [45, 36]. This makes the OpenMP runtime a suitable point for studying whether communication granularity can be changed transparently below the source-level programming abstraction.

Unlike manual Argument Fusion, ICF does not require the programmer to reorganize application data structures or modify kernel interfaces. It operates inside the OpenMP data-mapping path, where the runtime observes the set of mapped variables before issuing the corresponding device allocations and host-to-device transfers. ICF selects eligible input mappings, packs their data into runtime-managed pinned staging buffers, transfers staged chunks into a fused device allocation, and then rebinds the original OpenMP mappings to slices of that allocation. In this way, the logical OpenMP mapping abstraction is preserved while the physical organization of communication is changed.

In the title, *static* refers to Argument Fusion performed through an explicit application-level data layout, whereas *dynamic* refers to In-Copy Fusion performed by the offloading runtime according to the mappings encountered during execution. These two forms correspond to the two main levels at which communication granularity is investigated in this thesis.

The title of this thesis refers to both fusion and splitting techniques. In the work presented here, *fusion* refers to the aggregation of multiple logical communication objects into a smaller number of physical transfers and allocations. *Splitting* refers to the controlled chunking of fused payloads inside the ICF transfer pipeline. A naive runtime fusion strategy would pack all participating data into one large staging buffer and then issue one monolithic transfer. Such a design reduces transfer fragmentation, but it may also serialize hidden stages of the communication. ICF therefore splits the fused payload into calibrated chunks so that host-side staging and device-side transfer can proceed as a pipeline. The thesis thus studies both sides of communication-granularity control: aggregating fragmented mappings to reduce runtime overhead, and splitting the fused transfer into staged chunks to preserve overlap and avoid monolithic serialization.

1.1 Problem Statement

In loosely coupled heterogeneous systems, offloaded execution commonly follows a sequence of device allocation, host-to-device copy-in, kernel execution, device-to-host copy-out, and deallocation [17, 40]. When a kernel or OpenMP `target` region uses multiple input objects, this sequence is often replicated per argument or per mapped variable according to the mapping structure exposed to the runtime [45, 36]. The resulting execution pattern introduces two related inefficiencies.

First, segmented communication increases runtime orchestration overhead. Each allocation, transfer submission, synchronization point, and deallocation may contribute fixed cost independent of the useful payload. When many mappings are processed independently, these fixed costs accumulate.

Second, segmented communication can reduce effective transfer efficiency. Small transfers often underutilize the host–device interconnect because transfer startup costs, protocol overheads, and runtime preparation costs are amortized over too few bytes [56, 46, 40]. A sequence of small transfers can therefore take longer than a single aggregated transfer of the same total size.

These inefficiencies are especially relevant in directive-based offloading models such as OpenMP. The programming model exposes a convenient abstraction in which variables are mapped independently, but the runtime must eventually lower those mappings into concrete allocation and transfer operations. This creates a mismatch between the logical structure of the program and the physical granularity of communication. The problem addressed in this thesis is how to reduce this mismatch while preserving program semantics and without assuming that every application can be manually rewritten around fused data layouts.

1.2 Thesis Statement

The thesis advanced in this work is the following:

> Fragmented host–device communication is a first-order source of overhead in loosely coupled heterogeneous systems, and its cost can be reduced by changing the physical granularity of data movement and allocation. Manual Argument Fusion provides a controlled way to characterize this effect, while runtime-level In-Copy Fusion demonstrates that part of the same benefit can be obtained transparently inside a high-level offloading runtime, implemented here in LLVM OpenMP, by aggregating eligible mappings through a staged fused transfer path.

The thesis does not claim that fusion universally accelerates all offloaded workloads. The expected benefit depends on the number of independently transferred objects, their size distribution, the relative cost of communication and computation, the behavior of the runtime allocator and transfer path, and the ability of the runtime to preserve mapping semantics. The contribution is therefore not a universal replacement for existing data-movement optimizations, but a systematic study and implementation of communication-granularity optimization for a specific and common class of offloading inefficiencies.

1.3 Terminology

Because host–device communication is discussed across several abstraction levels, this thesis uses the following terminology consistently.

Term	Meaning in this thesis
Host–device	Adjectival form referring to communication, memory management, or execution involving both the host processor and the accelerator device.
Host-to-device (H2D)	Directional transfer from host memory to device memory. In OpenMP terminology, this corresponds primarily to copy-in for input mappings.
Device-to-host (D2H) / Writeback	Directional transfer from device memory to host memory. In OpenMP terminology, this corresponds primarily to copy-out for output mappings.
Copy-in	Runtime action that makes host data available on the device before accelerator execution.
Copy-out	Runtime action that makes device results available to the host after accelerator execution.
Fusion	Aggregation of multiple logical data objects into fewer physical communication or memory-management operations.
Splitting	Partitioning of a fused payload into staged chunks to avoid monolithic serialization and to enable overlap between host-side packing and device-side transfer.
Communication granularity	The physical organization of communication operations, including the number and size of transfers, allocations, synchronizations, and deallocations.

Table 1.1: Terminology used for host–device communication and communication-granularity optimization.

1.4 Research Questions

This thesis is organized around three high-level research questions, each supported by more specific operational objectives.

RQ1. How does communication granularity affect host–device communication performance, and under which conditions can Argument Fusion improve it?

The corresponding operational objectives are:

(a) to determine how transfer segmentation affects host–device communication efficiency in loosely coupled heterogeneous systems;

(b) to identify the transfer-size and argument-count regimes in which Argument Fusion improves performance relative to a baseline multi-argument offloading implementation;

(c) to determine how much of the observed benefit comes from transfer aggregation itself and how much is associated with secondary runtime effects such as reduced allocation and deallocation multiplicity; and

(d) to evaluate how Argument Fusion interacts with established CUDA communication optimizations, especially pinned host memory and asynchronous transfer submission.

RQ2. Can communication fusion be moved from application-level restructuring into the offloading runtime while preserving the programming model semantics?

The corresponding operational objective is to determine whether fusion can be moved from application-level data-layout restructuring into lower-level programming-model offloading runtime layers while preserving source-level code, kernel interfaces, and mapping semantics.

RQ3. How can runtime-level fusion be organized effectively, and under which workloads does it improve performance?

The corresponding operational objectives are:

(a) to determine how a runtime-managed fused transfer should be organized so that explicit host-side staging does not become a new serialization bottleneck; and

(b) to determine in which synthetic and application-level workloads runtime-level fusion improves copy-in and end-to-end execution time, and where its benefits saturate or degrade.

Table 1.2 summarizes the relationship between these high-level research questions, the main thesis contributions, their operational focus, and the chapters in which they are addressed.

Table 1.2: Mapping between the high-level research questions, thesis contributions, operational objectives, and supporting chapters.

RQ	Contribution	Operational focus	Main chapter
RQ1	Communication-granularity characterization through Argument Fusion	Transfer segmentation; transfer-size and argument-count regimes; allocation, H2D transfer, and deallocation effects; pinned memory and asynchronous transfers.	Chapter 4
RQ2	In-Copy Fusion	Runtime-level fusion while preserving source-level code, kernel interfaces, and mapping semantics.	Chapter 5
RQ3	Runtime organization and experimental evaluation of ICF	Organization of runtime-managed fused transfers; synthetic and application-level evaluation; favorable and limiting regimes.	Chapter 5

1.5 Contributions

This thesis makes three principal contributions, each supported by more specific analytical, experimental, or implementation contributions.

1. **Systematic characterization of communication granularity through Argument Fusion.** The thesis frames transfer segmentation as a runtime and communication-granularity problem rather than only as a question of total transferred bytes. It distinguishes between device inactivity between transfers, active-copy inefficiency caused by fragmented transfer payloads, and allocator/runtime overheads surrounding the transfer path.

    - **Controlled evaluation of manual Argument Fusion.** The thesis evaluates Argument Fusion as a controlled application-level transformation for reducing communication fragmentation, extending the analysis beyond earlier transfer-aggregation and operand-fusion work [27, 24, 51]. The evaluation separates device allocation, host-to-device transfer, and deallocation effects; studies the impact of argument count and total transfer size; and compares fusion against pinned-memory and asynchronous-transfer configurations.
    - **Interaction with standard CUDA communication optimizations.** The thesis evaluates not only Argument Fusion in isolation, but also its combination with pinned host memory and asynchronous transfers. This shows that fusion, pinned memory, and asynchronous submission affect partially distinct components of the communication pipeline and can therefore be complementary.

2. **In-Copy Fusion, a runtime-level fusion mechanism in LLVM OpenMP offloading.** The thesis presents ICF, an implementation inside LLVM's `libomptarget` runtime that aggregates eligible OpenMP input mappings without requiring application source-code changes, kernel-interface changes, or application-visible host-memory layout restructuring [52]. The implementation uses runtime-managed pinned staging buffers, fused device allocation, present-table rebinding, fallback behavior, and cleanup at mapping-lifetime granularity.

    - **A chunked staged transfer pipeline for runtime-managed fusion.** The thesis shows that naive monolithic runtime fusion can serialize host-side packing before device-side transfer. To address this, ICF partitions the fused payload into calibrated staged chunks, allowing host-side packing and device-side DMA transfer to overlap. This controlled splitting of the fused payload is used to make runtime fusion practical under pageable host-memory conditions.

3. **An experimental evaluation across synthetic and application-level workloads.** The thesis evaluates the proposed mechanisms on controlled synthetic benchmarks and application-level workloads, including regular fragmented inputs, vector addition with computation and copy-out, a regular MD-2D update kernel, an irregular BFS workload, a dynamic Service B+Tree query scenario, and a Kessler-style interleaved operand-layout case. The evaluation characterizes both favorable regimes and boundary cases where computation,

copy-out, irregular mapping structure, synchronization, or already efficient transfers limit the observed end-to-end benefit.

1.6 Scope and Non-Goals

The scope of this thesis is host–device data movement in loosely coupled heterogeneous systems, with particular emphasis on CUDA-based systems and OpenMP GPU offloading through LLVM's `libomptarget`. The underlying optimization principle is not specific to OpenMP: it concerns the physical granularity with which required communication operations are realized. OpenMP is the implementation target because its directive-level mapping abstraction exposes a concrete runtime point at which multiple logical mappings can be inspected before they are materialized as backend allocation and transfer operations [45, 30, 52].

The work focuses on communication organization around offloaded kernels and target regions, especially the allocation and host-to-device copy-in path for multiple independently mapped inputs. It does not propose a new accelerator architecture, a new interconnect, or a new programming model. It also does not attempt to optimize the computational kernels themselves. Kernel execution time, algorithmic arithmetic intensity, and device-side memory hierarchy optimization are treated as workload properties rather than as direct optimization targets.

The results should therefore be interpreted as evidence for a specific systems claim: communication granularity matters, manual fusion can expose and quantify the associated opportunity, and a runtime can recover part of that opportunity transparently when the mapping structure and execution regime are favorable. Detailed limitations of the current implementation and broader future-work directions are discussed in Chapter 6.

1.7 Thesis Organization

The remainder of the thesis is organized as follows.

Chapter 2 provides the architectural and programming-model background required by the rest of the thesis. It discusses the historical shift toward heterogeneous computing, the role of the memory wall, the standard offloading execution model in loosely coupled systems, and the reason transfer segmentation matters for host–device communication. It also positions accelerator programming models and explains why OpenMP offloading is a relevant target for runtime-level data-movement optimization.

Chapter 3 positions the thesis with respect to prior work on host–device communication, transfer scheduling, transfer aggregation, runtime communication management, and OpenMP offloading. The chapter distinguishes between techniques that reduce the amount of data movement, techniques that hide communication through overlap or scheduling, and techniques that change the physical granularity of communication. In this literature background, the thesis identifies the specific gap addressed by the work: systematic characterization of communication

granularity in multi-argument offloading and runtime-level fusion of eligible OpenMP mappings while preserving directive-level semantics.

Chapter 4 evaluates Argument Fusion as a controlled application-level transformation. It first analyzes the runtime manifestation of transfer inefficiencies, distinguishing device inactivity from active-copy fragmentation. It then evaluates manual fusion, pinned memory, asynchronous transfers, and their combination across synthetic benchmarks and application-level kernels derived from PolyBench. The goal of the chapter is to characterize the performance gain of Argument Fusion across a controlled parameter space, analyze the behavior of the optimizable runtime components, and show that Argument Fusion is complementary to standard CUDA optimizations for host–device communication.

Chapter 5 presents In-Copy Fusion. It motivates runtime-level fusion from the limitations of manual Argument Fusion and identifies the pageable transfer staging phase of CUDA as a potential fusion point and the LLVM OpenMP `targetDataBegin()` path as the relevant integration point. The chapter then describes the ICF implementation, including fusion eligibility, fused device allocation, chunk construction, asynchronous transfer submission, present-table rebinding, fallback behavior, and cleanup. Finally, it evaluates ICF across synthetic and application-level OpenMP workloads.

Chapter 6 concludes the thesis. It summarizes the thesis-level findings, answers the research questions introduced in Chapter 1, discusses the main limitations of the proposed techniques, and outlines future work on backend-aware staging, adaptive profitability control, unified fusion–tiling policies, and broader OpenMP/runtime support.

2. Background

2.1 From Scaling Limits to Heterogeneous Computing

2.1.1 From Frequency Scaling to Specialization

Beginning in the 1970s, the dominant trajectory of computer architecture was shaped by the steady advancement of semiconductor technology. Moore's observation [37] that the number of transistors on an integrated circuit would grow exponentially over time established a long-standing expectation of continuous performance improvement. As described by Hennessy and Patterson [19], sustained improvements in transistor density provided architects with an expanding implementation budget, enabling successive generations of processors to deliver higher performance through increasingly sophisticated microarchitectural techniques.

A key enabler of this progress was the scaling framework introduced by Dennard et al. [13]. Dennard scaling showed that, as transistors were reduced in size, power density could remain approximately constant provided that voltage and current were scaled appropriately. For several decades, this relationship allowed industry to improve performance by increasing clock frequency, transistor count, and microarchitectural complexity.

During this period, performance gains were largely driven by higher clock frequencies and by exploiting instruction-level parallelism within a single core. The expanding transistor budget enabled deeper pipelines, out-of-order execution, speculative execution, and increasingly sophisticated cache hierarchies [19]. This path eventually encountered physical and energy constraints. As voltage scaling slowed, power density increased, leading to the *power wall* [14]. At the same time, further increases in clock frequency became thermally constrained, leading to the *frequency wall* [19].

The breakdown of frequency scaling forced a fundamental shift in architectural design. Rather than relying primarily on higher clock speeds, processors increasingly exploited parallelism through multicore architectures. This shift improved throughput by executing multiple threads concurrently, but it also exposed limits. Amdahl's Law [3] states that the achievable speedup from parallelization is constrained by the fraction of a program that must execute sequentially. In addition, parallel execution introduces costs associated with synchronization, communication, locality, and data movement.

As a result, the field progressively shifted toward domain-specific and heterogeneous architectures. Instead of relying only on general-purpose cores, modern systems combine CPUs with accelerators specialized for particular forms of computation. GPUs, FPGAs, tensor accelerators, and other domain-specific processing units can provide higher throughput or energy efficiency for suitable workloads [19]. This shift improves compute capability, but it also makes system performance increasingly dependent on how effectively data are supplied to specialized

processing units.

2.1.2 The Memory Wall in Heterogeneous Systems

The shift toward heterogeneous computing does not eliminate the memory bottleneck. It often amplifies it. Wulf and McKee originally described the *memory wall* as the growing disparity between processor speed and memory access latency [57]. While this formulation focused primarily on latency, the modern memory wall also includes bandwidth limitations, energy cost, data placement, and the cost of moving data across distinct memory domains.

In heterogeneous systems, specialized accelerators can execute computations at very high throughput. However, this throughput is useful only when the required data can be delivered efficiently. When the host processor and accelerator have physically distinct memories, data must be transferred explicitly or implicitly between them. In discrete GPU systems, this transfer commonly crosses PCIe or NVLink. In embedded heterogeneous platforms, communication may instead occur through platform-specific memory ports, DMA engines, or on-chip interconnects. In both cases, the accelerator is not an isolated compute resource: it is part of a system in which data placement and movement are first-order performance factors.

Contemporary analyses emphasize that data movement can dominate both execution time and energy consumption, particularly in data-intensive workloads [21]. This changes how accelerator performance must be interpreted. A kernel may execute quickly once its operands are resident on the device, but the end-to-end offload may still be limited by allocation, transfer, synchronization, coherence, or memory-management overheads. Therefore, the performance of heterogeneous execution depends not only on the peak throughput of the accelerator, but also on the efficiency of the communication path that feeds it.

This thesis focuses on loosely coupled heterogeneous systems, where host and device memory are physically distinct and communication between them requires explicit or runtime-managed data movement. It does not study tightly integrated accelerator designs or in-memory computing architectures as primary targets [49]. Instead, it addresses systems in which offloaded execution requires data to be allocated, transferred, synchronized, and released across host and device memory domains.

2.2 Programming Models for Heterogeneous Offloading

Heterogeneous systems require programming models that expose accelerator execution while managing the complexity of distinct processing units and memory spaces. These models differ in how much control they give to the programmer, how portable they are across devices, and how much responsibility is delegated to the compiler or runtime. This distinction is important for the thesis because the data-movement problem studied here is shaped not only by hardware, but also by the software layer that turns program objects into concrete memory-management and transfer

operations.

Low-level accelerator APIs such as CUDA expose explicit control over device allocation, host-to-device transfers, device-to-host transfers, streams, events, synchronization, and kernel launches [43, 40]. This control allows programmers to tune communication behavior directly, for example by using pinned host memory, asynchronous copies, and streams [40]. However, it also requires accelerator-specific code and places much of the burden of data orchestration on the programmer.

Other heterogeneous programming models aim to improve portability and abstraction. OpenCL provides a standard API for programming systems composed of CPUs, GPUs, and other accelerators, while HSA-oriented approaches and SYCL-style single-source programming models provide different abstractions for heterogeneous execution and data management [53, 20, 25]. Their details differ, but they share a common systems challenge: data objects described at the program level must eventually be represented in a form that can be allocated, transferred, accessed, synchronized, and released by the underlying runtime and device backend.

Directive-based programming models provide another point in this design space. OpenMP `target` offloading allows programmers to annotate regions of code for accelerator execution, and OpenMP `target data` regions and `map` clauses describe how variables are associated with a device data environment [45]. Compared with low-level APIs, this model raises the abstraction level: the programmer specifies logical mapping intent, while the compiler and runtime are responsible for lowering that intent into concrete offloading operations. In this thesis, *lowering* refers to the translation of source-level directives and mapping clauses into compiler-generated runtime calls, device entry points, and backend allocation or transfer operations. This improves programmability and portability, but it also makes runtime implementation choices central to performance [36, 22].

Table 2.1 summarizes the distinction relevant to this thesis: the degree of control retained by the programmer and the extent to which data-movement decisions are delegated to lower software layers.

OpenMP is therefore not treated in this thesis as the only GPU programming model, nor as a universal replacement for CUDA or other accelerator APIs. It is used as a representative high-level offloading model in which runtime-managed data mapping is explicit enough to analyze, but abstract enough that physical communication decisions are hidden from the programmer. In particular, LLVM's OpenMP offloading runtime, `libomptarget`, provides the runtime layer through which OpenMP target execution is connected to device-specific plugins [30]. In the implementation studied in this thesis, this layer processes mapping lists, allocates device memory, submits transfers through device plugins, records present-table entries, and preserves OpenMP mapping semantics. This makes it a natural target for studying whether the runtime can change the physical granularity of communication while preserving the logical abstraction exposed by the programming model.

The central observation for the rest of the thesis is that programming models expose *logical*

Table 2.1: High-level comparison of heterogeneous programming models from the data-movement perspective considered in this thesis.

Model	Programming style	Data-movement control	Role relevant to this thesis
CUDA	Explicit accelerator API	Programmer explicitly controls allocations, transfers, streams, and synchronization.	Provides direct control over the communication path and the mechanisms evaluated in Chapter 4.
OpenCL	Explicit heterogeneous API	Programmer explicitly manages device objects and communication through a portable API.	Illustrates a portable model that still exposes explicit heterogeneous data management.
SYCL / HSA-oriented models	Higher-level heterogeneous abstractions	More data-management responsibility can be delegated to the programming model and runtime.	Illustrate alternative abstractions for expressing heterogeneous execution and data access.
OpenMP target offloading	Directive-based offloading	The programmer expresses mapping intent, while the compiler and runtime realize concrete allocation and transfer operations.	Provides the runtime-managed mapping layer targeted by ICF in Chapter 5.

data objects, but hardware and runtime systems operate on *physical* communication operations. When a program maps several arrays, fields, or buffers independently, the runtime may materialize them as independent allocations and transfers. This behavior is semantically natural, but it can be inefficient when the aggregate data volume could be communicated more effectively through fewer larger operations.

2.3 Standard Offloading Paradigm in Loosely Coupled Systems

Modern heterogeneous systems commonly contain a host processor and one or more accelerators, each with its own execution resources and memory domain. When these memories are physically distinct, accelerator execution requires an offloading process in which data are made available to the device before computation can proceed. Figure 2.1 illustrates the execution model considered throughout this thesis.

The execution flow consists of several stages. First, input data reside in host memory and are prepared under the control of the CPU. Second, corresponding buffers must be allocated in device memory. This allocation phase introduces latency and resource-management overhead because device memory is managed separately from host memory. Third, input data are transferred from host memory to device memory. The organization of host memory also affects this transfer path. Ordinary application memory is typically pageable: its physical pages are managed

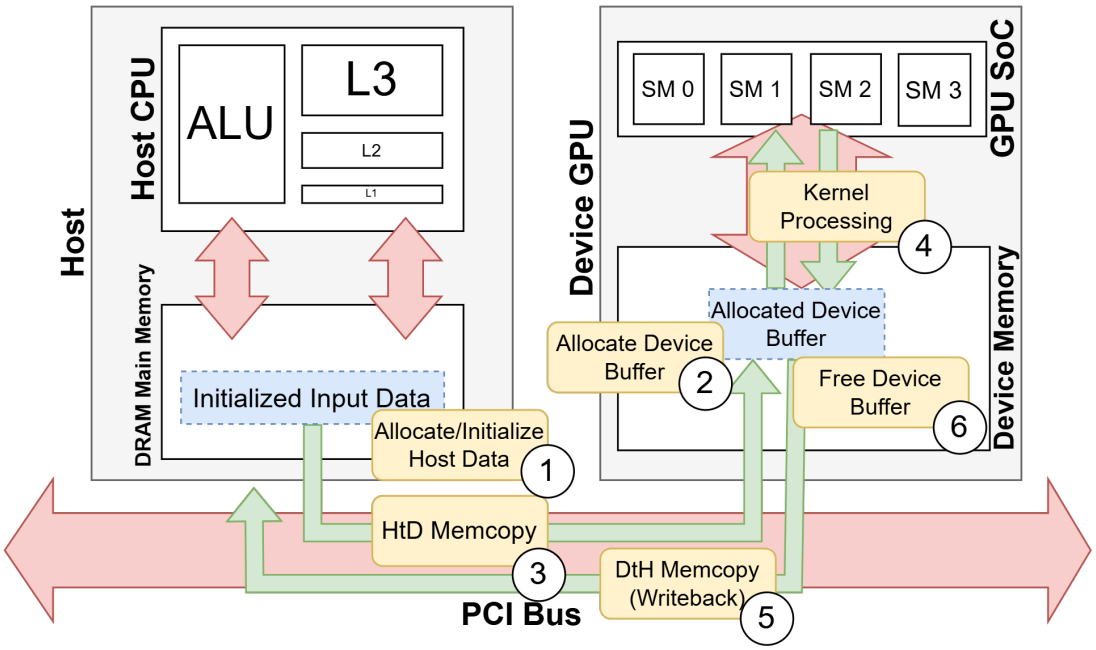


Figure 2.1: Standard offloading execution model in loosely coupled heterogeneous systems.

by the operating system and are not permanently fixed in memory. Device DMA transfers, however, require a stable host-memory region. Accelerator runtimes and drivers may therefore stage data through page-locked, or *pinned*, host memory before initiating the device transfer. Applications can also allocate pinned memory explicitly, avoiding part of this implicit staging path at the cost of more explicit memory management. This distinction becomes important later in the thesis because pageable and pinned host memory expose different communication overheads. In loosely coupled systems, this host-to-device transfer crosses an interconnect such as PCIe or NVLink and therefore incurs bandwidth, latency, and synchronization costs. Prior studies of GPU data movement show that such transfers can dominate execution time, require explicit performance modeling, or significantly affect the interpretation of accelerator speedup, particularly for data-intensive workloads [17, 56, 46].

Once the data are resident on the device, the kernel executes on the accelerator. For compute-intensive applications with high arithmetic intensity, kernel execution may dominate total runtime. For memory-bound or communication-heavy workloads, however, host–device communication may account for a substantial fraction of end-to-end execution time [17, 56, 46, 8].

After kernel execution completes, results may need to be transferred back from device memory to host memory. This device-to-host transfer incurs interconnect cost and synchronization overhead because the host must observe completed device computation before consuming the results. Finally, device memory may be deallocated, introducing additional runtime overhead.

This sequence—device allocation, host-to-device transfer, kernel execution, device-to-host transfer, and device deallocation—constitutes the standard offloading paradigm considered in this thesis. Although conceptually straightforward, it exposes multiple opportunities for inefficiency. Repeated buffer allocation, fragmented transfers across multiple arguments or

mapped variables, synchronization between host and device, and repeated deallocation can all contribute to end-to-end overhead.

This model is not merely an implementation detail of one programming framework. It follows from the architectural separation between host and accelerator memory spaces. High-level abstractions such as OpenMP `target` offloading or CUDA Unified Memory automate parts of this process, but they still rely on mechanisms that allocate, transfer, synchronize, migrate, or maintain coherence between host and device memory domains [45, 43, 40].

This thesis focuses on the data-movement and memory-management phases of this model. In particular, it targets inefficiencies that arise when several logical data objects are materialized as several independent communication operations. The next section explains why this segmentation matters.

2.4 Transfer Segmentation and Communication Granularity

A central implication of the standard offloading model in Figure 2.1 is that host–device communication is not a negligible implementation detail, but a first-order performance concern [17]. NVIDIA's CUDA Best Practices Guide states that the bandwidth available between host memory and device memory is far lower than the bandwidth available within device memory itself, and therefore recommends minimizing host–device transfers even when this requires keeping additional computation on the device [40]. This asymmetry becomes especially important in loosely coupled systems, where communication takes place over an external interconnect rather than through a unified on-chip memory hierarchy.

Within this setting, *transfer segmentation* refers to the decomposition of what could be communicated as one larger transfer into multiple smaller and independent transfers. This pattern arises naturally when multiple arrays, structure fields, or mapped arguments are transferred separately. Although functionally correct, such segmentation is inefficient because each transfer incurs fixed costs in addition to payload-dependent transmission time. A useful high-level model is therefore

$$T(N) = \alpha + \frac{N}{B_{\max}}, \tag{2.1}$$

where N is the transfer size, α is the fixed transfer overhead, and $B_{\max}$ is the asymptotic link bandwidth. Equivalently, the effective bandwidth derived from Equation 2.1 is

$$B_{\text{eff}}(N) = \frac{N}{\alpha + N/B_{\max}}. \tag{2.2}$$

Equations 2.1 and 2.2 are consistent with latency-plus-bandwidth models used in prior work on CPU–GPU data-transfer performance [56]. They are also consistent with empirical CUDA communication studies showing that measured transfer behavior depends on transfer size, memory placement, synchronization methodology, and interconnect topology [46]. Its

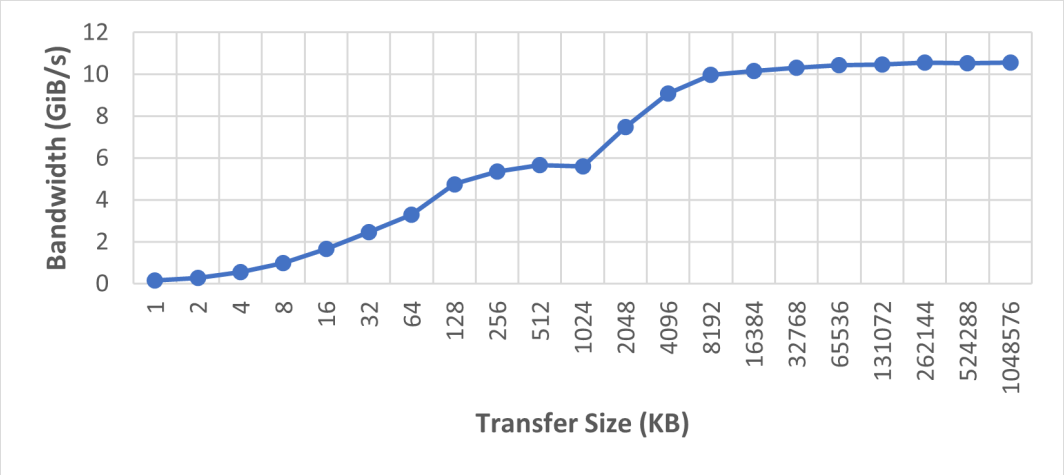


Figure 2.2: Example host–device bandwidth curve as a function of transfer size. Small transfers underutilize the interconnect, while larger transfers approach the sustained bandwidth limit.

consequence is immediate: when N is small, the fixed cost α dominates, and achieved bandwidth is far below the peak of the link.

Figure 2.2 illustrates this behavior using a measured CUDA host-to-device transfer sweep from the experimental context of this thesis. For small transfer sizes, measured bandwidth is low because setup overhead is amortized over too little useful data. As transfer size increases, the fixed overhead is spread over a larger payload, and achieved bandwidth rises toward saturation. The practical consequence is that a program issuing many small transfers repeatedly operates in the low-efficiency region of the curve, whereas aggregating those transfers can move communication closer to the saturation region.

This effect is not merely theoretical. NVIDIA's programming guidance recommends minimizing host–device transfers and batching many small transfers into larger transfers when possible, because each transfer carries overhead beyond the payload movement itself [40]. Published measurements on real GPU workloads also show that communication can occupy a substantial fraction of end-to-end execution time. Belviranli et al., in their study of transfer-aware GPU scheduling, report that data transfers occupy a considerable portion of a CUDA application's lifetime [8]. Similarly, Guo et al. study OpenMP target offloading and motivate transfer optimization from the contribution of data-transfer overhead to total execution cost [18].

For the purposes of this thesis, the key observation is therefore not only that host–device transfers are slower than on-device memory accesses. The key observation is that *segmented* communication amplifies this disadvantage. If the same total payload is divided into many independent operations, each operation can reintroduce startup cost, runtime orchestration, synchronization behavior, and memory-management overhead. This motivates the communication-granularity view used throughout the thesis: performance depends on the physical organization of communication operations, not only on aggregate byte volume.

2.5 Argument Fusion as a Communication-Granularity Optimization

A direct consequence of transfer segmentation is that heterogeneous runtimes may materialize each kernel argument or mapped object as an independent communication unit. For kernels with multiple inputs, this can produce a sequence of per-argument device allocations, host-to-device transfers, and deallocations. Even when the aggregate transferred volume is large, the runtime still pays the overheads associated with several smaller communication and memory-management operations. This observation is consistent with prior transfer-aggregation work, which identifies repeated transfer and allocation overhead as a source of inefficiency in GPU-based heterogeneous systems [27, 24].

In this thesis, *Argument Fusion* denotes the aggregation of the data-movement and memory-management operations associated with multiple logical kernel arguments. Instead of allocating and transferring each argument independently, the host-side offloading code constructs a single contiguous communication buffer that contains the participating arguments. The corresponding device allocation is also created as one contiguous region, and the original logical device pointers are assigned to offsets inside that fused allocation.

From the perspective of the kernel, the computational interface can remain unchanged: the kernel still receives one pointer per logical argument. The transformation instead changes the surrounding host-side communication path. Multiple logical arguments are packed into one physical payload, transferred as a larger host-to-device operation, and later addressed through offset-based device pointers. In this way, Argument Fusion changes communication granularity without modifying the device-side computation.

The purpose of introducing Argument Fusion at this point is methodological. It provides a concrete and controlled transformation for studying the performance consequences of communication granularity. By comparing a baseline multi-argument offload against a fused implementation, the evaluation can isolate how the number of allocations, transfer invocations, synchronization points, and deallocation operations affects execution time. This controlled characterization extends the preliminary Argument Fusion study reported in [51].

Detailed prior work on lazy allocation, transfer fusion, operand transfer fusion, and related aggregation techniques is discussed in Chapter 3. Chapter 4 then uses Argument Fusion as an experimental vehicle for quantifying when transfer aggregation is beneficial, how its benefits decompose across runtime components, and how it interacts with pinned memory and asynchronous transfer mechanisms.

2.6 From Manual Fusion to Runtime-Level Fusion

Manual Argument Fusion exposes the opportunity created by communication granularity, but it is not always a practical programming technique. It requires the programmer to reorganize host-side data layout, compute offsets, modify allocation and transfer code, and ensure that logical arguments are correctly reconstructed as device pointers. These changes can be invasive, especially in applications where data are produced by libraries, where different kernels consume different subsets of data, or where input sizes and mappings are determined dynamically. This limitation motivates shifting the fusion decision from application code toward the offloading runtime, as pursued by ICF in Chapter 5 [52].

A runtime system that receives a complete set of mappings before an offload operation can, in principle, reason about the aggregate communication pattern and alter its physical organization. This is especially relevant for directive-based programming models such as OpenMP, where the programmer expresses logical data mapping and the runtime performs the concrete allocation and transfer operations [45, 36, 30].

However, runtime fusion is more constrained than manual fusion. The runtime must preserve the programming model's semantics, respect mapping directions, maintain correct device pointer associations, avoid unsafe mappings such as pointer-containing structures when they cannot be handled correctly, and fall back to the baseline behavior when fusion is not applicable. In addition, the runtime must avoid introducing a new bottleneck by serializing all host-side packing before device transfer begins.

These constraints explain the transition from the background material in this chapter to the technical contributions of the thesis. Chapter 4 first studies manual Argument Fusion as a controlled way to characterize the communication-granularity opportunity. Chapter 5 then investigates whether part of the same opportunity can be recovered inside the LLVM OpenMP runtime through In-Copy Fusion, while preserving source-level code, kernel interfaces, and OpenMP mapping semantics.

2.7 Chapter Summary

This chapter established the background required by the two main experimental parts of the thesis. In loosely coupled heterogeneous systems, accelerator performance is affected not only by computation but also by the allocation, transfer, synchronization, and deallocation operations required to move data across separate memory domains. Because transfer efficiency depends on transfer size, decomposing the same aggregate payload into several independent operations can reduce communication efficiency.

Argument Fusion provides the application-level mechanism used in Chapter 4 to study this communication-granularity effect under controlled conditions. OpenMP target offloading subsequently provides the runtime-managed setting considered in Chapter 5, where logical

mappings are translated into concrete memory-management and communication operations. These observations provide the bridge from the general background of heterogeneous offloading to the characterization and runtime optimization studied in the remainder of the thesis.

3. Related Work

This chapter positions the thesis with respect to prior work on host–device communication, heterogeneous programming models, transfer aggregation, runtime communication management, memory-management overheads, and OpenMP offloading. The goal is not to survey GPU optimization broadly. Instead, the chapter identifies the research directions that intersect with the thesis: communication bottlenecks in loosely coupled heterogeneous systems [17, 56, 46], software abstractions for accelerator offloading [43, 53, 45], scheduling and pipelining of data movement [7, 4, 8, 28], aggregation of fragmented transfers [27, 24, 39], memory-allocation effects [58, 48], and optimization of directive-based accelerator runtimes [36, 22, 33, 34].

The central distinction used throughout the chapter is the distinction between reducing the *amount* of data movement and changing the *granularity* of data movement. Many prior systems attempt to reduce unnecessary transfers [35, 18, 33], overlap transfers with computation [7, 4, 23], improve data placement or scheduling [8, 38], or hide migration through unified-memory mechanisms [2, 12]. This thesis focuses on a complementary problem: when a set of transfers is still required, their physical organization may be inefficient because the runtime materializes several logical arguments or mappings as separate allocation and transfer operations. Argument Fusion and In-Copy Fusion address this granularity problem directly [51, 52].

3.1 Host–Device Communication Characterization

The performance limitations of host–device communication in accelerator systems have been widely studied. Gregg and Hazelwood emphasize that CPU–GPU performance comparisons cannot be interpreted independently of data location and data movement, since the cost of moving data to and from the accelerator can dominate the apparent benefit of GPU execution [17]. This observation is directly aligned with the motivation of this thesis: accelerator throughput alone is not sufficient when the offloaded computation requires substantial communication across separate memory spaces.

Performance models for CPU–GPU transfers further show that transfer cost depends on both fixed overheads and payload-dependent bandwidth terms. Van Werkhoven et al. model CPU–GPU data transfers and show that accurate performance reasoning must account for transfer direction, size, and system characteristics [56]. Such latency–bandwidth behavior is the basis for the transfer-segmentation argument used in this thesis: many small transfers repeatedly pay fixed costs and may operate below the efficient bandwidth regime of the interconnect.

CUDA communication behavior has also been characterized empirically. Pearson et al. present Comm|Scope, a benchmark suite for evaluating CUDA communication primitives on high-bandwidth interconnects [46]. Their work highlights that communication performance depends on factors such as transfer size, synchronization methodology, memory placement, and

topology. This supports the thesis claim that host–device communication should not be treated as a simple function of total transferred bytes alone. The organization of those bytes into individual runtime operations is also performance-relevant.

Other work has examined offload latency, command submission, and synchronization behavior. Lustig and Martonosi study mechanisms for reducing GPU offload latency through fine-grained CPU–GPU synchronization [31]. Tanasic et al. analyze the GPU command execution model in the context of preemptive multiprogramming [54]. Bakita and Anderson further investigate NVIDIA GPU internals relevant to reliable GPU management, including behavior that affects predictable accelerator use [6]. Although these works do not target transfer fusion, they reinforce the broader point that accelerator execution overhead includes runtime, command, synchronization, and device-management effects surrounding the kernel itself.

These characterization studies establish the performance basis for this thesis. They show that data movement is a first-order concern and that transfer size, synchronization, runtime behavior, and device-management mechanisms affect observed performance [17, 56, 46, 31, 54, 6]. The contribution of this thesis begins from that premise and narrows the focus to one specific source of inefficiency: fragmented multi-argument or multi-mapping communication.

3.2 Programming Models and Offloading Abstractions

Heterogeneous programming models differ in the amount of control they expose to the programmer and in the amount of responsibility assigned to the compiler or runtime [43, 53, 25, 20, 45]. This distinction is relevant to the thesis because communication granularity is not determined by hardware alone. It is also shaped by the programming interface through which data movement is expressed and by the runtime implementation that materializes logical data objects as physical allocations and transfers.

CUDA exposes low-level control over GPU execution and memory management, including explicit device allocation, host-to-device and device-to-host transfers, streams, events, synchronization, and unified-memory mechanisms [43, 40]. This level of control enables manual optimizations such as pinned memory, asynchronous transfer submission, stream-based overlap, and application-level argument packing. However, it also ties the program to a vendor-specific programming model and requires the programmer to manage accelerator data movement explicitly.

OpenCL was introduced as a standard for programming heterogeneous systems composed of CPUs, GPUs, and other accelerators [53]. It provides a more portable abstraction than CUDA, but still exposes explicit concepts of devices, kernels, command queues, buffers, and data transfers. Other heterogeneous programming ecosystems, including SYCL and HSA-oriented approaches, similarly address portability and programmability across heterogeneous devices, but differ in their memory models, execution abstractions, and runtime responsibilities [25, 20].

Directive-based programming models occupy a different point in this design space. OpenMP `target` offloading allows programmers to annotate code regions for accelerator execution, while

`target data` regions and `map` clauses describe data movement and device data-environment association [45]. Instead of manually writing allocation and transfer calls, the programmer expresses mapping intent, while the compiler and runtime lower that intent into device allocation, host-to-device copy-in, device-to-host copy-out, synchronization, and mapping-table management. This abstraction improves programmability, but it also makes runtime implementation choices central to performance [36, 22].

The relevance of OpenMP to this thesis follows from this abstraction boundary. OpenMP is not treated as the GPU programming standard, nor as a replacement for lower-level APIs. It is treated as a representative high-level offloading model in which the runtime has visibility over a complete mapping list and is responsible for materializing that list into concrete communication operations [45, 30]. This makes OpenMP a natural context for studying whether a runtime can change the physical granularity of communication while preserving source-level directives, kernel interfaces, and mapping semantics [52].

Backend portability is also relevant. LLVM's OpenMP offloading infrastructure separates the language-level runtime from device-specific plugins and backend mechanisms [30, 29]. Related backend ecosystems such as AMD ROCm and oneAPI Level Zero expose their own device-management and memory-transfer interfaces [1, 55]. This thesis implements ICF in the LLVM OpenMP CUDA path, but the broader research question is not CUDA-specific: if a runtime observes several logical mappings before issuing device transfers, then communication-granularity optimization may be relevant to other backend implementations as well.

3.3 OpenMP Offloading and Data-Mapping Optimization

OpenMP `target` offloading provides a directive-based programming model for heterogeneous execution. It exposes data movement through mapping clauses while delegating concrete allocation, transfer, synchronization, and mapping-table management to the implementation [45, 36]. In LLVM, this execution path is realized through the OpenMP offloading runtime and device plugins [30, 22]. This abstraction makes OpenMP a natural target for data-movement optimization, but also introduces challenges because the programmer, compiler, and runtime may have different visibility into data lifetimes, mapping structure, and backend behavior.

Monsalve Díaz et al. analyze OpenMP 4.5 offloading implementations with respect to correctness and overhead [36]. Their work shows that OpenMP offloading performance depends not only on the device kernel, but also on the runtime mechanisms used to implement the target model. Huber et al. study efficient execution of OpenMP on GPUs through compiler and runtime improvements in LLVM [22]. These works provide important context for ICF: OpenMP offloading performance is shaped by the implementation of the runtime and device plugins, and optimization opportunities exist below the source-level directive interface.

Several works focus specifically on OpenMP data movement and mapping behavior. Mishra et al. present a data-transfer and reuse-analysis tool for GPU offloading using OpenMP, with the

goal of identifying data that need not be transferred between host and device [35]. Marzen et al. propose OMPDart, which statically generates efficient OpenMP offload data mappings by analyzing host–device data dependencies and transforming source-level mapping constructs [33]. Guo et al. also study OpenMP offloading data-transfer optimization, focusing on avoiding unnecessary data transfers in OpenMP offload programs [18]. More recently, OMPDataPerf introduced dynamic analysis methods for detecting inefficient OpenMP data-transfer and memory-allocation patterns and reporting their optimization potential [34]. Together, these works show that OpenMP data movement is a recognized optimization target at both static-analysis and dynamic-analysis levels.

These OpenMP data-movement analyses and optimizations are closely related to ICF, but they operate at a different level. They primarily aim to reduce unnecessary transfers, improve mapping placement, restructure data regions, or expose data-reuse opportunities so that data can remain valid on the device across target regions [35, 18, 33, 34]. ICF assumes that a mapping event still occurs and that a set of input mappings must be copied to the device. It then optimizes the physical organization of those required transfers by aggregating eligible mappings inside the runtime [52].

Thus, OpenMP data-lifetime optimization and ICF are orthogonal. Static or dynamic mapping analyses reduce how often communication is needed; ICF reduces the cost of eligible communication that remains necessary. This distinction is important for the scope of the thesis. ICF does not replace static OpenMP mapping optimization, dependence analysis, or programmer-managed target data regions. Instead, it complements them by improving the copy-in path at the moment the runtime observes the concrete mapping set. This is particularly relevant for dynamically supplied data, irregular mapping sets, or applications where a static fused layout is difficult to construct or maintain.

3.4 Transfer Scheduling, Pipelining, and Runtime Overlap

A substantial body of work attempts to reduce the visible cost of communication by overlapping data movement with computation, pipelining communication and computation phases, or scheduling transfer operations more effectively [7, 4, 5, 8, 23, 28]. These approaches are related to this thesis because they address the same host–device communication bottleneck, but they generally operate along a different optimization dimension.

Bastem et al. study overlapping data transfers with GPU computation using tiled execution [7]. Their work partitions computation and communication so that data movement for one tile can overlap with computation on another. CoCoPeLia models communication–computation overlap for GPU-accelerated linear algebra and predicts efficient pipelined execution configurations [4]. PARALiA extends this direction through a performance-aware runtime for auto-tuning linear algebra on heterogeneous systems [5]. These works are relevant because they show that effective heterogeneous execution often depends on balancing pipeline stages rather than simply minimizing

one isolated component.

Recent work by Li et al. studies the performance implications of pipelining data transfers in CPU–GPU heterogeneous systems [28]. This is particularly relevant to the "splitting" side of the present thesis. In-Copy Fusion does not merely aggregate all mapped inputs into one monolithic transfer. Instead, it splits the fused payload into calibrated, same-sized staged chunks so that host-side packing and device-side transfer can proceed as a pipeline [52]. The motivation is similar to communication pipelining work, but the pipeline stages are different: ICF balances host-side packing into pinned staging buffers against host-to-device transfer of already packed chunks.

Runtime systems have also been proposed to schedule or reorganize communication more effectively. CuMAS schedules data transfers and kernels from multiple CUDA applications on shared GPUs to improve utilization and reduce interference [8]. Hidden Unified Memory overlaps host-to-device copies with computation by exploiting unified-memory behavior and transparent runtime mechanisms [23]. PROACT targets inefficient fine-grained communication in multi-GPU systems by converting remote fine-grained accesses into more interconnect-friendly transfers while preserving a peer-to-peer programming model [38]. These systems are related because they improve communication behavior without requiring extensive application rewriting. However, they do not primarily target the physical granularity of OpenMP map entries within one offload preparation phase. CuMAS schedules communication and computation tasks across applications, Hidden Unified Memory hides copies through unified-memory mechanisms, and PROACT targets inter-GPU fine-grained communication. ICF instead aggregates several eligible OpenMP input mappings belonging to the same offload operation before the backend materializes them as independent transfers.

The work in this area is therefore complementary to the thesis. Scheduling, overlap, and pipelining can reduce the visible cost of communication [7, 4, 8, 23, 28], while Argument Fusion and ICF reduce fragmentation in the communication operations themselves [51, 52]. ICF additionally borrows the general principle of pipelining, but applies it internally to a runtime-managed staged copy-in path.

3.5 Transfer Aggregation, Operand Fusion, and Static Layout Optimization

The closest prior work to the communication-granularity component of this thesis is transfer aggregation, lazy allocation, operand fusion, and fine-grained communication packing [27, 24, 39]. These approaches recognize that independently transferring several operands can introduce repeated startup, allocation, protocol, and runtime overheads. They therefore attempt to merge multiple logical data objects or fine-grained communication events into fewer physical communication operations.

Li and Kessler propose lazy allocation and transfer fusion for GPU-based heterogeneous systems [27]. Their work reduces allocation and transfer overhead by deferring and merging memory-management and data-transfer operations. This is directly related to the Argument Fusion transformation evaluated in this thesis. The key difference is the role played by Chapter 4: Argument Fusion is not claimed as a new transformation, but is used as a controlled experimental mechanism for characterizing the performance effects of transfer granularity, allocator multiplicity, pinned memory, asynchronous transfer submission, and application-level argument structure [51].

Kessler later formulates operand transfer fusion as a global optimization problem over a kernel–operand dependence graph [24]. The goal is to reorder operand arrays in memory so that transfers and allocations of adjacent operands can be merged, reducing accumulated startup overheads. This approach is more global and static than the manual fusion experiments in Chapter 4. It assumes that enough information is available to construct or optimize an operand layout before execution. By contrast, ICF performs fusion at runtime inside the OpenMP offloading infrastructure, using the concrete mapping set observed for a particular offload event.

This distinction supports the static/dynamic framing of the thesis. Static operand-layout optimization attempts to arrange operands globally so that future transfers can be coalesced [24]. Manual Argument Fusion provides an application-level, explicitly programmed version of the same communication-granularity principle [51]. ICF moves the principle into the runtime, where fusion decisions are made dynamically for the mapping set observed at a concrete offload boundary [52]. Chapter 5 revisits this distinction experimentally through a Kessler-style interleaved benchmark, comparing a static layout baseline against runtime fusion inside OpenMP.

FinePack addresses fine-grained communication aggregation in a different setting. Muthukrishnan et al. propose mechanisms for transparently improving the efficiency of fine-grained transfers in multi-GPU systems by packing fine-grained communication into more efficient interconnect messages [39]. Although FinePack targets a different layer of the system, its motivation is conceptually aligned with this thesis: fine-grained communication can incur disproportionate protocol and transfer overheads, and aggregation can improve effective communication efficiency. The distinction is that FinePack focuses on multi-GPU fine-grained transfers, whereas this thesis targets host–device allocation and transfer fragmentation in CUDA and OpenMP offloading.

This thesis builds on the same general principle as transfer aggregation work but contributes in two specific ways. First, it provides a systematic evaluation of manual Argument Fusion across transfer sizes, argument counts, runtime components, and combinations with pinned and asynchronous communication [51]. Second, it introduces ICF as a runtime-level mechanism that applies fusion inside LLVM's OpenMP offloading runtime without requiring application-visible data-layout restructuring [52]. The novelty is therefore not the abstract idea that larger transfers can be better than smaller transfers; prior work has already established transfer aggregation and operand fusion as useful optimization directions [27, 24, 39]. The contribution is the characterization of this effect across runtime components and the demonstration that part of the benefit can be recovered transparently inside the OpenMP runtime.

3.6 Runtime Memory Management and Allocation Effects

Host–device communication cost is not limited to the time spent in payload transfer. Device allocation, deallocation, synchronization, and runtime memory-management behavior can also contribute substantially to offload overhead [58, 42, 48]. This is particularly relevant for multi-argument offloading, where a baseline implementation may allocate and release one device buffer per argument.

CUDA memory-management calls have been shown to introduce non-trivial timing and synchronization behavior. Yang et al. discuss pitfalls that arise when using NVIDIA GPUs for real-time tasks and identify memory-management and synchronization effects as important sources of unpredictability [58]. NVIDIA's CUDA runtime documentation also distinguishes operations by their synchronization behavior and documents cases where runtime calls may block or synchronize depending on the operation and memory involved [42]. NVIDIA's stream-ordered allocator provides an alternative allocation/deallocation mechanism intended to order memory-management operations with respect to CUDA streams [41].

Recent allocator-characterization work further supports the view that GPU memory-management cost can be non-uniform. Rodriguez et al. analyze GPU memory allocation characteristics and report behavior relevant to understanding allocation overheads and size-dependent effects [48]. More general work on custom and dynamic memory allocation has long shown that allocator behavior can materially affect performance, locality, and runtime overhead [9, 15, 32]. Although these latter works are not specific to GPU offloading, they reinforce the broader point that allocation policy and allocation multiplicity can be performance-relevant system concerns.

This thesis uses these observations to interpret the allocation and deallocation components of Argument Fusion. In the baseline multi-argument case, each argument may induce its own device allocation and device deallocation. In the fused case, several logical arguments can be represented by one physical device allocation. Therefore, the benefit of fusion may arise not only from larger host-to-device transfers, but also from reduced allocator and deallocator multiplicity. Chapter 4 evaluates this directly by separating allocation, host-to-device transfer, and deallocation timing [51].

3.7 Unified, Shared, and Managed Memory

Unified, shared, and managed-memory systems address host–device data movement at a different abstraction level. Instead of requiring every transfer to be expressed explicitly, these systems provide shared address-space abstractions, implicit migration, page-based movement, or runtime-managed movement between host and device memory domains [43, 2, 12, 23].

Allen and Ge analyze the behavior of Unified Virtual Memory for GPU-accelerated computing [2]. Cooper et al. examine the design and performance implications of shared virtual memory across diverse applications [12]. Hidden Unified Memory similarly exploits unified-memory

mechanisms to overlap host-to-device copies with computation [23]. These approaches can reduce programmer burden and may hide or automate parts of data movement.

The distinction from ICF is the level at which optimization occurs. Unified and shared virtual memory mechanisms modify or hide migration between memory domains, often through page-based movement, runtime migration policies, or memory-system support [2, 12, 23]. ICF operates on explicit OpenMP mapping events. It assumes that a copy-in operation has been requested by the program and then modifies the physical organization of that required copy-in while preserving OpenMP mapping semantics [45, 52]. It does not replace OpenMP mapping semantics with page-fault-driven migration, nor does it attempt to infer data lifetime across arbitrary program regions.

The two directions are therefore complementary. Managed-memory systems can reduce the need for explicit transfers or hide migration cost in some cases. Communication-granularity optimization reduces the cost of explicit transfers that still occur, especially when the runtime receives several independent mappings at the same offload boundary.

3.8 Positioning of This Thesis

The categories in the table correspond to the literature groups discussed above: communication characterization [17, 56, 46], programming-model abstractions [43, 53, 45], OpenMP data-mapping optimization [35, 18, 33, 34], transfer scheduling and overlap [7, 4, 8, 23, 28], transfer aggregation and operand fusion [27, 24, 39], runtime memory management [58, 42, 48], and unified or shared memory [2, 12].

Table 3.1 summarizes how the main related-work directions motivate and delimit the contributions of this thesis. Communication characterization, transfer aggregation, and runtime memory-management work provide the basis for the systematic study of communication granularity through Argument Fusion; OpenMP data-mapping work provides the context for moving fusion into the runtime; and transfer scheduling and pipelining provide the context for organizing runtime-managed fused communication efficiently.

Table 3.1: Positioning of this thesis relative to related work.

Research direction	**Primary objective**	**Relation to this thesis**
Host–device communication characterization	Measure transfer costs, bandwidth behavior, synchronization effects, topology dependence, and offload overhead.	Provides the performance basis for studying transfer segmentation and communication granularity.
Programming models and offloading abstractions	Expose accelerator execution through low-level APIs, portable heterogeneous models, or directive-based abstractions.	Provides the abstraction context. The thesis targets OpenMP not as the only GPU model, but as a high-level runtime-managed mapping model.
OpenMP data-mapping optimization	Reduce unnecessary mappings, improve data reuse, generate better target data regions, or detect inefficient mapping patterns.	Orthogonal. These approaches reduce when communication is needed; ICF reduces the cost of eligible communication that still occurs.
Transfer scheduling and overlap	Hide communication cost through scheduling, pipelining, concurrent execution, or overlap with computation.	Complementary. ICF also uses pipelining internally, but its main objective is to change transfer granularity.
Transfer aggregation and operand fusion	Merge multiple logical operands or transfers into fewer physical communication operations.	Closest prior work. This thesis characterizes manual Argument Fusion experimentally and implements a dynamic runtime-level OpenMP version through ICF.
Runtime memory management and allocation effects	Characterize or reduce allocation, deallocation, synchronization, and allocator-induced overheads.	Supports the component-level analysis of Argument Fusion, where reduced allocation and deallocation multiplicity contribute to speedup.
Unified/shared/managed memory	Automate, hide, or migrate data between host and device memory spaces.	Related as transparent data movement, but ICF operates on explicit OpenMP map entries rather than replacing them with page-fault-driven migration.

The position of the thesis can therefore be stated precisely. Prior work already establishes that host–device transfers are expensive [17, 56, 46], that overlap and scheduling can reduce visible communication cost [7, 4, 8, 23], that transfer aggregation and operand fusion can

reduce startup and communication overhead [27, 24, 39], and that OpenMP data mapping can be analyzed or optimized [35, 18, 33, 34]. What remains less explored is the runtime granularity of already-required OpenMP map-entry transfers: that is, the case where communication cannot simply be removed, hidden, or statically rearranged, but its physical organization can still be changed at the moment the runtime observes the concrete mapping set.

The contribution of this thesis is therefore to examine communication granularity as a measurable runtime effect in multi-argument offloading, to quantify how manual Argument Fusion interacts with allocation, transfer, deallocation, pinned memory, and asynchronous execution [51], and to demonstrate that a runtime implementation inside LLVM OpenMP can transparently aggregate eligible input mappings through a staged fused transfer path [52].

In this sense, the thesis connects two previously separate perspectives. From transfer-fusion and operand-layout work it inherits the idea that several logical objects can sometimes be communicated more efficiently as one physical payload [27, 24]. From OpenMP data-mapping work it inherits the concern for preserving directive-level semantics and runtime mapping behavior [45, 36, 22]. In-Copy Fusion combines these perspectives by preserving OpenMP's logical mapping abstraction while modifying the physical communication path used to materialize eligible host-to-device input mappings [52].

4. Evaluation of Argument Fusion

4.1 Chapter Overview

Chapter 2 established that, in loosely coupled heterogeneous systems, host–device communication is governed by a latency–bandwidth trade-off, and that fragmented transfers operate in a low-efficiency regime of the interconnect. It also introduced Argument Fusion as a transformation that alters the granularity of communication by aggregating multiple kernel arguments into a single transfer.

This chapter evaluates how Argument Fusion changes communication granularity and how this affects offloaded execution, extending the preliminary Argument Fusion study reported in [51].

The analysis addresses RQ1 and its operational objectives. Section 4.2 examines how transfer segmentation appears at runtime as both device inactivity and reduced active-copy efficiency. The synthetic benchmark then identifies the transfer-size and argument-count regimes in which Argument Fusion improves performance and separates the effects of transfer aggregation from allocation and deallocation consolidation. Finally, the comparison with pinned memory, asynchronous transfers, and Fusion+ evaluates how Argument Fusion interacts with established CUDA communication optimizations.

To answer these questions, the evaluation is structured along two complementary axes. The first is a parametric synthetic benchmark that enables independent control over the number of kernel arguments and total data size. This allows isolation of communication effects from computation. The second is an application-level evaluation based on representative kernels from the PolyBench benchmark suite [47], which serves to validate whether the trends observed in controlled experiments persist under realistic workloads.

The chapter proceeds as follows. Section 4.2 examines the runtime-level manifestation of transfer inefficiencies and relates them to measurable system behavior. Section 4.3 describes the methodology for applying Argument Fusion and defines the evaluated optimization variants. Section 4.4 details the experimental setup and measurement methodology. Sections 4.5 and 4.6 present the synthetic and application-level evaluations, respectively. Finally, Section 4.7 discusses the implications of the results and outlines the limitations and applicability of Argument Fusion.

4.2 Motivation and Communication Effects

The practical manifestation of the transfer segmentation inefficiency is best understood by examining the runtime behavior of offloaded executions. Prior characterization studies show that host–device transfer performance depends on transfer size, synchronization behavior,

memory placement, and runtime effects rather than only on aggregate byte volume [17, 56, 46]. In this section, we refine the discussion from a systems-model perspective to a runtime perspective, identifying the concrete mechanisms through which segmented communication degrades performance.

4.2.1 Execution Actors and Communication Pipeline

Host–device communication in discrete heterogeneous systems involves the interaction of multiple hardware and software components operating across distinct memory spaces. In CUDA-based systems, communication is orchestrated through the combined activity of the host processor, the CUDA runtime and driver stack, and device-side copy engines responsible for executing memory transfers [40, 10, 6].

From the application perspective, communication operations are initiated through runtime API calls such as `cudaMalloc`, `cudaMemcpy`, `cudaMemcpyAsync`, and `cudaFree`. These calls are issued by the host CPU and translated by the CUDA runtime and driver stack into work submitted to the accelerator [43, 10].

The CPU therefore acts as the control-side orchestrator of communication. It invokes the relevant runtime API calls, passes the source and destination addresses and transfer sizes to the runtime, and enforces the synchronization and ordering constraints required by the program. The device copy engines perform the actual data movement, but they do so only after the corresponding operations have been issued and ordered by the host-side runtime.

The actual movement of data between host and device memory is performed by the device DMA engines, commonly exposed in NVIDIA profiling tools as copy engines [6]. Once transfer commands have been issued and all required transfer conditions have been satisfied, the DMA engines execute the communication independently of the host CPU.

Importantly, application-level transfers become visible on the device only after the corresponding runtime calls have been issued and ordered by the host-side software stack [43, 10]. Consequently, host-side runtime behavior directly influences the utilization of the copy engines and the effective communication throughput of the system.

Figure 4.1 conceptually illustrates the communication pipeline considered throughout this chapter.

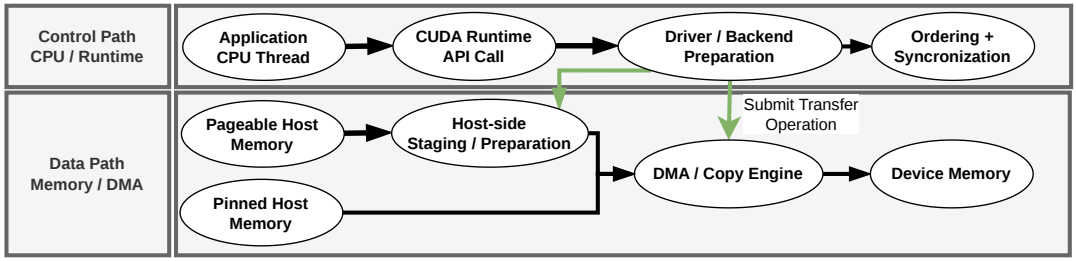


Figure 4.1: Conceptual communication pipeline for host–device transfers in CUDA-based systems.

An important distinction exists between pageable and pinned host memory. Pageable memory is managed by the operating system through virtual memory mechanisms and is not guaranteed to remain resident at fixed physical addresses. Consequently, it cannot always be accessed directly by the DMA engines [40].

Pinned memory, by contrast, is page-locked and remains resident in physical memory for the duration of its allocation. Because of this property, pinned buffers can be directly accessed by the device DMA engines without requiring the pageable-memory staging path used for ordinary pageable host allocations [40].

When pageable memory transfers are issued, the CUDA runtime internally performs additional staging operations before DMA execution can begin [43]. As a result, portions of the communication pipeline involve both host-side memory-copy activity and device-side DMA activity before the transfer fully completes.

This execution model has several important implications for communication behavior.

First, communication overhead is not limited to the raw DMA transfer itself. Runtime submission, synchronization behavior, allocation management, and transfer staging may all contribute to total execution time [43, 40, 42].

Second, the CPU and DMA engines may operate partially asynchronously. Consequently, host-side API activity and device-side DMA activity do not necessarily appear as strictly serialized phases in execution traces.

Third, communication performance depends not only on total transferred bytes, but also on how communication is organized and scheduled by the runtime. Fragmented communication patterns may repeatedly incur runtime overheads and synchronization delays even when the aggregate transferred data volume remains unchanged [56, 46, 27, 24].

These observations form the basis for the transfer inefficiencies analyzed in the following sections.

4.2.2 Runtime Manifestation of Transfer Inefficiencies

In typical offloading runtimes, each kernel argument or mapped object is handled as an independent communication unit, resulting in a sequence of allocation, transfer, and synchronization operations [51, 24, 27, 45]. This behavior induces a fragmented execution pattern in which communication is decomposed into multiple short-lived transactions. The resulting host–device interaction can be visualized as a timeline of alternating activity and inactivity phases, where both host and device incur overheads between successive transfers.

Two distinct inefficiencies emerge from this runtime behavior. The first is a host-level inefficiency: the device copy engine remains idle between successive transfers even though additional data remain to be communicated. These idle intervals arise from host-side transfer preparation, pageable-memory handling, API and driver overheads, DMA command setup, and synchronization requirements between transfer operations. The second is a device-level

active-transfer inefficiency: even when the idle gaps between transfers are excluded, several small device-side copy operations require more cumulative active transfer time than one larger copy operation carrying the same total payload. This second effect is subtler because it occurs while the copy engine is active, and therefore concerns the efficiency of the transfer operations themselves rather than the gaps between them.

Device Inactivity and Inter-Transfer Overheads

The first inefficiency is the presence of idle periods between consecutive device-side copy intervals. Each transfer invocation requires host-side work before the corresponding copy operation becomes visible on the device timeline. This work may include API invocation, runtime and driver preparation, DMA command setup, pageable-memory handling, and synchronization required by blocking transfer semantics. During these intervals, the device copy engine is not transferring payload data, even though additional transfers remain pending.

This effect can be observed by comparing the elapsed duration of a sequence of transfers with the cumulative duration of the visible device-side copy intervals inside that sequence. Pinned memory can reduce part of this idle component by avoiding the pageable-memory handling path, but it does not by itself remove the per-transfer structure or the synchronization semantics of blocking transfer calls [40]. Similarly, asynchronous transfer mechanisms can reduce this effect by allowing the host to enqueue successive transfers more compactly, thereby overlapping part of the control-side preparation with ongoing device-side copy activity.

This inefficiency is therefore primarily a scheduling and orchestration effect.

Transfer Fragmentation and Active-Copy Efficiency

The second inefficiency arises from the fragmentation of a fixed communication volume into multiple smaller device-side copy operations. This effect is distinct from the idle gaps discussed above. Even if the idle intervals between transfers are excluded, the cumulative active copy time of many small transfers can be larger than the active copy time of one aggregated transfer carrying the same total payload.

This behavior reflects the fact that active transfer time is not determined only by the number of payload bytes. Each DMA operation also incurs transfer-level overheads associated with command setup, transaction formation, protocol handling, and finite payload granularity. These costs are difficult to isolate individually from user-level measurements, and the experiments in this chapter do not claim to identify their exact microarchitectural origin. What can be measured directly is their aggregate effect: segmented transfers achieve lower effective bandwidth than an aggregated transfer of the same total size, even when only device-side active copy intervals are considered.

Consequently, transfer fragmentation affects the duration of the active copy intervals themselves.

4.2.3 Measurement Methodology for Runtime Analysis

To quantify the runtime effects described in the previous sections, we rely on profiling-based measurements of host–device communication. The qualitative distinction introduced above is formalized by separating the elapsed wall-clock interval of a transfer sequence from the cumulative time during which device-side copy activity is visible in the profiler timeline. This allows us to distinguish idle intervals between transfers from inefficiency inside the active transfer intervals themselves.

We define the following metrics:

- **Wall-clock transfer time (T_{wall}):** the elapsed time between the initiation of the first host-to-device transfer and the completion of the last transfer in a given offloading sequence.
- **Active DMA time (T_{DMA}):** the cumulative time during which the device DMA engine is actively transferring data, as reported by profiling tools.
- **Idle time (T_{idle}):** the difference between wall-clock transfer time and active DMA time.

$$T_{\text{idle}} = T_{\text{wall}} - T_{\text{DMA}} \tag{4.1}$$

With this notation, the first inefficiency introduced above corresponds to an increase in T_{idle}: the wall-clock transfer interval grows because visible device-side copy operations are separated by idle gaps. The second inefficiency corresponds to an increase in T_{DMA}: even after removing idle gaps, a segmented sequence of small transfers can spend more cumulative time in active copy operations than a single aggregated transfer of the same total payload.

These metrics allow us to distinguish between inefficiencies originating from device inactivity (idle time) and inefficiencies intrinsic to the transfer process itself (active DMA time).

The system used for the analysis of these effects is a machine featuring an NVIDIA Quadro P400 GPU with CUDA 12, specified in Section 4.4.1. Measurements are obtained via NVIDIA Nsight Systems, which enables direct observation of DMA activity intervals and synchronization gaps between transfers [44].

The following subsections use these metrics to isolate and quantify the individual effects contributing to transfer inefficiency.

4.2.4 Quantifying Device Inactivity

While Section 4.2.2 introduced device inactivity as a qualitative property of segmented transfer execution, this subsection isolates and quantifies this effect.

To characterize the inefficiency derived from device inactivity, we use an experiment consisting of 16 independent host–to–device transfers of 64 KiB each. This transfer size is selected in the

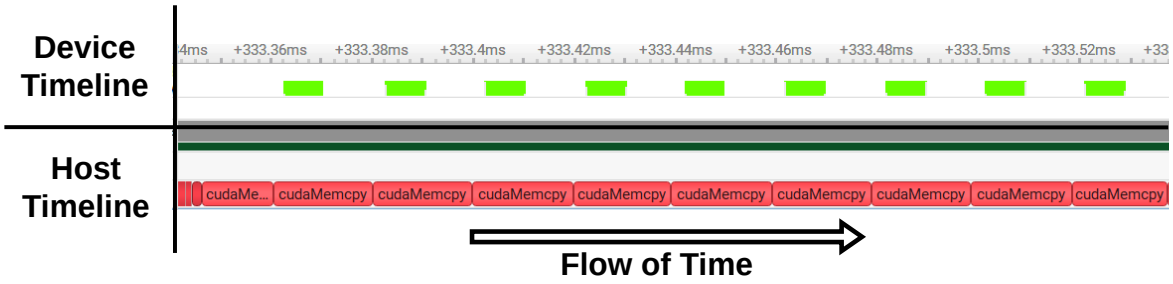


Figure 4.2: Nsight Systems trace of consecutive host–to–device transfers showing gaps in DMA activity.

sub-megabyte regime so that per-transfer overheads are large relative to the payload transmission time, making idle intervals visually and quantitatively apparent.

Figure 4.2 shows the corresponding Nsight Systems trace for the host–to–device transfer phase. The device timeline contains a sequence of discrete DMA activity intervals separated by visible gaps. These gaps correspond to periods in which no data movement is performed by the device copy engine, and therefore contribute to the idle component T_{idle} defined in Section 4.2.3.

We further quantify this effect by sweeping the total transferred data size while maintaining a fixed number of transfers. For each point, we measure the wall-clock transfer time T_{wall} and the cumulative active DMA time T_{DMA}. Their difference gives the cumulative idle time T_{idle}.

Figure 4.3a[1] reports the relative contribution of DMA activity and inactivity to the total host–to–device transfer interval. The results show that, for small and medium transfer sizes, a substantial fraction of the total transfer interval is spent outside active DMA execution. This confirms that segmented transfers are penalized not only by the duration of the individual data movements, but also by the idle intervals introduced between them.

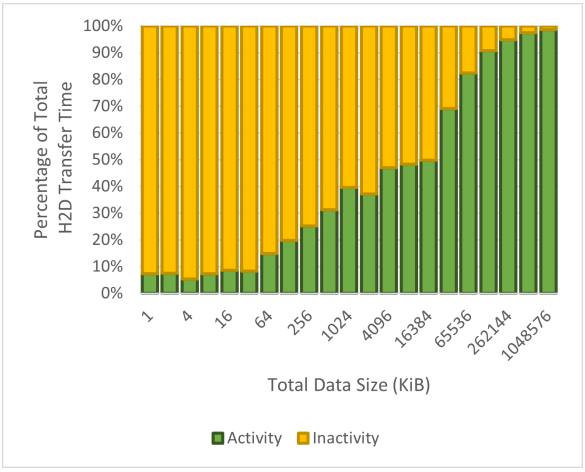
4.2.5 Quantifying Fragmentation of Host-to-Device Transfers

The previous subsection quantified idle time between DMA operations. We now isolate the second effect: the loss of transfer efficiency of the device's copy engine caused by decomposing a fixed communication volume into multiple smaller transfers.

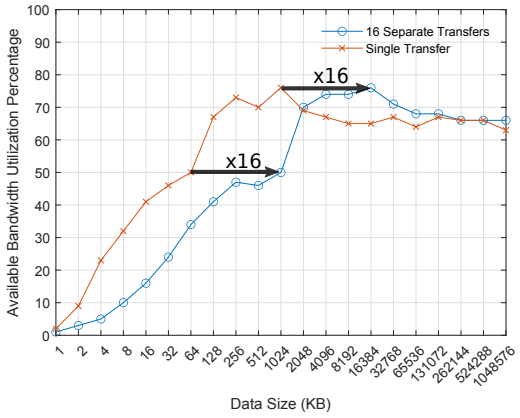
To isolate this effect, we compare two transfer patterns with identical total data volume. The first transfers the data as 16 independent host–to–device copies. The second transfers the same amount of data as a single aggregated copy. The comparison is performed using the active DMA time T_{DMA}, excluding idle intervals between transfer operations. This isolates inefficiency inside the transfer process itself, rather than inefficiency caused by gaps between transfers.

Figure 4.3b reports the effective bandwidth utilization of the two configurations calculated using only active DMA time T_{DMA}. The segmented case reaches the high-utilization region of

[1]The trace reveals that host-side API activity and device-side DMA activity are not perfectly aligned in time, and that multiple `cudaMemcpy` calls may appear on the host timeline before the corresponding DMA operations are executed on the device. This apparent discrepancy reflects additional complexity in the runtime and execution model. A more detailed discussion of this behavior is provided in Section 4.3.



(a) Relative contribution of active DMA time and idle time for 16 separate host–to–device transfers.



(b) Bandwidth utilization of DMA activity for segmented vs. aggregated transfers.

Figure 4.3: Transfer inefficiencies: (a) device inactivity and (b) transfer fragmentation.

the interconnect at approximately 16 times larger total data sizes than the aggregated case. This indicates that fragmentation increases the cumulative DMA execution time even when the total transferred payload is unchanged.

The key distinction between this effect and device inactivity is that fragmentation affects T_{DMA} directly [56]. Device inactivity increases the difference between T_{wall} and T_{DMA}, whereas fragmentation increases T_{DMA} itself. These two effects form the basis for evaluating communication optimizations in the following subsection.

4.2.6 Unified Interpretation and Optimization Implications

The decomposition of transfer inefficiency into device inactivity (T_{idle}) and transfer fragmentation (T_{DMA}) provides a structured framework for interpreting the effect of common communication optimizations. Importantly, these optimizations do not act on a single isolated mechanism, but may influence multiple components of the execution pipeline to varying degrees.

Pinned memory avoids the pageable-memory staging path for the transferred source buffer, thereby reducing the overhead associated with pageable transfer preparation. As a result, it can reduce the idle intervals between consecutive DMA operations, leading to a decrease in T_{idle}.

Asynchronous transfer mechanisms primarily target device inactivity. By relaxing the strict ordering constraints between transfer invocations, they enable the host to issue multiple transfer requests without blocking, allowing portions of transfer preparation to overlap with ongoing DMA execution. This reduces the cumulative idle time T_{idle}, while leaving the structure and number of DMA operations unchanged.

Argument Fusion operates on a fundamentally different dimension by altering the granularity of communication. By aggregating multiple transfers into a single operation, it reduces the number of DMA invocations and eliminates repeated per-transfer overheads. This directly reduces

the cumulative DMA execution time T_{DMA}. Additionally, by reducing the number of API calls, DMA programming operations, and synchronization points, it also reduces T_{idle}.

Taken together, these observations indicate that communication optimizations act along partially overlapping but distinct axes. Pinned memory and asynchronous execution reduce idle time by addressing pageable-memory handling and synchronization barriers, respectively, while Argument Fusion reduces both idle time and transfer fragmentation. Consequently, these techniques are complementary rather than mutually exclusive, and their combined effect must be evaluated explicitly rather than inferred from isolated behavior.

4.3 Argument Fusion Methodology

This section formalizes the execution configurations used throughout the evaluation. Rather than introducing a new technique, Argument Fusion is treated as a controlled transformation applied to the standard offloading model. The goal is to isolate its effect on communication behavior and to evaluate its interaction with existing optimization mechanisms.

We first define the baseline multi-argument offloading model, then describe the manual transformation that implements Argument Fusion, and finally introduce the set of evaluated configurations combining fusion with other communication optimizations.

4.3.1 Baseline Multi-Argument Offloading

In the baseline implementation, each kernel argument is allocated, transferred, and deallocated independently. For a kernel with N input arguments, this produces N allocation calls, N host-to-device transfers, and N deallocation calls, partitioning the total communication volume into multiple smaller operations.

This baseline configuration serves as the reference point for all subsequent transformations and evaluations.

4.3.2 Manual Argument Fusion Transformation

Argument Fusion transforms the baseline execution model by aggregating multiple kernel arguments into a single contiguous memory buffer. Instead of allocating and transferring each argument independently, all arguments are packed into a unified buffer prior to offloading.

Formally, given a set of arguments $\{A_1, A_2, \ldots, A_N\}$ with sizes $\{S_1, S_2, \ldots, S_N\}$, the transformation constructs a single buffer of size:

$$S_{\text{total}} = \sum_{i=1}^{N} S_i \tag{4.2}$$

The execution flow is modified as follows:

Code 4.1: Standard offloading paradigm code for a generic kernel with 3 input arguments.

```
// Host Allocation
A = malloc(n); B = malloc(m); C = malloc(z);
//1) Device Allocations
cudaMalloc((void **)&A_gpu,n);
cudaMalloc((void **)&B_gpu,m);
cudaMalloc((void **)&C_gpu,z);
//2) 3 Host--to--Device Transfers
cudaMemcpy(A_gpu,A,n,HostToDevice);
cudaMemcpy(B_gpu,B,m,HostToDevice);
cudaMemcpy(C_gpu,C,z,HostToDevice);
//3) Kernel Launch
kernel<<< grid1, block1 >>>(A_gpu,B_gpu,C_gpu);
//4) Writeback of Results
cudaMemcpy(result,A_gpu,n,DeviceToHost);
//5) 3 Device De--Allocations
cudaFree(A_gpu);
cudaFree(B_gpu);
cudaFree(C_gpu);
```

1. A single device allocation of size S_{total} is performed
2. A single host-to-device transfer is issued for the aggregated buffer
3. Individual arguments are accessed through offset-based addressing within the fused buffer
4. A single device deallocation replaces multiple deallocation operations

To preserve the original kernel interface, pointer variables corresponding to each argument are reassigned to offsets within the fused allocation. This allows the computational code to remain unchanged, while modifying only the data movement path.

Listings 4.1 and 4.2 illustrate the baseline and fused implementations for a representative multi-argument kernel. The scenario under inspection features three inputs A, B, and C of sizes n, m, and z, respectively.

In the baseline version (Listing 4.1), each argument is allocated and transferred independently; thus, three device allocations, three host-to-device transfers, and three deallocations are required.

In the fused version (Listing 4.2), a single allocation, host-to-device transfer, and deallocation are used. In order to avoid modifications to the kernel, pointer assignment is required for all but the first argument both for the host address and the device address.

In summary, manually applying *Argument Fusion* to a multi-input offload requires the developer to replace the separate host and device allocations with unified allocations whose size equals the aggregate size of all fused inputs. The original logical argument pointers are then reassigned to offsets within these unified allocations. The per-argument host-to-device transfers and device deallocations are replaced by one transfer and one deallocation over the fused buffer. This preserves the kernel's logical argument interface, but moves layout management and offset calculation into the host-side offloading code.

Code 4.2: 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 HtD 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);
```

Code 4.3: Pinned memory transformation.

```
1  //Standard host allocation
2  A = malloc(size);
3  //Pinned host allocation
4  cudaMallocHost((void **)&A, size);
5
6  //Standard host de-allocation
7  free(A);
8  //Pinned host de-allocation
9  cudaFreeHost(A);
10
```

Pinned Memory and Asynchronous Transfer Transformations

In addition to Argument Fusion, we apply two localized code transformations corresponding to pinned host memory and asynchronous transfers. These transformations modify host allocation and host–device transfer calls without altering kernel logic or data layout.

Pinned Memory. Pinned memory is introduced by replacing standard host memory allocation and deallocation calls with their page-locked counterparts. The transformation is shown in Listing 4.3.

Asynchronous Transfers. Asynchronous execution is enabled by replacing blocking transfer calls with their non-blocking counterparts using the default stream. A synchronization point is inserted after the final transfer to ensure completion. The transformation is shown in Listing 4.4.

Composability. These transformations are applied independently and can be combined with both baseline and fused implementations without additional structural changes.

Code 4.4: Asynchronous transfer transformation using the default stream.

```
1  //Synchronous transfer
2  cudaMemcpy(dst, src, size, kind);
3  //Asynchronous transfer (default stream)
4  cudaMemcpyAsync(dst, src, size, kind, 0);
5
6  //Synchronization before using transferred data
7  cudaDeviceSynchronize();
8
```

4.3.3 Optimized Configurations

The evaluation compares five communication configurations, each corresponding to a distinct organization of host–device transfers featuring an initial larger transfer followed by three smaller, same-sized transfers. These configurations are selected to isolate the main mechanisms available for reducing the inefficiencies introduced by segmented host-to-device communication: pageable-memory handling, synchronization-induced device inactivity, transfer fragmentation, and the combined effect of these mechanisms.

Figure 4.4 illustrates the expected host–device activity trace for each configuration. The scenario shown consists of one larger input transfer followed by several smaller input transfers, representing a common multi-argument offloading pattern. Time progresses from left to right. The upper lane of each configuration represents device-side DMA activity, while the lower lane represents host-side API activity, pageable-memory staging, and waiting intervals.

The baseline configuration, shown in Figure 4.4A, corresponds to the standard pageable synchronous transfer path. Each argument is handled independently, resulting in one transfer invocation per argument. Since the host buffers are pageable, each transfer may follow the pageable-memory staging path before the corresponding host-to-device DMA operation is executed. In addition, the blocking semantics of synchronous transfers introduce waiting intervals between consecutive transfers. As a result, the baseline exposes both forms of inefficiency discussed previously: idle periods in the device copy engine and fragmentation of the total communication volume into multiple smaller DMA operations.

The pinned-memory configuration, shown in Figure 4.4B, preserves the same per-argument transfer structure but replaces pageable host buffers with pinned host buffers. This avoids the pageable-memory staging path for the transferred source buffers and allows the device DMA engine to access the host buffers directly. However, the communication remains segmented: each argument is still transferred independently, and waiting intervals still appear between consecutive blocking transfers. Pinned memory therefore targets pageable-memory handling overhead, but does not by itself address transfer fragmentation.

The asynchronous-transfer configuration, shown in Figure 4.4C, also preserves the per-argument transfer structure, but replaces blocking transfer calls with asynchronous transfer calls. This allows the host to enqueue successive transfers more compactly, reducing the idle

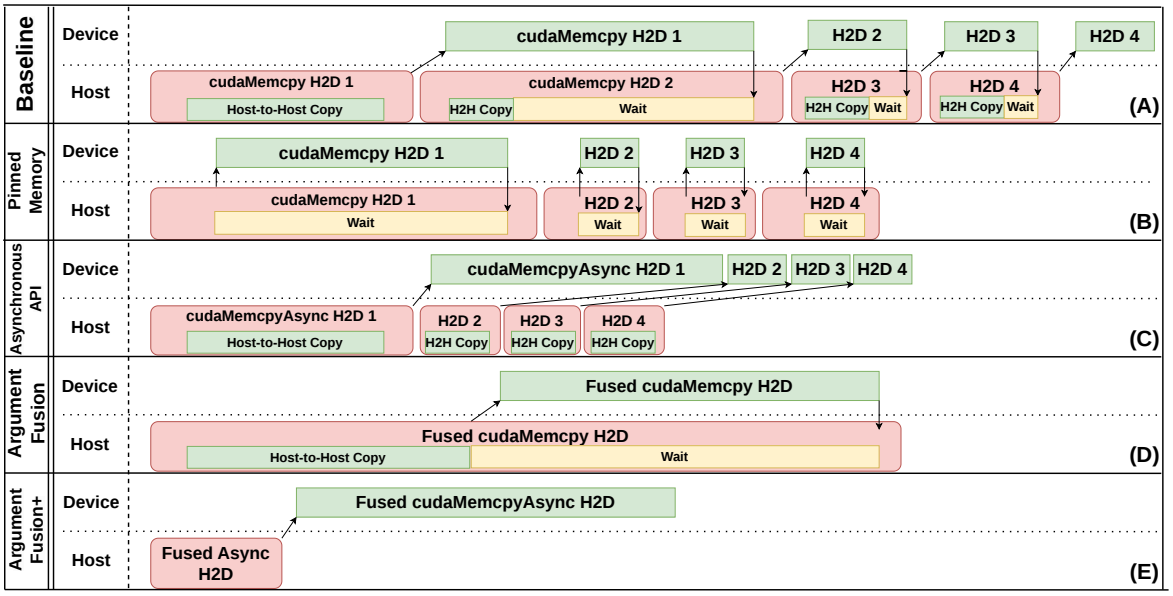


Figure 4.4: Conceptual host–device activity traces for the five evaluated communication configurations: (A) baseline pageable synchronous transfers, (B) pinned-memory synchronous transfers, (C) pageable asynchronous transfers, (D) Argument Fusion with a single fused pageable transfer, and (E) Argument Fusion+ with a single fused pinned asynchronous transfer. The figure illustrates how the configurations affect host-side staging, device-side DMA activity, transfer fragmentation, and waiting intervals.

intervals caused by host-side blocking between transfer submissions. Since the host buffers remain pageable in this configuration, pageable-memory handling may still be required before the corresponding DMA activity can occur. Asynchronous transfers therefore primarily target synchronization-induced device inactivity, while leaving the number and granularity of DMA operations unchanged.

The Argument Fusion configuration, shown in Figure 4.4D, changes the communication granularity itself. Instead of issuing one host-to-device transfer per argument, the input arguments are packed into a single fused buffer and transferred through one larger host-to-device operation. This removes the repeated per-transfer invocation overhead and shifts the communication pattern toward a larger burst transfer. Since the configuration still uses pageable memory and a synchronous transfer call, the fused transfer may still include pageable-memory handling and a final wait for completion. However, the fragmentation of device-side DMA activity is removed and idleness is significantly reduced.

The final configuration, denoted Argument Fusion+, is shown in Figure 4.4E. It combines the three mechanisms: fused communication granularity, pinned host memory, and asynchronous transfer submission. The input arguments are aggregated into a single fused buffer, allocated in pinned host memory, and transferred using an asynchronous host-to-device operation. This configuration therefore removes the per-argument transfer structure, avoids the pageable-memory staging path, and reduces host-side blocking during transfer submission. It represents the fully optimized configuration evaluated in this chapter.

Table 4.1: Evaluated communication configurations.

	Configuration	Transfer granularity	Host memory	Transfer API
A)	Baseline	Separate arguments	Pageable	Synchronous
B)	Pinned Memory	Separate arguments	Pinned	Synchronous
C)	Asynchronous Transfers	Separate arguments	Pageable	Asynchronous
D)	Argument Fusion	Fused arguments	Pageable	Synchronous
E)	Argument Fusion+	Fused arguments	Pinned	Asynchronous

Table 4.1 summarizes the five configurations and the communication mechanisms enabled in each case.

These configurations are not intended to form a complete factorial combination of all possible mechanisms. Instead, they correspond to the experimentally evaluated cases and to the conceptual traces in Figure 4.4. The comparison isolates the effect of each major optimization direction: pinned memory reduces pageable-memory handling overhead, asynchronous transfers reduce blocking-induced device inactivity, and Argument Fusion reduces communication fragmentation by changing the transfer granularity. Argument Fusion+ evaluates the combined effect of applying all three mechanisms simultaneously.

4.4 Experimental Setup

4.4.1 Platforms

The evaluation is conducted on two systems representative of distinct GPU architectures. Table 4.2 summarizes the hardware and software specifications of the experimental platforms.

Table 4.2: Experimental platforms.

Component	**System A**	**System B**
CPU Model	Intel(R) i7-8700K CPU @ 3.70GHz	13th Gen Intel(R) i9-13900K
Cores/Threads	6/12	24/48
GPU Model	NVIDIA Quadro P400	NVIDIA RTX 4090
Architecture	Pascal	Ada
Device Memory	2 GB	24 GB
Interconnect	PCIe Gen3	PCIe Gen4
CUDA Version	12.2	13.1
Compiler	NVCC	NVCC
Operating System	Ubuntu 20.04	Ubuntu 20.04
Driver Version	579.133.20	595.58.03
Nsight Systems	3.1	3.1

Ubuntu 20.04 had reached the end of its standard support period by the time of thesis submission. It is reported here because it was the operating-system version used for the experiments; no operating-system change or re-execution of the reported measurements was performed as part of the thesis revision.

The platforms in this chapter differ from those used later in the ICF evaluation because the two chapters study different system layers. This chapter evaluates manual Argument Fusion as a CUDA-level application transformation and therefore uses two locally controlled NVIDIA GPU systems that allow detailed component-level timing and profiling of `cudaMalloc`, `cudaMemcpy`, and `cudaFree`. Chapter 5, by contrast, evaluates a modified LLVM OpenMP runtime and therefore uses the systems available for that runtime implementation and deployment, including an HPC-class accelerator platform. The platform sets should therefore be interpreted as complementary rather than identical: Chapter 4 characterizes CUDA-level communication behavior, while Chapter 5 evaluates runtime-level OpenMP fusion.

4.4.2 Measurement Methodology

Host-Side Timing Measurements. All benchmark implementations are written as native CUDA applications (`.cu` files). Timing measurements are collected on the host side using the `std::chrono` library.

For the synthetic benchmarks, execution is decomposed into the following stages:

1. Device memory allocation
2. Host-to-device transfer
3. Kernel computation
4. Device-to-host transfer
5. Device memory deallocation

Each stage is timed independently. To ensure that the measured duration reflects the completion of device-side activity rather than only host-side API invocation time, a `cudaStreamSynchronize()` call is inserted at the end of each measured stage. As a result, each timing interval spans from the initial host-side API invocation to the completion of the corresponding device operation.

This synchronization strategy intentionally prevents overlap between stages during measurement. Consequently, interactions such as overlap between the API call to launch computation on the host and an in-flight host–to–device transfer are not reflected in the reported synthetic timings. The purpose of this methodology is to isolate the individual contribution of each execution stage rather than measure fully overlapped execution.

In addition, the purpose of the synthetic evaluation is to isolate and analyze the optimizable stages of the offload. As such, **compute time and device-to-host transfers are excluded** from

Code 4.5: CUDA warmup routine used prior to benchmark execution.

```
inline void cuda_warmup() {
    cudaFree(0);

    void* d = nullptr;
    unsigned char h[1024] = {0};

    cudaMalloc(&d, 1024);
    cudaMemcpy(d, h, 1024, cudaMemcpyHostToDevice);
    cudaMemcpy(h, d, 1024, cudaMemcpyDeviceToHost);
    cudaFree(d);

    cudaDeviceSynchronize();
}

```

the synthetic evaluation results. The synthetic "communication-path" metric used in the following sections is therefore the sum of device allocation, host-to-device transfer, and device deallocation time.

Statistical Methodology. Each synthetic benchmark configuration is executed 100 times. The reported value corresponds to the arithmetic mean after outlier removal using the Interquartile Range (IQR) method.

To avoid initialization effects and capture steady-state behavior, all benchmarks include a device warm-up phase prior to measurement, shown in Listing 4.5.

Application-Level Benchmark Measurements. For the application-level benchmarks, no intermediate synchronization points are inserted between execution stages. Instead, only the total execution time of the offloading pipeline is measured, spanning from the initial device allocation to the completion of device memory deallocation. This methodology preserves the natural execution behavior of the application and allows runtime overlap effects to emerge organically.

4.4.3 Synthetic Benchmark Parameters

The synthetic benchmark suite is designed to isolate and evaluate the communication behavior of multi-argument offloading independently of application-specific computational characteristics. In particular, the benchmarks focus on the performance of device allocation, host-to-device transfer, and device memory deallocation.

Controlled Parameters. The synthetic benchmark sweeps two independent communication dimensions:

- the total transferred data size S_{total} and

- the number of kernel arguments N_{args}.

The total transfer size ranges from 1 KiB to 1 GiB using powers-of-two scaling, while the argument count is varied across 2, 4, 8, and 16 arguments.

For each configuration, the size of the individual argument transfers is derived from these two parameters according to:

$$S_{\text{arg}} = \frac{S_{\text{total}}}{N_{\text{args}}} \tag{4.3}$$

where S_{total} is the total transferred data size and N_{args} is the number of kernel arguments.

Consequently, increasing the number of arguments increases communication fragmentation while preserving the total communication volume.

Synthetic Kernels and Validation. The synthetic benchmark uses a checksum kernel. This validation kernel is excluded from the reported timing measurements.

4.4.4 Application Benchmarks

In addition to the synthetic benchmark suite, the evaluation includes a set of application-level kernels drawn from the PolyBench benchmark suite [47]. PolyBench is a widely used collection of computational kernels designed for the evaluation of compiler optimizations and high-performance computing systems. The suite includes a diverse set of linear algebra and stencil-based workloads with well-defined memory access patterns and parameterized dataset sizes.

The selected benchmarks are `gemver`, `mvt`, `bicg`, `atax`, and an additional vector addition benchmark, `vadd`, included as a representative memory-bound baseline.

These benchmarks were selected due to their multi-array communication structure and varying degrees of transfer fragmentation. In particular, they expose different numbers of transferred arrays and combinations of one-dimensional and two-dimensional data structures, making them suitable for evaluating the effect of Argument Fusion on realistic offloading workloads.

Benchmark Characteristics. Table 4.3 summarizes the communication characteristics of the evaluated benchmarks.

Table 4.3: Communication characteristics of evaluated benchmarks.

Benchmark	1D Arrays	2D Arrays	Total Arguments
GEMVER	8	1	9
MVT	4	1	5
BICG	4	1	5
ATAX	3	1	4
VADD	3	0	3

Benchmark Descriptions.

- **GEMVER:** Dense matrix-vector update and multiplication kernel involving one two-dimensional matrix and multiple vector operands.
- **MVT:** Matrix-vector transpose multiplication benchmark operating on a dense matrix and multiple vectors.
- **BICG:** BiConjugate Gradient kernel involving matrix-vector multiplications with multiple vector inputs and outputs.
- **ATAX:** Matrix transpose and matrix-vector multiplication benchmark with mixed one-dimensional and two-dimensional data structures.
- **VADD:** Vector addition benchmark included as a simplified memory-bound reference workload.

Dataset Scaling. All PolyBench kernels are parameterized by a single variable N, where one-dimensional arrays contain N elements and two-dimensional arrays contain $N \times N$ elements. However, using identical values of N across benchmarks does not produce comparable communication volumes. For example, the total transferred data size of `gemver` is significantly larger than that of `vadd` for the same value of N due to the presence of additional arrays and two-dimensional operands.

Since Argument Fusion primarily targets communication behavior rather than element count, the PolyBench datasets were modified so that the evaluated configurations correspond to comparable total transferred data sizes across benchmarks. This normalization allows the evaluation to compare the effect of communication fragmentation independently of benchmark-specific array composition.

The original values of N for these benchmarks correspond to approximate total data sizes of 8 KiB for the MINI datasets and up to approximately 60 MiB for the EXTRA LARGE datasets, both of which are well within the 1 KiB to 1 GiB evaluation range.

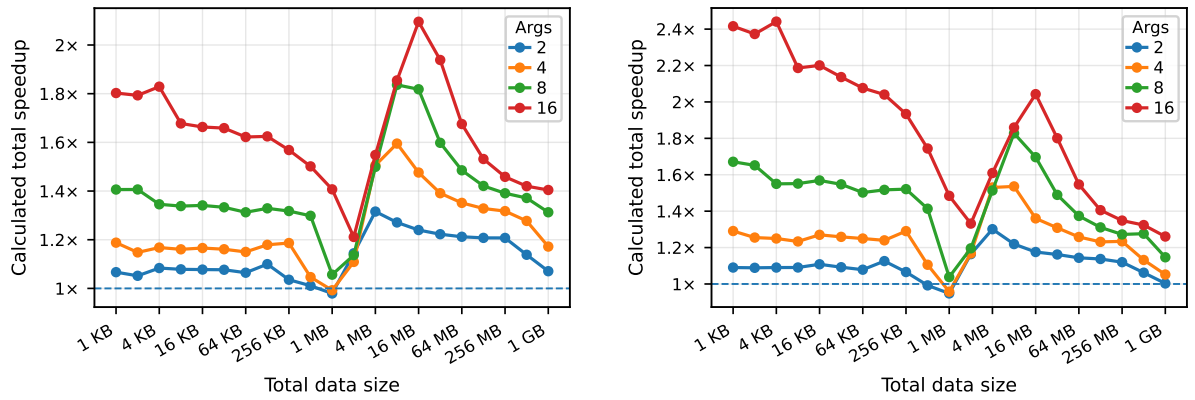
4.5 Synthetic Benchmark Evaluation

This section evaluates the behavior of Argument Fusion under controlled synthetic workloads. Using the benchmark methodology described in Section 4.4.3, we systematically vary the total transferred data size and the number of kernel arguments in order to characterize the relationship between communication fragmentation, transfer size and offloading performance.

The evaluation proceeds progressively from communication-path performance characterization to runtime component analysis and finally to the interaction between Argument Fusion and complementary communication optimizations.

This structure follows the operational objectives associated with RQ1. The communication-path sweep maps where fusion is profitable as a function of total data size and argument count, while the component-level decomposition identifies whether the observed gains originate from host-to-device transfer aggregation, allocation consolidation, deallocation consolidation, or their combination.

4.5.1 Communication-Path Impact of Argument Fusion



(a) Quadro P400

(b) RTX4090

Figure 4.5: Communication-path speedup of Argument Fusion over the baseline pageable synchronous implementation.

The qualitative expectation that aggregating several small transfers can reduce per-transfer overhead follows directly from the communication model established in Chapter 2 and from prior transfer-aggregation work. The purpose of the experiments in this section is therefore not to establish this expectation itself, but to quantify its practical range and magnitude, determine how it changes with argument count and transfer size, and establish how much of the resulting communication-path benefit is attributable to transfer aggregation versus allocation and deallocation consolidation.

We begin by evaluating the communication-path impact of Argument Fusion on synthetic multi-argument offloading workloads. Figure 4.5 presents the calculated communication-

path speedup[2] of the fused pageable synchronous implementation over the baseline pageable synchronous implementation while sweeping both the total transferred data size and the number of kernel arguments.

Several important trends emerge from the results.

First, the effectiveness of Argument Fusion scales strongly with the number of kernel arguments participating in the offload operation. Across all evaluated transfer sizes, higher argument counts consistently produce larger speedups. While the two-argument configuration achieves relatively modest gains, the 16-argument configuration approaches and surpasses speedups of 2× in multiple regions of the sweep. This behavior directly reflects the fragmentation model established in Section 4.2.2: increasing the number of arguments increases the number of independent allocations, transfers, and deallocation operations performed by the baseline runtime.

Second, the effectiveness of Argument Fusion is not uniform across transfer sizes. Large speedups are observed in the small-to-medium transfer regime, where the baseline implementation operates far from bandwidth saturation and repeated communication overhead dominates execution time. In this region, Argument Fusion significantly reduces the repeated incurrence of transfer setup costs and synchronization overheads by replacing multiple fragmented transfers with a single aggregated communication operation.

As transfer sizes increase further, the relative benefit of transfer aggregation becomes less pronounced. Larger transfers operate closer to the bandwidth-efficient region of the interconnect, reducing the proportional contribution of fragmentation overhead to total communication time. Nevertheless, measurable speedups persist even at large transfer sizes in which the global maximum of performance is observed. This indicates that allocation and deallocation multiplicity also notably contribute to the observed benefit.

Another notable observation is the presence of local discontinuities and non-monotonic regions across the sweep. In particular, several transfer-size regions exhibit sudden changes in relative speedup despite continuous scaling of the communication volume. These effects are especially visible in higher-argument configurations. Since the communication-path metric includes allocation and deallocation stages in addition to host-to-device transfers, these discontinuities suggest that CUDA allocator behavior contributes significantly to the observed runtime characteristics. This interpretation is examined directly in the component-level allocation analysis of Section 4.5.2.

Overall, the results demonstrate that Argument Fusion is fundamentally a communication-granularity optimization. Its effectiveness increases with transfer fragmentation and is most pronounced when the baseline execution is dominated by repeated runtime overheads rather than pure bandwidth-limited data movement.

[2]For the synthetic benchmark, communication-path speedup is derived from the sum of device allocation, host–to–device transfer, and device deallocation times. Checksum computation and device-to-host writeback are intentionally excluded in order to isolate the stages directly affected by Argument Fusion.

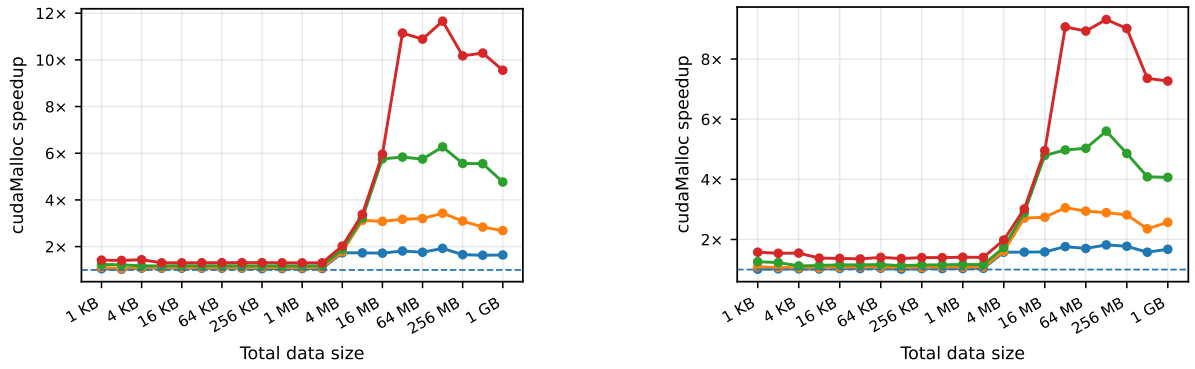
4.5.2 Runtime Component Analysis of Fusion

While the previous subsection established the communication-path impact of Argument Fusion, the observed speedups originate from multiple independently optimized runtime stages. In particular, Argument Fusion reduces the number of device memory allocations, host-to-device transfers, and device memory deallocations performed during the execution of the benchmark.

To isolate the contribution of each stage and to understand the performance trends of Figure 4.5, we now examine the speedup achieved individually for device allocation, host-to-device transfer, and device deallocation operations. All measurements compare the fused pageable synchronous implementation against the baseline pageable synchronous implementation.

Device Allocation Performance

Figure 4.6a presents the measured speedup of device memory allocation for the Quadro P400 platform. Figure 4.6b presents the equivalent results for the RTX4090 platform.



(a) Quadro P400

(b) RTX4090

Figure 4.6: Device allocation speedup of Argument Fusion over the baseline pageable synchronous implementation.



Figure 4.7: Nsight Systems trace of consecutive `cudaMalloc` calls with allocation sizes selected so that the highlighted region of each row reaches approximately 2 MiB of cumulative requested allocation. All rows are shown on the same time scale. The trace shows an expensive allocation followed by shorter allocation calls, suggesting behavior consistent with coarse-region allocation and subsequent suballocation. The final 4 MiB allocation is included as a reference for behavior beyond the apparent 2 MiB region.

Two important trends emerge from the allocation measurements.

First, allocation speedup scales strongly with argument count. This behavior is expected, since Argument Fusion replaces multiple independent allocation calls with a single aggregated device allocation. Consequently, the reduction in allocator invocations increases proportionally with the degree of communication fragmentation.

Second, unlike the host-to-device transfer speedup analyzed later, allocation speedup increases significantly for larger transfer sizes. For high argument-count configurations, speedups exceeding 9× are observed on both platforms for large allocation volumes. This behavior indicates that device allocation overhead is not constant with respect to allocation size [48].

The results additionally reveal discontinuities and sharp transitions across the sweep. In particular, the abrupt increase in speedup beyond the multi-megabyte regime suggests that the visible cost of `cudaMalloc` is affected by internal allocation thresholds or coarse-grained allocator behavior. To further inspect this behavior, Figure 4.7 shows an Nsight Systems trace of consecutive `cudaMalloc` calls. The experiment issues repeated allocations whose individual sizes are shown on the left of each row. For the 256 KiB, 512 KiB, 1 MiB, and 2 MiB rows, the sequence of allocations reaches an aggregate requested size of approximately 2 MiB; the final row shows a single 4 MiB allocation for comparison.

The trace shows that a relatively expensive allocation is followed by a sequence of shorter allocation calls. Once the cumulative requested size reaches the apparent multi-megabyte region, a subsequent allocation again incurs a larger cost. This pattern is consistent with state-dependent allocator behavior: the visible cost of an allocation may depend not only on the requested size of the current call, but also on allocator state and on whether the request crosses an internal management boundary. Recent allocator-characterization work similarly reports non-uniform CUDA allocation behavior and size-dependent effects in GPU memory management [48].

The experiment does not expose the internal CUDA allocator implementation and therefore does not prove that CUDA uses any one of these specific allocator designs. However, the Nsight Systems trace is consistent with the same general mechanism: an expensive operation appears when a coarse allocation region must be obtained or extended, while subsequent requests can be satisfied more cheaply until that region is exhausted. This provides timing evidence that the visible cost of `cudaMalloc` depends not only on the requested byte count of an individual call, but also on allocator state and apparent coarse allocation boundaries.

A second observation from Figure 4.7 is that a single larger allocation can be substantially cheaper than a sequence of smaller allocations whose cumulative requested size is comparable. In particular, the single 4 MiB allocation shown in the final row does not reproduce the cumulative cost of several smaller allocations that together reach the same multi-megabyte scale. This indicates that the allocator cost is not simply additive with respect to requested bytes. Instead, repeated allocator invocations introduce per-call overheads and may cross allocator-state boundaries multiple times, while one larger allocation can be served through a single allocator transaction. This behavior provides a plausible explanation for the abrupt increase in the `cudaMalloc` speedup

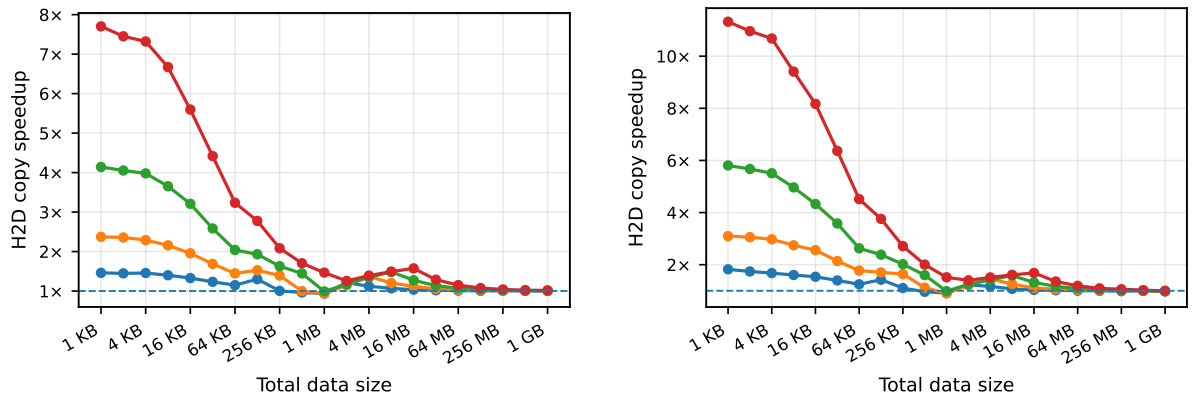
curves of allocations larger than 2 MiB: once the fused implementation replaces several smaller baseline allocations with one larger allocation, it can avoid both repeated invocation overhead and unfavorable interactions with coarse allocator regions.

This interpretation helps explain the structure of Figure 4.6. In the baseline implementation, the total allocation volume S_{total} is divided across N_{args} independent calls, each of size $S_{\text{total}}/N_{\text{args}}$. In the fused implementation, the same total volume is allocated through one call of size S_{total}. The trace suggests that these two organizations are not equivalent from the allocator's perspective: one large allocation can be cheaper than several smaller allocations with comparable aggregate size. If allocator cost changes around an apparent multi-megabyte coarse-region boundary, then the baseline and fused implementations cross that boundary at different points in the sweep. The fused implementation reaches the boundary when S_{total} itself reaches the relevant allocation scale, whereas the per-argument baseline reaches it only when each individual allocation, $S_{\text{total}}/N_{\text{args}}$, reaches that scale. This staggered crossing is consistent with the discontinuities and sudden speedup bursts observed across the allocation-speedup curves.

Most importantly, the results demonstrate that allocation behavior contributes substantially to the overall effectiveness of Argument Fusion. The communication-path speedups observed previously are therefore not solely attributable to transfer aggregation, but also to the consolidation of allocator activity. Argument Fusion changes both the number of allocator invocations and the allocation-size regime exercised by the runtime.

Host-to-Device Transfer Performance

Figure 4.8a presents the host-to-device transfer speedup achieved by Argument Fusion on the Quadro P400 platform, while Figure 4.8b presents the equivalent measurements for the RTX4090 platform.



(a) Quadro P400

(b) RTX4090

Figure 4.8: Host-to-device transfer speedup of Argument Fusion over the baseline pageable synchronous implementation.

The transfer measurements exhibit a substantially different behavior compared to device

allocation.

For the smallest transfer sizes, Argument Fusion achieves very large transfer-side speedups, reaching nearly 8× for the 16-argument configuration on the Quadro P400 platform and 11× for the RTX4090. This region corresponds directly to the fragmented low-efficiency regime discussed previously in Section 4.2.2.

However, unlike allocation speedup, transfer-side speedup decreases steadily as total transfer size increases. Beyond the multi-megabyte regime, the speedup approaches unity across all argument configurations. This behavior reflects the transition of the interconnect toward its bandwidth-efficient operating region. As transfer size increases, the proportional contribution of transfer setup overhead diminishes relative to the payload transmission time, reducing the relative advantage of transfer aggregation.

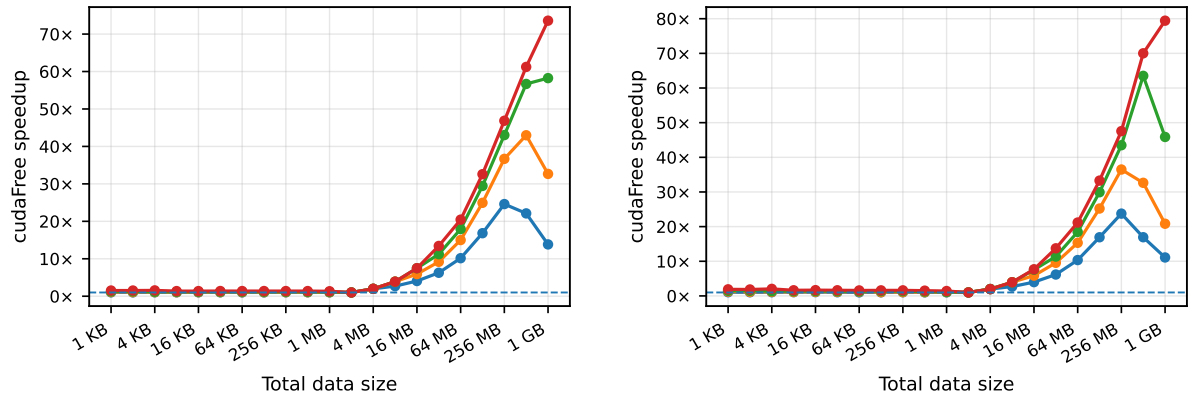
Importantly, this behavior confirms that Argument Fusion primarily optimizes communication fragmentation in transfers rather than raw peak bandwidth. Once transfers become sufficiently large to saturate the communication link efficiently, further aggregation yields diminishing returns on the transfer stage itself.

The transfer results additionally explain an important observation from the previous subsection. Although transfer-side speedup decreases significantly for large transfers, the communication-path speedup remains substantial. This indicates that other runtime stages, particularly allocation and deallocation, continue contributing to the overall effectiveness of Argument Fusion even after transfer efficiency improvements saturate.

Device Deallocation Performance

The final component isolated in the synthetic benchmark is device deallocation. Although deallocation is often treated as a small teardown operation, the results show that it can become a major source of speedup for Argument Fusion. In the baseline configuration, each mapped argument is associated with an independent device allocation and therefore with an independent device deallocation at the end of the offloaded region. In the fused configuration, the corresponding data are represented by a single aggregate device allocation and released with one deallocation.

Figure 4.9a presents the measured device deallocation speedup on the Quadro P400 platform, while Figure 4.9b presents the corresponding RTX4090 results.



(a) Quadro P400

(b) RTX4090

Figure 4.9: Device deallocation speedup of Argument Fusion over the baseline pageable synchronous implementation.

The speedup remains close to one for small data sizes, where all deallocation calls are cheap and API overhead dominates. At larger sizes, however, the speedup increases sharply. This behavior is especially visible as the number of baseline arguments increases, because the baseline performs one `cudaFree` call per argument, while the fused configuration performs only one `cudaFree` call for the aggregate allocation.

The deallocation-speedup curves do not follow a smooth monotonic trend. Instead, they exhibit size-dependent peaks whose position changes with the number of baseline arguments. For two, four, and eight arguments, the maximum appears at progressively larger total data sizes. This indicates that the relevant quantity is not only the aggregate data size S, but also the per-argument allocation size S/N in the fragmented baseline, where N is the number of mapped arguments. When S/N enters a high-cost deallocation regime, the baseline accumulates several expensive deallocation calls, whereas the fused version still pays only one deallocation for the aggregate allocation. Consequently, the peak of the deallocation speedup shifts to larger total data sizes as the number of arguments increases.

To better understand this behavior, Figure 4.10 reports the individual `cudaFree` timings of the fragmented pageable synchronous baseline for four and eight buffers across the full data-size sweep. Each colored curve corresponds to one deallocation in the baseline teardown sequence. The dashed black curve reports the single fused deallocation for the same aggregate data size. The figure is not intended to reverse engineer the CUDA allocator, but to expose the sequence-level behavior that produces the large deallocation speedups in Figure 4.9.

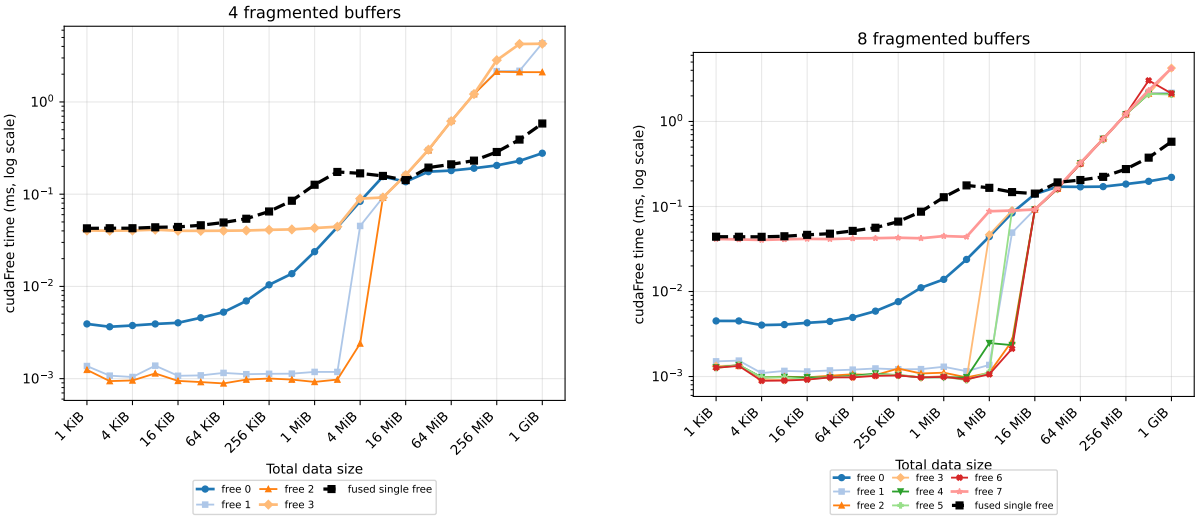
Several observations follow from this decomposition. First, the first baseline deallocation behaves differently from the rest of the sequence. It grows gradually with data size and remains comparatively cheap over most of the sweep. Second, the last deallocation is already more expensive at small sizes and follows a different trajectory. Third, the intermediate deallocations do not all transition at the same point. They exhibit smaller preliminary steps followed by larger jumps into higher-cost regimes. This qualitative structure is visible for four and eight fragmented

buffers, although the exact transition points and the number of affected deallocations vary with the number of buffers.

This behavior explains why the deallocation speedup can increase and later decrease. The fused deallocation remains substantially cheaper in absolute time, but it is not constant: the time of the single fused `cudaFree` still grows with the aggregate allocation size. Once the fragmented baseline has already entered a high-cost deallocation regime, its additional growth may slow down, while the fused deallocation continues to increase. In such regions, the speedup ratio can decrease even though the fused path remains faster in absolute time.

These measurements should be interpreted as application-level timing evidence, not as a precise model of CUDA allocator internals. The CUDA allocator and driver implementation are not exposed by the programming model, and the exact internal mechanism responsible for the observed transitions cannot be determined from these measurements alone. Nevertheless, the result is consistent with known CUDA memory-management behavior. Prior work has shown that CUDA memory-management calls such as `cudaFree` can introduce synchronization effects in real-time GPU workloads [58]. NVIDIA also documents that CUDA runtime calls may exhibit synchronization or blocking behavior depending on the operation and the required runtime resources, and provides stream-ordered allocation and deallocation as an alternative mechanism for ordering allocation lifetimes with respect to CUDA streams [42, 41]. Furthermore, recent allocator-characterization work reports non-uniform CUDA allocation behavior and size-class-like effects in GPU memory management [48]. The measurements in this subsection are therefore consistent with prior evidence that CUDA memory management can exhibit non-trivial, non-uniform timing behavior, while the specific first-free and multi-free-tail pattern remains an empirical observation of this benchmark and platform.

The important conclusion for Argument Fusion is structural. The fragmented baseline exposes the CUDA runtime to a sequence of independent device deallocations whose cost is non-uniform across both position and allocation size. Argument Fusion replaces this fragmented teardown with a single deallocation. The resulting speedup is therefore not merely a reduction in the number of API calls; it also avoids the expensive tail of a multi-deallocation sequence.



(a) Four fragmented buffers.

(b) Eight fragmented buffers.

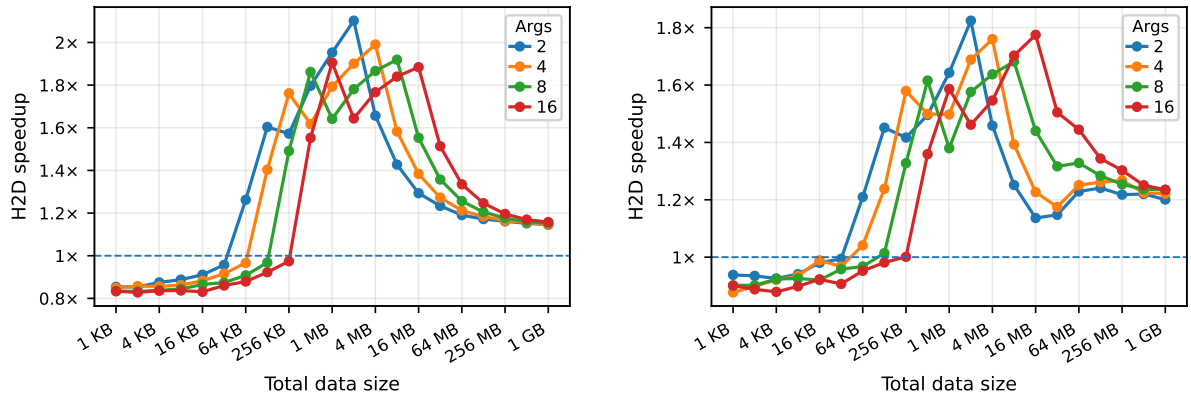
Figure 4.10: Individual `cudaFree` timings for the fragmented pageable synchronous baseline. Each colored curve corresponds to one deallocation in the baseline teardown sequence. The dashed black curve reports the single fused deallocation for the same aggregate data size. Across four and eight buffers, the first deallocation remains comparatively cheap, while later deallocations exhibit step-like transitions into higher-cost regimes. Argument Fusion avoids this multi-deallocation tail by replacing the fragmented teardown sequence with one fused device deallocation.

4.5.3 Impact of Pinned Host Memory

The previous subsections addressed the first three operational objectives associated with RQ1: characterizing transfer segmentation, identifying when Argument Fusion is beneficial, and decomposing its component-level sources of speedup. The remainder of the synthetic evaluation addresses the final operational objective by examining how Argument Fusion relates to established CUDA communication optimizations. We first isolate the impact of pinned host memory on host-to-device transfer behavior.

As discussed previously in Section 4.3.2, pinned memory avoids the pageable-memory staging path by exposing page-locked host buffers directly to the device DMA engine. Consequently, the runtime can avoid part of the pageable transfer preparation required for ordinary host allocations.

Figure 4.11a presents the measured host-to-device transfer speedup of pinned memory over pageable memory on the Quadro P400 platform, while Figure 4.11b presents the corresponding RTX4090 results.



(a) Quadro P400

(b) RTX4090

Figure 4.11: Host-to-device transfer speedup of pinned memory over pageable memory for varying transfer sizes and argument counts.

The results reveal a markedly different behavior compared to Argument Fusion.

For very small transfer sizes, pinned memory provides little benefit and may even introduce slight slowdowns relative to pageable memory. In this regime, the transfer payload is too small for DMA efficiency to dominate execution time, while the overhead associated with managing pinned allocations remains non-negligible.

As transfer sizes increase, however, pinned memory begins providing substantial transfer-side acceleration. Across all argument configurations, the speedup rises sharply in the sub-megabyte to low-megabyte regime, reaching speedups approaching 2× on the Quadro P400 platform.

Unlike Argument Fusion, the effectiveness of pinned memory does not scale monotonically with argument count. Instead, the curves exhibit similar overall behavior shifted across different transfer-size regions. This suggests that pinned memory primarily targets a transfer-side inefficiency that depends more strongly on transfer size than on communication fragmentation itself.

Beyond the multi-megabyte regime, the relative benefit of pinned memory gradually decreases, although measurable speedups persist even at large transfer sizes. This behavior indicates that pageable-memory overhead becomes proportionally less significant once transfers enter the bandwidth-efficient operating region of the interconnect.

4.5.4 Impact of Asynchronous Transfers

We next evaluate the effect of asynchronous transfer mechanisms on host-to-device communication behavior.

As discussed previously in Section 4.2.6, synchronous transfer operations serialize portions of the communication pipeline by forcing the host thread to wait before additional transfer work can be scheduled. Asynchronous transfers relax this behavior by allowing transfer operations to be enqueued without immediate host-side blocking.

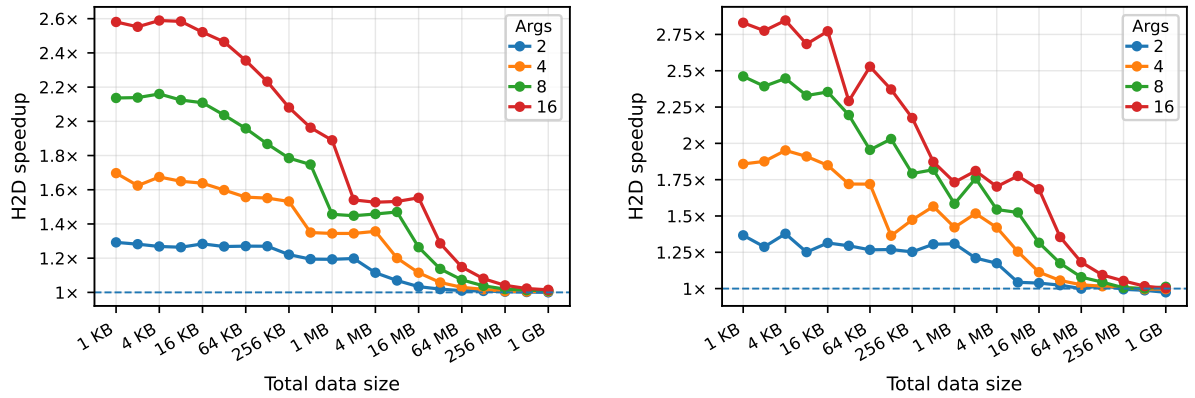
However, asynchronous transfers are closely tied to the host memory model used by the runtime. According to NVIDIA documentation, the host-visible synchronization behavior of memory-copy calls depends on transfer direction and on whether host memory is pageable or pinned [42]. When pageable memory is used, the CUDA runtime may internally synchronize portions of the transfer process due to the implicit staging operation required before DMA transfer can begin. Consequently, pageable asynchronous transfers may not achieve the same degree of overlap and scheduling efficiency as pinned asynchronous transfers.

To isolate these effects, we evaluate two separate configurations:

1. pageable asynchronous transfers compared against pageable synchronous transfers,
2. pinned asynchronous transfers compared against pageable synchronous transfers.

Pageable Asynchronous Transfers

Figure 4.12a presents the measured host-to-device transfer speedup of pageable asynchronous transfers over pageable synchronous transfers on the Quadro P400 platform, while Figure 4.12b presents the corresponding RTX4090 results.



(a) Quadro P400 (b) RTX4090

Figure 4.12: Host-to-device transfer speedup of pageable asynchronous transfers over pageable synchronous transfers.

The results show that asynchronous transfers provide substantial acceleration for highly fragmented communication patterns. The largest gains are observed for high argument-count configurations and small transfer sizes, where the baseline synchronous implementation incurs significant host-side serialization overhead between successive transfers.

Unlike pinned memory, whose effectiveness peaks in the medium transfer-size regime, asynchronous execution exhibits its strongest benefit in the highly fragmented small-transfer region. This behavior aligns directly with the device inactivity model established previously in Section 4.2.2. By allowing transfer operations to be queued without immediate blocking,

asynchronous execution reduces idle gaps between consecutive DMA operations and improves transfer scheduling efficiency.

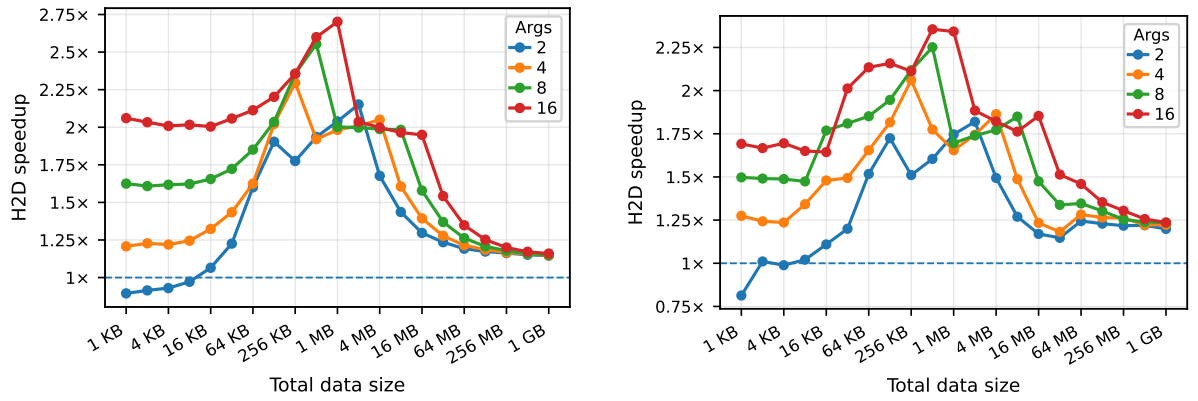
As total transfer size increases, however, the relative benefit of asynchronous execution decreases steadily across all argument configurations. Once transfers become sufficiently large, the execution becomes increasingly dominated by the transfer payload itself rather than by inter-transfer scheduling overhead. Consequently, the proportional contribution of asynchronous scheduling optimizations diminishes.

Importantly, the results additionally show that pageable asynchronous transfers alone do not eliminate all communication overhead. Although substantial improvements are observed, the performance gains remain bounded relative to the theoretical limit of fully overlapped asynchronous communication. This behavior is consistent with the pageable-memory staging model discussed previously, where portions of the transfer process may still require implicit synchronization before DMA execution can begin.

The speedup trend of pageable asynchronous transfers is qualitatively similar to that of Argument Fusion in the small-transfer regime, because both reduce visible overheads associated with segmented communication. However, direct comparison shows that Argument Fusion outperforms pageable asynchronous transfers across most of the total-size sweep, indicating that changing transfer granularity removes overheads that asynchronous submission alone cannot eliminate.

Pinned Asynchronous Transfers

Figure 4.13a presents the measured host-to-device transfer speedup of pinned asynchronous transfers over the baseline pageable synchronous implementation on the Quadro P400 platform, while Figure 4.13b presents the corresponding RTX4090 measurements.



(a) Quadro P400 (b) RTX4090

Figure 4.13: Host-to-device transfer speedup of pinned asynchronous transfers over the baseline pageable synchronous implementation.

Combining pinned memory with asynchronous execution produces substantially larger

speedups across almost the entire sweep. In contrast to pageable asynchronous transfers, the pinned asynchronous configuration simultaneously benefits from reduced pageable-memory handling overhead and improved transfer scheduling behavior.

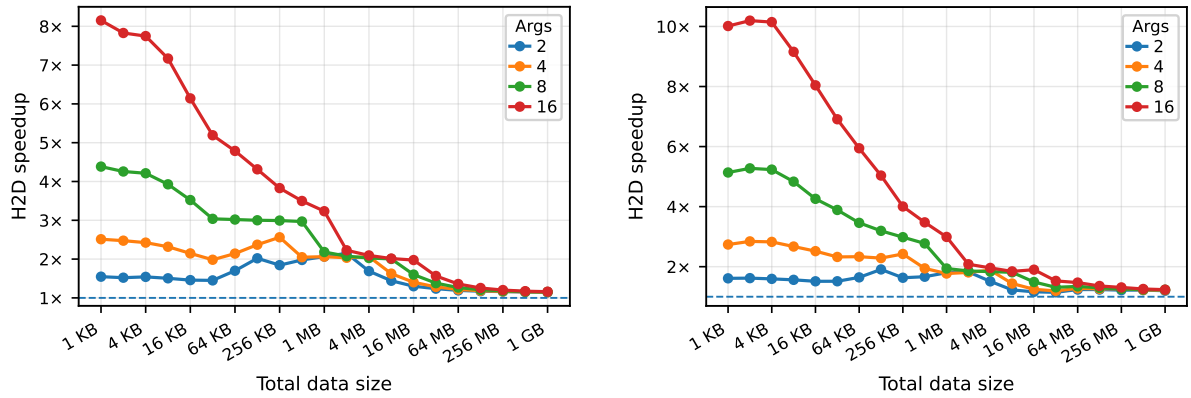
The results reveal a broad high-performance region spanning from small fragmented transfers to medium-sized communication volumes. In particular, high argument-count configurations achieve speedups exceeding 2× on both experimental platforms.

4.5.5 Fusion+ and Combined Communication Optimizations

We finally evaluate the fully optimized configuration, denoted as *Fusion+*, which combines Argument Fusion, pinned host memory, and asynchronous transfers. The objective of this evaluation is to determine whether the three optimization mechanisms remain complementary when applied simultaneously.

Fusion+ Host-to-Device Transfer Performance

Figure 4.14 presents the host-to-device transfer speedup of Fusion+ over the baseline pageable synchronous implementation on the Quadro P400 and RTX4090 platforms.



(a) Quadro P400 (b) RTX4090

Figure 4.14: Host-to-device transfer speedup of Fusion+ over the baseline pageable synchronous implementation.

Fusion+ produces the highest transfer-side speedups among the evaluated configurations across most of the parameter sweep. Compared with pageable asynchronous transfers, Fusion+ benefits not only from more compact transfer submission but also from the removal of per-argument transfer fragmentation. Compared with pinned asynchronous transfers, Fusion+ adds the reduction in the number of DMA transfer invocations. Thus, while pinned asynchronous execution improves the efficiency and scheduling of individual transfers, Fusion+ additionally changes the transfer granularity.

The largest transfer-side gains appear in the fragmented small-to-medium transfer regime, especially for high argument counts. In this region, the baseline repeatedly incurs transfer setup

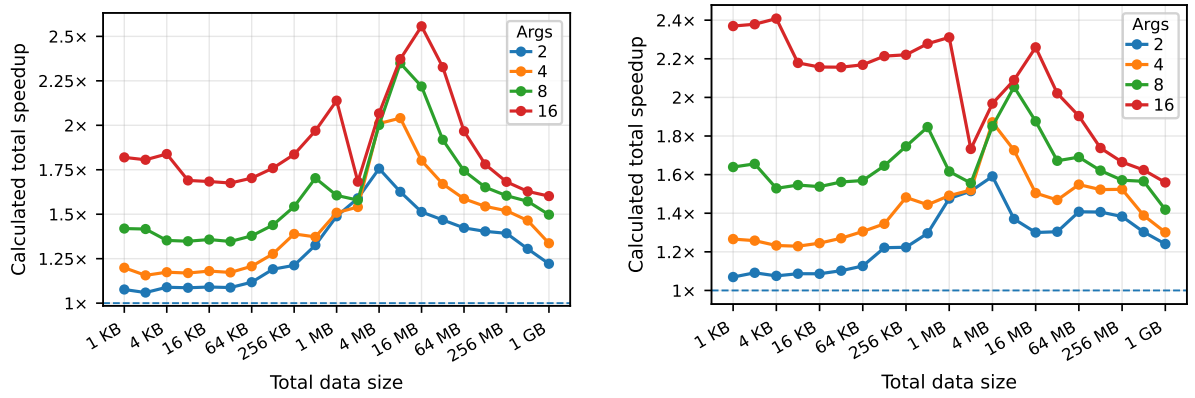
overheads, pageable-memory handling costs, and synchronization effects. Fusion+ reduces all three effects simultaneously: it aggregates the payload, uses pinned host memory, and submits the resulting transfer asynchronously.

At very small transfer sizes, however, the interaction between the mechanisms is not always uniformly beneficial. Pinned memory may introduce overheads that are not fully amortized by the small payload. This explains why some low-data-size regions do not improve as strongly as the medium-size region, particularly on the higher-bandwidth RTX4090 platform. As the transfer size increases into the sub-megabyte and low-megabyte range, the benefits of pinned asynchronous communication become more visible and Fusion+ outperforms the individual optimizations more consistently.

As total transfer size increases further, the relative benefit of Fusion+ gradually decreases. At sufficiently large transfer sizes, communication approaches the bandwidth-saturation regime of the interconnect, reducing the proportional contribution of transfer-side startup and scheduling overhead. Nevertheless, measurable gains can persist because Fusion+ still reduces the number of transfer invocations and the associated runtime orchestration.

Fusion+ Communication-Path Performance

Figure 4.15 presents the communication-path speedup of Fusion+ over the baseline pageable synchronous implementation on the Quadro P400 and RTX4090 platforms.



(a) Quadro P400

(b) RTX4090

Figure 4.15: Communication-path speedup of Fusion+ over the baseline pageable synchronous implementation.

The communication-path results confirm that the evaluated mechanisms are largely complementary. Compared with standalone Argument Fusion, Fusion+ consistently raises the performance envelope because it combines reduced allocation and transfer multiplicity with a more efficient host-memory and transfer-submission path.

The strongest improvements occur in the medium-transfer regime. In this region, the cost of transfer fragmentation, pageable-memory handling, and device inactivity are all still significant,

while the transferred payload is large enough to amortize the overhead of pinned-memory use. This makes the medium-size regime the point at which the three mechanisms interact most constructively.

The comparison with pinned asynchronous transfers is also important. Pinned asynchronous transfers improve the communication path without changing argument granularity. Fusion+ retains those benefits while also reducing the number of allocation, transfer, and deallocation operations. This explains why Fusion+ dominates the total communication-path measurements even in cases where its transfer-side advantage over pinned asynchronous execution is less visually pronounced.

Overall, the results demonstrate that Argument Fusion, pinned host memory, and asynchronous transfer submission act on distinct but interacting components of the communication pipeline. Argument Fusion reduces communication granularity and allocator multiplicity, pinned memory avoids the pageable-memory staging path, and asynchronous execution reduces transfer serialization and device inactivity. When combined, these effects produce the strongest communication-path performance observed in the synthetic evaluation.

4.5.6 Synthetic Evaluation Synthesis

The synthetic evaluation shows that the communication-path speedup of Figure 4.5 results from different mechanisms in different parts of the parameter space. Table 4.4 summarizes the main relationships established by the component-level experiments.

Table 4.4: Summary of the runtime factors influencing the synthetic communication-path results.

Factor	Where it is most relevant	Observed effect
Transfer fragmentation	Small and medium per-argument transfers, especially at high argument counts	Aggregation reduces repeated transfer overhead and improves active-copy efficiency; the benefit diminishes as individual transfers approach the bandwidth-efficient regime.
Device inactivity and transfer submission	Highly fragmented communication with repeated transfer invocations	Asynchronous submission reduces gaps between transfers, while fusion also reduces the number of transfer operations that must be submitted.
Device allocation	Multiple independent allocations, with increasingly visible effects at larger sizes	Fusion replaces multiple `cudaMalloc` calls with one allocation; non-uniform allocator behavior contributes to the communication-path speedup.
Device deallocation	Multiple independent deallocations, particularly in higher-cost size regimes	Fusion replaces the fragmented `cudaFree` sequence with one deallocation, avoiding both multiplicity and the expensive tail observed in the baseline sequence.
Pageable-memory handling	Transfer sizes for which staging overhead remains significant	Pinned memory removes the pageable-memory staging path and is complementary to fusion rather than a replacement for it.

The resulting communication-path behavior therefore cannot be attributed to a single component. At small and medium transfer sizes, host-to-device transfer aggregation and reduced transfer-side overhead are major contributors. As individual transfers become larger and their transfer-side benefit approaches saturation, allocation and deallocation consolidation account for a larger share of the remaining communication-path gain. Fusion+ further combines these effects with pinned-memory and asynchronous-transfer benefits. This decomposition explains why the total speedup in Figure 4.5 does not follow the host-to-device transfer-speedup curve alone.

4.6 Application-Level Evaluation on PolyBench

The synthetic evaluation addresses the operational objectives associated with RQ1. This section evaluates whether the same conclusions remain visible when the communication path is embedded

in complete application-level kernels. The purpose is not to introduce a new research question, but to test the external validity of the synthetic trends under realistic argument structures, computation, copy-out, and benchmark-specific data layouts.

4.6.1 Benchmark Selection

To evaluate whether the trends observed in the synthetic benchmark persist in application-level workloads, Argument Fusion was evaluated using a subset of kernels from the PolyBench benchmark suite [47]. The selected benchmarks expose different argument structures, transfer granularities, and compute-to-communication ratios while preserving an explicit CUDA-style offloading model.

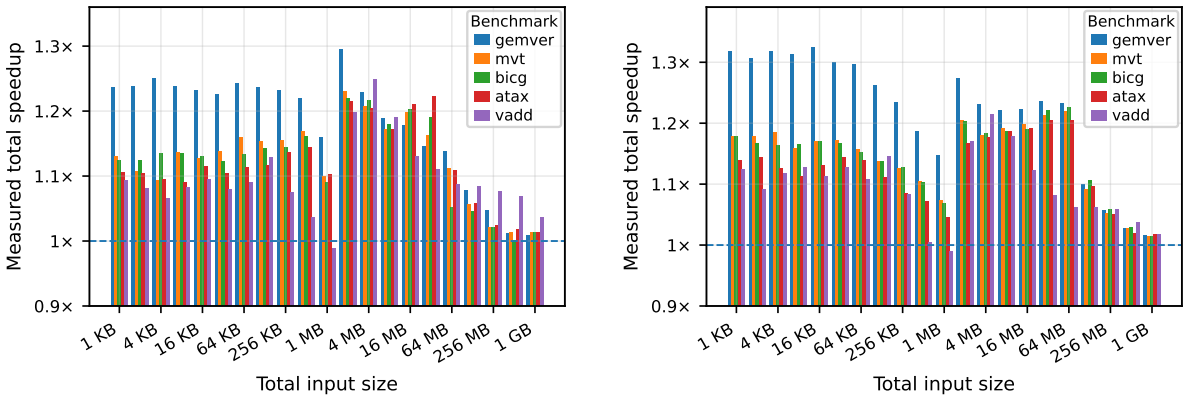
The evaluation includes `gemver`, `mvt`, `bicg`, `atax`, and a vector-addition benchmark, `vadd`. The PolyBench kernels represent dense linear-algebra patterns involving combinations of two-dimensional matrix operands and one-dimensional vector operands. The `vadd` benchmark is included as a simplified memory-bound reference because its uniformly sized vector arguments approximate the idealized transfer structure used in the synthetic evaluation.

4.6.2 Argument Counts and Dataset Sizes

The benchmark configurations used throughout the evaluation are summarized in Table 4.3. Dataset sizes were selected so that the overall transferred data volume spans multiple orders of magnitude, from kilobytes to gigabytes. The aggregated transfer volume reported in the figures corresponds to the total communication volume obtained from PolyBench-style dataset scaling. The original PolyBench size classes correspond approximately to 8 KiB for MINI, 60 KiB for SMALL, 640 KiB for STANDARD, 15 MiB for LARGE, and 60 MiB for EXTRA LARGE datasets.

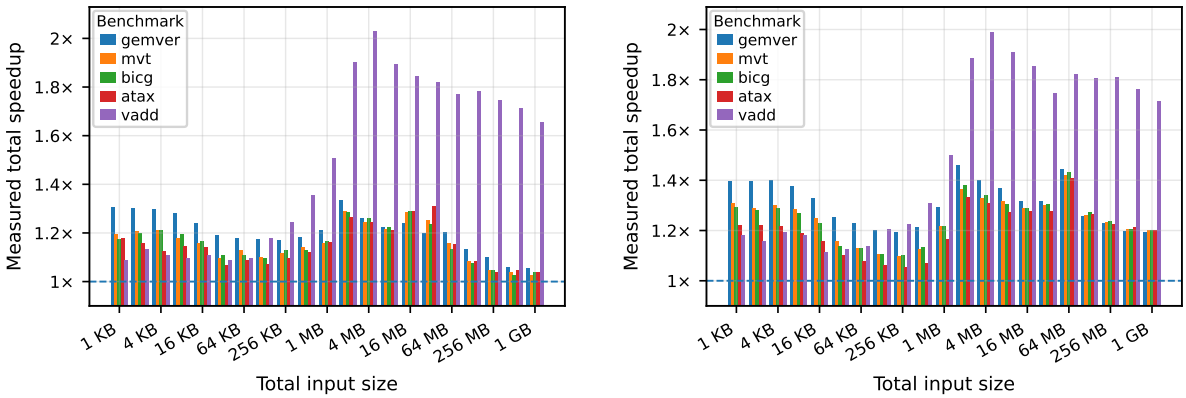
For each benchmark, the problem dimensions were adjusted so that configurations with comparable total transfer volumes could be evaluated across applications. This normalization is necessary because the same value of the PolyBench parameter N does not imply the same communication volume across benchmarks. A benchmark with one $N \times N$ matrix and several vectors transfers substantially more data than a benchmark containing only vectors for the same value of N.

Despite this normalization by total transferred volume, the internal argument composition remains different across benchmarks. The PolyBench kernels contain a dominant matrix transfer together with several smaller vector transfers, whereas `vadd` contains uniformly sized vector transfers. This distinction is important for interpreting the results: equal total transferred bytes do not imply equal communication fragmentation.



(a) Quadro P400 (b) RTX4090

Figure 4.16: Measured end-to-end speedup of Argument Fusion relative to the baseline implementation across PolyBench applications.



(a) Quadro P400 (b) RTX4090

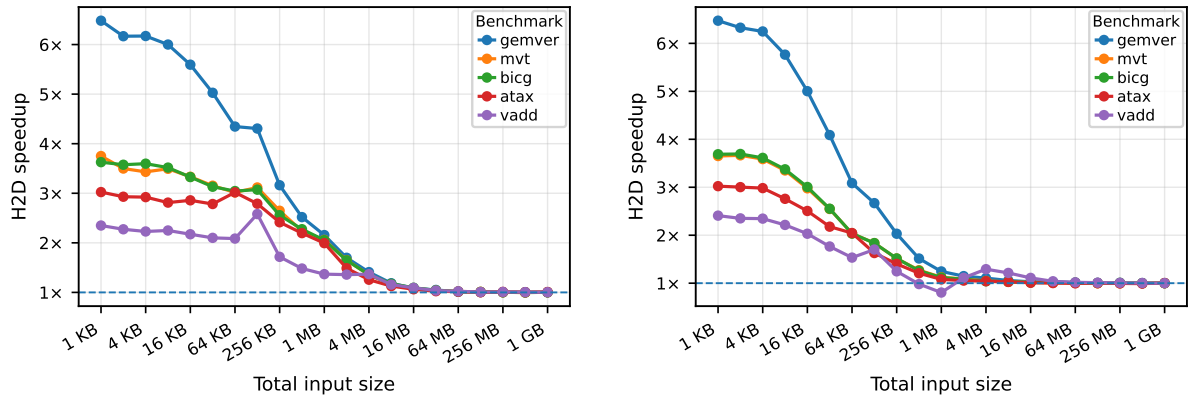
Figure 4.17: Measured end-to-end speedup of Fusion+ relative to the baseline implementation across PolyBench applications.

4.6.3 Performance Results

Figures 4.16 and 4.17 report the measured end-to-end speedups of Argument Fusion and Fusion+, respectively, relative to the baseline implementation.

The end-to-end results show that Argument Fusion remains beneficial beyond the controlled synthetic benchmark, but with lower and more benchmark-dependent speedups. This is expected because the application measurements include kernel execution, writeback, allocation and deallocation behavior, and benchmark-specific data layout. The synthetic benchmark should therefore be interpreted as a favorable reference case for equal-size fragmented arguments, while the application results show how much of that opportunity survives in complete kernels.

In Figure 4.16, standard Argument Fusion produces measurable speedups across the evaluated benchmarks, but the magnitude differs substantially by application. Among the PolyBench kernels, `gemver` is generally the strongest case in these measurements. This follows from its



(a) Quadro P400

(b) RTX4090

Figure 4.18: Measured host-to-device transfer speedup of Argument Fusion relative to the baseline implementation across PolyBench applications.

larger argument set: `gemver` transfers one matrix and eight vectors, creating more independent baseline allocation and transfer operations than `mvt`, `bicg`, or `atax`. The matrix still dominates the byte volume at larger sizes, but the additional vector operands increase the runtime operation count that can be consolidated by fusion.

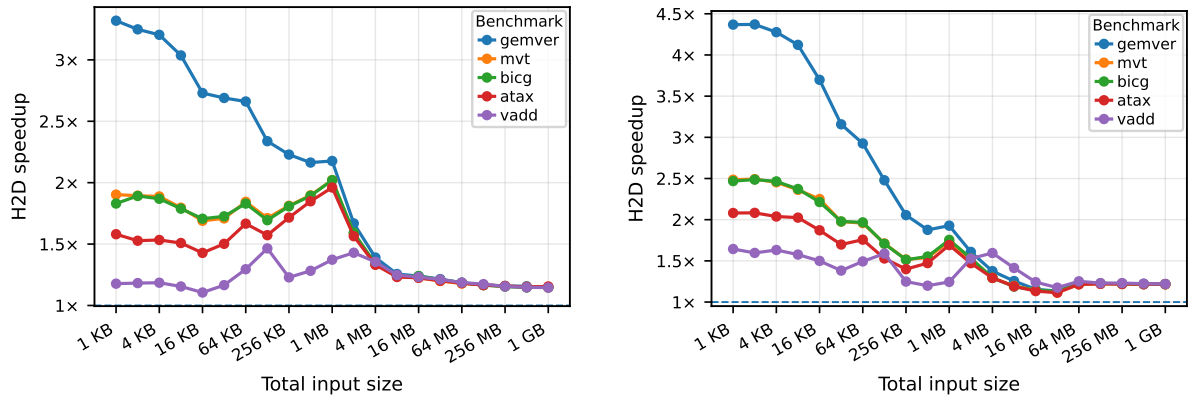
The `vadd` benchmark exhibits a distinct trend. Its operands are uniformly sized one-dimensional vectors, so its argument-size distribution is more regular than that of the matrix-based kernels. This makes it structurally closer to the equal-size synthetic case. However, `vadd` has only three transferred arguments and its measured end-to-end interval includes computation and writeback.

Figure 4.17 shows that Fusion+ changes the end-to-end behavior in a benchmark- and size-dependent way. The combination of fusion, pinned memory, and asynchronous transfer submission can produce larger gains when pageable-memory overhead, transfer scheduling, and fragmented communication are visible in the measured interval. This is particularly apparent in regions where the regular vector-only structure of `vadd` exposes the combined communication optimizations effectively. However, Fusion+ does not act as a uniform multiplier over standard Argument Fusion: its benefit depends on the interaction between pinned-memory behavior, asynchronous submission, allocator effects, computation, and writeback.

Figures 4.18 and 4.19 isolate the host-to-device transfer component.

The transfer-only figures show the direct effect of changing communication granularity. In both figures, `gemver` exhibits the largest transfer-side benefit among the PolyBench kernels. This result is consistent with the synthetic evaluation: transfer aggregation is most effective when the baseline contains many separate communication objects. In `gemver`, the nine transferred operands create substantially more baseline transfer multiplicity than in the other selected PolyBench kernels.

The H2D results also show why nominal argument count alone is insufficient. For example, `atax` has four transferred arguments, but its behavior still differs from the equal-size synthetic



(a) Quadro P400

(b) RTX4090

Figure 4.19: Measured host-to-device transfer speedup of Fusion+ relative to the baseline implementation across PolyBench applications.

case. In the synthetic benchmark, the total payload is divided evenly across the participating arguments. In `atax`, most transferred bytes belong to the two-dimensional matrix, while the remaining one-dimensional vectors contribute a smaller fraction of the total payload. The effective fragmentation opportunity is therefore smaller than the nominal argument count suggests. In the measured H2D results of Figure 4.19, `atax` reaches a maximum speedup of approximately 2× and 1.7× on the Quadro P400 and RTX 4090 platforms, respectively, compared with the synthetic four-argument H2D maximum of approximately 2.8× and 2.2× of Figure 4.14.

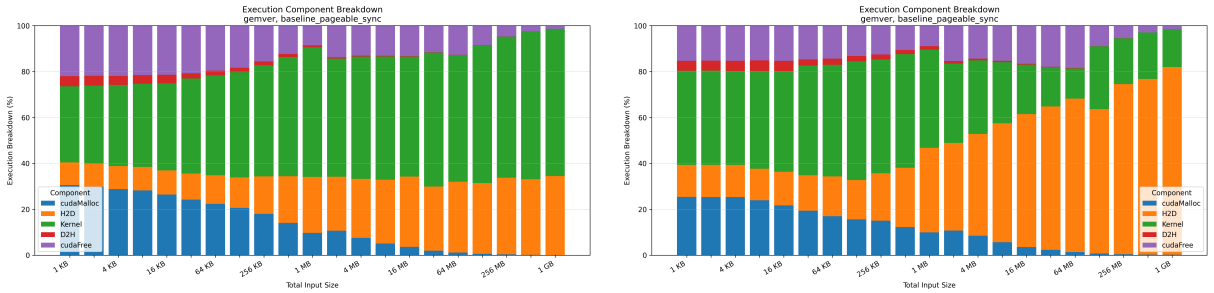
The same interpretation applies to `mvt` and `bicg`. Both contain a dominant matrix operand and four smaller vector operands. Fusion reduces the number of transfer invocations, but the largest transfer in the baseline already amortizes part of the transfer startup cost. Their transfer-side speedups are therefore lower than those of an equal-size synthetic benchmark with comparable nominal argument count.

The `vadd` transfer results reflect the opposite tradeoff. Its vector operands are regularly sized, so their byte distribution is favorable for fusion. However, only three host-to-device transfers are consolidated. Consequently, `vadd` can behave more regularly than the matrix-based kernels without necessarily exceeding the speedups of benchmarks with larger argument sets.

Figure 4.19 combines the transfer-side effects of fusion, pinned memory, and asynchronous submission. The resulting ordering is not identical to standard Argument Fusion because the measured speedup now includes changes in host-memory transfer behavior and transfer scheduling, not only transfer-count reduction. This explains why some benchmarks or size ranges benefit disproportionately under Fusion+: the regularity of the source operands, the number of eliminated transfers, and the pinned/asynchronous transfer behavior all contribute to the observed result.

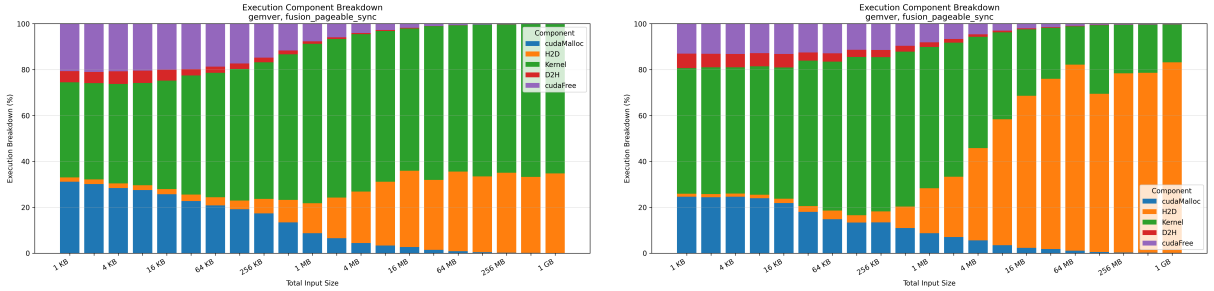
Figures 4.20 and 4.21 present normalized execution breakdowns for the `gemver` benchmark under the baseline and fused pageable synchronous implementations.

The breakdowns explain why transfer-side speedup and end-to-end speedup differ. Argument



(a) Quadro P400 (b) RTX4090

Figure 4.20: Normalized execution breakdown of the baseline pageable synchronous implementation for `gemver`.



(a) Quadro P400 (b) RTX4090

Figure 4.21: Normalized execution breakdown of the fused pageable synchronous implementation for `gemver`.

Fusion reduces allocation, host-to-device transfer, and deallocation components. As these components shrink, kernel execution and writeback occupy a larger fraction of normalized runtime. This does not indicate that those stages become slower; it indicates that the optimized communication-path stages have been reduced.

Overall, the application-level evaluation preserves the qualitative conclusions of the synthetic benchmark, but the magnitude depends on the real argument structure of each application. The decisive factors are the number of transfers removed, how evenly bytes are distributed across arguments, and how much of the measured interval is occupied by the optimized communication path.

4.6.4 Interpretation

The PolyBench evaluation provides three main conclusions.

First, the synthetic benchmark identifies the favorable regime for Argument Fusion, but overestimates application-level benefit because it isolates communication-dominated stages. Real applications dilute this effect through computation, writeback, and non-uniform operands.

Second, total transferred bytes are not sufficient to predict the benefit of fusion. Applications with similar aggregate communication volumes can respond differently depending on how that volume is distributed across arguments.

Third, manual fusion is effective but intrusive. The transformation is practical for controlled experiments and simple kernels, but it becomes increasingly intrusive as application data structures, map lifetimes, and ownership relationships become more complex.

The application-level results therefore reinforce the thesis-level motivation for runtime-level automation. If communication fragmentation is a runtime manifestation of independently handled arguments or mappings, then part of the optimization opportunity should be recoverable inside the runtime without requiring each application to be manually rewritten around fused data layouts.

4.7 Discussion

The experimental section addressed RQ1 through its operational objectives: when Argument Fusion improves performance, how much of the improvement comes from transfer aggregation rather than allocator effects, and how fusion interacts with pinned memory and asynchronous transfer submission. This discussion consolidates the main conclusions from the synthetic and application-level results.

4.7.1 Conditions for Effective Argument Fusion

Argument Fusion is most effective when the baseline execution contains several independently handled communication objects whose individual sizes are small or medium relative to the efficient bandwidth regime of the interconnect. In this regime, each separate argument incurs allocation, transfer submission, synchronization, and deallocation overheads that are large relative to the useful payload.

The favorable condition is therefore not simply large data movement, but fragmented data movement. A workload with several similarly sized arguments exposes more optimization headroom than a workload in which most bytes already belong to one large transfer. This explains the difference between the synthetic benchmark and the application-level benchmarks: the synthetic benchmark represents a favorable equal-size fragmentation case, whereas the PolyBench kernels often concentrate most transferred bytes in one two-dimensional matrix operand.

Argument count and byte distribution must therefore be considered together. A benchmark with more nominal arguments does not necessarily expose more fusion opportunity if most of the communication volume is already concentrated in one large operand. Conversely, a smaller number of regularly sized operands can still expose a useful fusion opportunity when those operands occupy a significant fraction of the measured communication path.

At the application level, the effect is additionally bounded by the fraction of execution time occupied by the optimized stages. Argument Fusion directly reduces allocation, host-to-device transfer, and deallocation activity for the fused inputs. If kernel execution, device-to-host writeback, or other non-optimized runtime stages dominate the measured interval, the end-to-end

speedup is necessarily smaller than the isolated communication-path speedup.

4.7.2 Transfer Aggregation versus Allocator Consolidation

The component-level results show that Argument Fusion should not be interpreted only as a host-to-device bandwidth optimization. It changes both the number of transfer operations and the number of memory-management operations issued by the runtime.

The transfer-side benefit is strongest for low data sizes. In this region, many small transfers repeatedly pay startup, submission, and synchronization overheads, and the interconnect is not yet operating near its bandwidth-efficient regime. Fusing the arguments into a larger transfer improves the amortization of these costs and reduces active-copy fragmentation.

As transfer sizes increase, the host-to-device transfer benefit decreases because individual transfers become large enough to approach efficient bandwidth utilization. At this point, reducing transfer count yields progressively smaller transfer-side gains. This explains why the host-to-device transfer speedup curves approach saturation or unity in large-transfer regions.

The allocation and deallocation behavior is complementary. Unlike the transfer-side speedup, allocator-side gains can remain significant or even become more visible at larger data sizes. The allocation results and Nsight Systems traces indicate that the visible cost of `cudaMalloc` is affected by allocation size, invocation count, and allocator state. Replacing several allocations with one larger allocation can therefore avoid repeated allocator invocations and can move the fused implementation into a different allocator regime.

The same principle applies to deallocation. Replacing several `cudaFree` calls with a single deallocation reduces deallocation multiplicity and may reduce allocator-side synchronization overhead. Consequently, the overall communication-path speedup is the combined result of transfer aggregation, allocation consolidation, and deallocation consolidation. This is why Argument Fusion can continue to provide communication-path gains even after the isolated host-to-device transfer benefit has diminished.

4.7.3 Interaction with Pinned Memory and Asynchronous Transfers

The experiments also show that Argument Fusion is not an alternative to pinned memory or asynchronous transfer submission. These mechanisms act on different parts of the communication pipeline and can therefore be combined.

Pinned memory primarily changes the host-memory transfer path. By avoiding the pageable-memory staging path for the transferred source buffer, pinned memory improves the efficiency of individual host-to-device transfers, especially once the payload is large enough to amortize pinned-memory overhead.

Asynchronous transfer submission primarily targets blocking-induced device inactivity. By allowing transfer operations to be submitted without immediate host-side blocking, asynchronous execution can reduce gaps between device-side copy operations and improve transfer scheduling.

Its benefit is most visible when the baseline contains several transfer operations whose submission and synchronization overheads are significant.

Argument Fusion targets communication granularity. It reduces the number of allocation, transfer, and deallocation operations by aggregating several logical arguments into one physical communication path. It therefore addresses a different source of overhead from pinned memory and asynchronous transfer submission.

Fusion+ demonstrates the complementarity of these mechanisms. In the synthetic communication-path evaluation, combining fusion, pinned memory, and asynchronous transfer submission produces the strongest overall behavior because it reduces transfer fragmentation, avoids the pageable-memory transfer path, and reduces blocking-induced inactivity. In the application-level benchmarks, the same combination remains beneficial, but the magnitude is benchmark- and size-dependent because computation, writeback, allocator effects, and argument-size distribution also affect the measured interval.

The key conclusion is therefore not that Fusion+ uniformly dominates every configuration in every regime. Rather, the results show that pinned memory and asynchronous transfers remain applicable after fusion. They optimize different mechanisms and should be viewed as complementary communication optimizations rather than substitutes for argument aggregation.

4.7.4 Limits of Manual Fusion and Runtime Implications

Manual Argument Fusion is useful as an experimental transformation because it exposes the performance effect of communication granularity under controlled conditions. However, it is limited as a general application optimization strategy.

First, manual fusion requires explicit restructuring of host and device memory layouts. Logically independent arrays must be packed into unified buffers, and the original logical pointers must be reconstructed through offset calculations. This increases host-side code complexity and couples application data layout to the communication strategy.

Second, the transformation is application-specific. The PolyBench results show that even within a small set of dense linear-algebra kernels, the usefulness of fusion depends on argument count, byte distribution, and benchmark-specific execution structure. A fused layout that is effective for one kernel may not be suitable for another kernel with different access patterns, writeback requirements, or data lifetimes.

Third, manual fusion only affects the parts of the offload path that are explicitly reorganized. It does not optimize kernel execution, device-to-host writeback, application-level synchronization, or cases where the communication path is already bandwidth-efficient.

These limitations motivate runtime-level automation. A substantial part of the measured benefit comes from operations controlled by the offloading runtime: allocation calls, transfer submissions, synchronization points, and deallocation calls. A runtime mechanism can therefore attempt to recover part of the same benefit without requiring the programmer to construct fused

layouts manually.

This motivates the transition to In-Copy Fusion in the next chapter. Instead of requiring application-visible fused buffers, ICF investigates whether the OpenMP runtime can aggregate eligible input mappings dynamically, preserve the logical mapping abstraction, and use staged transfer execution to avoid replacing transfer fragmentation with host-side serialization.

Table 4.5 summarizes the practical implications of the experimental results.

Table 4.5: Practical guidelines derived from the Argument Fusion evaluation.

Optimization	**Most favorable conditions**	**Main limitation**
Argument Fusion	Multiple independently transferred arguments, particularly when a substantial fraction of the payload is distributed across small or medium-sized operands.	Benefit decreases when transfers are already individually bandwidth-efficient or when non-optimized execution stages dominate total time.
Pinned memory	Transfer sizes for which pageable-memory staging contributes materially to transfer time.	Provides limited benefit for very small payloads and does not reduce transfer fragmentation.
Asynchronous transfers	Several independently submitted transfers with visible blocking or inter-transfer idle time.	Does not change the number or granularity of transfer operations.
Fusion+	Workloads in which transfer fragmentation, pageable-memory handling, and transfer submission overhead are simultaneously significant.	End-to-end gain remains bounded by computation, writeback, and other unaffected runtime stages.

4.8 Summary

This chapter evaluated Argument Fusion as a controlled transformation for studying communication granularity in CUDA-style offloading.

The synthetic evaluation showed that host-to-device transfer fusion is most effective at low and medium data sizes, where fragmented transfers repeatedly pay startup and submission overheads. As transfer sizes increase and individual transfers approach the bandwidth-efficient regime of the interconnect, the isolated transfer-side benefit decreases.

The component-level evaluation showed that the total communication-path improvement is not caused by transfer aggregation alone. Allocation and deallocation consolidation contribute substantially, especially at larger data sizes where allocator behavior and deallocation overhead remain significant. Argument Fusion therefore combines two complementary effects: it reduces transfer fragmentation, and it reduces allocation/deallocation multiplicity.

The evaluation of pinned memory and asynchronous transfer submission showed that these optimizations remain applicable after fusion. Pinned memory, asynchronous submission, and Argument Fusion target different communication-path mechanisms, and their combination in Fusion+ provides the strongest synthetic communication-path behavior.

The PolyBench evaluation confirmed that the same effects remain visible in complete application-level workloads, but with smaller and more benchmark-dependent end-to-end gains. The observed benefit depends on argument count, byte distribution across arguments, communication-to-computation ratio, writeback behavior, and the fraction of execution time occupied by the optimized communication path.

Overall, the chapter shows that transfer organization is a critical performance factor in heterogeneous runtimes, while also exposing the limitation of manual Argument Fusion: it can reduce communication overhead, but it requires application-specific data-layout restructuring. This limitation motivates the runtime-level mechanism developed in the next chapter.

With respect to the thesis research questions, this chapter provides the experimental basis for RQ1. It shows that transfer segmentation affects both idle time and active-copy efficiency, that Argument Fusion is most effective when communication is fragmented across multiple small or medium-sized arguments, that allocation and deallocation consolidation are significant contributors to the total communication-path speedup, and that fusion remains complementary to pinned memory and asynchronous transfer submission. These conclusions provide the baseline characterization needed for the runtime-level ICF design evaluated in Chapter 5.

5. In-Copy Fusion: Runtime Argument Fusion for OpenMP GPU Offloading

5.1 From Manual Fusion to Runtime Fusion

The previous chapter showed that aggregating fragmented argument transfers can reduce host–device communication overhead. It also exposed the central limitation of manual Argument Fusion: the application, compiler, or code generator must explicitly construct fused host layouts, assign offsets, and preserve the correspondence between logical program objects and physical memory regions. This requirement makes manual fusion intrusive and difficult to generalize across applications whose mappings are dynamic, irregular, or not statically visible.

This chapter moves the fusion decision into the offloading runtime layer. The proposed mechanism, In-Copy Fusion (ICF), is implemented inside LLVM's `libomptarget` runtime and targets the copy-in path generated by OpenMP mappings. OpenMP is the implementation vehicle because its mapping abstraction exposes a concrete runtime point at which logical data-mapping intent has already been collected, but backend allocation and transfer operations have not yet been fully materialized. Instead of requiring application-visible fused buffers, ICF observes eligible input mappings during runtime mapping processing, constructs a fused physical communication path, and rebinds the original logical mappings to slices of a fused device allocation.

Runtime-level fusion introduces two constraints that do not appear in manual fusion. First, the original host objects may be unrelated pageable allocations, so the runtime cannot treat them as one contiguous source buffer. Second, explicitly packing those objects into a staging buffer can create a new serialization bottleneck if all host-side packing is completed before device transfer begins. ICF addresses these constraints through runtime-managed pinned staging buffers and a chunked staged transfer pipeline.

5.1.1 Research-Question Scope

This chapter addresses the runtime-fusion research questions introduced in Chapter 1. The design and implementation of ICF address RQ2, which asks whether communication fusion can be moved from application-level restructuring into the offloading runtime while preserving the programming-model semantics. The staged transfer design and experimental evaluation address RQ3, which concerns how runtime-level fusion can be organized effectively and under which workloads it improves performance.

5.1.2 Contributions

This chapter makes the following contributions:

Code 5.1: Simplified OpenMP target data mapping example.

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

1. It identifies the `targetDataBegin()` path in LLVM OpenMP as a runtime point where multiple mappings can be inspected before the baseline allocation and transfer sequence is materialized.

2. It presents In-Copy Fusion, a runtime-level fusion mechanism that aggregates eligible OpenMP input mappings without requiring source-level data-layout restructuring.

3. It introduces a chunked staged transfer pipeline that uses runtime-managed pinned staging buffers to aggregate disjoint pageable mappings while avoiding monolithic host-side packing.

4. It evaluates ICF across synthetic, Kessler-style layout-conflict, and application-level OpenMP workloads, characterizing when runtime fusion improves copy-in and end-to-end execution, when its benefits saturate or degrade, and how it compares against a static operand-layout fusion baseline.

5.1.3 Chapter Organization

The chapter first identifies the runtime fusion opportunity in LLVM OpenMP, then develops the staging and chunking design required to make runtime fusion practical. It then describes the implementation in `libomptarget`, evaluates ICF on synthetic workloads, a Kessler-style layout-conflict case, and application-level workloads, and concludes with a discussion of applicability, limitations, and portability.

5.2 The Runtime Fusion Opportunity in LLVM OpenMP

5.2.1 OpenMP Target Mapping Syntax

OpenMP offloading exposes accelerator execution through directive constructs rather than through explicit device API calls in application code. A simplified example is shown in Listing 5.1.

The `target` construct creates a device execution region. The surrounding `target data` construct creates a device data environment. The `map(to:)` clause requests copy-in from host memory to device memory, `map(from:)` requests copy-out from device memory to host memory,

and `map(tofrom:)` combines both directions. From the programmer's perspective, these clauses describe logical data-mapping intent. The allocation of device storage, host-to-device transfer, device-to-host transfer, synchronization, and mapping-table bookkeeping are delegated to the OpenMP offloading runtime.

In compiler terminology, *lowering* is the translation of source-level constructs such as `#pragma omp target data` into lower-level compiler and runtime operations. In LLVM OpenMP offloading, lowering converts OpenMP directives and map clauses into runtime calls that pass mapping metadata to `libomptarget`. This metadata includes host addresses, mapped sizes, and map-type flags. ICF operates after this lowering step, inside the runtime path where the logical mapping list is already visible but before the backend has fully materialized the corresponding allocation and transfer operations.

OpenMP target offloading therefore separates logical mapping specification from physical communication execution. The programmer describes mappings and target data environments through directives such as `target`, `target data`, `target enter data`, and `target exit data`, while the runtime materializes the corresponding device allocation, transfer, synchronization, bookkeeping, and cleanup behavior [45, 30].

This distinction is central to ICF. Runtime-level fusion is possible only if the runtime observes the mapping set before the backend has committed to independent allocation and transfer operations. LLVM's OpenMP offloading implementation provides such a point in the data-mapping path.

After lowering, OpenMP target constructs enter the `libomptarget` runtime through calls that pass mapping metadata, including host addresses, mapped sizes, and map-type flags. Although ordinary `target` regions and explicit data-management directives enter through different interface paths, both reach the runtime mapping infrastructure before device execution proceeds [30].

Figure 5.1 shows the relevant communication path after OpenMP source directives have been lowered to runtime calls. At the source level, constructs such as `target data`, `target enter data`, and `target` provide the mapping clauses shown in Listing 5.1. After lowering, ordinary target regions enter through the target-kernel path, while explicit data-management directives enter through the target-data path. Both paths converge on the mapping-processing stage represented by `targetDataBegin()`.

The optimization opportunity highlighted by the figure is that `targetDataBegin()` has collective visibility of the mapping list, while the baseline path below this point still materializes eligible mappings through independent allocation and transfer operations. ICF is inserted at this boundary to exploit that collective visibility before those independent backend operations are issued.

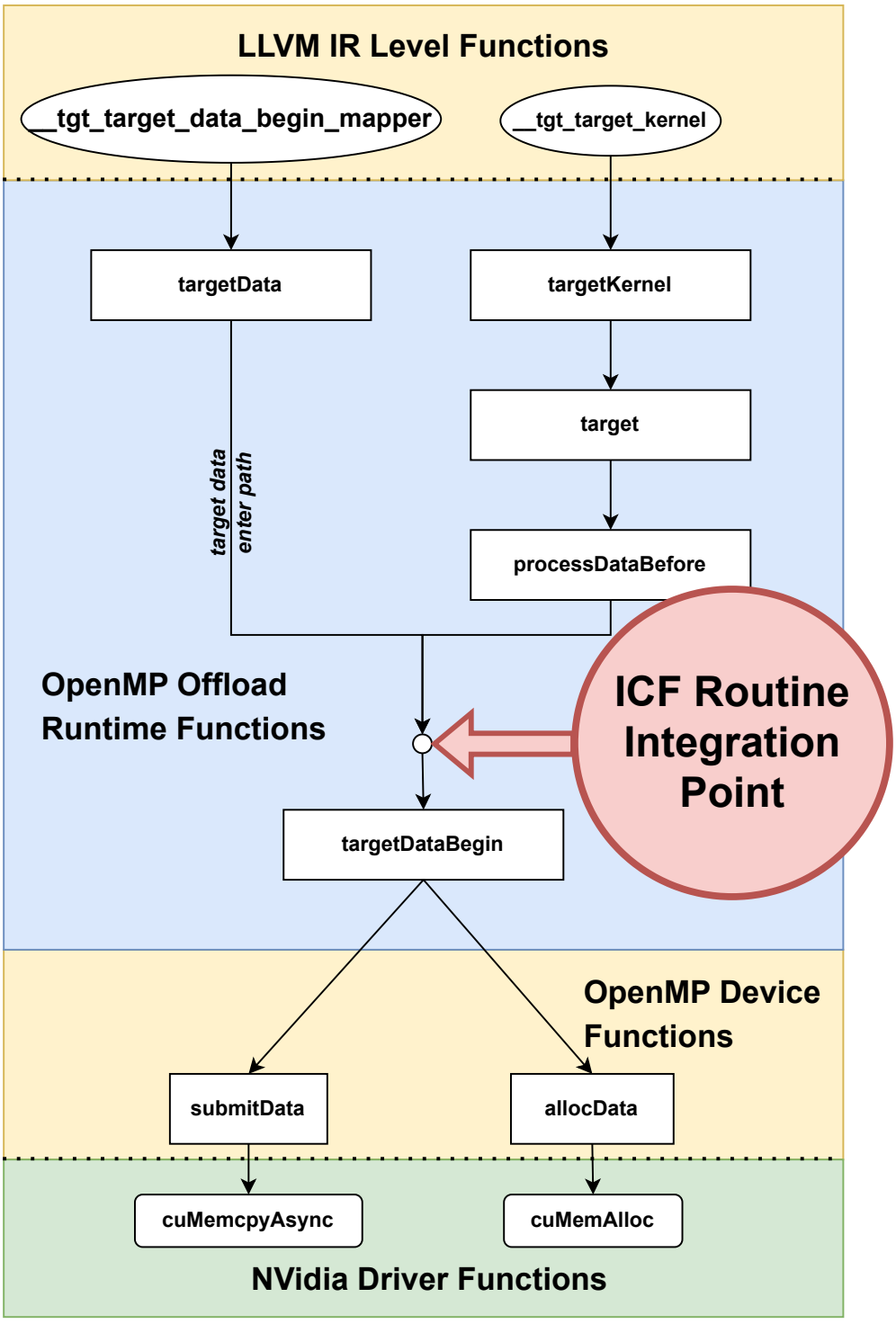


Figure 5.1: LLVM OpenMP offloading communication flow after source-level OpenMP directives and map clauses have been lowered to runtime calls. The ICF integration point is inside the runtime mapping path, where `libomptarget` has access to the mapping list before backend allocation and host-to-device transfer operations are fully materialized.

At `targetDataBegin()`, the runtime has received the mapping list for the region. For each mapping, it can inspect the host pointer, mapped size, and map semantics. In the baseline implementation, this collective visibility is not used to change communication granularity. Mappings are processed as independent entries: the runtime checks whether each mapping is already present, allocates device storage when required, submits the corresponding host-to-device transfer, and updates runtime bookkeeping.

Algorithm 1 summarizes this baseline structure.

Algorithm 1 Baseline communication flow inside `targetDataBegin()`.

```
for all m ∈ Mappings do
    e ← LookupMappingEntry(m)
    if RequiresAllocation(m, e) then
        d ← AllocateDeviceMemory(m)
        RegisterMapping(m, d)
    end if
    if RequiresHostToDeviceTransfer(m, e) then
        SubmitHostToDeviceTransfer(m, e)
    end if
    UpdateRuntimeBookkeeping(m, e)
end for
```

This organization is semantically robust because each mapping is handled according to its own map type and present-table state. Its cost is that communication is physically fragmented. When a region contains several input mappings that are not already present on the device, the runtime may issue several independent allocation and transfer operations even though the aggregate input volume is known.

5.3 Why Runtime Fusion Is Difficult

Runtime-level fusion differs from manual fusion because the runtime must preserve the memory layout and mapping semantics already exposed to the application. The runtime cannot require the application to allocate a single fused host buffer, nor can it change the programmer-visible identity of mapped variables.

5.3.1 Disjoint Host Allocations and OpenMP Mapping Semantics

Manual Argument Fusion constructs contiguity before offloading. The application or transformation pass allocates one large host buffer, assigns logical arguments to offsets inside that buffer, and transfers the resulting contiguous region to the device.

The OpenMP runtime does not have the same freedom. When `targetDataBegin()` receives the mapping list, the participating host objects may originate from unrelated allocations, library-owned buffers, or dynamically generated regions of the host address space. They are logically associated with the same offload operation, but they are not necessarily contiguous in memory.

This creates a physical obstacle. The backend transfer interface requires a contiguous source region, while a set of independent pageable mappings does not provide one. The runtime can observe that the mappings are temporally co-located, but it cannot reinterpret their original host addresses as one contiguous transfer source without changing application-visible memory layout.

There is also a semantic obstacle. OpenMP mappings carry map-type information, present-table state, ownership rules, and lifetime behavior. Some mappings may already be present on the device, some may require copy-in, some may require writeback later, and others may have semantics that make fusion unsafe. A runtime-level mechanism must therefore preserve the logical identity of each mapping even if it changes the physical organization of allocation and transfer.

Consequently, runtime fusion requires an intermediate communication representation: one that can assemble disjoint pageable host mappings into a contiguous transfer source while preserving their original OpenMP mapping identities.

5.3.2 Pageable-Memory Staging as the Enabling Layer

ICF is based on the observation that pageable host-to-device communication already involves a staging-like preparation path before data become available to the device DMA engine. In the baseline offloading path, this preparation is managed below the OpenMP runtime and is applied independently to each mapped object. Consequently, the runtime loses the opportunity to aggregate several logical mappings before they enter the physical transfer path.

ICF exposes an analogous staging point at the OpenMP runtime level. Instead of adding an unrelated communication phase, the runtime recreates the relevant pageable-transfer staging role explicitly using runtime-managed pinned buffers. Eligible source mappings are copied from their original pageable host locations into these pinned staging buffers, which provide contiguous transfer sources for fused host-to-device submission. Fusion therefore occurs at the staging boundary: the application still sees separate host objects and ordinary OpenMP mappings, while the runtime constructs a temporary fused communication representation for transfer submission.

This design does not depend on accessing CUDA's internal pageable-transfer buffers or modifying the CUDA driver path. It recreates the required staging functionality inside `libomptarget`, where the complete OpenMP mapping set is visible and where fusion decisions can be made before backend transfers are issued. The staged payload is then transferred to one fused device allocation, and the original logical mappings are exposed again through present-table entries pointing to slices of that allocation.

5.3.3 Observed Pageable Transfer Behavior

The previous subsection identifies staging as the runtime-level point where fusion can be introduced. The next question is whether this staging should be performed monolithically or as a pipeline. Although CUDA's internal pageable-memory transfer implementation is not fully specified, NVIDIA documents that memory-copy synchronization behavior depends on transfer direction and on whether host memory is pageable or pinned [42]. Pageable transfers therefore provide a plausible setting in which host-side preparation and device-side transfer behavior may differ. To characterize this behavior, pageable host allocations were transferred to the device using asynchronous host-to-device CUDA transfers and profiled with NVIDIA Nsight Systems [44].

Figure 5.2 shows representative traces for increasing transfer sizes. For smaller transfers, visible GPU copy-engine activity begins only after host-side API activity has completed. For larger transfers, GPU DMA activity increasingly overlaps with the host-side API interval. This suggests that pageable communication may internally expose overlap between preparation and device-side transfer for sufficiently large payloads.

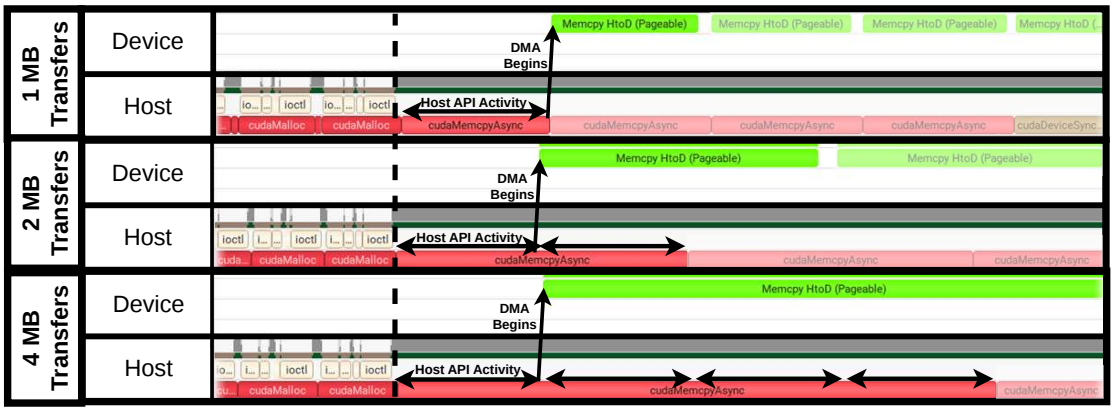


Figure 5.2: Nsight Systems traces of pageable asynchronous CUDA host-to-device transfers for increasing transfer sizes. The traces show behavior consistent with staged pageable communication: for smaller transfers, visible GPU copy-engine activity begins only after host-side API activity has completed, while larger transfers exhibit increasing overlap between host-side activity and device-side DMA execution.

The implication for ICF is precise. Runtime-managed staging solves the contiguity problem, but it can introduce a pipeline problem. A naive implementation that packs the entire fused payload into one staging buffer before submitting the device transfer may reduce transfer count while losing useful overlap between host-side preparation and device-side DMA.

Runtime fusion must therefore use staging without turning staging into a monolithic serialization phase.

5.3.4 Baseline Interleaving and Incidental Allocation–Transfer Overlap

The baseline OpenMP offloading path should not be interpreted as a fully naive implementation in which all device allocations are completed before any data transfer begins. In LLVM's OpenMP offloading runtime, data mappings are processed through the `targetDataBegin()` path on a per-entry basis. As a result, for a region with multiple mapped inputs, the runtime and device plugin may issue an allocation for one mapping, submit the corresponding host-to-device transfer, and then continue processing subsequent mappings. The resulting timeline is therefore an interleaved sequence of allocation and transfer operations rather than a strictly phase-separated sequence.

Figure 5.3 illustrates this behavior conceptually. In the baseline OpenMP case (A), each mapped argument is still handled independently, preserving the fragmented communication pattern. However, because host-to-device transfers are submitted asynchronously at the CUDA-driver level, a transfer already visible on the device copy engine may overlap with later host-side allocation work. This overlap is especially relevant when the later allocation calls are cheap. As discussed in Section 4.5.2 and more specifically depicted in Figure 4.7, the measured allocation behavior is consistent with a stateful allocator in which an expensive operation may acquire or activate a coarse allocation region, while subsequent allocation requests can be served more cheaply from already available allocator state. In such cases, a later allocation may correspond primarily to inexpensive allocator bookkeeping or pointer assignment, and its cost can be largely hidden behind an in-flight transfer.

This observation is important for interpreting In-Copy Fusion. ICF removes the per-argument allocation and transfer structure by first allocating one fused device region and then transferring the fused payload through the chunked staging pipeline described in Section 5.4.2. As shown in Figure 5.3B, this eliminates the baseline's allocation–transfer interleaving across individual arguments. The benefit is a reduction in allocator multiplicity and transfer fragmentation; the cost is that the schedule of host-side and device-side work changes. Therefore, ICF is not compared against an entirely unoptimized baseline, but against a baseline that may already hide part of its per-argument allocation work behind in-flight DMA activity.

This distinction explains why fusion can become less effective, or even slower, in adversarial cases. When the baseline's interleaved allocation–transfer schedule happens to hide cheap allocation calls behind ongoing DMA activity, the visible cost of the baseline is lower than a simple sum of independent allocation and transfer costs would suggest.

5.4 In-Copy Fusion Design

ICF implements the constraints identified above through a staged runtime design. Eligible mappings are assigned slices inside one fused device allocation, their source bytes are packed into runtime-managed pinned staging buffers, and staged chunks are transferred asynchronously

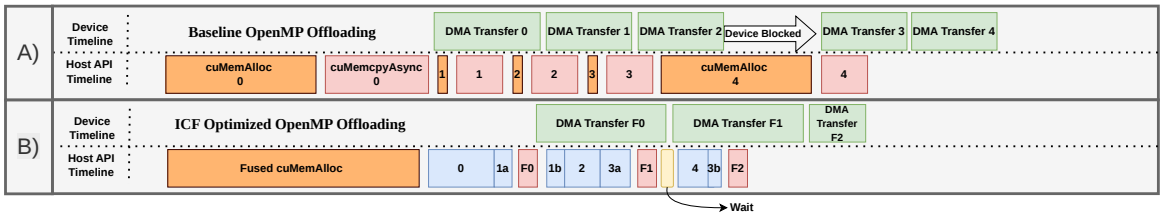


Figure 5.3: Conceptual comparison between the baseline OpenMP offloading path and the ICF optimized path. In the baseline case (A), `targetDataBegin()` handles mappings independently, producing an interleaved sequence of device allocations and asynchronous host-to-device transfers. This preserves transfer fragmentation, but may hide cheap allocation calls behind in-flight DMA activity. In the ICF case (B), eligible mappings are consolidated into one fused device allocation and transferred through the runtime-managed chunked staging pipeline. This removes per-argument transfer fragmentation and allocator multiplicity, but also changes the overlap structure of the offload.

to the corresponding device offsets.

The design has three core elements: a fused device layout, a pinned staging layer, and a chunked transfer pipeline.

5.4.1 From Monolithic Staging to Chunked Staged Fusion

A naive runtime-fusion design would identify eligible mappings, allocate one fused device buffer, pack all source data into one pinned staging buffer, and then submit one large host-to-device transfer. This design reduces transfer and allocation multiplicity, but it introduces a new ordering constraint: all host-side packing must complete before device-side transfer can begin.

Figure 5.4 illustrates this problem. The baseline pageable path can expose partial overlap between host-side preparation and device-side DMA activity. Naive monolithic fusion replaces this with a full host-side packing phase followed by a transfer phase. The resulting transfer is larger, but the pipeline can be worse.

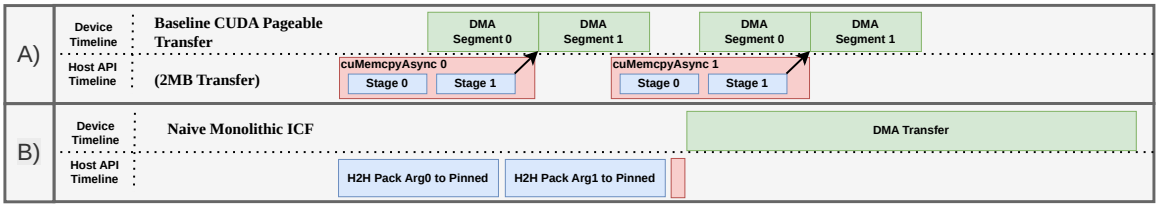


Figure 5.4: Conceptual comparison between baseline pageable CUDA communication and naive monolithic runtime fusion. Baseline pageable communication may overlap host-side preparation and device-side DMA execution, whereas naive monolithic fusion serializes explicit host-side packing before DMA submission.

ICF therefore uses chunked staged fusion. The fused payload is divided into chunks; each chunk is packed into reusable pinned staging memory and submitted asynchronously to the corresponding offset of the fused device allocation. While one chunk is in flight on the DMA

path, the runtime can prepare a later chunk, subject to staging-buffer reuse constraints. This preserves the aggregation benefit of fusion while recovering overlap between host-side staging and device-side communication.

5.4.2 Chunked Transfer Pipeline

Let K denote the total number of bytes selected for fusion and let S denote the staged chunk size. The fused communication is partitioned into

$$N = \left\lceil \frac{K}{S} \right\rceil \tag{5.1}$$

chunks. For each chunk, the runtime performs host-side packing from one or more original pageable mappings into pinned staging memory, followed by host-to-device transfer from the staging region to the corresponding offset in the fused device allocation.

The staged execution can be viewed as a producer–consumer pipeline. The host thread constructs staged chunks from disjoint OpenMP mappings. The GPU copy engine transfers completed chunks to device memory. If the chunk size is chosen appropriately, the two actors can operate with partial overlap.

The chunk size controls the tradeoff between transfer granularity and pipeline depth. Very small chunks increase the number of transfer submissions and synchronization events, reintroducing fragmentation overhead. Very large chunks reduce submission overhead, but reduce pipeline depth and approach monolithic staging.

A simplified model captures this tradeoff. Let $T_H(S)$ be the host-side staging time for a chunk of size S, let $T_D(S)$ be the device transfer time for that chunk, and let $\alpha(S)$ represent per-chunk runtime overhead, including submission and synchronization costs. Under a pipelined execution model, the total communication time can be approximated as

$$T_{\text{pipeline}}(K, S) \approx T_H(S) + T_D(S) + (N - 1)\max(T_H(S), T_D(S)) + N\alpha(S). \tag{5.2}$$

This model is not intended to predict exact runtime behavior. Its purpose is to identify the central design constraint: useful chunking must be large enough to amortize transfer overhead while small enough to preserve staged overlap.

5.4.3 Directive-Dependent Synchronization Effects

ICF is implemented inside the `targetDataBegin()` path, which is reached by both ordinary `target` regions and explicit data-management directives such as `target enter data`. These execution forms share mapping infrastructure, but their visible synchronization behavior differs.

In explicit data-management directives, the communication phase is exposed more directly. A `target enter data` directive completes only after the runtime reaches the required mapping

synchronization point. If staged transfers remain in flight at that point, the measured cost includes the corresponding wait.

In ordinary `target` regions, copy-in is followed by additional runtime activity and kernel execution. Portions of the staged transfer pipeline may be hidden by later target-region preparation or may be measured together with kernel execution and cleanup. As a result, the same ICF transfer path can produce different visible speedups depending on directive form.

This distinction matters for policy selection and evaluation. Explicit data-management experiments expose the optimized copy-in path most directly, while ordinary target experiments show how much of the communication benefit survives in the complete offload interval.

5.4.4 Calibration-Driven Chunk Selection

The analytical model is insufficient for selecting a universally optimal chunk size. The terms $T_H(S)$, $T_D(S)$, and $\alpha(S)$ depend on CPU performance, host memory bandwidth, interconnect generation, GPU copy-engine behavior, backend submission cost, OpenMP runtime synchronization, and the composition of the mapped arguments.

ICF therefore uses a calibration-driven policy. Offline profiling sweeps the total fused communication volume K and candidate chunk sizes S. For each pair (K, S), the benchmark measures the effective copy-in bandwidth of the ICF path. The selected policy maps ranges of total fused communication volume to empirically chosen chunk sizes.

At runtime, the implementation computes K during the fusion prescan and selects the corresponding chunk size from the calibrated policy table. The policy is platform-specific and fixed for the evaluated implementation. Online adaptation is left as future work.

5.5 Runtime Implementation in `libomptarget`

ICF is implemented in LLVM OpenMP offloading version 21.0.0 using the plugins-nextgen CUDA backend. The implementation extends the runtime communication path responsible for mapping preparation, device allocation, host-to-device transfer submission, present-table updates, and cleanup.

Table 5.1 summarizes the implementation mechanisms. The following subsections describe each mechanism at implementation level.

Table 5.1: ICF implementation mechanisms and runtime roles.

Mechanism	Runtime role
Fusion eligibility analysis	Selects mappings whose OpenMP semantics can be preserved under fused physical communication.
Fused device allocation	Creates one contiguous device region that stores selected logical mappings as slices.
Pinned staging buffers	Provide contiguous host regions for constructing staged chunks from disjoint pageable mappings.
Chunk construction	Copies byte ranges from original host mappings into the correct offsets inside staged chunks.
Asynchronous H2D submission	Transfers staged chunks to the fused device allocation while allowing staging and DMA to overlap.
Present-table rebinding	Makes each fused slice appear to the runtime as an ordinary independent present mapping.
Fallback and cleanup	Preserve correctness for unsupported cases and release shared fused allocations at the proper mapping lifetime.

5.5.1 Integration Point and High-Level Flow

ICF is implemented as a preprocessing phase inside `targetDataBegin()`. At this point, the mapping list is available, while baseline allocation and transfer processing has not yet completed.

Algorithm 2 summarizes the runtime flow. ICF selects eligible mappings, computes the aggregate payload size, chooses a calibrated chunk size, allocates one fused device buffer, transfers staged chunks to the corresponding device offsets, and rebinds the original mappings as present slices. Mappings not selected for fusion continue through the baseline path unchanged.

Algorithm 2 High-level ICF communication flow inside `targetDataBegin()`.

$B \leftarrow$ GetPinnedStagingBuffer()
$(F, K) \leftarrow$ SelectFusableMappings(ArgTypes, ArgSizes)
if $F = \emptyset$ **then**
    return ContinueBaselineRuntime
end if
$S \leftarrow$ SelectChunkSize(K)
$C \leftarrow \lceil K/S \rceil$
$D \leftarrow$ AllocateFusedDeviceBuffer(K)
for $c \leftarrow 0$ to $C - 1$ **do**
    ConstructStagedChunk(B, c, F)
    SubmitChunkTransfer(D, B, c)
end for
for all $i \in F$ **do**
    BindMappingToFusedSlice(Args$[i]$, D)
    ArgTypes$[i] \leftarrow$ ArgTypes$[i] \cup$ `OMP_TGT_MAPTYPE_PRESENT`
end for
return ContinueMappingProcessing(ArgTypes)

The algorithm is conservative: no baseline mapping state is modified until a valid fusion group and fused allocation have been established.

5.5.2 Fusion Eligibility Analysis

Eligibility analysis determines which mappings can be safely represented as slices of a shared fused allocation. The current implementation focuses on input mappings that require host-to-device communication at entry. Mappings with incompatible map types, existing present-table state, unsupported pointer semantics, invalid sizes, or insufficient fusion opportunity remain on the baseline path.

Algorithm 3 summarizes the prescan. The predicate IsFusableInputMapping represents the semantic and profitability checks used by the implementation.

Algorithm 3 ICF fusion eligibility analysis.

$F \leftarrow \emptyset$
$K \leftarrow 0$
for all $i \in$ MappingIndices **do**
    if IsFusableInputMapping(ArgTypes$[i]$, ArgSizes$[i]$) **then**
        $F \leftarrow F \cup \{i\}$
        $K \leftarrow K +$ AlignedSize(ArgSizes$[i]$)
    end if
end for
if $|F| < 2$ **then**
    return $(\emptyset, 0)$
end if
return (F, K)

The requirement $|F| \geq 2$ prevents ICF from replacing a single ordinary transfer with an equivalent staged transfer. The aligned-size accumulation ensures that each logical mapping receives a valid slice within the fused allocation.

Table 5.2 summarizes the practical eligibility policy.

Table 5.2: ICF fusion eligibility policy.

Mapping case	**Action**	**Reason**
Input mapping requiring H2D transfer	Fuse	Source bytes can be copied into the staged fused payload and rebound as a device slice.
Single eligible mapping	Fallback	Fusion would not reduce transfer count and would add staging overhead.
Already-present mapping	Fallback	Existing present-table state should not be rebound.
`map(from:)` without copy-in	Fallback for copy-in fusion	No H2D input payload exists at entry.
Overlapping or aliased mappings	Fallback	Independent rebinding could violate logical host-object relationships.
Pointer-containing structures	Fallback	Correct handling requires deeper pointer-attachment reasoning.
Zero-size or invalid-size mappings	Fallback	No useful payload is available for fusion.

5.5.3 Fused Device Allocation and Slice Layout

After the runtime selects a valid fusion set F, it allocates a single contiguous device buffer of size K. Each original mapping is assigned a slice inside this allocation.

Let D denote the base address of the fused device allocation. For each mapping $i \in F$, ICF computes an offset o_i and associates the logical mapping with the device slice

$$D_i = D + o_i. \tag{5.3}$$

The original application does not observe this fused layout directly. Kernel arguments are still resolved through OpenMP device-pointer translation, and each logical mapping is later rebound to its corresponding slice.

This design separates physical ownership from logical mapping identity. The fused allocation owns the memory, while each present-table entry describes one logical view into that memory.

5.5.4 Pinned Staging Buffer Management

A contiguous device allocation is not sufficient by itself; each transfer submission also requires a contiguous host source. Since the original OpenMP mappings may refer to disjoint pageable host allocations, ICF constructs each transfer source inside runtime-managed pinned staging memory.

Pinned staging buffers are allocated and retained by the runtime backend rather than allocated for each offload. This avoids making pinned allocation latency part of every fusion operation and allows staged chunks to reuse pre-allocated host storage.

When multiple staging regions are available, they are used in alternating fashion. CUDA events associated with each staging region prevent the runtime from overwriting staging memory whose previous contents are still participating in an active transfer. This enables host-side packing for one chunk to overlap with device-side transfer of a previously packed chunk, subject to backend synchronization constraints.

5.5.5 Chunk Construction and Host-Side Packing

ICF treats the fused payload as a logical contiguous byte interval of length K. Chunk construction maps each interval of this fused payload back to the original mappings that contribute bytes to that interval.

For chunk c, the runtime determines the fused byte interval

$$[cS, \min((c+1)S, K)) \tag{5.4}$$

This interval may overlap one mapping slice, multiple slices, or the boundary between adjacent slices. The runtime copies each overlapping subrange from the original host allocation into the corresponding offset within the staging buffer.

Algorithm 4 Chunk construction and host-side packing.

```
chunkBegin ← c · S
chunkEnd ← min((c + 1) · S, K)
for all i ∈ F do
    sliceBegin ← GetSliceOffset(i)
    sliceEnd ← sliceBegin + AlignedSize(ArgSizes[i])
    a ← max(chunkBegin, sliceBegin)
    b ← min(chunkEnd, sliceEnd)
    if a < b then
        src ← Args[i] + (a − sliceBegin)
        dst ← B + (a − chunkBegin)
        CopyHostRange(dst, src, b − a)
    end if
end for
```

Because chunk construction is independent of kernel code, the kernel continues to receive ordinary device pointers after present-table rebinding.

5.5.6 Asynchronous H2D Submission

Once a chunk has been packed into pinned staging memory, ICF submits an asynchronous host-to-device transfer from the staging buffer to the corresponding offset in the fused device allocation. For chunk c, the destination is

$$D + cS, \tag{5.5}$$

and the transferred byte count is

$$\min(S, K - cS). \tag{5.6}$$

Algorithm 5 Asynchronous submission of a staged chunk.

```
dst ← D + c · S
bytes ← min(S, K − c · S)
SubmitAsyncHostToDevice(dst, B, bytes)
RecordStagingEvent(B)
```

The event associated with the staging buffer records when the submitted transfer has consumed the buffer contents. Before the same staging region is reused, the runtime synchronizes with this event or otherwise verifies that reuse is safe.

5.5.7 Present-Table Integration and Device Pointer Rebinding

After the fused data has been transferred, the remaining OpenMP runtime must treat each original mapping as if it had been processed independently. ICF achieves this through present-table rebinding.

For each mapping $i \in F$, ICF creates or updates runtime mapping information so that the original host mapping is associated with the corresponding device slice $D_i = D + o_i$. The mapping is then marked as present before control returns to the baseline mapping path.

Algorithm 6 Present-table rebinding for fused mappings.

1: **for all** $i \in F$ **do**
2: $o_i \leftarrow \text{GetSliceOffset}(i)$
3: $P_i \leftarrow D + o_i$
4: $\text{CreatePresentEntry}(\text{Args}[i], P_i, \text{ArgSizes}[i])$
5: $\text{ArgTypes}[i] \leftarrow \text{ArgTypes}[i] \cup \texttt{OMP_TGT_MAPTYPE_PRESENT}$
6: **end for**

This step is central to semantic preservation. Each original OpenMP mapping retains its own host association, size, and present-table identity. The only difference is that the mapped device address points into a slice of a shared fused allocation rather than to a separately allocated device buffer.

5.5.8 Fallback Conditions and Runtime Safety

ICF is an opportunistic optimization. It must not force unsupported mappings through the fused path. Fallback is triggered when the runtime cannot form a profitable and semantically valid fusion group, when pinned staging storage is unavailable, when fused device allocation fails, or when the selected chunking policy would not reduce the baseline communication structure.

Mappings excluded from fusion are not modified. They continue through the standard LLVM OpenMP mapping path and are allocated, transferred, reused, or released according to baseline runtime rules.

5.5.9 Cleanup and Mapping Lifetime Management

Fused allocation changes the relationship between logical mappings and physical device memory. In the baseline path, an individual mapping is typically associated with an individual device allocation or existing present-table entry. Under ICF, multiple logical mappings may share one physical allocation. Cleanup must therefore distinguish logical mapping lifetime from physical allocation lifetime.

ICF preserves OpenMP exit semantics by treating each fused slice as an independent logical mapping while delaying deallocation of the underlying fused allocation until all participating

slices have ended. Operations such as `map(from:)` and `map(delete:)` continue to operate at the level of original OpenMP mappings.

Algorithm 7 High-level ICF cleanup flow inside `targetDataEnd()`.

```
for all e ∈ ExitMappings do
    if RequiresWriteback(e) then
        SubmitDeviceToHost(e)
    end if
    if IsIcfSlice(e) then
        r ← FindOwningAllocation(e)
        MarkSliceEnded(r, e)
        if AllocationCanBeReleased(r) then
            EraseSyntheticMappings(r)
            FreeFusedAllocation(r)
        end if
    else
        ContinueBaselineCleanup(e)
    end if
end for
```

Physical deallocation is therefore delayed until the final slice associated with the fused allocation has ended.

5.6 Experimental Methodology

This section describes the methodology used to evaluate ICF. The evaluation separates two questions. First, does ICF improve the copy-in path that it directly modifies? Second, do those improvements survive when computation, writeback, synchronization, irregular mappings, and dynamic runtime data structures are included in the measured interval?

5.6.1 Experimental Platforms

The evaluation uses two heterogeneous systems representative of different accelerator and interconnect regimes. System A is a local workstation with an NVIDIA Quadro P400. System C is an HPC node with an NVIDIA H100-class accelerator. Table 5.3 summarizes the relevant platform details.

Table 5.3: Experimental platforms used in the ICF evaluation.

Component	**System A**	**System C**
System	Local workstation	MareNostrum 5 ACC partition, BSC
CPU	Intel Core i7-8700K, 6 cores / 12 threads	2× Intel Xeon Platinum 8460Y+, 40 cores each
Host memory	32 GB DDR4	512 GB DDR5 per node
GPU	NVIDIA Quadro P400	NVIDIA H100 / Hopper GPU
GPU memory	2 GB	64 GB HBM per GPU
Host–GPU interconnect	PCIe Gen3	PCIe Gen5
Operating system	Ubuntu 20.04	Red Hat Enterprise Server
CUDA toolkit	12.2	12.2
LLVM/OpenMP runtime	LLVM 21.0.0, plugins-nextgen CUDA backend	LLVM 21.0.0, plugins-nextgen CUDA backend
Compilation target	`sm_61`	`sm_90`

5.6.2 Runtime Configurations

The evaluation compares the baseline LLVM OpenMP runtime against the modified runtime with ICF enabled. The baseline runtime processes OpenMP mappings independently through the standard per-mapping allocation and transfer path. The ICF runtime performs eligibility analysis, fused device allocation, chunked staged copy-in, present-table rebinding, and fused-allocation cleanup for selected mappings.

All evaluated applications use pageable host-memory allocations. This reflects ordinary OpenMP application behavior: the programmer does not explicitly allocate all mapped data in page-locked memory. ICF introduces runtime-managed pinned staging internally, but it does not require application-visible pinned buffers.

Communication is submitted through the default CUDA stream in the evaluated runtime configuration. CUDA events are used to protect staging-buffer reuse during asynchronous chunked transfer execution. ICF does not modify device-side computation, kernel code, source-level OpenMP directives, or application-visible host-memory layout.

5.6.3 OpenMP Execution Forms

The evaluation considers two OpenMP execution forms: explicit data-management directives and ordinary target regions, both defined by the OpenMP target offloading model [45].

In the explicit data-management form, copy-in is performed using `target enter data`,

computation is performed inside a later `target` region using already-present device data, and copy-out plus cleanup are performed using `target exit data`. This form exposes the communication path optimized by ICF most directly because copy-in and copy-out can be timed separately.

In the ordinary target form, data movement and device execution are coupled through a single `target` construct. Measurements using this form represent the common full offload path, where allocation, communication, runtime orchestration, kernel execution, synchronization, writeback, and deallocation all contribute to the observed time.

Unless stated otherwise, figures without explicit copy-in, copy-out, or total labels report end-to-end offload execution time.

5.6.4 Benchmark Design

The evaluation uses the term *kernel-style benchmark* for controlled offload patterns that isolate a specific runtime mapping behavior without attempting to reproduce a full application. These benchmarks still execute OpenMP `target` regions or `target data` regions, but their purpose is methodological: they expose how the runtime handles mapped inputs, allocation, copy-in, copy-out, and cleanup under a known argument structure.

In this chapter, the benchmark set is organized into three levels. The first level consists of synthetic kernel-style benchmarks, represented by EMPTY and XVADD. EMPTY isolates mapping and communication behavior with negligible computation, while XVADD adds lightweight device computation and output writeback. The second level is the Kessler-Interleaved stress case, which is still synthetic but is designed to compare runtime fusion against a static operand-layout strategy inspired by prior operand-transfer fusion work. The third level consists of application-level workloads, represented by MD-2D, BFS, and Service B+Tree, where application structure, irregularity, computation, and writeback all contribute to the measured interval.

Benchmark level	Role in the evaluation
Synthetic kernel-style benchmarks	EMPTY and XVADD isolate OpenMP mapping, copy-in, copy-out, and lightweight-kernel behavior under controlled buffer counts and total input sizes.
Static-layout stress case	Kessler-Interleaved isolates a multi-kernel operand-layout conflict and compares runtime ICF with a static operand-ordering strategy.
Application-level workloads	MD-2D, BFS, and Service B+Tree test whether the same runtime effect survives regular application structure, irregular graph mappings, and dynamic query-side mappings.

Figure 5.5: Benchmark taxonomy used in the ICF evaluation. Kernel-style benchmarks are controlled OpenMP offload patterns used to expose runtime mapping behavior before moving to application-level workloads.

The evaluation uses progressively less controlled benchmark categories. Table 5.4 summarizes the role of each workload.

Table 5.4: Benchmarks used throughout the ICF evaluation.

Benchmark	Category	Experimental purpose
EMPTY	Synthetic	Controlled upper-bound evaluation under regular fragmented mappings and negligible device computation.
XVADD	Synthetic	Evaluation of communication optimization under lightweight device computation and output write-back.
Kessler-Interleaved	Synthetic literature-positioning	Comparison of runtime ICF against a Kessler-style static operand layout under a multi-kernel pattern where one global operand order cannot make all kernel operand sets contiguous.
MD-2D	Application	Evaluation of ICF inside a regular multi-buffer application update kernel with prior offload.
BFS	Application	Evaluation under irregular mapping sizes and non-trivial graph-processing behavior.
Service B+Tree	Application	Evaluation under dynamic service-style query batches with runtime-supplied pageable host data.

The application benchmarks are further distinguished by the structure of their mapped data.

This is relevant because ICF operates on the runtime mapping list rather than on a source-level algorithmic description. The number of copy-in mappings, their relative sizes, and their regularity determine how much fragmented input communication can be consolidated. Table 5.5 summarizes this structure using semantic mapping groups rather than implementation variable names.

Table 5.5: Application-level mapping structure used in the ICF evaluation.

Benchmark	**Copy-in mappings**	**Input data structure**	**Measured output / cleanup**
MD-2D	4	Four equally sized 2D state vectors: position, velocity, acceleration, and force. Each has size $2N_p$ double-precision elements.	Two updated state vectors are copied back; the remaining input-only vectors are deleted.
BFS	6	Irregular graph input: node metadata, edge list, three frontier/state masks, and the cost array. Sizes scale with $\|V\|$ and $\|E\|$.	The cost array is copied back; graph structure and traversal-state arrays are deleted.
Service B+Tree	4	Dynamic query batch: query keys, current-node indices, offsets, and answer records. Sizes scale with the number of queries Q. Persistent tree data are transferred outside the measured interval.	Current-node indices, offsets, and answer records are copied back; query keys are deleted.

Table 5.5 shows the intended progression of the application evaluation. MD-2D is the regular case in a sequence of other offloads, with balanced input mappings that resemble the synthetic equal-size setting. BFS is the irregular case, where graph structure and traversal state produce mappings with different sizes and roles. Service B+Tree is the dynamic case, where the persistent tree is excluded from the measured interval and each measured invocation operates on runtime-supplied query-side mappings.

The synthetic benchmarks use the term buffer count to denote the number of logical mapped input buffers participating in the offload. The total input communication volume is partitioned across these buffers, allowing the experiments to vary both aggregate transfer size and mapping fragmentation.

EMPTY minimizes device-side computation and constructs highly fragmented OpenMP

mapping patterns. XVADD preserves fragmented input mappings while adding lightweight vector computation and output writeback.

The Kessler-Interleaved benchmark extends the synthetic evaluation from isolated single-offload fragmentation to a multi-kernel operand-layout conflict. It is inspired by operand-transfer fusion, where a static optimizer selects one global operand order to maximize adjacent transfer regions across a sequence of kernels. When the kernel–operand pattern does not admit a single order that makes every kernel's operands contiguous, the optimized static layout still leaves some kernels segmented. This benchmark isolates that case using simple vector kernels, allowing ICF to be compared against a Kessler-style static layout baseline.

The application-level evaluation includes MD-2D, BFS, and Service B+Tree. MD-2D is derived from the SOLLVE OpenMP benchmark suite, which is designed to evaluate OpenMP features, including offload behavior [50]. BFS and Service B+Tree are derived from Rodinia-style heterogeneous benchmark workloads, with B+Tree corresponding to the Rodinia B+Tree search workload [11, 16]. MD-2D evaluates a regular update phase with balanced input mappings. BFS evaluates irregular graph-processing communication, where mapped data differ in size and semantic role. Service B+Tree evaluates runtime-supplied query batches while excluding the persistent tree offload from the measured interval. The corresponding mapping structures are summarized in Table 5.5.

5.6.5 Timing Methodology

Each parameter configuration is executed for 101 iterations. Unless otherwise stated, the reported value is the median after warmup. Copy-in timing measures the interval directly affected by ICF: runtime mapping preparation, selected device allocation, host-side staging, and host-to-device transfer submission and completion. Total timing includes the corresponding full offload interval for the benchmark and execution form.

Speedup is computed as

$$\text{Speedup} = \frac{T_{\text{baseline}}}{T_{\text{ICF}}}. \tag{5.7}$$

Values above one indicate improvement over the baseline runtime.

5.6.6 Effective Bandwidth

For the chunking-policy calibration, effective copy-in bandwidth is computed as

$$B_{\text{eff}} = \frac{K}{T_{\text{copy-in}}}. \tag{5.8}$$

where K is the aggregate number of bytes selected for staged transfer and $T_{\text{copy-in}}$ is the measured copy-in interval. This metric is used only to calibrate and interpret the staged transfer path; application results are reported as speedup.

5.7 Experimental Results

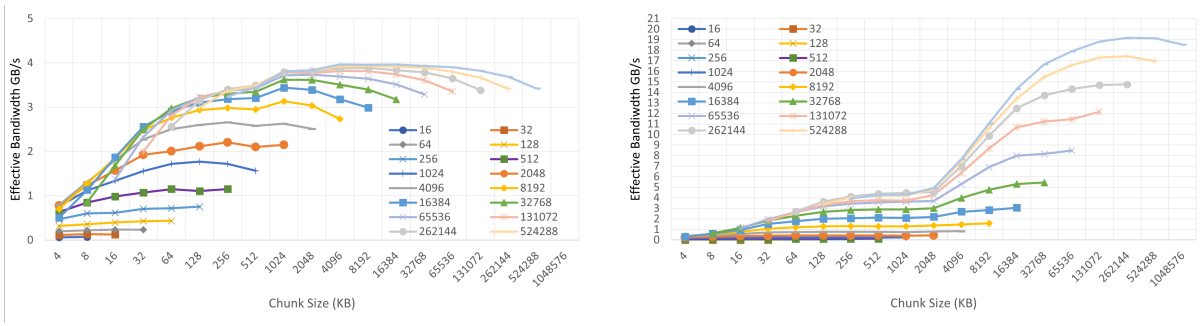
The results in this section provide the experimental evidence for RQ3: first by calibrating the staged transfer policy, and then by evaluating ICF across progressively less controlled workloads.

5.7.1 Chunking Policy Calibration

The first experiment evaluates the effect of staged chunk size on effective copy-in bandwidth. This experiment is necessary because ICF introduces a runtime decision that does not exist in the baseline: the fused communication volume must be partitioned into staged chunks.

If the selected chunk size is too small, ICF reintroduces per-chunk overhead and insufficient DMA payload size. If the selected chunk size is too large, execution approaches monolithic staging and may reduce overlap between host-side packing and device-side communication.

Figure 5.6 reports the measured effective copy-in bandwidth for different total communication volumes and chunk sizes.



(a) System A. (b) System C.

Figure 5.6: Effective copy-in bandwidth obtained by ICF under different staged chunk sizes and total fused communication volumes.

The results show that chunk size has a first-order impact on ICF performance. On both systems, very small chunks produce low effective bandwidth because execution is dominated by per-chunk overheads, repeated runtime submission, synchronization, and insufficient DMA payload size. As chunk size increases, bandwidth rises sharply and reaches a high-throughput region.

The comparison between systems shows that the useful chunking region is platform-dependent. System A stabilizes in a smaller chunk-size range, while System C benefits from larger chunks and reaches a higher effective-bandwidth region. This confirms that chunk selection cannot be treated as a universal constant.

The shift toward larger useful chunks on System C is consistent with the substantially different communication regime of the two platforms. System C combines an H100-class GPU with a PCIe Gen5 host–device link and DDR5 host memory, whereas System A uses a Quadro P400 over PCIe Gen3 with DDR4 host memory. On the higher-throughput System C, small staged

chunks expose per-chunk packing, submission, and synchronization overheads more strongly relative to the available transfer bandwidth. Larger chunks amortize these fixed costs over more data and provide sufficiently large transfers to exploit the faster communication path. The result should therefore be interpreted as a platform-dependent balance between host-side staging cost, per-chunk runtime overhead, and device-transfer throughput, rather than as a fixed architectural chunk size that generalizes across systems.

Bandwidth does not increase monotonically with chunk size for all total volumes. After the useful saturation region, some curves plateau or decline. This confirms the design motivation from Section 5.4.1: the goal is not simply to minimize the number of chunks, but to select a granularity that provides sufficient DMA payload size while preserving staged execution behavior.

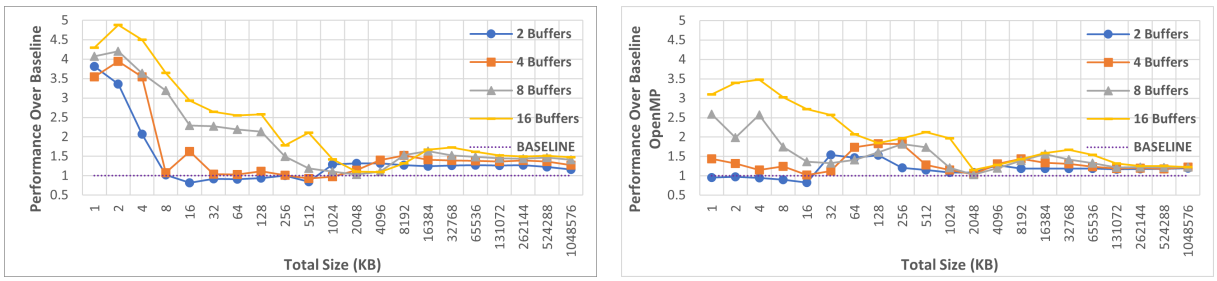
5.7.2 Synthetic Benchmark Evaluation

The synthetic benchmarks evaluate ICF under controlled communication structures before moving to application-level workloads. EMPTY removes nearly all device-side computation and approximates the upper-bound benefit of runtime fusion under regular fragmented mappings. XVADD preserves fragmented input mappings while adding lightweight vector computation and output writeback. Kessler-Interleaved then evaluates ICF against a static operand-layout baseline inspired by operand-transfer fusion, where conflicting multi-kernel operand sets prevent all kernels from being represented as one contiguous transfer region under a single global layout.

EMPTY: Isolated Mapping and Transfer Behavior

The EMPTY benchmark stresses OpenMP data mapping with multiple independently allocated input buffers and negligible computation. This is the most favorable configuration for ICF because the baseline runtime processes each mapped buffer separately, while ICF aggregates the same logical communication volume into a fused allocation and staged transfer sequence.

Figure 5.7 shows the total EMPTY benchmark speedup under explicit data-management execution.

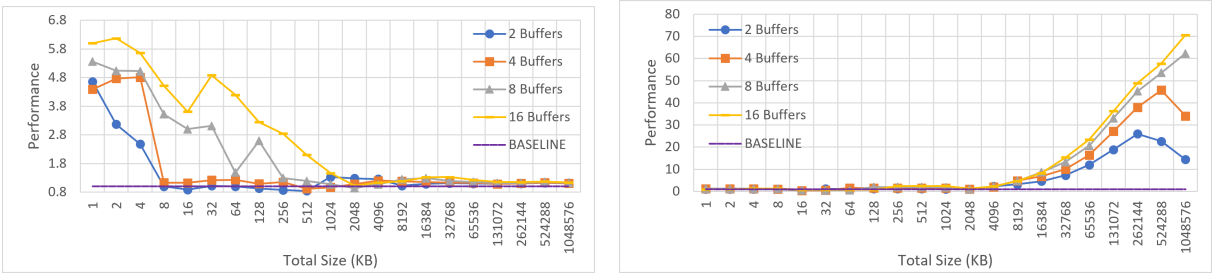


(a) System A. (b) System C.

Figure 5.7: EMPTY benchmark total speedup under explicit data-management execution.

The total results show that ICF is most effective when the communication volume is small to moderate and the buffer count is high. In this regime, the baseline execution is dominated by

fragmented runtime activity: repeated mapping processing, device allocation, transfer submission, synchronization, and cleanup. By reducing the number of independent communication objects, ICF reduces this overhead substantially. To better visualize the component-level behavior of ICF, Figure 5.8 reports the Copy-In and Copy-Out speedups of the EMPTY benchmark on System A. These measurements are shown on the same platform used for the manual Argument Fusion experiments in Chapter 4, allowing the observed trends to be compared qualitatively against Argument Fusion and Fusion+. The comparison is not component-equivalent: in ICF, the Copy-In interval includes not only host-to-device transfer, but also runtime mapping preparation, fused device allocation, host-side staging into pinned buffers, transfer submission and completion, and present-table updates. The figure should therefore be interpreted as a component-level view of the ICF runtime path rather than as a direct H2D-only comparison.



(a) Copy-In. (b) Copy-Out.

Figure 5.8: EMPTY benchmark component-level speedup on System A under explicit data-management execution.

The Copy-In results are the closest measurement of ICF's intended optimization target. They show the expected dependence on buffer count: higher buffer counts expose more fragmented baseline mapping and communication activity, and therefore provide more opportunity for runtime fusion. The largest benefits occur in the low-size regime, where repeated per-mapping allocation, staging, transfer submission, and runtime-management overheads dominate the baseline. As the total size increases, the speedup approaches the baseline because individual transfers increasingly amortize their fixed costs, while ICF continues to pay the cost of constructing the fused staged representation.

The trend is consistent with the manual Argument Fusion and Fusion+ results from Chapter 4: aggregation is most valuable when the baseline consists of many small communication objects. The difference is that manual Argument Fusion and Fusion+ operate on an application-visible fused host layout, whereas ICF must construct the fused representation dynamically inside the OpenMP runtime. Consequently, ICF can recover part of the same fragmentation benefit, but its Copy-In interval also includes staging, allocation, and mapping-management costs that are not present in the same form in the manual CUDA implementation.

The Copy-Out results should not be interpreted as evidence of symmetric device-to-host transfer fusion. The current ICF implementation primarily targets eligible input mappings. In the EMPTY benchmark, the Copy-Out interval includes exit-side runtime processing, present-table

handling, and cleanup of mappings created during entry. The observed Copy-Out speedups therefore indicate secondary runtime effects of ICF rather than direct D2H fusion. By consolidating eligible mappings into a fused allocation and rebinding their present-table entries, ICF can reduce part of the exit-side mapping and cleanup overhead even though the optimization is introduced on the copy-in path.

Overall, Figure 5.8 shows that ICF's EMPTY speedup is produced by two effects: direct reduction of fragmented Copy-In work and secondary reductions in exit-side runtime handling. This mirrors the component-level conclusion of Chapter 4, where communication-path speedup was not caused only by host-to-device transfer aggregation, but also by changes in allocation and deallocation behavior. Figure 5.9 shows the corresponding EMPTY behavior for ordinary `target` execution on System A.

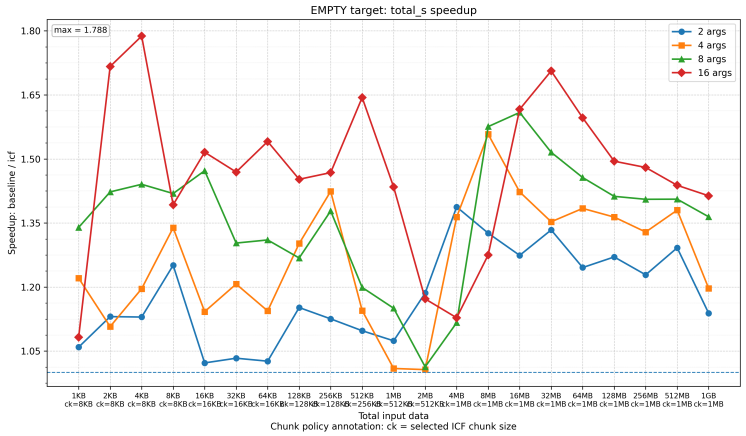


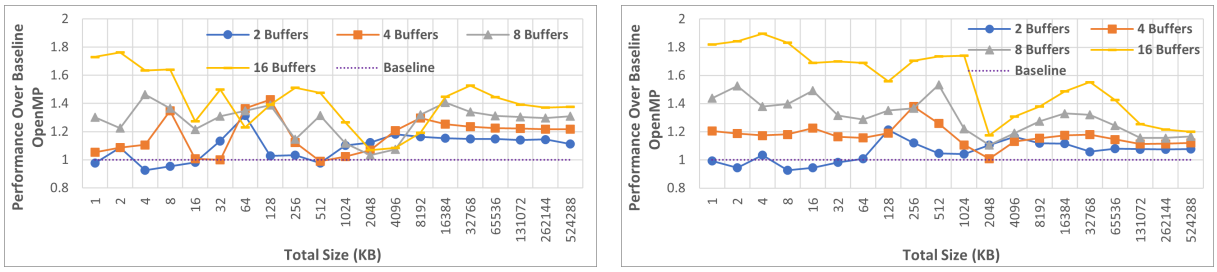
Figure 5.9: EMPTY benchmark speedup on System A under ordinary `target` execution.

The speedup remains visible, but the curves are more irregular because the measured interval includes the complete target-region path rather than only explicit data-management phases. This result shows how much of the copy-in benefit remains visible when OpenMP couples data movement, execution, and synchronization within one construct.

XVADD: Communication Optimization under Computation

The XVADD benchmark adds lightweight parallel computation to the fragmented-input structure used by EMPTY. Multiple input vectors are copied to the device, a vector-addition kernel computes the output, and the result is copied back to the host.

Figure 5.10 reports XVADD speedup under explicit data-management execution.



(a) System A.

(b) System C.

Figure 5.10: XVADD benchmark speedup under explicit data-management execution.

The results confirm that ICF remains beneficial after computation and output writeback are introduced, although the speedups are generally lower than in EMPTY. The same buffer-count trend remains visible: higher buffer counts generally produce stronger speedups because they create more fragmented baseline communication. At larger sizes, the curves tend to stabilize rather than grow without bound, indicating that ICF reduces overhead in fragmented regimes rather than increasing raw bandwidth for all transfers.

XVADD is therefore the bridge between the synthetic upper-bound case and the application results. It shows that ICF is not an empty-kernel artifact, but also that communication optimization does not translate linearly into total acceleration once computation and output movement contribute to the measured interval.

Runtime Fusion under Kessler-Style Layout Constraints

The previous synthetic benchmarks represent favorable cases for runtime fusion: the mapped inputs are regular, independently allocated, and belong to one offload structure. This subsection evaluates a different case, motivated by Kessler-style operand-transfer fusion [24]. In that setting, a static optimizer chooses one global operand layout for a sequence of kernels so that operands used together by a kernel are placed adjacently in memory. Adjacent operands can then be transferred as fewer contiguous regions. The difficulty is that one global operand order must serve all kernels in the sequence. If different kernels use conflicting operand subsets, no single linear layout can make every kernel's operand set contiguous.

The benchmark isolates this layout-conflict case. For readability, the conflict is first shown with eight logical operands,

$$A, B, C, D, E, F, G, H,$$

and four consecutive vector-addition kernels whose input sets are

$$K_1 = \{A, B, C, D\}, \quad K_2 = \{E, F, G, H\}, \quad K_3 = \{A, C, E, G\}, \quad K_4 = \{B, D, F, H\}.$$

The ordinary layout treats each operand as an independent mapped object. A Kessler-style static

layout instead rearranges the operands globally. For example, the layout

$$A, C, B, D, F, H, E, G$$

makes the operand sets of K_1, K_2, and K_4 contiguous, but leaves K_3 split into two contiguous regions, A, C and E, G. This is not an intentionally weak baseline: in this compact example, the static layout reduces the sequence from sixteen logical transfer regions in the ordinary baseline to five transfer regions. However, because one kernel remains split, it does not reach the four-transfer lower bound corresponding to one contiguous transfer region per kernel.

Kernel	**Operand set**	**Effect under layout** A, C, B, D, F, H, E, G
K_1	A, B, C, D	Contiguous region A, C, B, D
K_2	E, F, G, H	Contiguous region F, H, E, G
K_3	A, C, E, G	Split into A, C and E, G
K_4	B, D, F, H	Contiguous region B, D, F, H

Figure 5.11: Kessler-Interleaved operand-layout conflict. A single static operand order can make several kernel operand sets contiguous, but the conflicting operand set K_3 remains split. Runtime ICF differs by constructing a temporary fused communication representation for the concrete mapping set of each offload event.

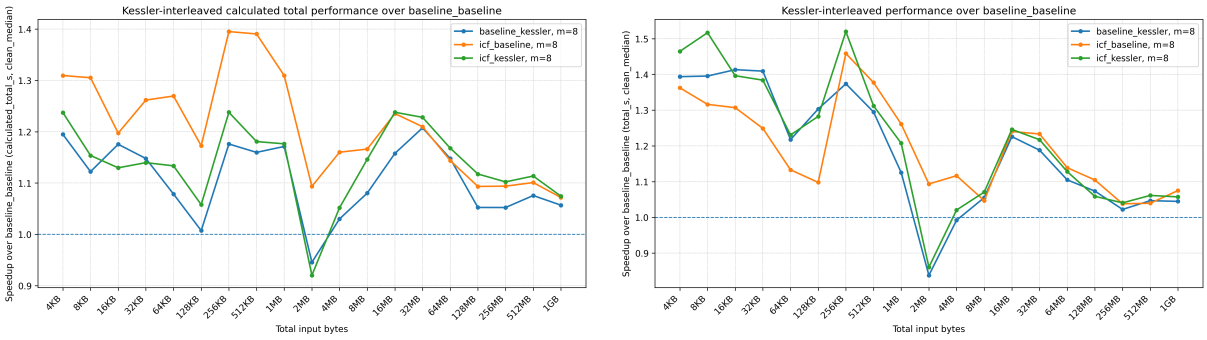
ICF addresses a different point in the design space. It does not require one global operand order to satisfy all kernels. Instead, it observes the concrete mapping set of each offload event and constructs a temporary fused communication representation for that event. The plots below use the sixteen-buffer, eight-mapped-buffer variant of the same layout-conflict pattern. The purpose of the Kessler-Interleaved benchmark is therefore to compare a static global layout strategy with a runtime per-offload fusion strategy under a case where the static layout cannot fully eliminate segmentation.

The experiment compares four configurations. The ordinary baseline uses the default OpenMP layout and the unmodified runtime. The *baseline-kessler* configuration applies the Kessler-style operand layout while preserving the baseline OpenMP execution path. The *ICF-baseline* configuration applies ICF to the ordinary layout. The *ICF-kessler* configuration combines ICF with the Kessler-style layout. The primary comparison is between *ICF-baseline* and *baseline-kessler*: it tests whether runtime fusion can outperform a static operand-layout strategy without requiring the application to adopt that layout. The secondary comparison is between *ICF-kessler* and *baseline-kessler*, which tests whether ICF adds benefit after the static layout has already reduced most of the fragmentation.

The results are reported on System A for the configuration with sixteen total buffers, eight mapped buffers per kernel, and four consecutive kernels. This is the most fragmentation-heavy

configuration in the benchmark and therefore the most relevant case for comparing transfer-fusion behavior. The experiment is used as a controlled literature-positioning evaluation rather than as a full cross-platform characterization.

Because the benchmark uses simple vector-addition kernels, raw target-region timing can obscure the communication effect with computation and runtime noise. Figure 5.12 therefore reports two total-time views. The explicit data-management result uses a calculated total: the measured communication path is combined with normalized computation time, so that the effect of communication-path changes can be evaluated under a controlled full-offload estimate. The ordinary `target` result reports the complete measured target-region interval.



(a) Explicit data-management calculated total.

(b) Ordinary `target` measured total.

Figure 5.12: Kessler-Interleaved total speedup on System A for the configuration with sixteen total buffers, eight mapped buffers per kernel, and four consecutive kernels. The explicit data-management result combines measured communication time with normalized computation time; the ordinary `target` result reports the complete measured target-region interval.

The calculated-total results show that *ICF-baseline* outperforms *baseline-kessler* across most of the evaluated sweep. This is the main comparison of the experiment: a runtime fusion mechanism, applied without Kessler-style global layout restructuring, exceeds the static layout baseline in most transfer-size regimes.

The combined *ICF-kessler* configuration also improves over *baseline-kessler* in most regions. This shows that ICF can still add benefit after static layout optimization. However, *ICF-kessler* usually does not reach the performance of *ICF-baseline*. This is expected: under the Kessler-style layout, most kernels are already represented as contiguous operand groups, and ICF mainly acts on the remaining split case. The remaining opportunity is therefore smaller than in the ordinary layout, where ICF can fuse the current mapping set for each kernel directly.

The ordinary `target` result provides a full-path visibility check rather than the primary comparison between static layout fusion and ICF. The measured interval includes mapping, copy-in, kernel execution, synchronization, copy-out, and cleanup, so the communication benefit does not translate linearly into total target speedup. In this execution form, the ordering is more mixed than in the calculated data-management result. The *Kessler-style* layout is stronger in parts of the small-size regime; *ICF-baseline* is strongest in the medium-to-large-size region; and *ICF-kessler* is competitive or best at several points. The ordinary `target` result therefore

confirms that the effect remains visible in the complete OpenMP target path, but it should not be interpreted as a clean component-level ranking of the mechanisms.

The main conclusion from this benchmark is that ICF provides a runtime alternative to static operand-layout fusion in cases where the global layout problem is structurally constrained. Kessler-style fusion chooses one physical operand order for a sequence of kernels; ICF constructs a temporary fused communication representation for each offload event. In the controlled data-management view, ICF outperforms the Kessler-style static layout baseline across most of the evaluated sweep. In the ordinary `target` view, the result is more mixed but still shows that ICF remains competitive with the static-layout baseline in the full target-region path. This does not imply that runtime fusion dominates static operand fusion in general. Rather, it shows that when a static global layout cannot fully eliminate segmentation, runtime-level fusion can recover additional communication-granularity benefits without requiring global source-level or application-level layout restructuring.

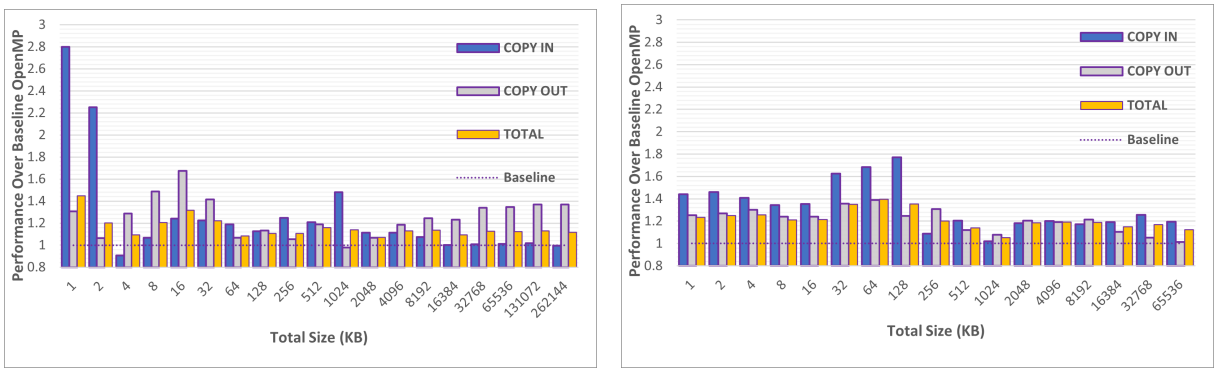
5.7.3 Application-Level Evaluation

The application benchmarks evaluate ICF under progressively less idealized communication structures. These workloads include application-specific control flow, non-uniform data structures, output phases, synchronization effects, and varying ratios between communication and computation.

The three applications stress distinct aspects of runtime-managed fusion. MD-2D evaluates a regular multi-buffer update phase. BFS evaluates irregular graph-processing communication. Service B+Tree evaluates dynamic query batches whose request buffers are supplied at runtime.

MD-2D: Regular Multi-Buffer Update

Figure 5.13 reports copy-in, copy-out, and total speedup for MD-2D under explicit data-management execution.



(a) System A. (b) System C.

Figure 5.13: MD-2D copy-in, copy-out, and total speedup under explicit data-management execution.

The copy-in curves are the most direct measurement of ICF because the optimization modifies the input-mapping path. The observed copy-in gains show that ICF remains effective for a structured application update kernel, not only for synthetic uniformly sized arrays.

Total speedup is lower than copy-in speedup, but remains positive across a broad part of the sweep. MD-2D therefore validates the regular application case: ICF's copy-in benefit can remain visible when structured application work is included.

Figure 5.14 reports MD-2D under ordinary `target` execution on System A.

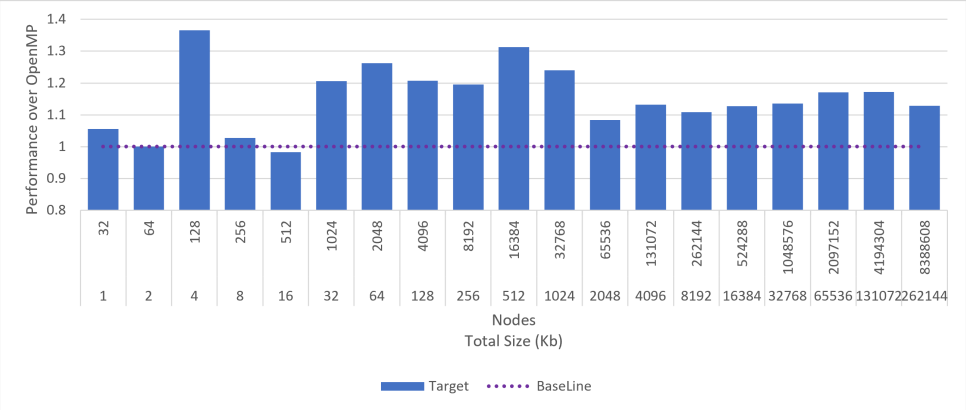


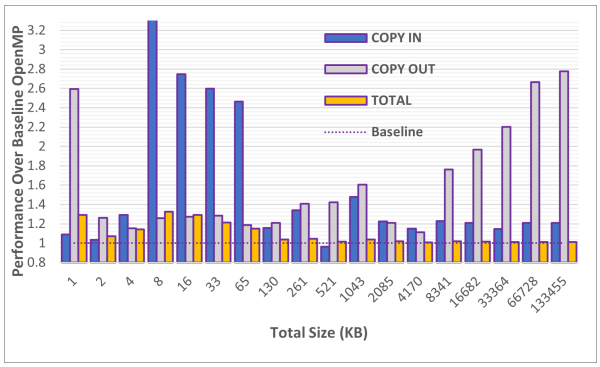
Figure 5.14: MD-2D speedup on System A under ordinary `target` execution.

This execution form is less decomposable because mapping, copy-in, kernel execution, synchronization, copy-out, and cleanup are integrated into one target-region interval. The result is important qualitatively: the MD-2D benefit is not solely an artifact of isolating `target enter data`; it remains visible in the ordinary OpenMP target path.

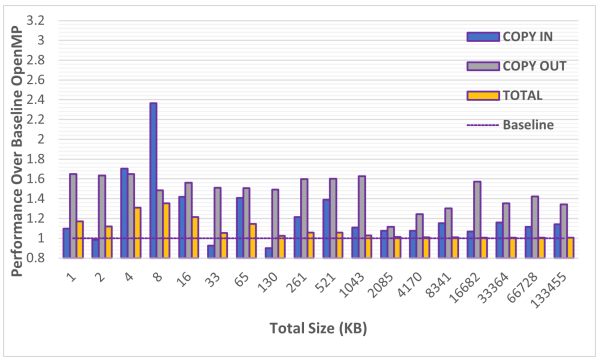
BFS: Irregular Graph Workload

BFS evaluates ICF on an irregular graph traversal. The copy-in phase contains a heterogeneous set of mapped arrays associated with graph structure, frontier state, update masks, visited-state data, and cost values. These arrays differ in size and semantic role, making BFS fundamentally different from EMPTY, XVADD, and MD-2D.

Figure 5.15 reports copy-in, copy-out, and total speedup for BFS.



(a) System A.



(b) System C.

Figure 5.15: BFS copy-in, copy-out, and total speedup under explicit data-management execution.

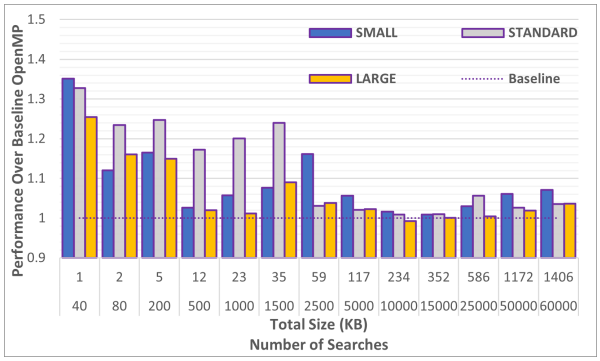
The copy-in results show that ICF can improve the input path even when mappings are non-uniform. This demonstrates that the implementation is not restricted to synthetic equal-size vectors. The slice layout and chunk-construction mechanism can aggregate irregular OpenMP mappings while preserving their logical identities.

Total speedup is more limited and less regular than in MD-2D. BFS should therefore be interpreted as a boundary case: ICF can improve irregular copy-in, but the end-to-end result depends on how much of the measured interval is represented by eligible input mappings.

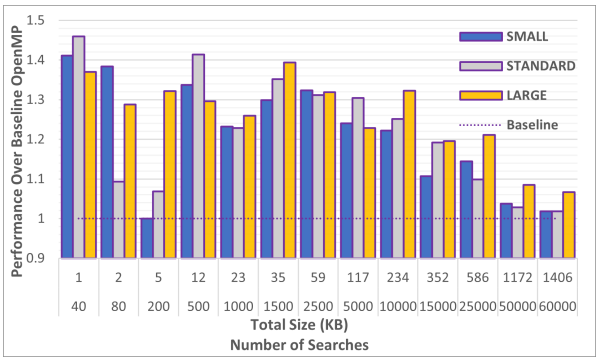
Service B+Tree: Dynamic Query Batches

Service B+Tree evaluates ICF in a dynamic query-processing scenario derived from the Rodinia B+Tree benchmark. The persistent tree structure is initialized and transferred before the measured interval. The reported measurements focus on the steady-state query phase, where request-side arrays are supplied at runtime.

Figure 5.16 reports Service B+Tree speedup as the number of searches in each batch increases.



(a) System A.



(b) System C.

Figure 5.16: Service B+Tree speedup for dynamic query batches. The initial tree offload is excluded from the measured interval.

The results show that ICF can improve dynamic service-style offloading, but the magnitude

depends on the balance between request-buffer movement, search computation, answer writeback, and runtime cleanup.

The non-monotonicity of the curves is expected. Increasing the number of searches increases the amount of request data that can benefit from fusion, but also increases search work and answer handling. When the service phase becomes dominated by query processing rather than input-buffer movement, the relative contribution of ICF decreases.

Service B+Tree complements BFS as a realistic boundary case. BFS stresses irregular mapping structure; Service B+Tree stresses runtime-supplied data whose communication pattern is materialized only at execution time. Together, they show that ICF is most valuable when the runtime observes multiple eligible input mappings whose fragmented copy-in cost is significant in the measured interval.

5.7.4 Cross-Benchmark Interpretation

Across the full evaluation, ICF shows a consistent behavioral pattern. It is most effective when three conditions hold: the offload contains multiple independently mapped input buffers, the total communication volume is large enough to amortize staged DMA but small enough that baseline fragmentation remains costly, and copy-in communication represents a significant fraction of the measured interval.

Table 5.6 summarizes the role of each benchmark.

Table 5.6: Cross-benchmark interpretation of ICF behavior.

Benchmark	**Communication structure**	**Observed ICF behavior**
EMPTY	Regular fragmented inputs with negligible computation.	Upper-bound case. ICF strongly reduces mapping and transfer fragmentation, especially for many buffers and small-to-moderate total sizes.
XVADD	Regular fragmented inputs with lightweight computation and output writeback.	Communication benefit remains visible, but total speedup is reduced by kernel execution, output transfer, synchronization, and cleanup.
Kessler-Interleaved	Multi-kernel operand-layout conflict inspired by static operand-transfer fusion.	Literature-positioning case. ICF outperforms a Kessler-style static layout baseline across most of the calculated data-management sweep on System A, while the ordinary `target` result remains mixed but competitive.
MD-2D	Regular application update with structured multi-buffer inputs.	Strongest application-level validation. Copy-in gains partially survive into total execution.
BFS	Irregular graph workload with non-uniform mappings and metadata.	Boundary case. Copy-in may improve, but total speedup is limited by irregular application behavior and phases outside the optimized path.
Service B+Tree	Dynamic runtime-supplied query buffers with persistent tree data.	Runtime-relevant case. ICF can optimize dynamically supplied pageable request arrays, but service-level speedup depends on query processing and answer movement.

The synthetic results establish that the mechanism works under the conditions it is designed for. The application results show how those benefits are bounded by real execution structure. ICF does not guarantee large end-to-end acceleration merely because multiple mappings exist; it improves performance when fragmented copy-in is a meaningful part of the measured execution interval.

5.8 Discussion

5.8.1 Scope of the Optimization

ICF optimizes the physical organization of OpenMP host-to-device input communication. It does not change the algorithm executed by the kernel, the application-visible host-memory layout, the kernel interface, or the OpenMP mapping clauses written by the programmer. Its scope is the runtime copy-in path: mapping inspection, device allocation, host-side staging, host-to-device transfer submission, present-table updates, and the synchronization required to complete entry mapping.

The Kessler-Interleaved experiment further distinguishes ICF from static operand-layout fusion. Static fusion attempts to choose an application-level operand order that maximizes contiguous transfer regions across a kernel sequence. ICF instead constructs a temporary fused representation for the mapping set observed at runtime. Its scope is therefore not global data-layout optimization, but runtime reconstruction of a fused copy-in path for eligible mappings.

This scope is narrower than general OpenMP data-mapping optimization. Static analyses and directive-rewriting approaches can reduce when communication is required by changing data-region placement or improving reuse across target regions. ICF instead assumes that a set of mappings has already been requested and reduces the cost of communicating eligible mappings.

Copy-in speedup is therefore the most direct evidence of ICF's effect. End-to-end speedup is more workload-dependent because it also includes computation, writeback, cleanup, synchronization, and runtime activity outside the optimized copy-in path.

5.8.2 When ICF Helps and When It Saturates

The cross-benchmark results show that ICF is most effective when fragmented copy-in is a substantial part of the measured interval. This condition holds most clearly in EMPTY and progressively weakens as computation, writeback, irregular mapping structure, or service-level work become dominant.

ICF saturates when the baseline communication is no longer dominated by fragmentation. This occurs when individual transfers already amortize startup overheads, when the fusion group is too small to justify staging, or when phases outside the optimized copy-in path dominate total execution time.

5.8.3 Runtime Effects: Synchronization, Allocation, and Chunking

The measured behavior of ICF is shaped by runtime effects in addition to raw transfer bandwidth.

First, directive form matters. Explicit data-management directives expose the copy-in path more directly, while ordinary `target` regions include additional runtime and kernel-execution behavior. Consequently, the same fused transfer pipeline can appear differently depending

on whether the measured interval is dominated by mapping completion or full target-region execution.

Second, ICF changes allocation behavior. Selected mappings share one fused device allocation instead of requiring independent allocations. This can reduce allocation multiplicity and can also change interaction with backend allocator behavior. These effects are runtime- and backend-dependent and should be interpreted as implementation behavior rather than as a universal property of all OpenMP runtimes.

Third, the chunking policy determines whether staged fusion preserves overlap or degenerates toward either transfer fragmentation or monolithic staging. Very small chunks reintroduce per-chunk submission and synchronization overhead. Very large chunks reduce pipeline depth and approach monolithic host-side packing before transfer. The calibrated policy used in this chapter selects practical chunk sizes for the evaluated platforms, but it is not claimed to be universally optimal.

5.8.4 Backend Dependence and Portability

ICF relies on a general runtime principle: disjoint logical mappings can be communicated through a fused physical path if the runtime can construct a safe contiguous transfer representation and preserve mapping-table semantics. This principle is not specific to one accelerator architecture.

The evaluated implementation, however, is CUDA-specific in its backend behavior. It uses pinned staging buffers, CUDA asynchronous transfers, CUDA events for staging-buffer reuse, and the plugins-nextgen CUDA path in LLVM OpenMP. The motivation for staged fusion is also informed by observed pageable-transfer behavior on CUDA systems. Therefore, the design principle is portable, but the measured performance and calibrated chunking policy are backend- and platform-dependent.

A port to another backend would need to re-establish three properties: an appropriate staging mechanism, asynchronous transfer submission with safe staging-buffer reuse, and correct integration with the backend's allocation and mapping-table behavior. This would require backend-specific work for plugin targets such as CUDA, ROCm, or Level Zero rather than a direct reuse of the measured CUDA policy [30, 1, 55]. The thesis therefore claims portability of the runtime abstraction, not portability of the exact CUDA performance results.

5.8.5 Relationship to Manual Fusion and Current Limitations

ICF extends the communication-granularity principle studied through manual Argument Fusion, but applies it at a different layer. Manual fusion constructs a fused layout before offloading and exposes that layout to the application or generated host code. ICF preserves the original application layout and constructs the fused representation inside the runtime copy-in path.

The current implementation is conservative. It focuses on eligible host-to-device input mappings and does not provide a general solution for all OpenMP mapping cases. Unsupported

map-type combinations, already-present mappings, pointer-containing structures, overlapping mappings, and cases without sufficient fusion opportunity fall back to the baseline path. Device-to-host writeback fusion is not treated symmetrically in the current implementation.

Energy consumption was not measured in this evaluation, and no assumption of zero energy overhead should be made. ICF introduces additional host-side work through mapping analysis and memory copies into the staging buffers, while also reducing the number of device allocations and host-to-device transfer operations. The net energy effect therefore cannot be inferred from execution-time speedup alone and remains outside the scope of the present evaluation.

The limitations of ICF therefore define its operating regime rather than invalidating the optimization. ICF is intended for offload regions in which multiple eligible input mappings are materialized as fragmented host-to-device communication and where the cost of staging and rebinding is smaller than the runtime overhead removed by fusion. This makes it complementary to manual data-region management, static OpenMP mapping optimization, and application-level layout transformations rather than a replacement for them.

5.9 Summary

This chapter presented In-Copy Fusion, a runtime-level implementation of communication-granularity optimization inside LLVM's OpenMP offloading runtime. ICF extends the principle studied through manual Argument Fusion by operating directly on eligible OpenMP input mappings. The implementation constructs a fused device allocation, stages pageable host data through runtime-managed pinned buffers, submits chunked asynchronous host-to-device transfers, and rebinds the original OpenMP mappings to slices of the fused allocation while preserving the standard mapping abstraction.

The evaluation showed that ICF improves copy-in and total offload performance when OpenMP data movement is fragmented across multiple mappings and when communication-relevant runtime overheads are significant. The results also identified the limits of the approach: benefits diminish when transfers are already bandwidth-efficient, when computation or writeback dominates, when the baseline incidentally overlaps allocation and transfer costs, or when mappings are not safely eligible for fusion. These findings complete the transition from manual fusion as a characterization tool to runtime fusion as a transparent OpenMP optimization. The next chapter summarizes the thesis-level conclusions and outlines directions for future work.

With respect to the thesis research questions, the chapter shows that runtime-level fusion is feasible for eligible OpenMP input mappings, that chunked staging is required to make runtime fusion practical under pageable host-memory conditions, and that the observed benefit depends on mapping fragmentation, directive form, backend behavior, and the fraction of execution time occupied by copy-in.

6. Conclusions and Future Work

6.1 Thesis Conclusions

This thesis studied the efficiency of host–device data movement in loosely coupled heterogeneous systems, with particular focus on discrete GPU platforms and OpenMP offloading through LLVM's `libomptarget` runtime. The central claim of the thesis is that offload performance depends not only on the total number of bytes transferred between host and device, but also on the physical granularity with which those bytes are allocated, transferred, synchronized, and released.

The work was developed in two stages. First, manual Argument Fusion was evaluated as a controlled application-level transformation. By replacing multiple per-argument allocations, transfers, and deallocations with a smaller number of fused operations, Argument Fusion exposes the performance cost of fragmented communication. The evaluation showed that communication fragmentation is most harmful when data movement is divided across several small or medium-sized arguments. It also showed that the benefit of fusion is not limited to host-to-device transfer aggregation: allocator and deallocator consolidation can contribute significantly to the observed speedup.

Second, the thesis presented In-Copy Fusion (ICF), a runtime-level mechanism that moves the fusion principle into LLVM's OpenMP offloading runtime. ICF identifies eligible input mappings at runtime, packs their data through pinned staging buffers, transfers the fused payload into a single device allocation, and rebinds the original OpenMP mappings to slices of that allocation. This preserves source-level code, kernel interfaces, and OpenMP mapping semantics while changing the physical organization of communication.

A central design result of ICF is that fusion must be combined with controlled splitting. Naive monolithic runtime fusion can serialize host-side packing before device-side transfer. ICF therefore divides the fused payload into calibrated chunks, allowing host-side packing and host-to-device DMA to proceed as a staged pipeline. In this sense, fusion and splitting are complementary communication-granularity transformations: fusion removes excessive fragmentation across logical mappings, while chunking introduces controlled segmentation to preserve overlap and avoid monolithic staging overhead.

The experimental evaluation showed the bounded nature of the optimization. ICF improves copy-in and total offload performance when OpenMP data movement is fragmented across multiple eligible mappings and when communication-relevant runtime overheads are significant. Its end-to-end benefit decreases when transfers are already bandwidth-efficient, when computation or writeback dominates total execution time, or when the baseline runtime incidentally hides part of allocation cost behind in-flight transfers. These results support the thesis claim without implying universal acceleration.

Quantitatively, the manual Argument Fusion characterization showed speedups of up to approximately 2.1×, increasing to approximately 2.6× when fusion was combined with pinned memory and asynchronous transfer submission in Fusion+. At the runtime level, ICF achieved synthetic speedups of up to approximately 2.1× on System A and 3.1× on System C. Across the evaluated application-level ICF workloads, host-to-device improvements reached approximately 2×, while the largest end-to-end improvements were on the order of 40%. These maxima occur in favorable communication-dominated configurations and should be interpreted together with the limiting regimes identified in Chapters 4 and 5.

Overall, the thesis establishes that communication granularity is a first-order systems concern in discrete heterogeneous offloading. Manual Argument Fusion provides a controlled method for characterizing this effect, while In-Copy Fusion demonstrates that part of the same opportunity can be recovered transparently inside the OpenMP runtime. The resulting contribution is a bounded runtime optimization for a specific but important class of offloading inefficiencies: multi-mapping communication patterns whose performance is limited by fragmented allocation and transfer behavior.

6.2 Answers to the Research Questions

The research questions introduced in Chapter 1 can now be answered as follows.

RQ1: How does communication granularity affect host–device communication performance, and under which conditions can Argument Fusion improve it? Transfer segmentation reduces communication efficiency by decomposing a fixed aggregate payload into multiple smaller physical operations. This increases both visible orchestration overhead between transfers and the cumulative active-copy time required to move the same total amount of data. Argument Fusion is therefore most effective when communication is distributed across several small or medium-sized arguments, especially as argument count increases. The observed benefit is not caused by transfer aggregation alone: allocation and deallocation consolidation can also contribute substantially to communication-path speedup. Pinned memory and asynchronous transfer submission remain complementary because they target different parts of the communication pipeline.

RQ2: Can communication fusion be moved from application-level restructuring into the offloading runtime while preserving the programming-model semantics? Yes, for a conservative subset of OpenMP mappings. ICF demonstrates that eligible input mappings can be fused inside LLVM's OpenMP runtime by allocating a shared device buffer, staging the participating host data, transferring the fused payload, and rebinding the original mappings to slices of the fused allocation. This preserves source-level code, kernel interfaces, and the logical OpenMP mapping abstraction while changing the physical communication path. The current

implementation deliberately falls back to the baseline path for mapping cases whose semantics cannot be preserved safely.

RQ3: How can runtime-level fusion be organized effectively, and under which workloads does it improve performance? Runtime-managed fusion must avoid replacing transfer fragmentation with a new host-side serialization bottleneck. ICF addresses this by splitting the fused payload into calibrated staged chunks, allowing host-side packing and host-to-device transfer to proceed as a pipeline. The evaluation shows that this approach is most effective when several eligible input mappings are present and fragmented copy-in represents a significant fraction of the measured interval. Its benefit saturates or degrades when individual transfers are already bandwidth-efficient, when the fusion group is too small to amortize staging, when computation or writeback dominates end-to-end execution, or when baseline runtime behavior already hides part of the relevant overhead.

6.3 Limitations

The scope of this thesis is limited to host–device communication in loosely coupled heterogeneous systems. The proposed mechanisms do not reduce the semantic amount of data required by an application, nor do they optimize the computational kernels themselves. They instead change how required data movement is physically organized.

Manual Argument Fusion requires application-level restructuring and is therefore not presented as a general programming methodology. Its role in the thesis is analytical: it provides a controlled transformation for isolating the cost of fragmented communication and memory-management multiplicity.

ICF removes the need for application-level restructuring, but only for mappings that can be fused conservatively while preserving OpenMP semantics. The current implementation focuses primarily on eligible host-to-device input mappings. It does not provide a general solution for arbitrary pointer-containing structures, all writeback cases, overlapping mappings, complex member mappings, or all forms of nested data structure. These restrictions are intentional: the runtime must preserve correctness before maximizing fusion coverage. They bound the current implementation to the class of mappings for which runtime fusion can be performed without changing source-level behavior.

The measured performance behavior is also backend- and platform-dependent. ICF relies on properties of pageable-memory staging, pinned buffers, device allocation, asynchronous transfer submission, and OpenMP device-plugin behavior. The general principle that communication granularity affects performance is portable, but the exact quantitative results and optimal chunking policy are specific to the evaluated systems and software stack.

Finally, the optimization is beneficial only when communication fragmentation is a relevant bottleneck. Workloads dominated by computation, device-to-host writeback, long-lived device-

resident data, or already-efficient large transfers may show limited end-to-end improvement. These cases define the operating regime of the technique rather than contradicting the thesis claim.

6.4 Future Work

The results of this thesis suggest several directions for future research. These directions do not change the conclusion of the thesis; they concern deeper integration, broader applicability, and more adaptive runtime policy for the same communication-granularity principle.

A first direction is deeper backend-aware integration of the staged transfer pipeline. ICF currently implements explicit staging and chunked host-to-device transfer at the OpenMP runtime level. This makes the mechanism transparent within LLVM OpenMP, but it also means that the runtime reconstructs part of the behavior that lower-level backends and drivers already perform internally for pageable transfers. A more integrated device-plugin, backend-level, or driver-adjacent implementation could expose finer control over staging buffers, transfer queues, and chunk scheduling. Such integration could reduce redundant host-side copying, improve overlap between packing and DMA, and allow chunking decisions to account for backend-specific transfer behavior more directly.

A second direction is adaptive profitability control. The current implementation uses conservative semantic eligibility checks and calibrated chunk-size policies. Future work could extend this with a runtime policy that decides not only whether a mapping set is semantically fusable, but also whether fusion is expected to be beneficial for the current transfer sizes, argument count, execution form, and backend state. Such a policy could combine empirical calibration, lightweight online measurements, and static information already available in the mapping list. This would allow ICF to remain opportunistic: applying fusion when fragmented communication is likely to dominate, and falling back to the baseline path when packing, staging, or changed overlap structure is expected to outweigh the benefit.

A third direction is compiler-assisted host-code fusion. Manual Argument Fusion shows that application-visible packing can expose substantial communication-granularity benefits, but writing and maintaining such layouts by hand is intrusive. A compiler or source-to-source transformation could generate host-side packing, offset calculation, and device-pointer reconstruction automatically for kernels whose argument structure is statically visible. Such an approach would sit between manual Argument Fusion and runtime ICF: more explicit than runtime-only fusion, but less burdensome than hand-written fused layouts. It could also cooperate with runtime ICF by exposing more regular mapping structures or by annotating cases where static fusion is profitable.

A fourth direction is the unification of fusion with host-side tiling. Fusion reduces excessive fragmentation by aggregating several small mappings, whereas tiling intentionally splits large data movement and computation into pipelineable units. A natural continuation is a unified granularity

policy that decides when to fuse several mappings, when to tile large mappings, and when to combine both. Such a policy would treat fusion and tiling as complementary transformations over the same communication pipeline: fusion reduces unnecessary fragmentation, while tiling introduces controlled segmentation to enable overlap and bounded staging.

A fifth direction is broader support for OpenMP mapping patterns and device backends. Future implementations could extend the same approach to additional map types, selected writeback cases, and more complex data layouts where correctness can be preserved through precise slice tracking and lifetime management. The mechanism could also be evaluated on non-CUDA OpenMP backends, such as AMD or Intel GPU plugins, to distinguish CUDA-specific staging and transfer effects from properties that generalize to OpenMP runtime-level communication optimization.

These directions extend the thesis without reopening its central claim. The work presented here establishes that communication granularity is measurable, that it materially affects offload performance, that manual fusion can characterize the opportunity, and that runtime-level fusion can recover part of that opportunity transparently for eligible OpenMP mappings. Future work can therefore focus on making the mechanism more adaptive, more deeply integrated, and applicable to a broader range of mappings and platforms, while preserving the semantic and methodological foundation established in this thesis.

References

[1] Advanced Micro Devices. *AMD ROCm Programming Guide*. https://rocm.docs.amd.com/en/latest/how-to/programming_guide.html. Accessed 2026-06-01.

[2] Tyler Allen and Rong Ge. "In-Depth Analyses of Unified Virtual Memory System for GPU Accelerated Computing". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 1–15. DOI: 10.1145/3458817.3476171.

[3] Gene M Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. 1967, pp. 483–485.

[4] Petros Anastasiadis et al. "CoCoPeLia: Communication-Computation Overlap Prediction for Efficient Linear Algebra on GPUs". In: *Proceedings of the 2021 IEEE International Symposium on Performance Analysis of Systems and Software*. ISPASS '21. IEEE, 2021, pp. 17–29. DOI: 10.1109/ISPASS51385.2021.00015.

[5] Petros Anastasiadis et al. "PARALiA: A Performance Aware Runtime for Auto-Tuning Linear Algebra on Heterogeneous Systems". In: *ACM Transactions on Architecture and Code Optimization* 20.4 (2023), pp. 1–25. DOI: 10.1145/3624569.

[6] Joshua Bakita and James H. Anderson. "Demystifying NVIDIA GPU Internals to Enable Reliable GPU Management". In: *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2024, pp. 294–305. DOI: 10.1109/RTAS61025.2024.00031.

[7] Burak Bastem et al. "Overlapping Data Transfers with Computation on GPU with Tiles". In: *Proceedings of the 46th International Conference on Parallel Processing*. ICPP '17. IEEE, 2017, pp. 171–180. DOI: 10.1109/ICPP.2017.26.

[8] Mehmet E. Belviranli et al. "CuMAS: Data Transfer Aware Multi-Application Scheduling for Shared GPUs". In: *Proceedings of the 2016 International Conference on Supercomputing*. ICS '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1–12. DOI: 10.1145/2925426.2926271.

[9] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. "Reconsidering Custom Memory Allocation". In: *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '02. Association for Computing Machinery, 2002, pp. 1–12. DOI: 10.1145/582419.582421.

[10] Roberto Cavicchioli et al. “Novel methodologies for predictable CPU-to-GPU command offloading”. In: *LEIBNIZ INTERNATIONAL PROCEEDINGS IN INFORMATICS* 133 (2019).

[11] Shuai Che et al. “Rodinia: A Benchmark Suite for Heterogeneous Computing”. In: *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*. IISWC ’09. IEEE, 2009, pp. 44–54. DOI: 10.1109/IISWC.2009.5306797.

[12] Bennett Cooper, Thomas R. W. Scogland, and Rong Ge. “Shared Virtual Memory: Its Design and Performance Implications for Diverse Applications”. In: *Proceedings of the 38th ACM International Conference on Supercomputing*. ICS ’24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 26–37. DOI: 10.1145/3650200.3656608.

[13] Robert H Dennard et al. “Design of ion-implanted MOSFET’s with very small physical dimensions”. In: *IEEE Journal of solid-state circuits* 9.5 (2003), pp. 256–268.

[14] Hadi Esmaeilzadeh et al. “Dark silicon and the end of multicore scaling”. In: *Proceedings of the 38th annual international symposium on Computer architecture*. 2011, pp. 365–376.

[15] Yi Feng and Emery D. Berger. “A Locality-Improving Dynamic Memory Allocator”. In: *Proceedings of the 2005 Workshop on Memory System Performance*. MSP ’05. Association for Computing Machinery, 2005, pp. 68–77. DOI: 10.1145/1111583.1111592.

[16] Jordan Fix, Andrew Wilkes, and Kevin Skadron. “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*. 2011.

[17] Chris Gregg and Kim Hazelwood. “Where Is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer”. In: *Proceedings of the 2011 IEEE International Symposium on Performance Analysis of Systems and Software*. ISPASS ’11. IEEE, 2011, pp. 134–144. DOI: 10.1109/ISPASS.2011.5762730.

[18] Hengliang Guo et al. “OpenMP offloading data transfer optimization for DCUs: H. Guo et al.” In: *The Journal of Supercomputing* 80.2 (2024), pp. 2381–2402.

[19] John L Hennessy and David A Patterson. “A new golden age for computer architecture”. In: *Communications of the ACM* 62.2 (2019), pp. 48–60.

[20] Heterogeneous System Architecture Foundation. *HSA Specifications*. https://hsafoundation.com/standards/. Accessed: 2026-06-29. 2015.

[21] Mark Horowitz. “1.1 computing’s energy problem (and what we can do about it)”. In: *2014 IEEE international solid-state circuits conference digest of technical papers (ISSCC)*. IEEE. 2014, pp. 10–14.

[22] Joseph Huber et al. “Efficient Execution of OpenMP on GPUs”. In: *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*. CGO ’22. IEEE, 2022, pp. 41–52. DOI: 10.1109/CGO53902.2022.9741290.

[23] Jaehoon Jung et al. “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*. PPoPP ’20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 321–335. DOI: 10.1145/3332466.3374531.

[24] Christoph Kessler. “Global Optimization of Operand Transfer Fusion in Heterogeneous Computing”. In: *Proceedings of the 22nd International Workshop on Software and Compilers for Embedded Systems*. SCOPES ’19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 49–58. DOI: 10.1145/3323439.3323981.

[25] Khronos Group. *SYCL 2020 Specification*. https://www.khronos.org/sycl/. Accessed: 2026-06-29. 2021.

[26] Andreas Kurth et al. “HERO: Heterogeneous embedded research platform for exploring RISC-V manycore accelerators on FPGA”. In: *arXiv preprint arXiv:1712.06497* (2017).

[27] Lu Li and Christoph Kessler. “Lazy Allocation and Transfer Fusion Optimization for GPU-Based Heterogeneous Systems”. In: *Proceedings of the 26th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*. PDP ’18. IEEE, 2018, pp. 311–315. DOI: 10.1109/PDP2018.2018.00054.

[28] Ruihao Li et al. “Performance Implications of Pipelining the Data Transfer in CPU-GPU Heterogeneous Systems”. In: *ACM Transactions on Architecture and Code Optimization* (2025). DOI: 10.1145/3746231.

[29] LLVM Project. *Building the OpenMP Libraries*. https://openmp.llvm.org/Building.html. Accessed 2026-06-01.

[30] LLVM Project. *LLVM/OpenMP Runtimes*. https://openmp.llvm.org/design/Runtimes.html. Accessed: 2026-06-24. 2026.

[31] Daniel Lustig and Margaret Martonosi. “Reducing GPU Offload Latency via Fine-Grained CPU-GPU Synchronization”. In: *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture*. HPCA ’13. IEEE, 2013, pp. 354–365. DOI: 10.1109/HPCA.2013.6522332.

[32] Martin Maas, Krste Asanović, and John Kubiatowicz. “Learning-Based Memory Allocation for C++ Server Workloads”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’20. Association for Computing Machinery, 2020, pp. 541–556. DOI: 10.1145/3373376.3378525.

[33] Luke Marzen, Akash Dutta, and Ali Jannesari. “Static Generation of Efficient OpenMP Offload Data Mappings”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’24. IEEE, 2024, pp. 1–15. DOI: 10.1109/SC41406.2024.00041.

[34] Luke Marzen, Junhyung Shim, and Ali Jannesari. “Dynamic Detection of Inefficient Data Mapping Patterns in Heterogeneous OpenMP Applications”. In: *Proceedings of the 30th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’26. New York, NY, USA: Association for Computing Machinery, 2026. DOI: 10.1145/3774934.3786454.

[35] Alok Mishra, Abid M. Malik, and Barbara Chapman. “Data Transfer and Reuse Analysis Tool for GPU-Offloading Using OpenMP”. In: *OpenMP: Portable Multi-Level Parallelism on Modern Systems*. IWOMP ’20. Springer, 2020, pp. 280–294. DOI: 10.1007/978-3-030-58144-2_18.

[36] Jose M. Monsalve Díaz et al. “Analysis of OpenMP 4.5 Offloading in Implementations: Correctness and Overhead”. In: *Parallel Computing* 89 (2019), p. 102546. DOI: 10.1016/j.parco.2019.102546.

[37] Gordon E Moore et al. *Cramming more components onto integrated circuits*. 1965.

[38] Harini Muthukrishnan et al. “Efficient Multi-GPU Shared Memory via Automatic Data Movement”. In: *Proceedings of the 48th Annual International Symposium on Computer Architecture*. ISCA ’21. IEEE, 2021, pp. 74–87. DOI: 10.1109/ISCA52012.2021.00015.

[39] Harini Muthukrishnan et al. “FinePack: Transparently Improving the Efficiency of Fine-Grained Transfers in Multi-GPU Systems”. In: *Proceedings of the 2023 IEEE International Symposium on High-Performance Computer Architecture*. HPCA ’23. IEEE, 2023, pp. 516–529. DOI: 10.1109/HPCA56546.2023.10070949.

[40] NVIDIA Corporation. *CUDA C++ Best Practices Guide*. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/. 2023.

[41] NVIDIA Corporation. *CUDA C++ Programming Guide: Stream-Ordered Memory Allocator*. https://docs.nvidia.com/cuda/cuda-programming-guide/04-special-topics/stream-ordered-memory-allocation.html. Accessed: 2026-06-22. 2026.

[42] NVIDIA Corporation. *CUDA Runtime API: API Synchronization Behavior*. https://docs.nvidia.com/cuda/cuda-runtime-api/api-sync-behavior.html. Accessed: 2026-06-22. 2026.

[43] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*. Version 3.2. 2010.

[44] NVIDIA Corporation. *NVIDIA Nsight Systems User Guide*. https://docs.nvidia.com/nsight-systems/UserGuide/. Accessed: 2026-05-27. 2024.

[45] OpenMP Architecture Review Board. *OpenMP Application Programming Interface Version 5.2*. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf. Accessed: 2026-06-24. Nov. 2021.

[46] Carl Pearson et al. "Evaluating Characteristics of CUDA Communication Primitives on High-Bandwidth Interconnects". In: *Proceedings of the Tenth ACM/SPEC International Conference on Performance Engineering*. ICPE '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 209–218. DOI: 10.1145/3297663.3310299.

[47] Louis-Noël Pouchet et al. "Polybench: The polyhedral benchmark suite". In: *URL: http://www. cs. ucla. edu/pouchet/software/polybench* 437 (2012), pp. 1–1.

[48] Marcos Rodriguez et al. "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)*. Ed. by Daniele Cattaneo et al. Vol. 127. Open Access Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, 1:1–1:15. ISBN: 978-3-95977-363-8. DOI: 10.4230/OASIcs.PARMA-DITAM.2025.1. URL: https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.PARMA-DITAM.2025.1.

[49] Abu Sebastian et al. "Memory devices and applications for in-memory computing". In: *Nature nanotechnology* 15.7 (2020), pp. 529–544.

[50] SOLLVE Project. *SOLLVE Benchmarks: Performance Benchmarks for OpenMP Features*. Version sc19uvm. GitHub repository (release tag sc19uvm), accessed 2026-02-08. Aug. 1, 2019. URL: https://github.com/SOLLVE/benchmarks.

[51] D.-Od. Sotiropoulos, A. Capotondi, and A. Marongiu. "Evaluating the Benefits of Argument Fusion: Optimizing Host-Device Communication". In: *Proceedings of the 23rd ACM International Conference on Computing Frontiers Companion*. CF Companion '26. New York, NY, USA: Association for Computing Machinery, 2026. DOI: 10.1145/3801488.3806378.

[52] Dionysios-Odysseas Sotiropoulos et al. "In-Copy Fusion: Runtime Argument Fusion for Efficient OpenMP GPU Offloading". In: *Proceedings of the 55th International Conference on Parallel Processing*. ICPP '26. Singapore, Singapore: Association for Computing Machinery, 2026. ISBN: 979-8-4007-2657-6. DOI: 10.1145/3832810.3832841. URL: https://doi.org/10.1145/3832810.3832841.

[53] J. E. Stone, D. Gohara, and Guochun Shi. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”. In: *Computing in Science and Engg.* 12.3 (May 2010), 66–73. ISSN: 1521-9615.

[54] Ivan Tanasic et al. “Enabling preemptive multiprogramming on GPUs”. In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 2014, pp. 193–204. DOI: 10.1109/ISCA.2014.6853208.

[55] Unified Acceleration Foundation. *oneAPI Level Zero Specification*. https://oneapi-src.github.io/level-zero-spec/level-zero/latest/index.html. Accessed 2026-06-01.

[56] Ben van Werkhoven et al. “Performance Models for CPU-GPU Data Transfers”. In: *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. CCGrid ’14. IEEE, 2014, pp. 11–20. DOI: 10.1109/CCGrid.2014.16.

[57] Wm A Wulf and Sally A McKee. “Hitting the memory wall: Implications of the obvious”. In: *ACM SIGARCH computer architecture news* 23.1 (1995), pp. 20–24.

[58] Ming Yang et al. “Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems”. In: *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*. Ed. by Sebastian Altmeyer. Vol. 106. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018, 20:1–20:21. ISBN: 978-3-95977-075-0. DOI: 10.4230/LIPIcs.ECRTS.2018.20. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2018.20.