國立陽明交通大學
資訊科學與工程研究所
碩士論文

Institute of Computer Science and Engineering
National Yang Ming Chiao Tung University
Master Thesis

基於編譯器導引之紋理硬體運用的圖形處理器執行緒束內快取利用率提升技術

Improving Intra-Warp Cache Utilization on GPUs through Compiler-Guided Exploitation of Texture Hardware

研究生：賴奕廷 (Lai, Yi-Ting)
指導教授：游逸平 (You, Yi-Ping)

中華民國一百一十五年九月
September 2026

基於編譯器導引之紋理硬體運用的圖形處理器執行緒束內快取利用率提升技術

Improving Intra-Warp Cache Utilization on GPUs through Compiler-Guided Exploitation of Texture Hardware

研 究 生：賴奕廷　　Student : Lai, Yi-Ting
指導教授：游逸平　　Advisor : You, Yi-Ping

國立陽明交通大學
資訊科學與工程研究所

碩士論文

A Thesis
Submitted to Institute of Computer Science and Engineering
College of Computer Science
National Yang Ming Chiao Tung University
in partial Fulfilment of the Requirements
for the Degree of
Master of
Science

September 2026
Hsinchu, Republic of China

中華民國一百一十五年九月

誌謝

本論文歷時三餘月完成，儘管仍有諸多不盡人意之處，已盡力使內容與論述臻於完整。期許這篇論文能為後來的學子帶來一些啟發；若有朝一日能成為未來學子的普通常識，甚至有人能指出哪裡寫錯了，那就再好不過了。

謝謝嘉賢陪我討論研究瓶頸、一起看效能分析報告，也陪我從組合語言中分析編譯器的行為，一同揣摩硬體可能的運作方式。還記得那些在實驗室裡看報告，以及走在回宿舍路上的討論；這份研究，也正是在這些來回推敲的過程中逐漸精鍊成形。若我的研究有第二作者，那非你莫屬。當助教的時候，有你在也總是特別安心；你解決問題的能力，大家都有目共睹。*謝謝你，嘉賢！*

謝謝柏仲不厭其煩地聽我講研究的長篇大論，讓我在一次次梳理與討論中，釐清研究脈絡與自己的思路。還記得那些在博愛徘徊時的談話；除了研究，我們也談了許多人生哲學。這些討論讓我在研究之外，有了更多的思考與收穫，也更認識自己；要說它們比研究本身重要，或許也不為過。若不是這一年，我不會有這樣的機會與你相處。也謝謝你帶我去健身。能遇見你，是我的幸運。*謝謝你，柏仲！*

謝謝柏勛推薦各種工具與資源。雖然我對嘗試新工具向來有些保守，但這些分享也讓我得以拓展自己的邊界與視野。若不是你一直推廣大語言模型的使用，我大概也不會開始嘗試，更不會因此提高不少工作效率。也謝謝你和我聊咖啡。

謝謝悅盛陪我打桌球。你的技術實在太好了，每次和你打球都十分盡興。

謝謝游逸平老師點出圖形處理器這個研究方向，讓我得以在這個領域深入探索。也謝謝老師給予實驗室的彈性，讓我能按照自己的步調完成研究。

實驗室的大家也構成了這段時間很重要的一部分。謝謝嘉賢、信華、娟鳴，我們一起寫下屬於我們這一代人的記憶；也謝謝柏仲、柏勛、悅盛、尚奇、品文、博凱、建樺、津昱。你們各自不同的興趣與個性彼此交織，恰好形成一種自在的和諧，讓實驗室總是充滿笑聲；大家一起去吃飯的時候也總是很開心。也謝謝牧恩、霆寬、忠義、嘉揚、秉達、張軒、柏偉。你們形塑了我對這裡最初的印象，也讓我聽見了實驗室過去的故事，使剛加入實驗室的我少了許多陌生與不安。

謝謝國立陽明交通大學，讓我得以在這裡完成學業。在這裡的時光，我很開心。

最後，謝謝我自己的努力與堅持。加油吧，勇敢去闖。

大語言模型使用說明

本論文大量借助 ChatGPT 進行文字編輯與校對；惟研究動機與關鍵方法之立論，皆由本人反覆推敲而成。[1] 背景知識之整體脈絡與實驗設計亦由本人確立，並對語言模型所擴寫或生成之內容來回修訂與精鍊。文獻回顧則輔以 NotebookLM 進行彙整與事實查核。至於實作部分，亦皆由本人撰寫。[2]

誌於 丙午年夏 國立陽明交通大學

賴奕廷

[1]現在回頭看來，當時對大語言模型的使用仍較為保守；若重新進行這項研究，我會更積極地與大語言模型互動，用以發想、檢視與反覆辯證研究想法，深化思考。

[2]撰寫期間曾使用基本的程式碼自動補全功能，但在主要實作階段（2025 年初）其能力尚有限。惟後續強化之讀後寫相依分析由 GitHub Copilot 協助完成。

基於編譯器導引之紋理硬體運用的圖形處理器執行緒束內快取利用率提升技術

學生：賴奕廷　　指導教授：游逸平 博士

國立陽明交通大學 資訊科學與工程研究所

摘　　要

圖形處理器經常處理邏輯上具有多維結構，但實際儲存於線性全域記憶體中的資料。當執行緒束所產生的存取分布與此實體組織方式不一致時，單一記憶體指令可能跨越多個快取區段與快取列，使執行緒束僅使用所擷取快取資料中的一小部分。本研究將此現象描述為較低的*執行緒束內快取利用率*。此類存取不僅會產生更多快取存取波前並增加記憶體系統負載，也更加依賴後續指令重用其餘已擷取的資料。在細粒度執行緒束排程下，執行緒束間的快取競爭可能增加重用距離，使這些資料在再次使用前即遭逐出，進一步降低快取效能。既有以軟體管理的資料布局轉換雖可改善資料的實體組織方式，卻通常需要明確的布局轉換、額外的位址計算或大幅修改程式。本研究則探討是否能選擇更符合資料邏輯存取結構的實體組織方式。

本研究顯示，輝達的紋理硬體所提供之硬體管理區塊線性布局，可作為適用於特定多維存取的實用布局映射機制。透過將執行緒束所要求的資料集中於較少的快取區段與快取列，此布局能提升執行緒束內快取利用率、減少快取存取波前，並降低記憶體系統的負載，而無須由軟體進行位址轉換或增加額外的硬體支援。為自動套用此機制，本研究開發 *AutoTex* 編譯器框架，用以辨識合適的全域記憶體物件、重建其邏輯多維存取、分析其執行緒束內存取行為與物件來源，並自動轉換相應的裝置端記憶體存取與主機端記憶體管理操作。

本研究使用三套基準測試程式，於輝達的 Ada 與 Ampere 架構圖形處理器上評估 AutoTex。在所有經 AutoTex 轉換的 52 個核心函式中，其核心函式層級效能提升之幾何平均分別達 1.27× 與 1.28×，個別核心函式最高分別可達 12.80× 與 7.99×。在應用程式層級，AutoTex 於 16 個基準測試程式上的端對端效能提升幾何平均則分別達 1.20× 與 1.25×。上述結果顯示，透過編譯器導引選用既有的硬體管理布局，能在不需修改來源程式的情況下提升執行緒束內快取利用率與執行效能。

關鍵字：區塊線性布局、編譯器最佳化、執行緒束內快取利用率、記憶體存取最佳化、重用距離、紋理硬體

Improving Intra-Warp Cache Utilization on GPUs through Compiler-Guided Exploitation of Texture Hardware

Student: Yi-Ting Lai Advisor: Dr. Yi-Ping You

Institute of Computer Science and Engineering
National Yang Ming Chiao Tung University

Abstract

Graphics processing units (GPUs) frequently process logically multidimensional data structures that are stored in linear global memory. When the access distribution generated by a warp does not align with this physical organization, a single memory instruction may span many cache sectors and cache lines, causing the warp to consume only a small fraction of the cache data fetched on its behalf. We characterize this property as poor *intra-warp cache utilization*. Such accesses increase cache-access wavefront generation and memory-system pressure while relying more heavily on cross-instruction reuse of the remaining fetched data. Under fine-grained warp scheduling, inter-warp cache contention can increase reuse distances and evict these data before reuse, further reducing cache effectiveness. Existing software-managed layout transformations can improve the physical organization of data, but typically require explicit layout conversion, additional address computation, or substantial program modification. This work instead investigates whether a physical organization of data can be selected to better match the logical access structure of the computation.

We demonstrate that the hardware-managed block-linear organization provided by NVIDIA texture hardware can serve as a practical layout-mapping mechanism for suitable multidimensional accesses. By consolidating the data requested by a warp into fewer cache sectors and cache lines, this organization can improve intra-warp cache utilization, reduce cache-access wavefront generation, and alleviate pressure on the memory system without software-managed address translation or additional hardware support. To apply this mechanism transparently, we develop *AutoTex*, a compiler framework that identifies suitable global-memory objects, reconstructs their logical multidimensional accesses, analyzes their intra-warp access behavior and object provenance, and automatically transforms the corresponding device-side accesses and host-side memory management.

We evaluate AutoTex across three benchmark suites on NVIDIA Ada and Ampere GPUs. Across all 52 kernels transformed by AutoTex, the framework achieved kernel-level speedups with geometric means of 1.27× and 1.28×, respectively, with individual kernels reaching speedups of up to 12.80× and 7.99×. At the application level, across 16 benchmarks, AutoTex achieved end-to-end speedups with geometric means of 1.20× and 1.25×, respectively. These results demonstrate that compiler-guided selection of an existing hardware-managed physical layout can improve intra-warp cache utilization and execution performance while remaining transparent to the source program.

Keywords: block-linear layout, compiler optimization, intra-warp cache utilization, memory-access optimization, reuse distance, texture hardware

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Efficient memory access is critical to the performance of GPU applications, particularly for computations operating on large multidimensional data structures. Although such data naturally possess multidimensional logical organization, they are commonly stored in linear global memory, whose physical layout may not align with the access distribution generated by a warp. This mismatch can lead to inefficient use of the cache hierarchy and increased pressure on the memory system. This work investigates an alternative approach to this problem based on selecting a physical memory layout that better matches the logical access structure of the computation. In particular, we explore the hardware-managed block-linear layout provided by NVIDIA texture memory and develop AutoTex, a compiler framework that transparently maps suitable global-memory objects to this layout. This chapter motivates the problem, introduces the proposed approach, summarizes the contributions of this work, and outlines the organization of the remaining chapters.

1.1 Motivation

Graphics processing units (GPUs) achieve high computational throughput by executing large numbers of threads in parallel. On NVIDIA GPUs, threads are organized into warps whose memory accesses are processed collectively by the memory system. The efficiency of a memory instruction therefore depends not only on the amount of data requested by each thread, but also on how the addresses requested across the warp are distributed in the physical memory layout. When those addresses fall within a small number of cache sectors and cache lines, the warp can consume a large fraction of the cache data fetched on its behalf. When they are dispersed across many sectors and lines, however, each memory instruction may fetch substantially more data than the warp actually consumes.

This distinction is particularly important for applications that operate on multidimensional data. Such data are typically represented in global memory using a linear layout, in which one logical dimension is contiguous while accesses along other dimensions may have large strides. Consequently, an access pattern that is regular in the logical coordinate space can nevertheless be poorly matched to the physical organization of memory. A warp traversing a non-contiguous dimension may therefore access only a small fraction of each cache sector or cache line that it causes to be fetched. We refer to this property as poor *intra-warp cache utilization*: the threads of a warp collectively consume only a small portion of the cache data fetched on behalf of a single memory instruction.

Poor intra-warp cache utilization affects both memory-system processing and the effectiveness of the cache hierarchy. When the addresses generated by a warp span many cache sectors and cache lines, the corresponding memory instruction requires more cache-access wavefronts, increasing pressure on the memory-access pipeline and downstream memory subsystem. At the same time, data not consumed by the issuing warp can contribute only if they remain resident until accessed by subsequent instructions. Under fine-grained warp scheduling, accesses from other warps may intervene before this reuse occurs, increasing the reuse distance and the likelihood that the data are evicted first. Higher intra-warp cache utilization therefore reduces dependence on cross-instruction reuse and makes cache utilization more robust to inter-warp cache contention. As examined in detail in Chapter 3, these effects influence cache-access wavefronts, memory-system stalls, cache effectiveness, memory traffic, and ultimately execution performance.

Existing approaches address inefficient GPU memory behavior from several directions. Access-pattern transformations reorganize computation so that threads access memory in a more favorable order. Scratchpad-memory strategies explicitly stage data in programmer- or compiler-managed on-chip storage. Hardware cache-management techniques modify how cache resources are allocated or how requests are processed. Memory-layout strategies instead reorganize the physical placement of data to better match expected accesses. Each approach can be effective, but it may also require substantial program transformation, explicit data movement, additional address computation, or hardware support. In particular, software-managed blocked layouts can improve the spatial organization of multidimensional data, but require explicit layout conversion and additional address computation to map logical coordinates to the transformed physical layout.

NVIDIA GPUs already provide a different form of physical organization through texture memory. Multidimensional CUDA arrays used by texture and surface objects provide a hardware-managed block-linear layout rather than the linear layout of conventional global-memory allocations. This layout groups nearby elements across multiple logical dimensions into localized physical regions, while the hardware performs the corresponding address mapping from multidimensional coordinates. Although originally exposed as part of the texture-processing interface, this organization suggests a broader opportunity: the underlying layout can be viewed as a hardware-supported memory representation whose physical structure may better match certain multidimensional GPU access patterns.

This observation motivates the central question of this work: rather than transforming an access pattern to better match linear global memory, can the physical memory layout instead be selected to better match the access pattern? If suitable global-memory objects can be mapped to hardware-managed block-linear storage, the addresses requested by a warp may be consolidated into fewer cache sectors and cache lines. Such consolidation can increase intra-warp cache utilization, reduce cache-access wavefront generation, and

improve the effectiveness of the memory hierarchy without requiring software-managed address translation or additional hardware support.

Realizing this opportunity transparently requires more than substituting one memory API for another. A compiler must determine which data objects exhibit access patterns that can benefit from the block-linear layout, preserve correctness when multiple kernel arguments refer to the same underlying allocation, select an appropriate read-only or writable access interface, rewrite device-side memory operations, and coordinate the corresponding changes to host-side allocation, data transfer, object management, and kernel launch. These requirements motivate the compiler-guided approach developed in this work. The following section introduces the proposed layout-mapping mechanism and the AutoTex framework that automates its application.

1.2 Proposed Approach

This work addresses poor intra-warp cache utilization by changing the physical organization of data rather than restructuring the access pattern. The key idea is to map suitable multidimensional objects from the linear layout of global memory to the hardware-managed block-linear layout provided by CUDA arrays. Because this layout groups nearby elements across multiple dimensions, accesses that are widely separated under a linear representation can be placed within fewer cache sectors and cache lines. The resulting organization can increase intra-warp cache utilization, reduce cache-access wavefront generation, and alleviate pressure on the memory system. Unlike a software-managed blocked layout, the address mapping from logical multidimensional coordinates to the block-linear physical representation is performed by existing GPU hardware and does not require explicit layout-conversion data movement or software-managed address computation.

We use texture memory as the primary mechanism for accessing this block-linear storage. Although texture memory is traditionally associated with graphics-oriented texture processing, its backing CUDA arrays expose a hardware-managed physical organization that can also benefit general-purpose GPU computations. Read-only data can be accessed through texture objects, while writable data can be accessed through surface objects backed by the same underlying storage. This separation allows the block-linear layout to be applied according to the access semantics of each object rather than restricting the approach to read-only data.

To apply this mechanism automatically, we develop AutoTex, a compiler framework that identifies global-memory objects suitable for block-linear storage and transparently transforms their representation and accesses. AutoTex reconstructs the logical multidimensional structure of device-side memory accesses, analyzes their intra-warp access distribution, and determines transformation eligibility. Because multiple kernel arguments may refer to the same allocation, the framework tracks argument provenance to ensure

that all accesses use a consistent memory representation. Eligible read-only objects are mapped to texture memory, whereas writable objects are mapped to surface memory. AutoTex then rewrites the corresponding device-side accesses and coordinates the required host-side changes to allocation, data transfer, object management, and kernel launch.

We evaluate AutoTex across three benchmark suites on two NVIDIA GPU architectures. Across all 52 kernels transformed by AutoTex, the framework achieved kernel-level speedups with geometric means of 1.27× on the Ada Desktop and 1.28× on the Ampere Laptop, and individual kernels reached speedups of up to 12.80× and 7.99×, respectively. The evaluation further shows that the resulting performance improvements are associated with reductions in cache-access wavefront generation and memory-system pressure, together with changes in cache effectiveness and memory traffic. At the application level, across 16 benchmarks, AutoTex achieved end-to-end speedups with geometric means of 1.20× and 1.25× on the two platforms, respectively, with the realized benefits depending on both the contribution of accelerated kernels to total execution time and the runtime costs introduced by the transformed memory representation. These results demonstrate that compiler-guided selection of a hardware-managed physical layout can improve GPU memory behavior and execution performance without requiring source-level modification of the original program.

1.3 Contributions

The main contributions of this work are as follows:

- We identify poor intra-warp cache utilization caused by a mismatch between logical multidimensional access structure and linear physical memory organization as a target for memory-layout optimization. Higher intra-warp cache utilization allows a warp instruction to consume more of the cache data fetched on its behalf. This property reduces dependence on cross-instruction reuse and makes cache utilization more robust to inter-warp cache contention.
- We demonstrate that the hardware-managed block-linear organization provided by NVIDIA GPUs can serve as a practical layout-mapping mechanism for suitable multidimensional accesses. By consolidating the data requested by a warp into fewer cache sectors and cache lines, this organization can improve intra-warp cache utilization and reduce cache-access wavefront generation without requiring software-managed address translation or additional hardware support.
- We develop AutoTex, a compiler framework that applies this layout-mapping mechanism transparently to existing CUDA programs. AutoTex reconstructs multidimensional memory accesses, analyzes their intra-warp access distribution, preserves allocation-level consistency through provenance analysis, selects texture or surface

memory according to access semantics, and transforms both device-side accesses and host-side memory management.

We evaluate these contributions across three GPU benchmark suites and two NVIDIA GPU architectures, characterizing their effects on kernel-level and application-level performance, cache-access wavefronts, memory-system stalls, cache effectiveness, memory traffic, and runtime overhead.

The remainder of this work is organized as follows. Chapter 2 provides the architectural, programming, compilation, and profiling background required for the subsequent discussion. Chapter 3 motivates the problem through case studies, reuse-distance analysis, and experimental characterization of the resulting memory-system behavior. Chapter 4 reviews existing approaches to improving GPU memory behavior and positions the proposed approach relative to access-pattern transformations, hardware cache-management techniques, scratchpad-memory strategies, and memory-layout strategies. Chapter 5 develops texture memory as a layout-mapping mechanism, characterizes its hardware-managed block-linear organization, and analyzes its effects on cache-access wavefront generation and reuse distance. Chapter 6 presents the AutoTex compiler framework and its analysis and transformation phases, including its extension to writable data through surface memory. Chapter 7 evaluates the kernel-level and application-level performance of AutoTex and characterizes the architectural effects underlying the observed performance. Chapter 8 discusses the limitations of the proposed layout-mapping mechanism and compiler framework together with opportunities for future work. Finally, Chapter 9 summarizes the findings and broader implications of this work.

Chapter 2
Background

Graphics processing units originated as specialized accelerators for the graphics pipeline, which transforms a three-dimensional scene into a two-dimensional image. Graphics workloads expose substantial parallelism because many operations on vertices, primitives, and pixels can be performed independently with relatively limited communication among processing elements [1], [2]. Moreover, graphics applications are typically throughput-oriented and exhibit high latency tolerance [2], where performance is determined primarily by the amount of work completed over time rather than the latency of an individual operation. These characteristics motivated the development of GPU architectures that emphasize massive parallelism and hardware-supported multithreading.

The suitability of this execution model for data-parallel applications beyond graphics soon led to the emergence of general-purpose computing on GPUs [3], [4]. In response, GPU vendors progressively extended their architectures to support both graphics and general-purpose computation within a unified hardware platform [2]. This work focuses on NVIDIA GPUs and adopts NVIDIA terminology throughout. The architectural details discussed correspond to the Ampere [5] and Ada Lovelace [6] GPU architectures.

This chapter provides the architectural, programming, compilation, and profiling background required for the remainder of this work. Section 2.1 presents the hierarchical organization and execution model of modern NVIDIA GPUs. Section 2.2 then discusses texture processing within the graphics pipeline, which underlies the texture-based mechanisms explored in this work. Sections 2.3 and 2.4 describe the DRAM and cache hierarchies, respectively. Section 2.5 introduces reuse-distance analysis as a framework for reasoning about cache behavior and memory locality, while Section 2.6 presents the memory-access pipeline through which requests are processed by the cache hierarchy. Section 2.7 explains the CUDA programming model and its mapping onto GPU hardware, and Section 2.8 presents LLVM's compilation model for CUDA applications. Finally, Section 2.9 introduces the profiling tools used throughout this work to examine kernel-level hardware behavior and application-level execution.

2.1 NVIDIA GPU Hierarchy

Modern NVIDIA GPUs integrate graphics and general-purpose computing capabilities within a hierarchical hardware architecture. Many architectural components, including the streaming multiprocessors (SMs) and memory subsystem, are shared between graphics and compute workloads. Figure 2.1 illustrates the organization of an Ada Lovelace GPU

at multiple levels of the hardware hierarchy, from the device level down to individual SM sub-partitions (SMSPs).

At the highest level, shown in Figure 2.1 (a), a GPU consists of multiple graphics-processing clusters (GPCs), each containing a collection of units required for graphics processing. The GPCs share a L2 cache, which is backed by DRAM through a set of memory controllers. Depending on the execution environment, GPU memory accesses may target on-board device DRAM, peer memory attached to another GPU, or pinned system memory. Throughout this work, the term device memory refers to the GPU's on-board DRAM. Work submitted by the host processor is distributed across the GPU by the GigaThread engine.

Within each GPC, illustrated in Figure 2.1 (b), a raster engine and raster-operation (ROP) partitions support graphics-rendering operations. These units are shared by multiple texture-processing clusters (TPCs), each of which contains the hardware resources required for geometry and texture processing.

As shown in Figure 2.1 (c), each TPC comprises a PolyMorph engine and two SMs. The PolyMorph engine is responsible for geometry-processing tasks within the graphics pipeline, while the SMs execute both graphics and general-purpose compute workloads.

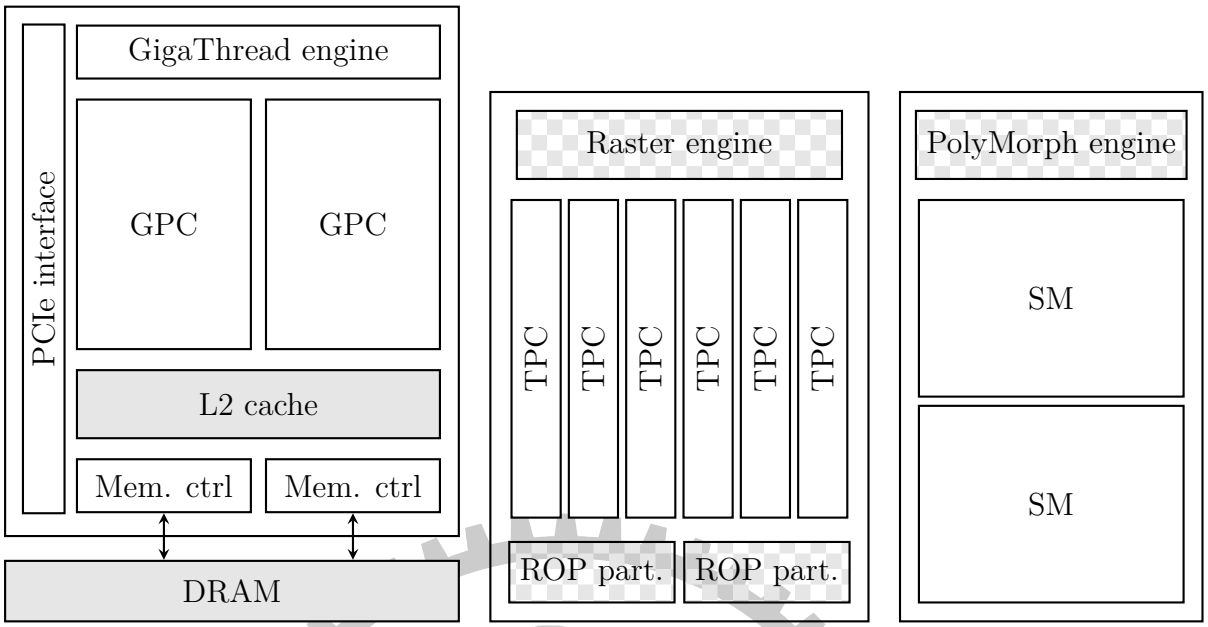
At the SM level, shown in Figure 2.1 (d), the SM is divided into multiple SM sub-partitions (SMSPs). The SMSPs share a L1 cache and a set of texture units. Texture units are specialized hardware responsible for texture-processing operations. In Ada Lovelace GPUs, each SM is additionally associated with a dedicated ray-tracing (RT) core that accelerates ray-tracing operations.

Figure 2.1 (e) illustrates the organization of an SMSP. Each SMSP contains a warp scheduler and dispatch unit that selects and issues instructions for execution. The execution resources include `FP32` and `INT32` arithmetic units,[1] load/store units (LSUs), special function units (SFUs), and tensor cores. These units share a local register file and access the memory hierarchy through the load-store subsystem. The special-function unit (SFU) accelerates transcendental operations and interpolation functions commonly used in graphics workloads, while the tensor core provides specialized support for matrix multiply-accumulate operations used in high-performance computing and machine-learning applications.

2.1.1 Parallel Execution Model

The parallel execution model describes how the GPU executes a workload to maximize throughput. A workload offloaded to the GPU is decomposed into a large number of independent *threads*. These threads are further organized into *thread blocks*, each comprising

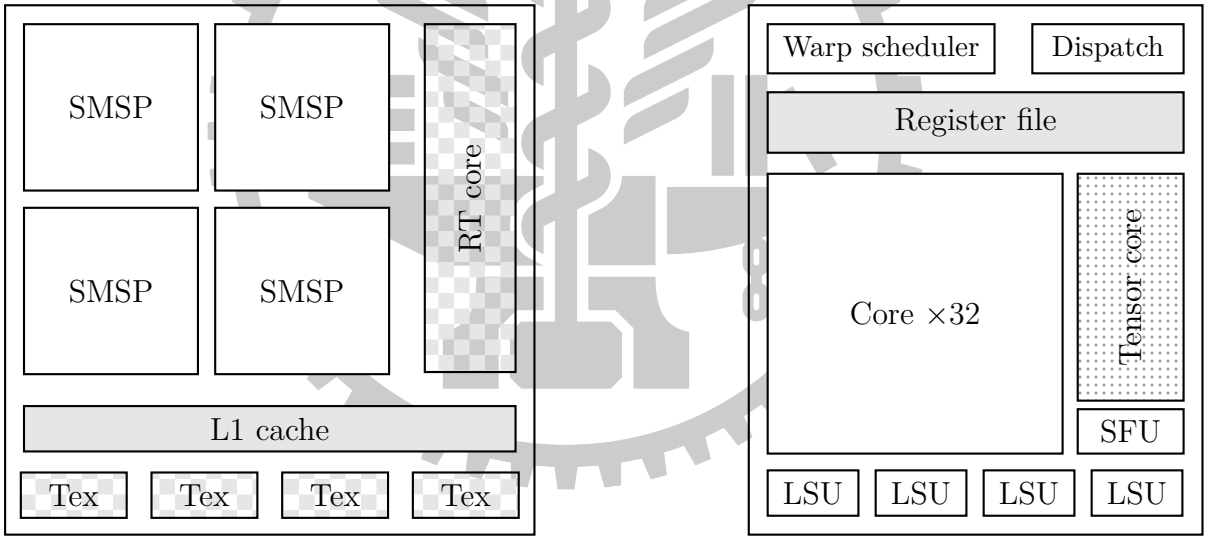
[1] `FP64` operations are supported by significantly fewer cores and therefore achieve lower throughput than `FP32` operations.



(a) Device-level organization connected to DRAM.

(b) Graphics-processing cluster (GPC).

(c) Texture-processing cluster (TPC).



(d) Streaming multiprocessor (SM).

(e) Streaming-multiprocessor sub-partition (SMSP).

Figure 2.1: Hierarchical organization of an NVIDIA Ada Lovelace GPU architecture. Gray-shaded components belong to the memory subsystem, checkerboard-patterned components are primarily associated with graphics processing, and dotted components are specialized for high-performance computing workloads. Unshaded components are shared across graphics and general-purpose computation. The quantities of architectural components shown in the GPC through SMSP subfigures correspond to those of the Ada Lovelace architecture. The device-level configuration in the first subfigure is illustrative; the number of graphics-processing clusters (GPCs) and memory controllers varies across GPU models.

a programmer-defined number of threads that execute on the same SM and can cooperate through shared memory and barrier synchronization. The GigaThread engine distributes thread blocks to the SMs, whose scheduling policy has been inferred from empirical studies to approximate a round-robin assignment [7]. Within each SM, the threads of a thread block are partitioned into groups of 32 called *warps*, which form the basic scheduling unit of the GPU.

Each SM maintains a pool of resident warps, commonly referred to as the *warp pool*. Warps residing in the warp pool are called *active warps*. If the number of thread blocks in the workload exceeds the combined capacity of all SMs, the remaining thread blocks are temporarily retained by the GigaThread engine and dispatched only after active thread blocks complete execution and release their resources. Registers are allocated for every thread, while shared memory is allocated per thread block. These resources remain reserved throughout the lifetime of the resident thread block. Because both resources are limited, increasing the register or shared-memory requirements reduces the number of warps that can simultaneously reside on an SM. The ratio between the achieved number of active warps and the architectural maximum is referred to as the *occupancy*.

A warp whose next instruction can be issued without waiting for data or execution dependencies is called an *eligible warp*. Each warp scheduler selects one eligible warp every cycle, if available. The selected warp is then forwarded to the dispatch unit, which issues its next instruction to the appropriate execution pipeline. By issuing instructions from different eligible warps in successive cycles, the GPU implements fine-grained warp scheduling, effectively realizing fine-grained multithreading [8, Sect. 3.11] at warp granularity and overlapping the execution of many independent warps.

This scheduling mechanism enables the hardware to hide execution latency by switching to other eligible warps whenever a warp stalls. Because instructions are issued in program order, however, the compiler also plays an important role in exposing instruction-level parallelism (ILP). It statically interleaves independent instructions to reduce dependency stalls, particularly for operations with fixed execution latencies. Together, hardware warp scheduling and compiler-generated ILP constitute the two complementary levels of latency hiding employed by modern GPUs [9].

SIMT Execution Model

Within a warp, all 32 threads execute the same instruction, which the dispatch unit issues simultaneously to the selected warp. At the hardware level, the execution units can therefore be viewed as SIMD lanes [10], each operating on the state of a corresponding thread. Unlike conventional SIMD execution, however, CUDA exposes independent threads rather than explicit vector lanes and execution masks to the programmer.

NVIDIA realizes this abstraction through a *single-instruction multiple-thread* (SIMT) execution model. The hardware dynamically detects control-flow divergence among threads

within a warp and executes divergent paths using predicate masks, allowing each thread to follow its own control flow while preserving the single-thread programming abstraction. Divergent paths are serialized and therefore require additional instruction issues, reducing execution efficiency. Thus, although SIMT provides greater programming flexibility than conventional SIMD execution, branch divergence remains an important source of performance degradation.

Because all threads within a warp execute the same instruction, memory instructions are likewise issued at a per-warp basis. As a result, the addresses generated by the threads of a warp are presented collectively to the memory subsystem, making their physical organization a key determinant of memory-access efficiency.

Tensor Hardware and Asynchronous Execution Model

Since the introduction of tensor cores in the Volta architecture and their subsequent evolution in later generations, the performance gap between computation and data movement has continued to widen. Tensor cores are specialized processing units designed to accelerate matrix multiply-accumulate operations.[2] In the Hopper architecture, NVIDIA further introduced the Tensor Memory Accelerator (TMA), a dedicated copy engine that performs bulk data transfers between global memory and shared memory.

Both tensor cores and TMA operate asynchronously with respect to the conventional warp instructions executed on CUDA cores. Consequently, explicit synchronization mechanisms are required to correctly coordinate data movement and computation. Because a small number of instructions can dispatch substantial amounts of work to these dedicated hardware units, the reliance on ILP for achieving high utilization is reduced.

At the same time, AI workloads, particularly general matrix multiplication, exhibit substantial data reuse that must be exploited to achieve optimal performance. Such kernels typically allocate large shared-memory buffers and many registers to retain reusable data on-chip. However, the resulting increase in resource consumption reduces thread-block residency, thereby limiting the number of warps available to the scheduler for latency hiding.

These architectural and workload trends have driven GPU programming beyond the traditional SIMT execution model toward increasingly sophisticated asynchronous execution models. One prominent example is warp specialization, a producer-consumer execution strategy in which one subset of warps is responsible for initiating asynchronous data transfers while the remaining warps perform computation. Because neither compiler optimizations nor hardware scheduling alone can adequately hide memory latency in this setting, multi-stage software pipelining is commonly employed to overlap computation with asynchronous data movement.

[2] $D = A \times B + C$

Tensor cores, TMA, and the associated asynchronous programming techniques are beyond the scope of this work. The analyses, concepts, and optimization techniques presented in the remainder of this work apply exclusively to conventional CUDA-core execution and do not consider tensor-core execution, TMA operations, or asynchronous execution models.

2.1.2 Memory Hierarchy

Figure 2.1 also highlights the organization of the GPU memory hierarchy. Registers reside within the SM and provide private storage for individual threads. The L1 cache is located on-chip and is shared among the SMSPs within an SM, while the L2 cache is shared across the entire GPU and serves as the last on-chip cache before accesses reach device DRAM. Together, these components form a hierarchical memory system that balances capacity, latency, and bandwidth requirements. Different memory spaces and hardware access paths, including the conventional load-store path and the texture subsystem, are provided to support diverse workload requirements. The DRAM-backed memory spaces are described in Section 2.3, while the cache hierarchy is discussed in Section 2.4.

2.2 Graphics Pipeline and Texture Hardware

Although modern GPUs support programmable shaders and general-purpose computation, graphics rendering remains a fundamental workload. Figure 2.2 illustrates the major stages of a simplified graphics pipeline [11, Fig. 8.1], [12], [13].

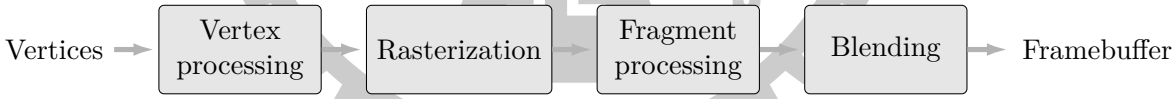


Figure 2.2: Simplified four-stage graphics pipeline.

The pipeline begins with vertex processing, where geometric vertices and their associated attributes, such as colors, normals, and texture coordinates, are transformed into screen-space representations. The resulting vertices are assembled into geometric primitives, typically triangles, and clipped against the viewing volume.

The rasterization stage then converts each primitive into a collection of screen-space fragments while interpolating vertex attributes across the primitive. These interpolated attributes are subsequently consumed by the fragment-processing stage, where shading computations determine the appearance of each fragment.

Finally, the various fragments corresponding to each pixel are combined in the blending stage before the resulting pixel values are written to the framebuffer.

In this work, we are interested in texture processing and its interaction with the memory hierarchy. Therefore, we focus on the stages of the graphics pipeline that in-

volve texture operations. Texture processing primarily occurs during fragment processing. Given the texture coordinates interpolated during rasterization, fragments perform texture sampling to obtain values from texture maps. This process includes level-of-detail selection, mipmap access, and texture filtering to reconstruct the texture values used for shading. The following subsections introduce the fundamental concepts of texture processing in computer graphics, including texture mapping and texture sampling, before discussing the hardware mechanisms that support these operations.

2.2.1 Texture Processing

A texture map, or simply a texture, is an image projected onto a three-dimensional object's surface to enhance its visual appearance. A texture consists of individual elements called *texels* (texture elements), which store the color and material information used to reproduce surface details such as patterns, reflections, and shadows.

Applying a texture to a surface requires establishing a correspondence between points on the surface and locations within the texture image. Let S denote a surface embedded in three-dimensional space and T denote the two-dimensional texture domain. The texture mapping function ϕ is defined as

$$\begin{aligned} \phi : S &\to T, \\ (x, y, z) &\mapsto (u, v), \end{aligned}$$

where (x, y, z) are spatial coordinates on the surface and (u, v) are texture coordinates within the texture domain [11, Sect. 11.1].

During rasterization, texture coordinates associated with the vertices of a primitive are interpolated to produce a texture coordinate (u, v) for each generated fragment.

These coordinates are subsequently used for texture sampling, which retrieves the corresponding texture value from the texture map. Because (u, v) generally does not coincide with a discrete texel location, texture sampling reconstructs a continuous value from neighboring texels through addressing and filtering operations.

To support rendering across a wide range of viewing distances and screen resolutions, textures are typically stored in a hierarchy of progressively lower-resolution images known as a *mipmap*. The highest-resolution image is referred to as the base level, while each subsequent level halves the dimensions of the previous one [11, Sect. 11.3.3], [14]. For example, a 1024×1024 texture is followed by a 512×512 level, then a 256×256 level, and so forth. During rasterization, texture-coordinate gradients are computed to estimate the appropriate level of detail (LOD) for subsequent texture sampling. The selected LOD determines the mipmap level used during sampling, allowing distant surfaces to be sampled from lower-resolution representations and thereby reducing aliasing artifacts.

Texture filtering determines how texels are combined during texture sampling. Because the area represented by a fragment rarely corresponds exactly to a single texel, the sampled value is typically reconstructed from multiple neighboring texels and, in some cases, multiple mipmap levels. For example, trilinear filtering blends samples from two adjacent mipmap levels to provide smooth transitions between LODs [14].

The efficiency of texture sampling depends not only on the sampling and filtering algorithms, but also on how texture data are organized and accessed in memory.

2.2.2 Texture Unit

Texture units, shown in Figure 2.1 (d), are specialized hardware units responsible for texture processing. They are designed to efficiently perform the address generation, memory access, and filtering operations required to reconstruct texture values from discrete texels. In the graphics pipeline, texture units operate during fragment processing, where interpolated texture coordinates generated during rasterization are used to retrieve and filter texture data.

Texture units support these operations through a dedicated processing pipeline [15], illustrated in Figure 2.3. Texture requests issued by the SM first enter the texture-input (TEXIN) stage, which retrieves texture metadata such as dimensionality, texel format, and sampling parameters from the texture descriptor. The request is then forwarded to the LOD stage.

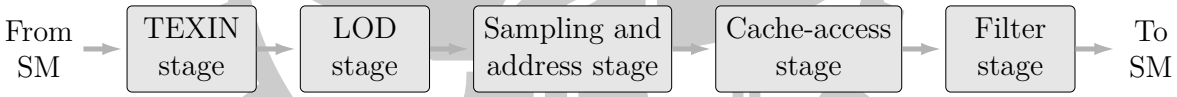


Figure 2.3: Hardware implementation of texture processing within a texture unit. The pipeline performs level-of-detail selection, texture sampling, memory access, and filtering to reconstruct texture values from discrete texels.

The LOD stage determines the appropriate mipmap level based on the spatial variation of texture coordinates. In graphics applications, texture coordinates are processed in groups of 2×2 fragments, known as quads, allowing the hardware to estimate texture-coordinate gradients with respect to screen-space coordinates. These gradients are subsequently used to select an appropriate mipmap level for texture sampling.

After LOD selection, the sampling and address stage computes the memory locations of the required texels. This stage also applies addressing modes such as clamping and wrapping when texture coordinates fall outside the texture boundaries. The resulting addresses are then forwarded to the cache-access stage.

The cache-access stage retrieves the requested texels from the memory hierarchy. Texture data may be supplied directly from the texture cache on a hit or fetched from lower levels of the memory hierarchy on a miss. Once the required texels have been obtained, the

filter stage performs the requested filtering operation, such as nearest-neighbor, bilinear, or trilinear filtering, to reconstruct the final texture value.

Finally, the filtered result is returned to the SM for subsequent fragment-shading computations. Through this dedicated processing pipeline, texture units provide hardware support for texture processing, enabling efficient texture sampling and filtering within the graphics pipeline. The storage and organization of texture data in memory are discussed in Section 2.3.3.

2.3 Device Memory and Memory Layout

Device memory corresponds to the DRAM attached to the GPU, as illustrated in Figure 2.1 (a). Although physically implemented by the same DRAM subsystem, device memory is exposed through multiple memory spaces that differ in their visibility, access semantics, and hardware access paths. These memory spaces therefore provide different views of the same underlying storage and are distinguished primarily by how the hardware interprets and accesses the data.

This section introduces the memory spaces relevant to this work, namely global memory and texture memory. Particular attention is given to the memory layouts employed by these spaces, as the organization of data in physical memory plays a central role in determining spatial locality and memory-system efficiency.

2.3.1 Memory Layout

A memory layout defines the mapping between the logical organization of data and its physical placement in memory. While application data are often represented as multidimensional structures, physical memory is inherently one-dimensional and addressed through a linear address space. Consequently, a layout function is required to translate logical coordinates into physical addresses. Let (x, y) denote the coordinates of a two-dimensional logical data structure and let i denote the corresponding physical address. A memory layout M is defined as

$$M : (x, y) \mapsto i.$$

Different layout functions preserve different notions of locality and therefore influence how effectively the memory hierarchy can exploit spatially related accesses.

2.3.2 Global Memory with Linear Layout

Global memory is the general-purpose memory space used for ordinary data storage. Data stored in global memory are organized using a *linear layout*, typically in either row-major or column-major order. Without loss of generality, this work assumes row-major order.

For a two-dimensional data structure of width W, the row-major linear layout M_{L} is defined as

$$M_{\mathrm{L}}^{(W)}(x, y) = yW + x.$$

The resulting organization is illustrated in Figure 2.4. Under this mapping, the elements of each logical row are placed consecutively in physical memory, while successive rows are separated by a stride of W elements. Row-wise traversals therefore tend to access contiguous memory locations, whereas column-wise traversals generate larger address strides.

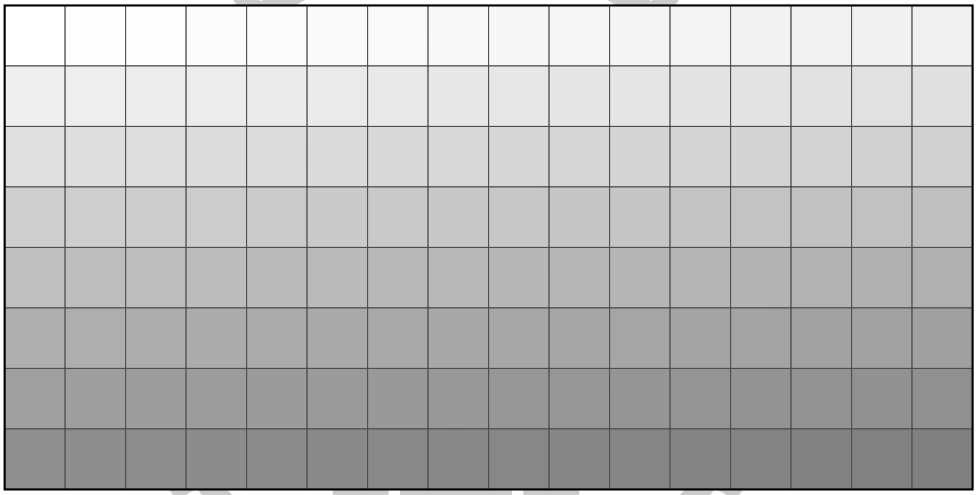


Figure 2.4: Linear layout of a 16×8 two-dimensional data structure. Each cell represents a data element, and the grayscale gradient indicates the order of elements in physical memory, from the first element (lightest) to the last element (darkest).

The layout function M_{L} is realized through address computations generated by the compiler [16, Sect. 5.1], [17, Sect. 6.4.3]. Specifically, memory accesses expressed in terms of logical coordinates are translated into linearized addresses according to the layout function, with the required computations performed either at compile time or at runtime depending on whether the dimensions and coordinates are known statically. The hardware operates only on the resulting linear addresses and does not need to interpret the logical organization of the underlying data.

Linear layouts are particularly effective when the dominant access pattern follows the dimension along which data are stored contiguously. In such cases, logically adjacent elements remain physically adjacent, allowing the memory hierarchy to efficiently exploit spatial locality through coalescing and cache-line transfers.

Global-Memory Access Coalescing

A global-memory instruction is issued collectively by the active threads of a warp and may generate one address for each participating thread. The hardware groups these thread-level accesses according to the memory regions containing the requested data, a process generally referred to as *access coalescing* [18]. Accesses concentrated within fewer memory requests are considered more highly coalesced, whereas accesses requiring many separate requests are considered less coalesced or *uncoalesced*.

The degree of coalescing depends on the spatial distribution of the addresses generated by the warp. In the worst case, the participating threads access distinct transfer units, potentially requiring one request per thread. When multiple addresses fall within the same transfer unit, however, one request can represent the accesses of several threads [19, Sect. 2.3.4.1]. The accessed elements need not reside in strictly consecutive memory locations or be accessed by the participating threads in any particular order for coalescing to occur.

Global-memory access coalescing is governed by the transfer granularities supported by the memory hierarchy. The sector-cache organization described in Section 2.4 uses a 128-byte cache line divided into four 32-byte sectors. Memory requests may therefore cover a single 32-byte sector, two adjacent sectors, or an entire 128-byte cache line, corresponding to 32-, 64-, and 128-byte transfers, respectively. Accesses represented by fewer sectors or cache lines require fewer requests, reduce the amount of cache processing, and limit the potential traffic toward the L2 cache and device memory. Conversely, accesses distributed across many sectors or cache lines generate more requests, consume more pipeline capacity, and may require more lower-level memory transfers.

Access coalescing is performed transparently by the hardware. It reduces the number of requests generated from a warp instruction but does not imply that each resulting request can be accepted in its entirety by a subsequent memory-pipeline stage in a single cycle. The partitioning into wavefronts and processing of these requests are discussed in Section 2.6.

2.3.3 Texture Memory with Block-Linear Layout

Texture memory is a memory space designed to support texture processing (Section 2.2.1). Unlike data in global memory, texture data may be accessed from a wide range of orientations with respect to the underlying memory layout. The same texture may be traversed horizontally, vertically, diagonally, or at varying levels of detail depending on the viewing direction and distance [14]. Consequently, a linear layout that favors a single traversal orientation is insufficient. To provide locality that is less sensitive to access orientation, texture memory employs a *block-linear layout* [20].

In a block-linear layout, texels belonging to a mipmap level are partitioned into a hierarchy of blocks. Each block consists of one or more groups of bytes (GOBs), which serve as the fundamental units of physical organization. The dimensions of a block are adjusted according to the size of the individual mipmap level. This adjustment is necessary because the dimensions of a mipmap level do not generally align with an integer number of fixed-size blocks, particularly for lower-resolution mipmap levels. By adapting the block dimensions, the layout reduces the amount of padding and unused storage that would otherwise arise while preserving locality. The resulting hierarchical organization is illustrated in Figure 2.5.

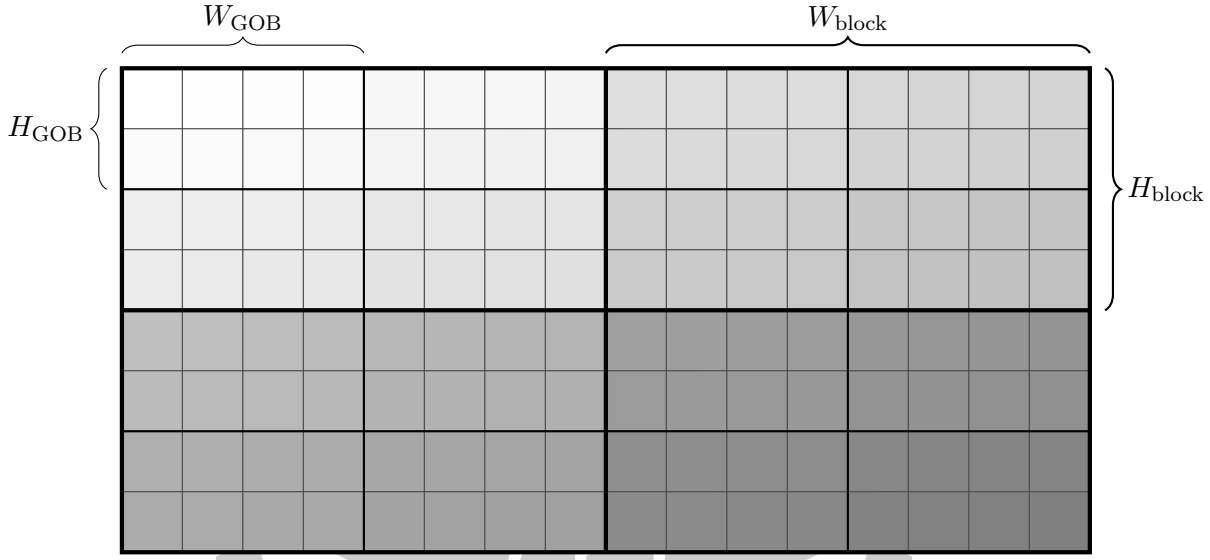


Figure 2.5: Example block-linear layout of a 16×8 two-dimensional mipmap level. Each cell represents a texel, and the grayscale gradient indicates the order of texels in physical memory, from the first texel (lightest) to the last texel (darkest). In this illustrative example, texels are first grouped into GOBs of dimensions $W_{\text{GOB}} \times H_{\text{GOB}} = 4 \times 2$ texels. GOBs are subsequently grouped into blocks of dimensions $W_{\text{block}} \times H_{\text{block}} = 2 \times 2$ GOBs, corresponding to 8×4 texels per block.

As shown in Figure 2.5, each block contains one or more GOBs, and each GOB contains a fixed-size region of texels. Within a GOB, texels are arranged according to an XYZ traversal order, in which neighboring texels along the x, y, and z dimensions are placed close together in physical memory. Likewise, GOBs within a block and blocks within a mipmap level are organized using the same principle. As a result, texels that are spatially nearby in multiple dimensions tend to remain physically nearby in memory.

Address translation in a block-linear layout therefore differs from that of a linear layout. Rather than directly linearizing texture coordinates, the hardware first identifies the block containing the requested texel, then the GOB within that block, and finally the texel position within the GOB. Let W_{GOB} and H_{GOB} denote the width and height of a GOB in texels, and let W_{block} and H_{block} denote the width and height of a block in units

of GOBs. For a mipmap level of width W, the block-linear layout function M_{BL} can then be expressed as

$$
\begin{aligned}
&M_{\mathrm{BL}}^{(W,W_{\mathrm{GOB}},H_{\mathrm{GOB}},W_{\mathrm{block}},H_{\mathrm{block}})}(x,y) \\
&= W_{\mathrm{block}}H_{\mathrm{block}}W_{\mathrm{GOB}}H_{\mathrm{GOB}}\left(\left\lfloor\frac{y}{H_{\mathrm{block}}H_{\mathrm{GOB}}}\right\rfloor\frac{W}{W_{\mathrm{block}}W_{\mathrm{GOB}}}+\left\lfloor\frac{x}{W_{\mathrm{block}}W_{\mathrm{GOB}}}\right\rfloor\right) && (2.1)\\
&+ W_{\mathrm{GOB}}H_{\mathrm{GOB}}\left(\left\lfloor\frac{y \bmod (H_{\mathrm{block}}H_{\mathrm{GOB}})}{H_{\mathrm{GOB}}}\right\rfloor W_{\mathrm{block}}+\left\lfloor\frac{x \bmod (W_{\mathrm{block}}W_{\mathrm{GOB}})}{W_{\mathrm{GOB}}}\right\rfloor\right) && (2.2)\\
&+ (y \bmod H_{\mathrm{GOB}})W_{\mathrm{GOB}} + (x \bmod W_{\mathrm{GOB}}). && (2.3)
\end{aligned}
$$

The physical address is obtained hierarchically by first computing the offset of the block, comprising $W_{\mathrm{block}} \times H_{\mathrm{block}}$ GOBs (Equation (2.1)), then the offset of the GOB within that block (Equation (2.2)), and finally the offset of the texel within the selected GOB (Equation (2.3)). This hierarchical translation enables the layout to preserve locality across multiple dimensions without changing the logical texture representation.

Unlike the linear-layout function, which is realized through compiler-generated address computations, the exact block-linear layout function M_{BL} is implemented in hardware (Section 2.2.2) and is not publicly documented. To efficiently compute texel addresses, GOB and block dimensions are typically powers of two, allowing division and modulus operations to be replaced by bitwise shifts and masks.

A key characteristic of block-linear layouts is that they provide more uniform locality when the dominant access orientation cannot be predicted in advance. Compared with linear layouts, which favor a single traversal direction, block-linear layouts reduce the average distance between spatially related texels in memory and generally incur fewer page crossings for arbitrarily oriented access patterns. They therefore improve cache utilization and memory-access efficiency for texture-processing workloads, albeit at the cost of reduced locality for any particular traversal orientation.

Texture and Surface Interfaces

The block-linear storage is exposed through two distinct memory interfaces. Texture memory provides a read-only interface intended for texture processing, whereas surface memory provides a writable interface to the same underlying storage. Consequently, both interfaces benefit from the locality-preserving properties of the block-linear layout.

Although both interfaces access the same underlying storage, only the texture interface supports texture-processing semantics, including configurable addressing modes, coordinate normalization, level-of-detail selection, and texture filtering. The surface interface instead provides direct element accesses without these capabilities.

Surface memory supports writes, but its reads may be satisfied from cached copies that are not coherent with writes performed during the same kernel invocation. As a

result, the surface interface does not guarantee read-after-write coherence, and a surface read following a surface write is not guaranteed to observe the updated value.

2.4 Cache Hierarchy

Caches are built from SRAM, have smaller capacities than backing memory, and are physically located closer to the compute units, providing latency roughly an order of magnitude lower than that of the next level [21]. This work focuses on the two-level data cache hierarchy for NVIDIA architectures, as illustrated in Figure 2.1.

Both L1 and L2 caches are sectored caches with 128-byte cache lines, each subdivided into four 32-byte sectors [22], [23]. The cache line structure is shown in Figure 2.6. These lines are aligned to 128-byte boundaries, mapping consecutive memory addresses to sequential sectors to preserve the physical layout of device memory until the line boundary is crossed. Both cache levels are believed to behave similarly to set-associative caches, as observed in microbenchmarking [21], [24], [25], [26], [27], featuring complex address-to-set mapping and replacement policies that deviate from standard least-recently-used (LRU) algorithms. However, NVIDIA does not disclose these details.

This chapter examines these cache properties in detail and evaluates how miss rates can be modeled using reuse-distance analysis.

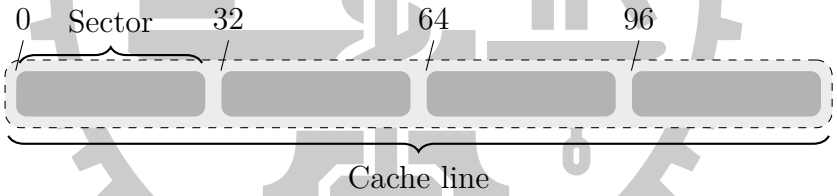


Figure 2.6: Cache-line structure of a sector cache. The numbers denote the byte offsets of sectors within the cache line.

2.4.1 L2 Cache

The L2 cache serves as the GPU's shared last-level cache and maintains data coherency across all SMs. It can retain data across kernel launches, facilitating inter-kernel data reuse. Modern architectures provide persistence controls that allow a specific cache portion to be reserved for persistent data. This mechanism mitigates cache contention by adjusting eviction priorities between persistent and streaming data streams [19, Sect. 4.13], [28, Sect. 9.7.9.8].

The L2 cache is physically partitioned into L2 slices (LTSs) distributed across multiple memory partitions (MPs). Each MP contains multiple LTSs and a memory controller responsible for device memory communication. SMs communicate with MPs through

a network-on-chip (NoC) interconnect. Consequently, L2 cache has non-uniform access latency across partitions due to interconnect differences [29]. The pipeline organization of an LTS is discussed in Section 2.6.

Transfer Granularity and Write Policy

The L2 cache maintains cache lines as the unit of allocation, replacement, and writeback, but supports sector-granular data transfers for servicing requests and cache fills [27]. Upon a cache miss, the corresponding cache line is allocated in the L2 cache, but by default only the requested 32-byte sector is fetched from device memory. This behavior is configurable via low-level cache operators that can increase the transfer granularity by prefetching multiple sectors or dual cache lines [28, Sect. 9.7.9.8].

The L2 cache employs a write-back policy with write allocation by default, functioning as a write buffer that coalesces data before performing bulk transfers to device memory [27], [30]. The architecture also supports cache operators that modify the default behavior, including a write-through scheme that immediately propagates updates to device memory [28, Sect. 9.7.9.11], [31].

2.4.2 Unified L1 Cache

The L1 cache is implemented as a unified on-chip storage structure whose capacity is partitioned between a hardware-managed data cache and a software-managed shared memory [32], [33]. This structure is private to each SM[3]; consequently, data cached in one SM's L1 cache are not required to be coherent with the L1 caches of other SMs. The pipeline organization of the L1 cache is discussed in Section 2.6.

L1 Data Cache

The L1 data cache employs the same cache-line and sector granularities as the L2 cache but cannot perform hardware prefetching. Because coherence is not maintained among L1 caches on different SMs, global-memory stores use a write-through policy. By default, stores still allocate cache lines in the L1 cache (i.e., write allocation) [27], although cache operators can modify this behavior [28, Sect. 9.7.9.8].

The L1 data cache is shared by global-memory, texture-memory, and surface-memory accesses. Consequently, texture and surface requests are serviced by the same cache structure as ordinary global-memory requests. Any differences in observed locality therefore arise primarily from the physical organization of texture data, particularly the block-linear layout (Section 2.3.3), rather than from a dedicated L1 cache for texture memory [15].

[3]Modern GPU architectures allow communication and data exchange with the shared memory of other SMs within the same graphics processing cluster, which consists of a group of SMs.

Shared Memory

Shared memory is a software-managed portion of the unified L1 structure.[4] Unlike the L1 data cache, whose contents are accessed through tag lookup and managed by hardware replacement policies, data stored in shared memory persist until they are explicitly overwritten by software.

The shared-memory portion inherits the bank organization of the underlying SRAM structure. It is organized into 32 banks, each capable of serving a 4-byte access at a time. Consequently, requests targeting different banks can be serviced simultaneously. Because shared memory bypasses the tag-lookup mechanism used by the cache hierarchy, accesses are mapped directly onto the bank structure. This organization effectively exposes a two-dimensional view of the underlying SRAM, with a row width of $4 \times 32 = 128$ bytes.

A bank conflict occurs when two or more addresses within the same request access different locations in the same bank. When such a conflict is detected, the request cannot be serviced in a single pass. Instead, the conflict resolver partitions the request into non-conflicting subsets. The non-conflicting portions are serviced immediately, whereas the conflicting portions are deferred and replayed in subsequent passes until all accesses have been completed [34]. Consequently, bank conflicts increase the effective latency of shared-memory accesses and reduce the achievable throughput.

2.5 Cache Analysis and Reuse Distance

Cache miss rate is an important performance metric that reflects a cache's ability to retain data for reuse. Miss rates depend not only on cache structures, such as capacity and associativity, but also on runtime access behavior. The 3Cs model classifies cache misses into three categories: *compulsory*, *capacity*, and *conflict* misses [35], [36, Sect. 5.3], [8, Sect. 2.1].

Under the cache organization considered in this work, compulsory misses occur when a cache sector is accessed for the first time and must therefore be fetched. Capacity misses occur when the finite cache cannot retain all cache lines needed for subsequent reuse, whereas conflict misses arise when limited associativity causes cache lines mapping to the same set to evict one another despite sufficient cache capacity elsewhere. The remainder of this section introduces the analytical concepts and modeling approaches used to reason about these cache misses.

2.5.1 Reuse-Distance Analysis

A common approach to cache analysis traces the sequence of memory accesses. For each datum, the number of distinct accesses between two consecutive accesses to the

[4] The capacity available to the L1 data cache decreases as more storage is allocated to shared memory.

same datum is defined as the *reuse distance* [37]. In a non-prefetch cache, a compulsory miss occurs when no prior access exists, as indicated by an infinity reuse distance. In a fully-associative LRU cache, a capacity miss occurs if and only if the reuse distance is not below the cache capacity. An N-way associative LRU cache can be modeled as N smaller caches, where reuse distance is measured only among accesses mapped to the same cache set. In set-associative caches, conflict misses are often intertwined with capacity misses because increasing cache capacity may also increase associativity or the number of sets [36, Sect. 5.3]. Consequently, conventional reuse-distance analysis typically does not distinguish between conflict and capacity misses.

In practice, caches perform implicit prefetching because the transfer granularity is larger than a single datum. In sector caches, the transfer unit is a sector; therefore, reuse distance is measured in terms of sector accesses rather than datum accesses. Additional reuse captured at this coarser granularity is referred to as spatial reuse. For sector caches that fetch at sector granularity but evict at cache-line granularity, two forms of reuse distance must be maintained to correctly characterize different miss types: compulsory misses are determined at the sector granularity, whereas capacity and conflict misses are determined at the cache-line granularity.

For regular access patterns, cache behavior can often be approximated using a repeating access interval. Miss rates are then estimated from the reuse-distance distribution within that interval, which serves as a representative steady-state behavior. Consider a single-cache-line sector cache configured with two sectors, each containing two elements. The cache fetches data at sector granularity while performing eviction at cache-line granularity. The analyzed access pattern alternates sequentially between two arrays, A and B, both aligned to cache-line boundaries. In the i-th iteration, elements `A[4i]`, `A[4i+1]`, and `A[4i+2]` are accessed, followed by `B[4i+2]`, after which element `A[4i+3]` is accessed. Figure 2.7 illustrates the first ten accesses corresponding to the initial two iterations. Figure 2.8 presents the resulting reuse distances at both sector and cache-line granularities, together with the interval boundaries. Each interval consists of five element accesses. The resulting access classifications are summarized in Table 2.1. In this example, the access to `A[2]` is classified as a compulsory miss because its sector-granularity reuse distance is infinite, whereas the access to `A[3]` is classified as a capacity miss because its cache-line-granularity reuse distance exceeds the number of cache lines. Based on this reuse-distance analysis, the resulting miss rate of the access pattern is 20%.

Table 2.1: Classification of memory access outcomes within Interval 1. Abbreviations denote compulsory (comp.) and capacity (cap.) cache misses.

Access	`A[0]`	`A[1]`	`A[2]`	`B[2]`	`A[3]`
Cache hit/miss	Miss (comp.)	Hit	Miss (comp.)	Miss (comp.)	Miss (cap.)

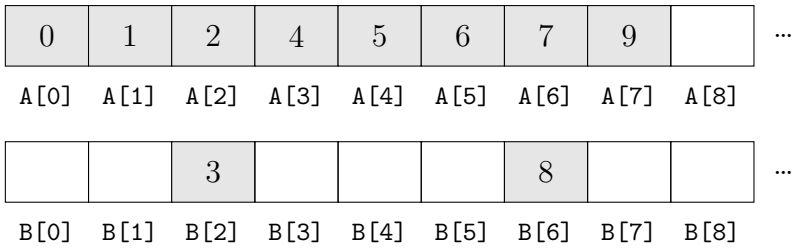


Figure 2.7: Interleaved access pattern across arrays A and B in the first two iterations ($i = 0, 1$). Shaded cells represent the active memory footprint and are annotated with their sequential access order from 0 to 9.

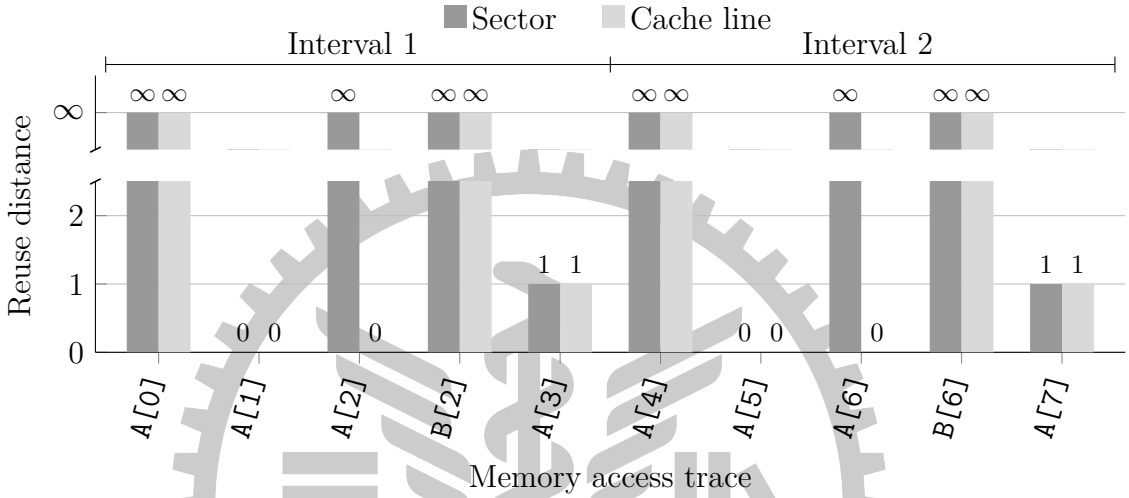


Figure 2.8: Reuse distance profiles at sector and cache-line granularities for the initial ten memory accesses of the trace. Shaded bar pairs contrast the tracking differences, while horizontal indicators demarcate the access boundaries for intervals across arrays A and B.

2.5.2 Reuse-Distance Analysis on GPUs

Reuse-distance analysis, however, requires an ordered memory-access trace, which is non-trivial to obtain in the presence of concurrent warp execution on GPUs. In the GPU execution model, all warps within an SM share the same L1 cache, and the ordering of memory accesses can be affected by various stall events. For example, under a *greedy-then-oldest* (GTO) warp-scheduling policy [38], a dependency stall causes the currently executing warp to yield execution resources to the oldest eligible warp. Warp execution may also stall due to exhaustion of *miss-status holding registers* (MSHRs), which track in-flight memory requests. Each MSHR entry corresponds to a unique cache-line request, while requests targeting the same cache line are merged into a single entry. Both the number of MSHR entries and the number of supported merges are limited [25], [39], [40]. Furthermore, although the instruction queue may continue to accept instructions from scheduled warps, its capacity is also finite [41]. Consequently, accurate reuse-distance analysis for GPUs requires modeling both the warp execution behavior and the associated stall events.

2.5.3 GPU Cache Modeling Approaches

Tang et al. [42] model a set-associative L1 cache with an LRU replacement policy under the simplifying assumption that only one warp from a thread block executes at a time. Instructions from different warps are interleaved in a *round-robin* (RR) manner, followed by warps from other thread blocks, thereby enforcing uniform progress among all warps. They formulate inter-thread access sequences using augmented access vectors [43]. However, the RR warp-scheduling assumption deviates from actual GPU behavior, where multiple warps may execute in parallel.

Nugteren et al. [25] extend the model by incorporating non-uniform warp progress caused by memory latency and MSHR exhaustion. Eligible warps are maintained in a first-in first-out queue, and a stalled warp is returned to the back of the queue only after the corresponding latency elapses. In addition, after accounting for coalescing behavior (Section 2.3.2), memory requests within a warp are issued according to thread index order. Nevertheless, their model still relies on RR warp scheduling, which may produce inaccurate inter-warp ordering.

Kiani et al. [40] further incorporate block reservation failures as a source of stalls. Such failures occur when all cache blocks within a set are already reserved by in-flight misses, and a new access attempts to reserve an additional block in the same set. Their work considers both RR and GTO warp-scheduling policies. Parallel warp execution is modeled by interleaving requests into a sequential trace, and the L2 cache is modeled analogously to L1 cache misses.

Huang et al. [44] propose GPUMech, which shows that a carefully selected representative warp can approximate the behavior of other warps with high accuracy. To reduce the impact of varying degrees of control-flow divergence, GPUMech employs the k-means clustering algorithm to identify representative warps for modeling.

Wang et al. [45] present the Memory Divergence Model (MDM), which attributes prolonged stall durations in divergent workloads to poor spatial locality and high memory-request volume, leading to congestion in the NoC and DRAM subsystems. Lee et al. [23] further develop the GPU Core Model (GCoM) and argue that structural hazards in functional units and cache resources also contribute significantly to stall behavior.

2.6 Memory-Access Pipeline

When a memory instruction is issued, the resulting request traverses a sequence of processing pipelines that connect the SM to the memory hierarchy. Figure 2.9 presents the high-level model used to describe this path, based on the memory-pipeline model introduced by NVIDIA [22, Sect. 2.3.1] and extended with the additional components discussed throughout this section. The figure should be interpreted as a conceptual model rather

than a literal microarchitectural representation: an implementation may contain multiple parallel pipelines, and each illustrated component may itself comprise several processing stages. This section focuses on how the accesses generated by a warp are routed, organized, and processed through the memory-access pipeline, with particular emphasis on the shared L1 cache-processing stages.

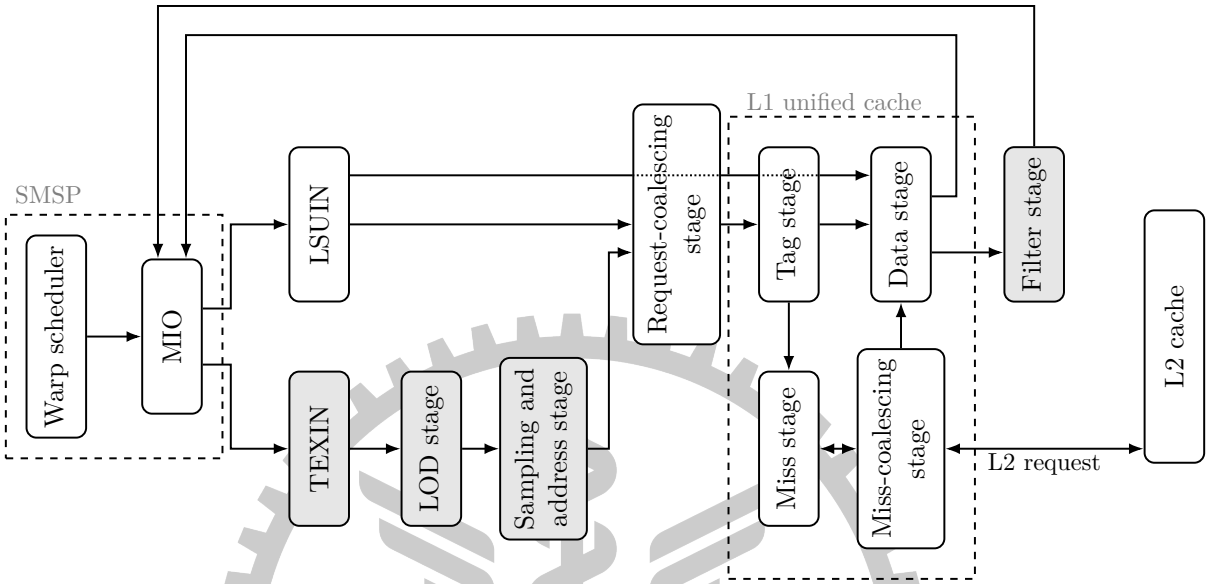


Figure 2.9: Conceptual memory-access pipeline for global-, texture-, and shared-memory instructions. Gray components denote texture-unit stages, and dotted line segments indicate bypassed processing stages.

2.6.1 Request Routing

Memory instructions issued by the warp schedulers are first forwarded to the memory input/output (MIO) unit, as shown in Figure 2.9. The MIO unit arbitrates among issued instructions and directs each request to the appropriate input interface [33]. Global- and shared-memory requests enter through the load/store-unit input (LSUIN) interface, whereas texture-memory requests through the texture-unit input (TEXIN) interface [12].[5]

After routing, LSUIN receives the addresses generated by the active threads for global- and shared-memory instructions and forwards global-memory accesses toward the L1 cache pipeline. Texture requests instead enter TEXIN and continue through the input and address-generation portion of the texture unit (Section 2.2.2), where the LOD and sampling-and-address stages map logical texture coordinates to texel addresses. The resulting global- or texture-memory addresses are then organized for cache access.

[5]Because memory-access pipelines have variable latency, issued instructions may wait in pipeline queues until the required processing resources become available [46]. Global- and shared-memory instructions use the *LSU queue*, whereas texture-memory instructions use the *TEX queue*. These queues decouple instruction issue from request processing.

2.6.2 Request Coalescing and Wavefront Processing

The accesses generated by a warp are initially associated with individual threads. Before being presented to the cache pipeline, the request-coalescing stage groups accesses that refer to the same cache sectors or cache lines. This process, referred to as *request coalescing*, converts the thread-oriented address set into one or more memory-oriented requests [18].

For a global-memory instruction, coalescing operates on the linear addresses received through LSUIN. For a texture-memory instruction, it operates after the logical texture coordinates have been mapped to addresses within the texture resource. The resulting request organization may therefore differ even when the logical access patterns are similar, because global and texture memory use different physical layouts.

Coalescing represents accesses from multiple threads using a common request when they refer to data within the same sector or cache line. It does not, however, imply that the entire resulting request can be processed by every subsequent stage in a single cycle. As the request progresses through the tag, data, and miss stages, each stage may partition it according to its own processing capability.

When a request exceeds the processing capability of a pipeline stage, that stage processes the request as multiple smaller work packages known as *wavefronts*. A wavefront is the portion of a warp-level memory request that the stage can process in one pass. Multiple wavefronts associated with the same warp instruction are serialized across cycles.

Wavefront processing is related to, but distinct from, request coalescing. Coalescing determines the sectors or cache lines required by the participating threads, whereas wavefront processing determines how those requests are partitioned and scheduled through a stage with limited processing capacity. Because stages may differ in processing capacity, the same request may be partitioned differently as it progresses through the pipeline.

For example, the tag stage can examine four sets containing four sectors each per cycle [47]. A request whose sectors cannot be accommodated within this capability requires multiple tag-stage wavefronts. The data stage, in contrast, can read only one cache line at a time [48]. The accesses contained in one tag-stage wavefront may therefore require multiple data-stage wavefronts. The miss stage similarly has its own processing limits. The number of wavefronts is consequently determined by both the spatial distribution of the requested sectors and the capability of the stage processing them.

The effectiveness of request coalescing and the resulting number of wavefronts directly affect the amount of work performed by the cache pipeline. Accesses concentrated within fewer sectors or cache lines require fewer sector requests, whereas spatially scattered accesses increase the number of sectors that must be processed. Such requests may occupy individual stages for more cycles, compete for processing capacity with requests from other warps, and increase queueing in the corresponding pipeline.

2.6.3 Memory-Specific Processing Paths

The processing path depends on the memory space targeted by the instruction. Global- and texture-memory requests traverse the cache-processing stages, where each stage may process the request as one or more wavefronts. Shared-memory requests instead follow the LSU path while bypassing the processing performed by the request-coalescing, tag, and miss stages, as indicated by the dotted segments in Figure 2.9.

A global-memory wavefront enters the tag stage, which determines whether its requested sectors are resident in the L1 cache. On a hit, the wavefront proceeds to the data stage, which reads the requested data and returns them to the requesting threads through the *LSU-data path*.

A texture-memory request first passes through the input and address-generation portion of the texture unit (Section 2.2.2). The resulting texel accesses are coalesced and processed as wavefronts through the same tag and data stages used by global-memory requests. On a cache hit, the retrieved texels return to the texture unit through the *TEX-data path* and proceed through the texture-filter stage before the resulting texture value is returned to the requesting threads.

Shared-memory requests enter through LSUIN but follow a shorter path. Because shared memory is directly addressed and explicitly managed by the program, its requests proceed to the LSU-data path without undergoing cache-tag lookup or miss processing. Requests are instead partitioned according to the shared-memory bank structure and its conflict-resolution capability. Accesses to different banks can be serviced in parallel, whereas those to different locations within the same bank may require multiple service cycles. Once the required accesses complete, the data return through the LSU-data path.

Thus, global and texture memory share the central cache-access stages but differ in address generation and result processing. Shared memory shares the load/store input and LSU-data paths with global memory but bypasses the cache-access stages. These differences lead to distinct performance constraints: global-memory requests are affected by request coalescing, cache behavior, and LSU-side pipeline capacity; texture-memory requests additionally consume texture-specific address-generation and TEX-data-path resources; and shared-memory requests are primarily constrained by bank conflicts and the throughput of the LSU-data path.

2.6.4 Miss Processing

When the tag stage determines that a requested sector is not resident in the L1 cache, the corresponding wavefront is forwarded to the miss stage. The miss stage tracks the outstanding access and prepares one or more requests for the L2 cache. Before an L2 request is issued, the miss-processing logic may determine that an outstanding miss already targets the same cache line. The new miss can then be associated with the existing

request rather than generating a redundant L2 access [34]. This mechanism, referred to as *miss coalescing*, allows the returned cache line to satisfy multiple pending accesses.

Miss coalescing is distinct from request coalescing. Request coalescing groups the thread-level accesses produced by one warp instruction before the initial cache lookup. Miss coalescing instead combines outstanding misses after tag lookup and may operate across requests that reach the miss stage at different times.[6] Request coalescing reduces the number of sector requests presented to the L1 cache pipeline, whereas miss coalescing reduces redundant traffic between the L1 and L2 caches.

When the requested data return from L2, the miss-processing path associates them with the pending accesses and makes them available to the data stage. A completed global-memory request then returns through the LSU-data path, whereas a completed texture-memory request continues through the TEX-data and texture-filter stages before returning to the requesting threads.

Cache misses increase both request latency and pressure on the miss-processing and lower-level memory pipelines. Miss coalescing can reduce redundant L2 requests when several pending accesses refer to the same cache line, but it cannot combine requests to different cache lines. The spatial distribution of the requested data therefore affects not only the initial cache access but also the traffic generated toward the L2 cache.

2.7 CUDA Programming Model

To enable programmers to exploit the hierarchical hardware resources of NVIDIA GPUs, NVIDIA proposed the CUDA programming model [19, Sect. 1.2]. The programming model is exposed through C++[7] with a small set of language extensions and built-in variables, forming what is commonly referred to as CUDA C++. In addition to the language extensions, CUDA provides a rich set of runtime APIs for managing memory, launching kernels, and coordinating execution between the CPU and the GPU.

The CUDA programming model assumes a heterogeneous computing system consisting of CPUs and GPUs, referred to as the *host* and *device*, respectively. Code executed on the CPU is termed *host code*, whereas code executed on the GPU is termed *device code*. To perform a computation on the device, the host first transfers the required data from host memory to device memory, launches a kernel for execution on the device, and later retrieves the computation results from device memory after the kernel completes.

The remainder of this section introduces the two primary aspects of the CUDA programming model that govern this interaction. Memory management describes how data are allocated and transferred between the host and the device, whereas kernel programming describes how computations are expressed, organized, and launched on the device.

[6] Using a pending request table.

[7] CUDA started much closer to C, but supported the vast majority of C++ as it evolved.

2.7.1 Memory Management

Memory management refers to the allocation, movement, and deallocation of data between the host and the device, all of which are initiated from the host code. Although the host and device maintain separate physical memory, CUDA provides a unified virtual address space in which both host and device allocations reside in distinct address ranges. Pointers therefore uniquely identify the location of an allocation without explicitly indicating whether it resides in host or device memory.

Despite sharing a unified virtual address space, the host and device generally access only data residing in their respective physical memories. Data required by the device must therefore be transferred from the host before kernel execution, and computation results are typically copied back after execution completes. CUDA also provides managed memory, in which these data movements are performed automatically by the runtime through on-demand page migration. Because this work focuses on explicit memory management, only programmer-controlled data transfers are discussed.

Host memory is typically allocated using standard C++ mechanisms, such as the new operator, whereas device memory is allocated through CUDA runtime APIs according to the target memory space (Section 2.3). Data are then transferred between the host and device using dedicated memory-copy APIs. For pageable host memory, CUDA generally stages the data through temporary page-locked (pinned) memory before transferring them via direct memory access [49], whereas transfers involving pinned memory can proceed directly. Once computation is complete, both host and device allocations are released using their respective deallocation mechanisms.

Global Memory Management Through Linear Pointers

For global memory, the managed object is a linear region of device memory identified by a pointer. The pointer simultaneously identifies the allocated storage and serves as the access handle within kernels. The same pointer is therefore passed among CUDA runtime APIs on the host and subsequently dereferenced by load and store instructions on the device, without requiring an intermediate access object or additional address translation. Table 2.2 summarizes the primary APIs involved in managing global memory.

Table 2.2: Representative APIs used in each step of memory management with global memory as device memory.

Step	APIs
Allocate device storage	`cudaMalloc`, `cudaMallocPitch`, `cudaMalloc3D`
Transfer data	`cudaMemcpy`, `cudaMemcpy2D`, `cudaMemcpy3D`
Kernel access	Linear pointer dereference
Release storage	`cudaFree`

CUDA also provides allocation routines for logically multidimensional arrays, such as `cudaMallocPitch` and `cudaMalloc3D`, which introduce padding to satisfy alignment requirements and improve memory-access efficiency. Nevertheless, these allocations remain linear regions in the device address space. The returned pointer therefore continues to serve as the primary programming abstraction, while any padding must be explicitly incorporated into the application's indexing calculations.

Texture Memory Management Through Opaque Objects

Unlike global memory, texture memory is not exposed through a linear pointer. Instead, CUDA manages texture memory through a *texture object*, an opaque handle that encapsulates both the underlying storage and the metadata required to access it. As a result, kernels cannot directly dereference the underlying memory. Instead, texture data are accessed exclusively through dedicated texture-fetch instructions using the texture object as the access handle. CUDA also provides *surface objects*, which similarly provide opaque handles to the same underlying storage through a writable surface interface. The remainder of this section therefore focuses exclusively on texture objects.

Creating a texture object involves three elements. First, storage is allocated, typically using a CUDA array whose physical organization follows the implementation-defined block-linear layout (Section 2.3.3). Second, a resource descriptor identifies the storage associated with the texture object. Finally, a texture descriptor specifies how that storage should be interpreted during texture accesses. The resource and texture descriptors are then combined to create the texture object, which may subsequently be passed to kernels. When the texture is no longer needed, both the texture object and its underlying storage are explicitly destroyed.

Table 2.3 summarizes the primary APIs involved in managing texture memory. Compared with global memory, texture memory introduces an additional level of indirection: the storage itself is never accessed directly by kernels. Instead, kernels receive a texture object that encapsulates the underlying storage and its access semantics.

Table 2.3: Representative APIs used in each step of memory management with texture memory as device memory.

Step	APIs
Allocate device storage	`cudaMallocArray`, `cudaMalloc3DArray`, `cudaMallocMipmappedArray`
Transfer data	`cudaMemcpyToArray`, `cudaMemcpy2DToArray`, `cudaMemcpy3D`
Create access handle	`cudaCreateTextureObject`
Kernel access	Texture fetch functions
Destroy access handle	`cudaDestroyTextureObject`
Release storage	`cudaFreeArray`, `cudaFreeMipmappedArray`

The resource descriptor specifies the backing storage of the texture object, such as a CUDA array, mipmapped array, or linear memory. The texture descriptor, in turn, defines the semantics of texture accesses independently of the underlying storage. Together, these descriptors determine the texture-access properties summarized in Table 2.4. The descriptor specifies the dimensionality of the texture, allowing the storage to be viewed as one-, two-, or three-dimensional arrays of texels. It also specifies the texel type, which is restricted to basic integer and single-precision floating-point types and their corresponding vector types. Beyond the data representation, the descriptor determines how texture coordinates are interpreted, how out-of-range coordinates are handled, how fetched values are represented, and whether neighboring texels are interpolated during sampling. This separation allows the same physical storage to support different access semantics without altering its underlying organization.

Table 2.4: Principal properties governing texture accesses.

Property	Purpose
Resource type	Backing storage associated with the texture object.
Dimensionality	One-, two-, or three-dimensional array of texels.
Texel type	Data type stored in each texel.
Read mode	Texels returned in their original representation or converted to normalized floating-point values.
Coordinate mode	Absolute or normalized texture coordinates.
Addressing mode	Handling of texture coordinates outside the valid range.
Filtering mode	Nearest-texel value or interpolation of neighboring texels.

2.7.2 Kernel Programming

CUDA adopts a *single-program multiple-data* (SPMD) programming model, in which a single program is executed concurrently by many threads operating on different portions of the input data. In CUDA, this program is expressed as a *kernel* [19, Sect. 2.3],[8] which is a function executed on the device and invoked from the host. Unlike an ordinary C++ function, a kernel is executed concurrently by many threads, each maintaining its own execution context while following the same program. The corresponding execution on NVIDIA GPUs follows the SIMT execution model (Section 2.1.1), which preserves the abstraction that each thread has an independent execution path with its own register state and control flow while transparently executing groups of threads as warps. CUDA distinguishes host and device functions using execution-space specifiers. A kernel is declared

[8]This work considers only conventional SIMT kernels. CUDA 13.3 also introduces tile kernels, an alternative programming paradigm that expresses computation at tile rather than thread granularity.

Listing 2.1: A naive CUDA kernel for matrix multiplication. Matrix multiplication $A \times B$ is computed for matrices of dimensions $M \times K$ and $K \times N$, producing an output matrix C of dimension $M \times N$. Boundary handling is omitted for brevity by assuming that all matrix dimensions are multiples of the thread-block dimensions.

```
__global__ void
matmul(int M, int N, int K,
    float *A, float *B, float *C) {
  int x = blockIdx.y*blockDim.y
          + threadIdx.y;
  int y = blockIdx.x*blockDim.x
          + threadIdx.x;

  float acc = 0.0f;
  for (int i = 0; i < K; i++)
    acc += A[x*K + i]
         * B[i*N + y];
  C[x*N + y] = acc;
}
```

Listing 2.2: A tiled CUDA kernel for matrix multiplication using shared memory. Boundary handling is omitted under the same assumption as Listing 2.1.

```
#define TS 32 /* tile size */

__global__ void
matmulTiled(int M, int N, int K,
    float *A, float *B, float *C) {
  __shared__ float tileA[TS][TS];
  __shared__ float tileB[TS][TS];

  int tx = threadIdx.x;
  int ty = threadIdx.y;
  int x = blockIdx.y*TS + ty;
  int y = blockIdx.x*TS + tx;

  float acc = 0.0f;
  for (int t = 0; t < K; t += TS) {
    tileA[ty][tx] = A[x*K+t+tx];
    tileB[ty][tx] = B[(t+ty)*N+y];
    __syncthreads();
    for (int i = 0; i < TS; i++)
      acc += tileA[ty][i]
           * tileB[i][tx];
    __syncthreads();
  }
  C[x*N + y] = acc;
}
```

with the `__global__` specifier, indicating that it is invoked from the host and executed on the device.[9]

To express data parallelism, the workload is partitioned into fine-grained tasks and mapped onto individual threads. Each thread identifies its assigned portion of the computation through built-in variables that specify its position within the thread hierarchy. For example, in matrix multiplication, each thread may compute a single element of the output matrix. Listing 2.1 illustrates a straightforward implementation in which each thread computes one output element according to its two-dimensional thread index.

Although Listing 2.1 is functionally correct, every thread repeatedly loads the required elements directly from global memory, even when neighboring threads access the same data. CUDA provides a thread hierarchy that enables threads within the same block to cooperate through shared memory and synchronization. This hierarchy is introduced next.

[9]CUDA also provides the `__host__` and `__device__` execution-space specifiers for ordinary host and device functions.

Thread Hierarchy

CUDA organizes the threads that execute a kernel into a hierarchy consisting of a grid and thread blocks, or simply blocks, as illustrated in Figure 2.10. The entire set of threads launched by a kernel forms a grid of blocks, and each block contains a collection of threads. Both grids and blocks can be organized in up to three dimensions, allowing CUDA programs to naturally map multidimensional problem domains onto the hierarchy.

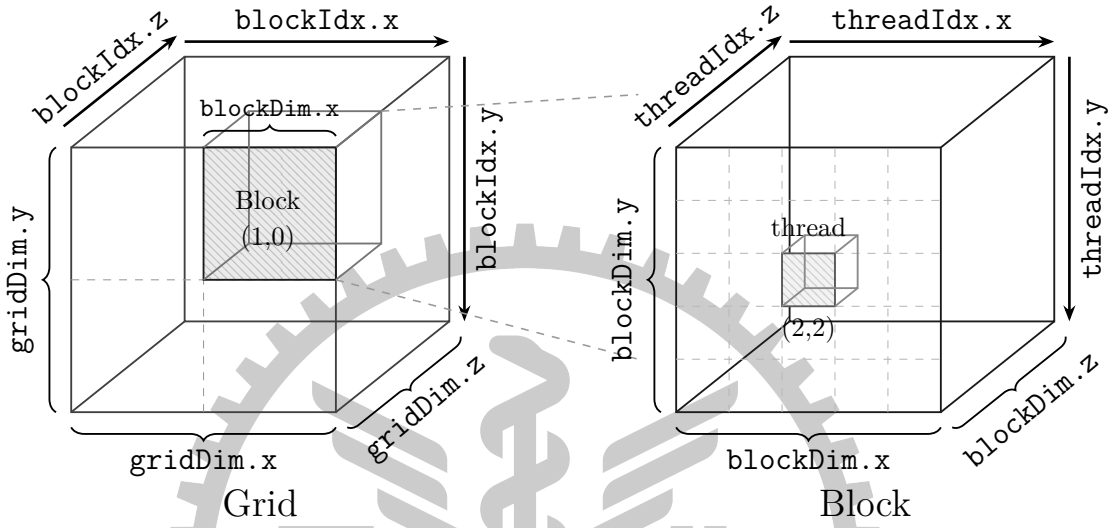


Figure 2.10: CUDA thread hierarchy. A kernel launch creates a grid of thread blocks, each containing a collection of threads. Blocks are indexed by `blockIdx` within the grid, whereas threads are indexed by `threadIdx` within each block. Adapted from Boehm [50].

Within a block, each thread is identified by its local thread index, `threadIdx`. The size of the block is given by `blockDim`. Similarly, each block is identified by its block index, `blockIdx`, within a grid whose size is given by `gridDim`. These built-in variables allow each thread to compute a global position in the problem domain. For the matrix-multiplication kernel in Listing 2.1, the global row and column indices of the output element are computed from `blockIdx`, `blockDim`, and `threadIdx`.

The thread block is also the basic unit of cooperation in CUDA. Threads within the same block execute on the same SM and can communicate through shared memory, whose variables are declared using the `__shared__` memory-space specifier. They can also synchronize using `__syncthreads()`, which ensures that all threads in the block reach the synchronization point before any of them proceed. Threads belonging to different blocks, however, cannot synchronize with one another within a kernel and do not share the same shared-memory allocation. Listing 2.2 shows a tiled version of matrix multiplication that exploits this cooperation. Each block computes one tile of the output matrix. Threads in the block collaboratively load tiles of the input matrices into shared memory, synchronize to ensure that the tiles are fully loaded, and then reuse the shared-memory data to compute their assigned output elements.

Kernel Launch

Once a kernel has been defined, the host launches it by specifying the *execution configuration* and kernel arguments. The execution configuration determines how the kernel is instantiated over the thread hierarchy, whereas the kernel arguments provide the data and scalar parameters shared by all threads, such as problem dimensions and device-memory access handles. Together, they provide the information required for the CUDA runtime to launch the kernel on the device.

The execution configuration reflects the mapping between the problem domain and the thread hierarchy. For the tiled matrix-multiplication kernel in Listing 2.2, each thread computes one output element and each thread block computes one `TS`×`TS` output tile. Accordingly, the block dimensions are chosen to match the tile dimensions, whereas the grid dimensions are determined by the number of tiles required to cover the output matrix.

CUDA expresses the execution configuration as part of the kernel-launch syntax. The launch expression, `kernel<<<gridDim, blockDim>>>(...)`, extends the conventional C++ function-call syntax with an execution configuration enclosed by triple chevrons. The first two configuration arguments specify the grid and block dimensions, respectively. CUDA further permits optional arguments for dynamic shared-memory size and execution stream, although these features are not considered in this work. Listing 2.3 illustrates the resulting launch for the tiled matrix-multiplication kernel.

Listing 2.3: Launching the tiled matrix-multiplication kernel. The execution configuration specifies one `TS`×`TS` thread block for each output tile. For brevity, the matrix dimensions are assumed to be multiples of the tile size `TS`, defined in Listing 2.2.

```
dim3 blockDim(TS, TS);
dim3 gridDim(N / TS, M / TS);

matmulTiled<<<gridDim, blockDim>>>(M, N, K, A, B, C);
```

From the host's perspective, launching a kernel does not directly initiate execution on the GPU. Instead, the launch operation records the execution configuration and kernel arguments in device memory and submits a corresponding launch command to a hardware work queue managed by the GPU [12], [51]. Thus, kernel execution is asynchronous with respect to the host. After issuing the launch, the host may continue executing subsequent host code and synchronize with the device only when the kernel results are required. Upon completion, the kernel writes its output in device memory, from which the host may later retrieve the results through an explicit data transfer.

2.8 LLVM Compilation Model

CUDA C++ combines conventional host code executed on the CPU with device code executed on the GPU. Consequently, a CUDA compiler must generate code for two different instruction-set architectures while preserving the interface between them.

The compiler framework proposed in this work is implemented using the LLVM compiler infrastructure [52], specifically LLVM 19.[10] Unlike NVIDIA's proprietary CUDA compiler (NVCC) [53], LLVM provides an open compiler framework that enables compiler analyses and transformations to be implemented as extensible LLVM passes. Accordingly, the compilation pipeline described in this section corresponds to the LLVM 19 implementation. The remainder of this section introduces LLVM's CUDA compilation model and the stages at which compiler passes can be incorporated.

2.8.1 Compilation Pipeline

Figure 2.11 illustrates LLVM's CUDA compilation pipeline [54], [55]. The pipeline consists of two sequential compilation phases: ❶ device compilation and ❷ host compilation. Because the generated fat binary is embedded into the host object, host compilation can begin only after device compilation has produced the fat binary. Each compilation phase comprises a sequence of stages that progressively transform the program toward the final executable.

During the ❶ device-compilation phase, Clang parses the complete CUDA source and lowers the device code into LLVM's intermediate representation (IR). The device IR is processed by the LLVM optimizer, where analyses and transformation passes are performed before the NVPTX code generator translates the optimized IR into NVIDIA's Parallel Thread Execution (PTX) code [28]. PTX serves as a virtual instruction set for NVIDIA GPUs and provides a stable interface between compiler-generated code and architecture-specific machine code. For each target GPU architecture, the optimizing assembler Ptxas assembles the PTX into GPU machine code (SASS) and packages it as a Cubin object file. Fatbinary then combines the generated Cubin files, optionally together with PTX, into a single fat-binary image.

After the fat binary has been generated, Clang recompiles the same CUDA source for the ❷ host-compilation phase. The fat binary is embedded into the host LLVM IR as a constant string literal, which is later emitted into a special section of the host object file. The host IR is likewise processed by the LLVM optimizer before being translated into native CPU machine code. Finally, the generated host object is linked with the CUDA

[10] LLVM 20 and later reorganize portions of the CUDA compilation pipeline by reusing the offloading infrastructure originally developed for OpenMP.

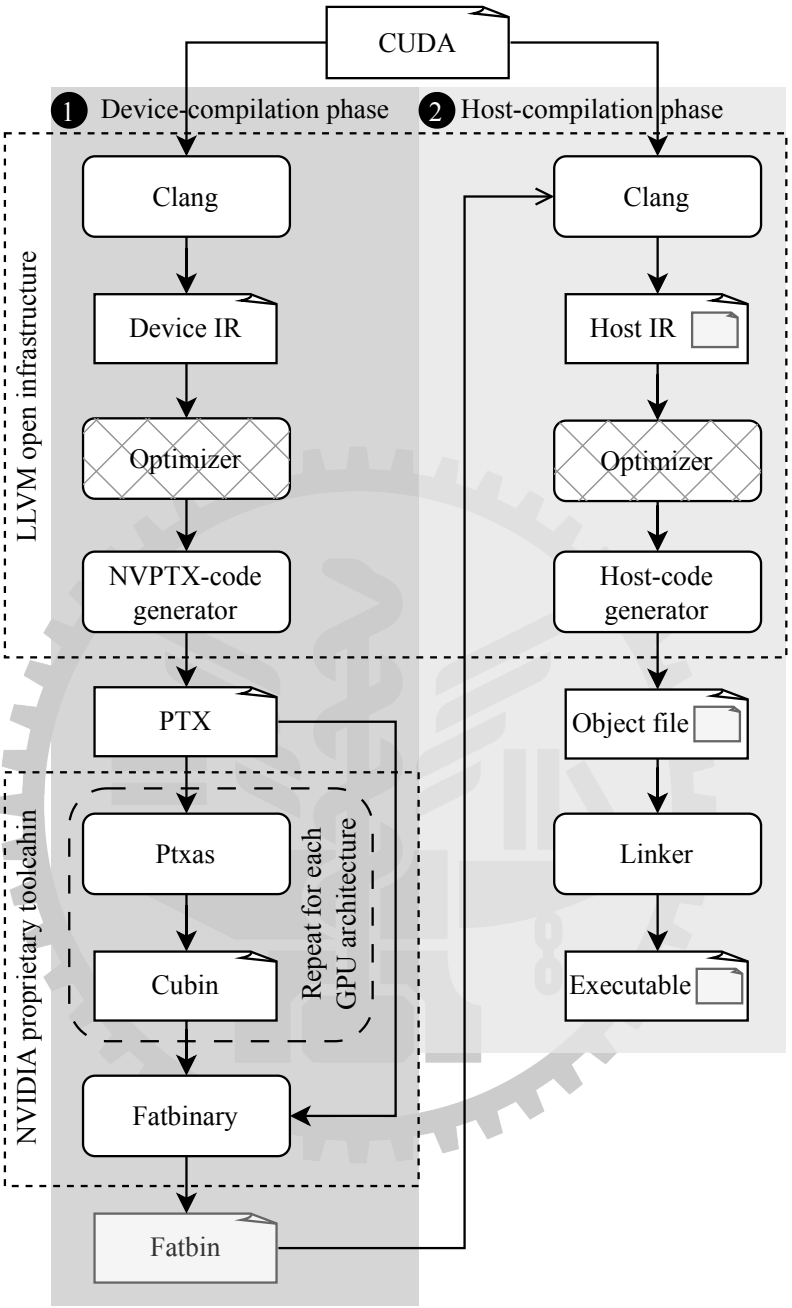


Figure 2.11: LLVM CUDA compilation pipeline. During the device-compilation phase, the device IR is optimized, translated into PTX, assembled into architecture-specific Cubin object files, and packaged into a fat-binary image. During the host-compilation phase, the fat binary is embedded into the host IR before native code generation. The double-hatched optimizer blocks denote the LLVM optimization stages, where analysis and transformation passes can be inserted. Filled arrows (→) indicate the normal compilation flow, whereas open arrows (⟶) represent binaries transferred across compilation phases.

runtime libraries to produce the executable. At program execution, the CUDA driver selects the machine code corresponding to the installed GPU or, if necessary, performs just-in-time compilation from the embedded PTX.

2.8.2 Merged-Parsing Model

LLVM differs from NVCC primarily in its compilation model. NVCC adopts a split-compilation model, in which the CUDA source is preprocessed into separate host and device source files before compilation. In contrast, Clang adopts a merged-parsing model, in which the complete CUDA source is parsed independently for both the device and host compilations. During device compilation, host code is parsed for semantic correctness but no host code is generated; conversely, during host compilation, device code is likewise parsed even though only host code is emitted. Although the source is parsed twice, the two compilations use different predefined macros and target-specific compilation options, preventing a single parsing pass from being reused. Nevertheless, both compilation phases reuse the same parser and semantic analysis but produce separate host and device binaries.

2.8.3 Kernel Stubs

During the host-compilation phase, Clang does not generate executable code for CUDA kernels. Instead, each kernel definition is replaced by a host-side *stub* function. The stub marshals the kernel arguments, receives the execution configuration specified by the kernel launch expression, and invokes the CUDA Runtime API to enqueue the corresponding kernel for execution. The generated fat binary embedded in the executable contains the compiled kernel images referenced by the stubs.

Because kernel launches are translated into calls to the CUDA Runtime API, compiler transformations can modify the generated stubs to change the kernel launch behavior or the representation of kernel arguments without modifying the CUDA runtime itself.

2.8.4 Extensible Compiler Infrastructure

LLVM exposes the optimization stages shown in Figure 2.11 through an extensible pass infrastructure, allowing custom analyses and transformations to be implemented as LLVM passes operating on either the device-side or host-side LLVM IR without modifying NVIDIA's proprietary tools. The compiler framework proposed in this work exploits this extensibility by inserting analysis and transformation passes into the LLVM optimization stages of both compilation phases.

2.9 Profiling Tools

NVIDIA provides two complementary profiling tools for analyzing CUDA applications: Nsight Systems [56] and Nsight Compute [57]. The two tools operate at different levels of abstraction. Nsight Systems provides an application-level view of execution, whereas Nsight Compute provides a detailed view of individual kernel executions.

2.9.1 Nsight Systems

Nsight Systems records application execution as a timeline containing CPU activity, CUDA runtime and driver API calls, memory transfers, synchronization operations, and kernel executions. Each kernel launch is represented as an execution interval, allowing its duration and its relationship with the surrounding host-side operations to be examined. This application-level view is useful for attributing execution time to different activities and identifying overheads associated with memory transfers, runtime calls, synchronization, and idle periods.

The NVIDIA Tools Extension library (NVTX) [58] can be used in conjunction with Nsight Systems to annotate the timeline with user-defined ranges, events, and metadata. These annotations associate regions of the application with higher-level program semantics, making it possible to relate the recorded CUDA operations and kernel executions to specific application phases. NVTX therefore provides additional context for interpreting the application-level timeline, although it does not expose the internal execution behavior of individual kernels.

2.9.2 Nsight Compute

Nsight Compute profiles individual kernel launches using hardware performance counters and derived metrics. The collected metrics can characterize different parts of GPU execution, including instruction issue, warp scheduling, occupancy, cache behavior, memory traffic, and warp stall conditions. Nsight Compute therefore enables the internal behavior of a kernel to be examined in substantially greater detail than is available from an application-level timeline.

The metrics collected during profiling can be selected through predefined or user-defined metric sets, allowing the profiling configuration to be tailored to the hardware components and execution phenomena under investigation. Because not all requested metrics can necessarily be collected simultaneously, Nsight Compute may divide their collection into multiple passes and replay the profiled kernel for each pass. Collecting more metrics can therefore increase profiling overhead by requiring additional kernel executions.

Nsight Compute also provides cache-control and clock-control mechanisms for improving measurement reproducibility. During kernel replay, cache control can flush the GPU

caches before each profiling pass, preventing cache state produced by an earlier replay from affecting the measurements collected during a later pass. Clock control can similarly reduce variability caused by dynamic clock-frequency changes. Together, these mechanisms establish more consistent execution conditions across profiling passes and repeated profiling runs.

The two profilers thus provide complementary perspectives. Nsight Systems shows how kernel executions and CUDA operations contribute to the execution of the complete application, whereas Nsight Compute explains the performance of a selected kernel in terms of its interaction with the GPU architecture. In this work, Nsight Compute is used throughout the subsequent chapters to validate the observed memory-system behavior and analyze the architectural effects of the proposed optimization. Nsight Systems is used in the evaluation to quantify the application-level contributions of kernel execution and CUDA runtime operations.

Chapter 3
Motivation

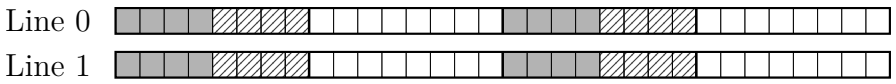
Non-unit-stride access patterns are prevalent in GPU workloads, particularly in computations over multidimensional data structures, as observed in a prior study [45] based on several widely used benchmark suites. When a warp traverses a logical dimension that is not contiguous under the physical memory layout, the addresses generated by a single warp instruction may span multiple cache sectors and cache lines. Although subsequent instructions may access the remaining data within those sectors, fine-grained warp scheduling can introduce many intervening accesses. The resulting cache-sector reuse distance may be sufficiently long for the sectors to be evicted before that reuse occurs. As a result, spatial locality exploited across warp instructions provides limited benefit under substantial cache pressure.

Rather than relying on reuse across warp instructions, cache data can be used more robustly when each warp instruction consumes as much as possible of the data fetched on its behalf. This property is closely related to the *intra-warp spatial locality* described by Liang et al. [59], which characterizes the spatial relationship among memory accesses generated by threads within the same warp. We quantify this property using *intra-warp cache utilization*. For a warp instruction, utilization at cache granularity g is defined as

$$U(g) = \frac{B_{\text{accessed}}}{N(g)S(g)},$$

where B_{accessed} is the number of distinct bytes accessed by the instruction, $N(g)$ is the number of cache units touched at granularity g, and $S(g)$ is the size of each unit. The metric represents the fraction of fetched cache data accessed by the instruction. At a given granularity, high utilization indicates that the requested data are concentrated within few fetched units, whereas low utilization indicates that they are distributed across many such units. Unlike locality that depends on reuse across instructions, intra-warp cache utilization depends only on the accesses of an individual warp instruction and does not require the fetched data to remain resident for subsequent accesses. A streaming access pattern may therefore provide no reuse across instructions while still achieving high intra-warp cache utilization.

Intra-warp cache utilization can be evaluated at cache-sector and cache-line granularities, yielding *intra-warp cache-sector utilization* (g = sector) and *intra-warp cache-line utilization* (g = line), respectively. Figure 3.1 illustrates that the two quantities capture different aspects of how a warp instruction consumes fetched cache data. In Figure 3.1 (a), the warp instruction accesses only half of each of the four touched sectors, and its



(a) Low sector- and line-level utilization across two cache lines: $U(\text{sector}) = 64/(4 \times 32) = 1/2$, $U(\text{line}) = 64/(2 \times 128) = 1/4$.

(b) Higher sector- and line-level utilization within one cache line: $U(\text{sector}) = 64/(3 \times 32) = 2/3$, $U(\text{line}) = 64/(1 \times 128) = 1/2$.

Figure 3.1: Intra-warp cache-sector and cache-line utilization for a warp instruction, which accesses sixteen 4-byte elements. Each row represents one cache line, divided into four 32-byte sectors and then into eight 4-byte elements per sector. Gray elements are accessed by the warp instruction; hatched elements are fetched as part of a touched sector but are not accessed by that instruction.

requested data likewise occupy a small fraction of the two cache lines containing those sectors. In Figure 3.1 (b), the requested data reside within a single cache line: they fill sector 0, the right-half of sector 1, and the left-half of sector 2. This access therefore touches three sectors and one cache line, achieving both higher cache-sector utilization and higher cache-line utilization.

Cache-sector utilization characterizes the fraction of each fetched sector accessed by the warp instruction. Because cache data are transferred at sector granularity (Section 2.4), higher sector utilization reduces the cache capacity and transfer bandwidth consumed by unaccessed data. It also reduces dependence on subsequent instructions accessing the remaining sector contents before eviction.

Cache-line utilization characterizes the fraction of each requested cache line accessed by the warp instruction. It reflects how the requested sectors are distributed across cache lines and is therefore closely related to memory-access coalescing and cache-access wavefront generation. When the requested sectors are concentrated within fewer cache lines, they can be served by fewer coalesced cache accesses and wavefronts (Section 2.3.2). Conversely, lower cache-line utilization increases the number of cache lines and wavefronts, placing greater pressure on the cache-access pipeline and downstream memory subsystem. The resulting contention can delay memory accesses and further increase their effective cache-sector reuse distances. Sector- and line-level utilization thus capture complementary effects: sector-level utilization determines how efficiently fetched sectors are consumed, whereas line-level utilization determines how many cache lines are occupied and how much memory-pipeline work is required to serve the instruction.

Non-unit-stride accesses do not inherently exhibit poor intra-warp cache utilization. Rather, poor utilization arises when the physical memory layout maps the addresses requested by a warp across many cache sectors and cache lines. This mismatch commonly occurs for multidimensional data when the logical direction traversed by the warp is poorly

aligned with the contiguous dimension of a linear global-memory layout. Reordering the physical data placement can instead consolidate the requested addresses, improve intra-warp cache utilization, and reduce cache-access wavefront generation. This observation motivates a layout-mapping mechanism that better aligns physical data placement with the logical multidimensional access structure of the computation.

The remainder of this chapter substantiates this motivation through controlled row-summation case studies. We first compare linear and transposed physical layouts that exhibit the minimum and maximum possible intra-warp cache utilization for aligned `FP32` accesses. We then analyze how their utilization affects intrinsic and effective cache-sector reuse distances and validate the predicted effects using cache-access wavefronts, warp stalls, cache hit rates, memory traffic, and execution time measured on real hardware.

3.1 Case Studies: Row Summation

We present two row-summation case studies with identical logical computations and thread mappings but different physical layouts: linear and transposed. Their contrasting intra-warp cache utilization allows us to examine its effects on cache reuse, cache-access wavefront generation, and execution performance.

3.1.1 Row Summation under Linear Layout

In this case study, we are given N matrices of size $Q \times K$. For each row, we sum the corresponding elements across all matrices, yielding an output vector of Q elements , as illustrated in Figure 3.2. All elements are of type `FP32` (4 bytes each). Each output element is assigned to one thread and computed independently. Listing 3.1 presents the CUDA kernel, in which the output vector and input matrices reside in global memory. The summation loop iterates K times, accessing the j-th column of all matrices in each iteration. The case study therefore exhibits a streaming access pattern in which each element is accessed exactly once.

The parameters are chosen as follows. The width K is a multiple of 32 to ensure that consecutive physical rows reside in different cache lines and that the first element of each physical row is always aligned to a cache-line boundary.[1] The height Q is likewise a multiple of 32 to ensure full warps and must not exceed the number of threads the GPU can schedule concurrently. The parameter N controls both the reuse distance and the total number of memory requests issued during kernel execution.

Under this setting, each thread within a warp accesses an element from a distinct logical row but the same logical column. Figure 3.3 illustrates the cache-line organization of a 32×32 logical input sub-matrix under the linear layout. Since consecutive physical

[1] Global memory allocations are guaranteed to be at least 256-byte aligned.

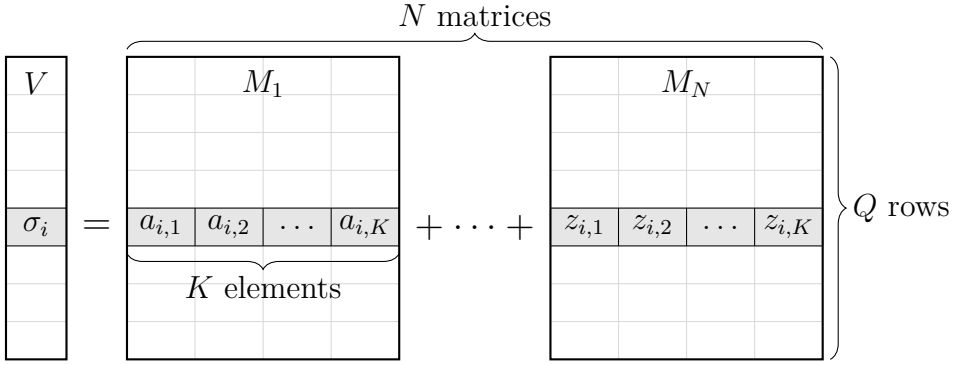


Figure 3.2: Logical concept of the row-summation case study. The summation of the corresponding rows of matrices M_1 through M_N yields the vector V; the elements shown in gray illustrate the contributing elements of the i-th row.

Listing 3.1: CUDA kernel for row-summation case study. Each thread accumulates the elements of a single logical row across all N input matrices. Ellipses denote omitted intermediate matrix arguments.

```
__global__ void sumRows(float *__restrict__ v,
        float *__restrict__ m1, ..., float *__restrict__ mN) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    float vi = 0;
    for (int j = 0; j < K; j++) {
        vi += m1[i * K + j] + ... + mN[i * K + j];
    }
    v[i] = vi;
}
```

rows are separated by at least one cache line, the 32 addresses generated by a warp map to 32 distinct cache sectors and 32 distinct cache lines. Accordingly, its intra-warp cache-sector and cache-line utilizations are

$$U(\text{sector}) = \frac{128}{32 \times 32} = \frac{1}{8}, \qquad U(\text{line}) = \frac{128}{32 \times 128} = \frac{1}{32}.$$

The low sector utilization leaves most of each fetched sector for later loop iterations. Exploiting this spatial locality therefore requires the sectors to remain resident until those accesses occur. The low line utilization additionally distributes the sectors across many cache lines, preventing their cache accesses from being consolidated and causing the warp load instruction to generate multiple cache-access wavefronts. The following analyses focus exclusively on memory loads and exclude the final store to the output vector.[2]

[2]The output is a $1 \times Q$ row vector stored contiguously in memory; the final store therefore exhibits full sector- and line-level utilization.

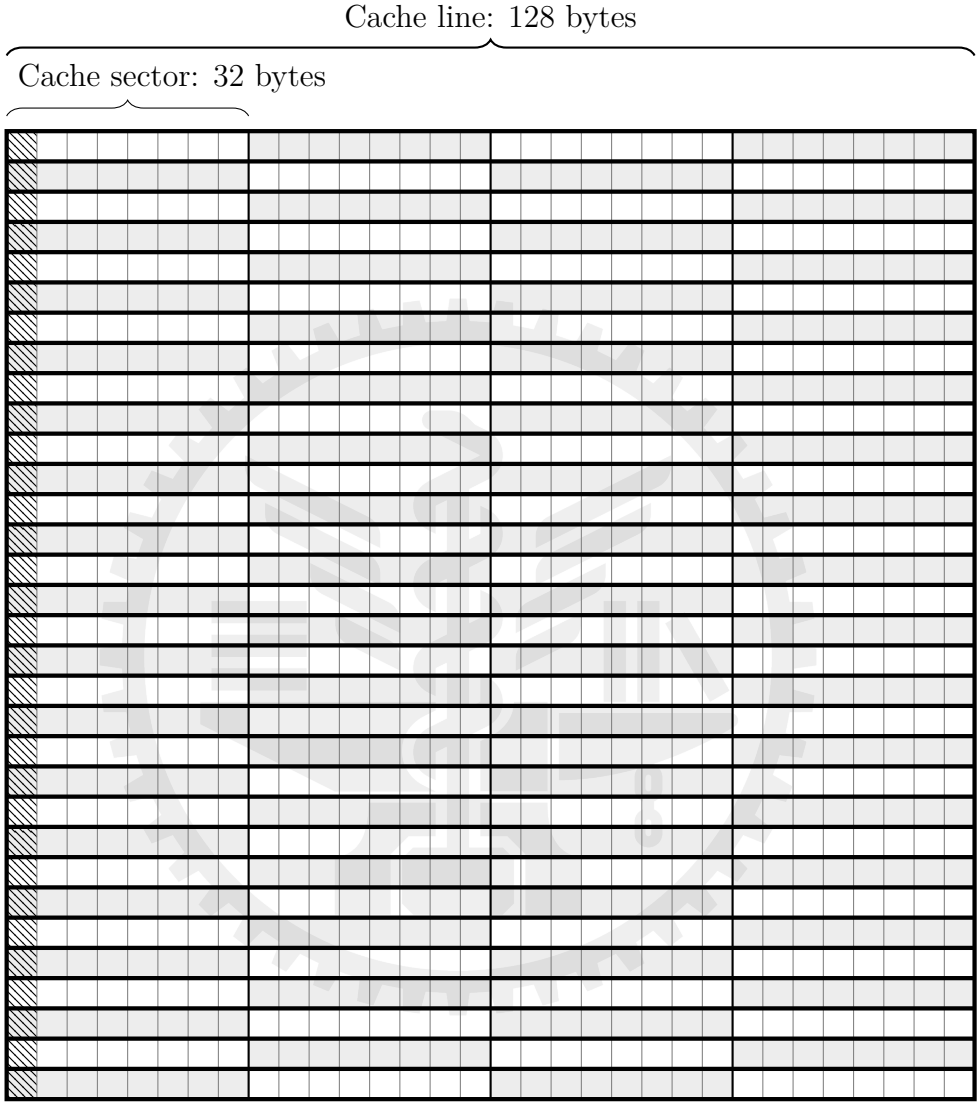


Figure 3.3: Cache-line organization of a 32×32 logical input sub-matrix accessed by a warp under the linear layout. Thick and thin boundaries indicate cache lines and sectors, respectively. The alternating checkerboard pattern distinguishes adjacent cache-sector regions and emphasizes the cache-sector boundaries. Shaded elements indicate the matrix elements accessed by a warp during the first iteration of the summation loop.

3.1.2 Row Summation under Transposed Layout

This case study mirrors the row-summation kernel of Section 3.1.1, except that the input matrices are transposed in global memory prior to kernel invocation,[3] effectively recasting the computation as a *column summation* from the perspective of the memory subsystem. The mapping onto the hardware is identical to the row summation counterpart, with the sole difference being the index expression used to access the matrices within the accumulation loop, as highlighted in Listing 3.2.

Listing 3.2: CUDA kernel for the row-summation case study under the transposed layout. Each thread accumulates elements along a single logical column of the transposed input matrices. The highlighted index expression marks the only difference from Listing 3.1.

```
__global__ void sumRowsT(float *__restrict__ v,
        float *__restrict__ m1, ..., float *__restrict__ mN) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    float vi = 0;
    for (int j = 0; j < K; j++) {
        vi += m1[ j * Q + i ] + ... + mN[ j * Q + i ];
    }
    v[i] = vi;
}
```

Because the transposed layout places elements of the same logical column in consecutive memory locations, all threads in a warp now advance through memory in lockstep, row after row of the transposed matrix, as illustrated in Figure 3.4. Because the 32 threads access contiguous 4-byte elements spanning four adjacent cache sectors within a single cache line, both utilization metrics are maximal:

$$U(\text{sector}) = \frac{32 \times 4}{4 \times 32} = 1, \qquad U(\text{line}) = \frac{32 \times 4}{1 \times 128} = 1.$$

The instruction thus consumes all data fetched on its behalf and does not depend on subsequent instructions to access the remaining contents of the sectors. Concentrating the four requested sectors within one cache line also allows their accesses to be coalesced into a single cache-access wavefront. The transposed layout consequently provides a controlled example of how higher intra-warp cache utilization reduces both dependence on cross-instruction spatial reuse and pressure on the cache-access pipeline.

[3] Transposition is generally impractical as a optimization strategy due to the additional preprocessing overhead; it serves here solely as a controlled baseline for comparison.

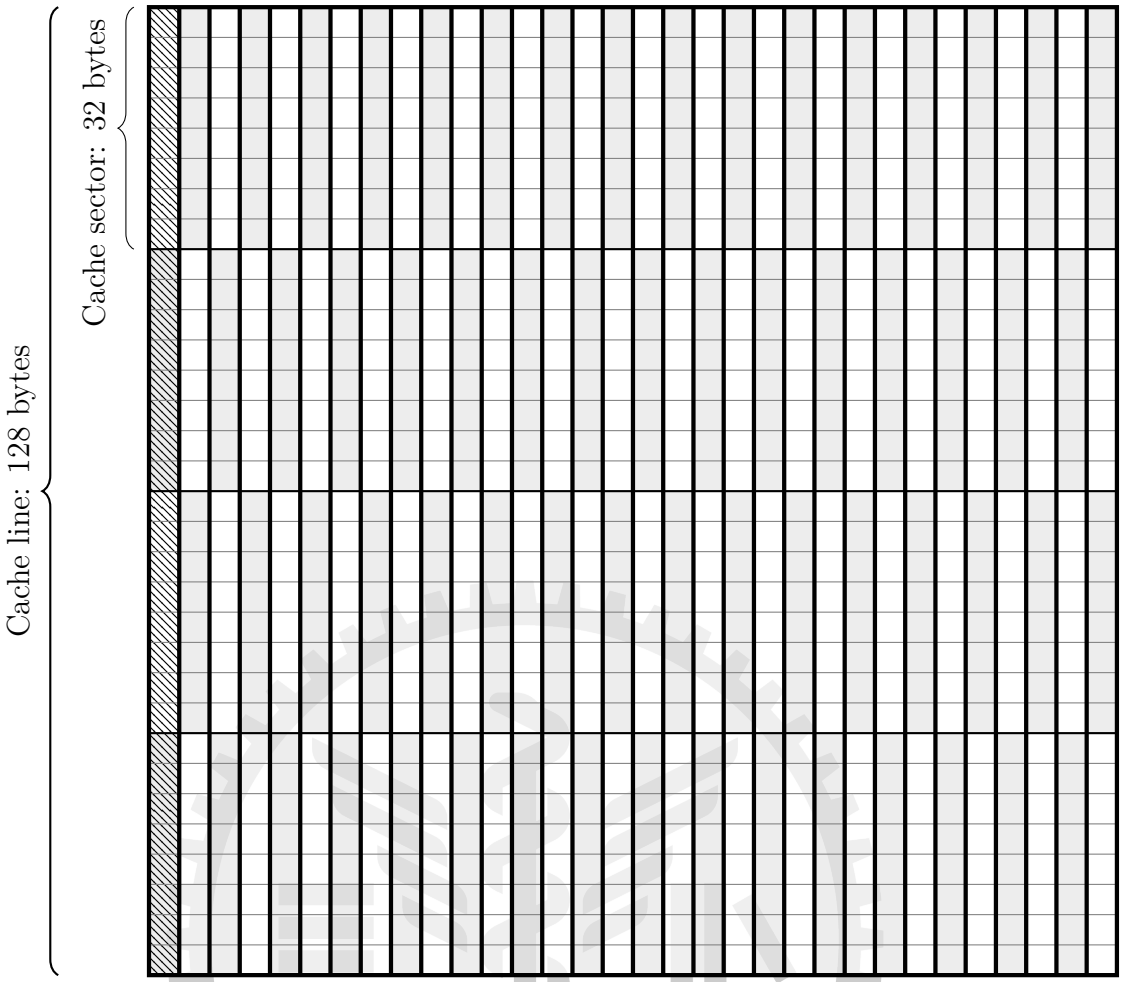


Figure 3.4: Cache-line organization of the 32×32 logical input sub-matrix accessed by a warp under the transposed layout. Compared with Figure 3.3, the physical organization is transposed while the shaded elements represent the same logical accesses performed during the first iteration of the summation loop. Thick and thin boundaries indicate cache lines and sectors, respectively.

3.2 Reuse-Distance Analysis

We first discuss the reuse distance under the assumption that contention in the memory subsystems is absent, and then progressively incorporate the effect of contention to describe how it amplifies reuse distance. For brevity, we hereafter refer to the row-summation kernels under the linear and transposed layouts as the *linear-layout case* and the *transposed-layout case*, respectively. Under the aligned `FP32` accesses considered here, the two cases exhibit the minimum and maximum possible intra-warp cache utilization.

We note that the GPU sector cache fetches data at sector granularity but evicts at cache-line granularity. In general, this implies that cache misses should be characterized using reuse distance (Section 2.5.1) at both granularities. Although their resulting distances are not numerically identical, the access patterns considered here traverse each cache line monotonically from its first sector to its last. Sector- and line-granularity reuse

distances therefore exhibit the same qualitative behavior. Because the following analysis focuses on this behavior rather than exact reuse-distance values, we consider only sector-granularity reuse distance.

3.2.1 Intrinsic Reuse Distance

The *intrinsic reuse distance* refers to the reuse distance induced solely by instruction ordering and warp scheduling, excluding the effects of memory-subsystem contention and pipeline backpressure. Under this optimistic assumption, the memory subsystem is modeled with unlimited hardware resources and infinite-sized queues that never stall the instruction stream. Consequently, as long as the warp scheduler selects an eligible warp with resolved dependencies, its instructions can always be dispatched through the memory pipeline. Because reuse-distance analysis requires an ordered memory-access trace (Section 2.5.1), the intrinsic reuse distance is determined by (1) the static instruction order, including interleaved dependencies, and (2) the dynamic execution order imposed by warp scheduling. The static instruction order can be obtained from the SASS code, which is the final compilation artifact executed by the hardware. Modeling the dynamic execution order, however, requires additional assumptions because many hardware details are undisclosed.

Cache-Structure Assumption We assume that the cache hierarchy is dedicated to global-memory data accesses. In particular, no shared memory is allocated from the L1 cache, and instruction-fetch traffic is excluded from the L2 cache. Because this analysis focuses only on intrinsic reuse distance rather than quantitative miss-rate modeling, the cache associativity and replacement policy are not considered.

Warp-Scheduling Assumption We use a GTO warp-scheduling policy (Section 2.5.2). GTO is widely used as a baseline model in GPU scheduling studies [60], [61], [62], [63], although the precise scheduling policies are undocumented. We interleave the requests of concurrently executed warps following the approach of Kiani et al. [40]. Reuse distance is analyzed for a representative warp, as suggested by GPUMech [44], to approximate the behavior of the remaining warps. Because the kernels in our case study do not exhibit control divergence, all warps are expected to follow similar execution trajectories and may therefore be treated interchangeably.

Transposed-Layout Case

The transposed-layout case exhibits a simple reuse-distance characteristic: all accesses have infinite intrinsic reuse distance, independent of both static instruction order and dynamic warp scheduling. A load instruction issued by a warp accesses four sectors within a single cache line and achieves full cache-sector utilization. Because the access pattern is purely streaming, every sector is accessed exactly once and is therefore compulsory.

The fetched sectors are then immediately consumed by the issuing warp, and neither instruction interleaving nor warp scheduling can interrupt it before the accesses complete.

Linear-Layout Case

The linear-layout case behaves differently. A single load instruction issued by a warp accesses 32 distinct sectors distributed across 32 cache lines, yielding low cache-sector and cache-line utilization. Although the first accesses to these sectors are compulsory, subsequent accesses to elements residing in the same sector may be reused later. Whether these sectors remain resident in the cache depends on the reuse distance introduced by instruction ordering and warp scheduling.

To analyze this behavior, we first examine the instruction ordering and dependency interleaving introduced by loop unrolling. The compiler-selected unrolling factor enables multiple accesses to the same input matrix within a single loop iteration. Load and use instructions from different input matrices are then statically interleaved. We define a *stall-free load sequence* (SFLS) as a maximal sequence of load instructions that can be issued continuously without stalling. Under the optimistic assumption used for intrinsic reuse distance, such stalls arise only from unresolved dependencies. The scheduler therefore continues issuing instructions from the same warp throughout an SFLS, while the dependent use instructions that follow may allow another warp to be selected.[4] Listing 3.3 shows an excerpt of an example instruction order involving three input arrays and an unrolling factor of 4. In this example, the instruction stream forms three SFLSs.

The intrinsic reuse distance of the linear-layout case can be categorized into three cases according to the referenced sector:

1. the sector has never been accessed before,
2. the sector was previously accessed in another SFLS, or
3. the sector was previously accessed earlier within the same SFLS.

Note that SFLSs from different loop iterations are considered distinct. The first access to a previously unseen sector (Case 1) always has infinite intrinsic reuse distance.

For sectors reused across SFLS boundaries (Case 2), the scheduler may issue instructions from other warps before the original warp resumes execution. Their accesses are therefore interleaved into the memory-access trace. Moreover, because the matrix width K is chosen such that each matrix row occupies exactly one cache line, different threads of a warp access disjoint cache lines and sectors. These intervening accesses therefore do not reference the sector being reused and only increase the number of distinct sectors accessed before reuse, resulting in a long reuse distance. Although the exact reuse distance depends on the detailed instruction ordering and scheduling behavior, accesses crossing SFLS boundaries consistently exhibit the longest finite intrinsic reuse distances.

[4] Compilers may therefore favor fewer, longer sequences when sufficient registers are available.

In contrast, sectors reused later within the same SFLS (Case 3) incur substantially shorter intrinsic reuse distances because fewer interleaved accesses occur before reuse. However, in the linear-layout case, such reuse occurs only when the loop-unrolling factor exceeds one. Across all evaluated values of N, the resulting reuse distances remain within the cache capacity.

Listing 3.3 illustrates all three cases. The accesses to `M1[i]`, `M2[i]`, and `M3[i]` correspond to Case 1 when `i` is a multiple of the number of elements per cache sector. Otherwise, the corresponding sectors were accessed during a preceding loop iteration and the accesses correspond to Case 2. The accesses to `M1[i+1]` and `M2[i+1]` are examples of Case 3 because the corresponding sectors were accessed earlier in the same SFLS. The accesses to `M3[i+1]`, `M1[i+2]`, and `M2[i+2]` are examples of Case 2 because the previous accesses to the same sectors occurred in SFLS_1. Likewise, the access to `M3[i+2]` is an example of Case 3 because the corresponding sector was previously accessed by `M3[i+1]` within SFLS_2. Finally, the accesses to `M1[i+3]`, `M2[i+3]`, and `M3[i+3]` are all examples of Case 2 because their previous accesses occur in earlier SFLSs.

3.2.2 Effective Reuse Distance

Effective reuse distance extends intrinsic reuse distance by accounting for additional stall events introduced by contention in the memory subsystem. Owing to its minimum cache-line utilization, the linear-layout case distributes each warp load across 32 cache lines and generates substantially more cache-access wavefronts than the transposed-layout case. When many such accesses are issued within a short interval, the available memory-level parallelism becomes limited by hardware resources such as cache ports, cache-bank conflicts, and MSHRs [23], [25], [40], [64]. Its minimum cache-sector utilization also causes each cache miss to transfer a full sector while consuming only one-eighth of its data. The resulting traffic increases pressure throughout the memory hierarchy, including the NoC and DRAM subsystem [45].

This contention increases the queuing delay incurred by memory requests. As outstanding requests accumulate, hardware resource limits may eventually constrain instruction issuance, preventing additional instructions from entering the queues. Once queue capacity is exhausted, issuance is suspended until sufficient entries become available. We refer to such stalls as *throttle stalls*. By delaying the remaining instructions of a warp and allowing accesses from other warps to intervene, throttle stalls increase the effective reuse distance beyond its intrinsic value.

The effective reuse distance is obtained by incorporating throttle stalls into the instruction stream. In the absence of throttle stalls, accesses reused later within the same SFLS (Case 3) can be issued continuously without interruption. In the linear-layout case, however, the low cache-line utilization of the load instructions generates many cache-access

Listing 3.3: An excerpt of an example instruction order for accesses to three matrices, M_1, M_2, and M_3, with an unrolling factor of 4. Consecutive load instructions form stall-free load sequences (SFLSs), shaded with alternating backgrounds.

```
// SFLS1:
a0 = M1[i]
b0 = M2[i]
c0 = M3[i]
a1 = M1[i+1]
b1 = M2[i+1]
// Dependency boundary:
use a0, b0, c0, a1, b1

// SFLS2:
c1 = M3[i+1]
a2 = M1[i+2]
b2 = M2[i+2]
c2 = M3[i+2]
// Dependency boundary:
use c1, a2, b2, c2

// SFLS3:
a3 = M1[i+3]
b3 = M2[i+3]
c3 = M3[i+3]
// Dependency boundary:
use a3, b3, c3
```

Listing 3.4: A possible execution order derived from Listing 3.3 after throttle stalls are introduced. Shading patterns match Listing 3.3 to track how individual SFLSs split across throttle boundaries.

```
// SFLS1:
a0 = M1[i]
b0 = M2[i]
c0 = M3[i]
// Throttle boundary
// SFLS1-continued:
a1 = M1[i+1]
b1 = M2[i+1]
// Dependency boundary:
use a0, b0, c0, a1, b1

// SFLS2:
c1 = M3[i+1]
a2 = M1[i+2]
b2 = M2[i+2]
// Throttle boundary
// SFLS2-continued:
c2 = M3[i+2]
// Dependency boundary:
use c1, a2, b2, c2

// SFLS3:
a3 = M1[i+3]
b3 = M2[i+3]
c3 = M3[i+3]
// Dependency boundary:
use a3, b3, c3
```

wavefronts, creating contention that may trigger a throttle stall before the subsequent reuse is issued.

When memory-subsystem contention is considered, a throttle stall conceptually introduces a new boundary within an original SFLS, dividing it into multiple smaller issue sequences. Although the reuse categories remain unchanged, accesses that would intrinsically be reused within the same SFLS may effectively cross a stall-induced boundary and allow accesses from other warps to intervene. More reuses therefore exhibit the behavior of Case 2 rather than Case 3, increasing their effective reuse distance. Listing 3.4 illustrates one possible stall-induced division of the instruction stream in Listing 3.3. Compared with the original instruction order, the accesses to `M1[i+1]` and `M3[i+2]` now cross stall-induced boundaries and consequently exhibit longer effective reuse distances.

3.3 Experimental Validation

The preceding analysis established how intra-warp cache utilization affects both cache reuse and memory-pipeline pressure. In the linear-layout case, minimum cache-sector utilization leaves most of each fetched sector to be accessed by subsequent instructions, making cache effectiveness dependent on the resulting reuse distance. Its minimum cache-line utilization also causes each warp load to generate many cache-access wavefronts, introducing contention that may further increase the effective reuse distance. By contrast, the transposed-layout case achieves maximum utilization at both granularities: each fetched sector is fully consumed by the issuing instruction, and the requested sectors are consolidated within a single cache line.

In this section, the case-study kernels were executed on real hardware to validate the predicted consequences of the reuse-distance analysis and assess their impact on actual performance. Execution time and detailed hardware performance counters were collected using NVIDIA Nsight Compute (Section 2.9).

Both kernels were evaluated on an NVIDIA GeForce RTX 4080 GPU, which is based on the Ada Lovelace microarchitecture and comprises 76 SMs with 128 CUDA cores each, co-scheduling up to 48 warps per SM, for a total capacity of 3,648 warps (116,736 threads). We used $Q = 58{,}368$ to achieve 50% occupancy,[5] with each SM hosting 24 warps, striking a balance between sufficient occupancy to observe the bottlenecks and avoiding excessive reuse distances that would accelerate cache thrashing (Section 3.3.3). The width was fixed at $K = 32$, which equals the number of `FP32` elements in a 128-byte cache line. Larger values of K that are multiples of 32 would merely replicate this access pattern without altering its fundamental cache behavior. The block size was set to 32 threads (1 warp), as the threads in both kernels neither communicate within blocks nor exhibit sensitivity to warp scheduling, making other multiples of 32 equivalent in behavior.

3.3.1 Cache-Access Wavefronts

Reuse distance alone does not characterize how efficiently a warp instruction uses the cache data fetched on its behalf or how much work it introduces into the cache pipeline. The transposed-layout case has no finite sector reuse because every fetched sector is fully consumed by a single warp instruction. The linear-layout case instead accesses each sector over multiple instructions. This cross-instruction spatial reuse arises precisely because each instruction consumes only one-eighth of every fetched sector.

The difference in intra-warp cache-line utilization determines how the requests are presented to the L1 cache. A cache-access wavefront comprises cache-line requests processed

[5] In this case study, occupancy is independent of N, as the register allocator maintains peak register usage below the occupancy-limiting threshold.

together by the cache data stage (Section 2.6). In the transposed-layout case, the four sectors requested by a warp load reside within one cache line. Its maximum cache-line utilization therefore allows the accesses to be coalesced into a single wavefront. In the linear-layout case, the 32 requested sectors reside in 32 distinct cache lines. Its minimum cache-line utilization consequently causes each warp load to generate 32 wavefronts, one for each requested line.

Figure 3.5 reports the number of wavefronts processed by the L1 cache across different values of N. The results highlight the fundamental distinction between the two memory layouts. The transposed-layout case consistently generated a small number of wavefronts because each cache line is accessed in a consolidated manner. In contrast, the linear-layout case generated substantially more wavefronts due to the low intra-warp cache-line utilization. Although later accesses may still exhibit reuse at the sector level, each of the 32 sectors must be serviced independently. This increase in memory traffic does not directly imply poor cache effectiveness; however, it establishes the conditions for memory-system contention and throttle stalls.

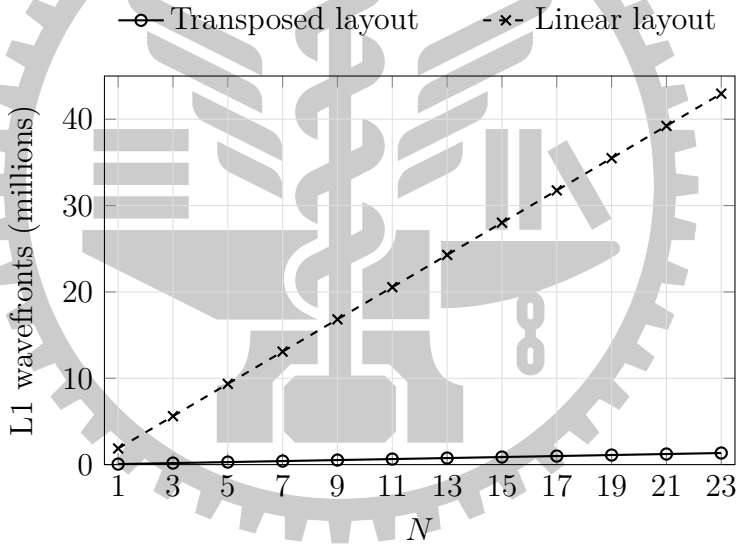


Figure 3.5: Number of wavefronts processed by the data stage of the L1 cache.

3.3.2 Scoreboard Stalls and Throttle Stalls

The larger number of wavefronts generated by the linear-layout case places additional pressure on the memory hierarchy, which leads to increased L2-cache access traffic, higher MSHR occupancy, and greater interconnect and DRAM request rates, all of which introduce queuing delays that are not presented by the intrinsic reuse distance. When such delays accumulate, instruction issuance becomes limited by hardware resource constraints, resulting in *LG throttle stalls*.[6] In this regime, instructions are prevented from issuing even

[6]LG denotes local and global memory.

after their dependencies have been resolved due to the unavailability of memory-system resources. In contrast, dependency-induced delays are captured by *long scoreboard stalls*,[7] which reflect the latency of outstanding global memory operations.

Figure 3.6 reports the normalized warp stall time attributed to long scoreboard stalls and LG throttle stalls. Stall time is normalized by the number of input matrices (N) to remove the linear increase in execution work as it grows, thereby exposing the effect of memory-system contention independently of kernel size. The transposed-layout case exhibited nearly constant long scoreboard stall time and negligible LG throttle stall time across all values of N, indicating that its high cache-line utilization causes little contention in the cache-access pipeline. The linear-layout case likewise exhibited long scoreboard stalls as warps waited for outstanding memory operations. More importantly, its normalized LG throttle stall time increased with N, showing that the wavefronts generated by its minimum cache-line utilization progressively saturate memory-system resources. Its measured long scoreboard stall time was lower because dependency waiting can overlap with or follow periods of throttle stalling.

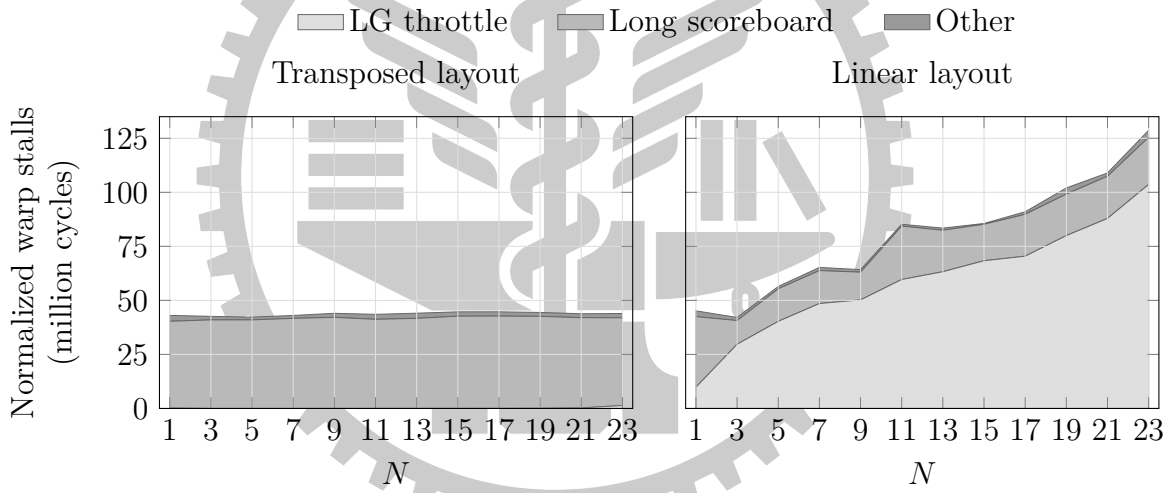


Figure 3.6: Normalized warp stall breakdown. Stall time is normalized by the number of input matrices (N). Distinct stall reasons are stacked to illustrate their contributions to the total stall time.

These results validate the distinction between intrinsic and effective reuse distance. Memory-subsystem contention introduces throttle stalls that interrupt otherwise stall-free load sequences and allow additional accesses from other warps to intervene. The resulting effective reuse distances are therefore longer than those predicted by instruction ordering and warp scheduling alone. Whether these longer distances cause cache misses depends on the cache's ability to retain partially consumed sectors until their remaining data are accessed, which is examined next.

[7]Long scoreboard stalls correspond to waiting for high-latency global-memory accesses. In contrast, short scoreboard stalls are associated with lower-latency shared-memory accesses.

3.3.3 Cache Effectiveness and Performance

After establishing the reuse-distance characteristics and the resulting stall behavior, Figure 3.7 examines their impact on cache effectiveness through the observed cache hit rates. In the transposed-layout case, each warp load accesses four sectors within a single cache line and fully consumes the fetched data. Because the access pattern is streaming, none of these sectors is accessed again. Every access is therefore compulsory, yielding 0% hit rates in both the L1 and L2 caches. Although this behavior appears unfavorable from the perspective of cache hit rates, it merely reflects the absence of reuse rather than inefficient use of the fetched data.

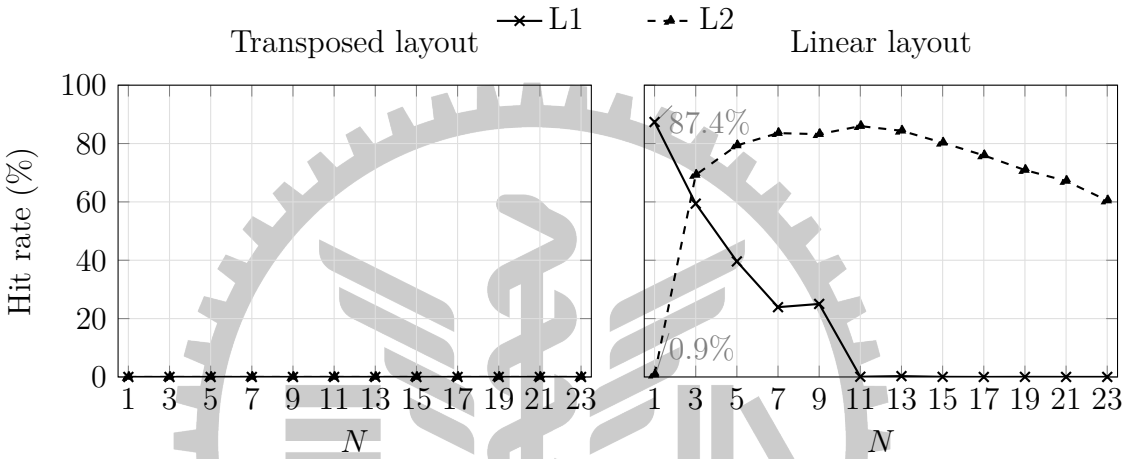


Figure 3.7: L1 and L2 cache hit rates.

The linear-layout case instead consumes each sector across eight separate load instructions because each instruction accesses only one of its eight `FP32` elements. Following the initial compulsory miss, the remaining seven accesses can therefore hit in the cache, giving an ideal L1 hit rate of 87.5%. The corresponding L2 hit rate remains 0% if all reuses are satisfied by the L1 cache. This ideal behavior was observed only at small values of N. In particular, when $N = 1$, the compiler selected an unrolling factor of eight, placing all eight accesses to a sector within a single SFLS. The sector is therefore fully consumed before warp scheduling can introduce cross-SFLS reuse, eliminating the long finite intrinsic reuse distances associated with Case 2.

Even with this near-ideal hit rate, the linear-layout case remained slower than the transposed case. Retaining the partially consumed sectors avoids repeated fetches from lower cache levels, but it does not reduce the 32 cache-access wavefronts generated by each load instruction. The resulting cache-pipeline pressure and queuing delays demonstrate that minimum intra-warp cache-line utilization can degrade performance even when the cache successfully preserves the spatial locality left for subsequent instructions.

As N increases, accesses to more input matrices are interleaved within the instruction stream, placing more distinct sectors between consecutive accesses to the same sector.

Increasing register pressure may also shorten SFLSs, causing more reuses to cross SFLS boundaries. Together, these effects increase the intrinsic reuse distance, while the resulting memory-system contention further increases the effective reuse distance. Consistent with these effects, capacity and conflict misses progressively reduced the L1 hit rate, leaving the L1 cache largely ineffective for $N \geq 11$. Reuses were then serviced increasingly by the L2 cache or DRAM.

The resulting memory traffic is quantified in Figure 3.8, which reports the number of sectors fetched at each level of the memory hierarchy. A single input matrix occupies 233,472 sectors; therefore, the gray dotted reference line corresponds to $233{,}472N$ sectors, representing the minimum number of sector fetches required to access each input matrix once. The transposed-layout case followed this baseline because each fetched sector was fully consumed by one warp instruction. By contrast, the linear-layout case issued 8 L1 sector accesses for every sector of input data, consistent with its cache-sector utilization of 1/8. Its L1 sector count was therefore uniformly 8× the reference value, even when the subsequent accesses hit in the L1 cache. As cache effectiveness deteriorated, the same sectors had to be fetched repeatedly from the L2 cache and DRAM, increasing traffic throughout the lower memory hierarchy.

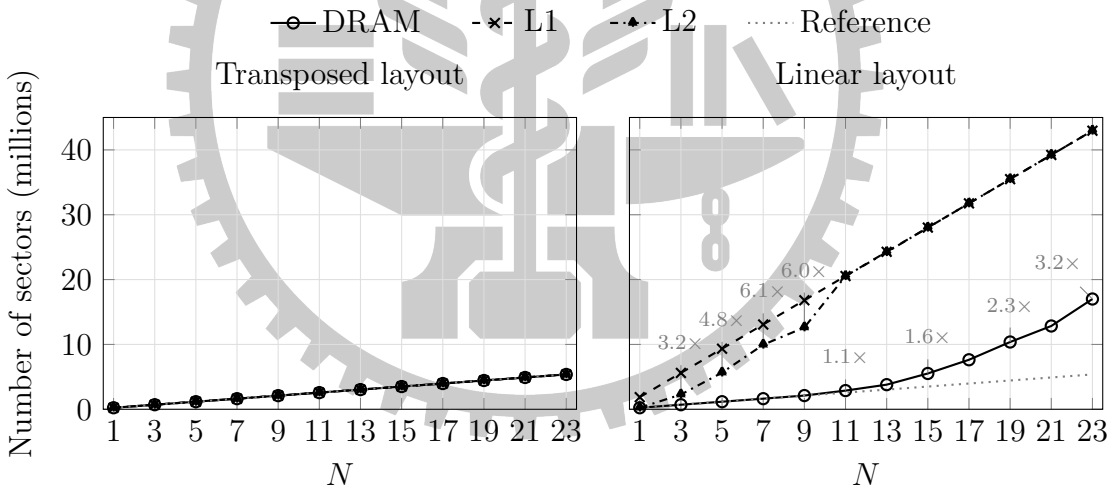


Figure 3.8: Number of sectors fetched from DRAM, L1, and L2. The gray dotted reference line indicates the total data volume of the N input matrices ($233{,}472N$ sectors). Annotations show the ratio relative to the reference; ratios are omitted for the transposed-layout case, where they are uniformly 1×, and for the L1 count in the linear-layout case, which is uniformly 8×.

The additional sector fetches in the linear-layout case increase pressure on the lower memory hierarchy, exacerbating memory-system contention and the associated queuing delays and throttle stalls (Figure 3.6). These effects accumulate as N increases and ultimately manifest as a widening execution-time gap between the linear- and transposed-layout cases, as shown in Figure 3.9.

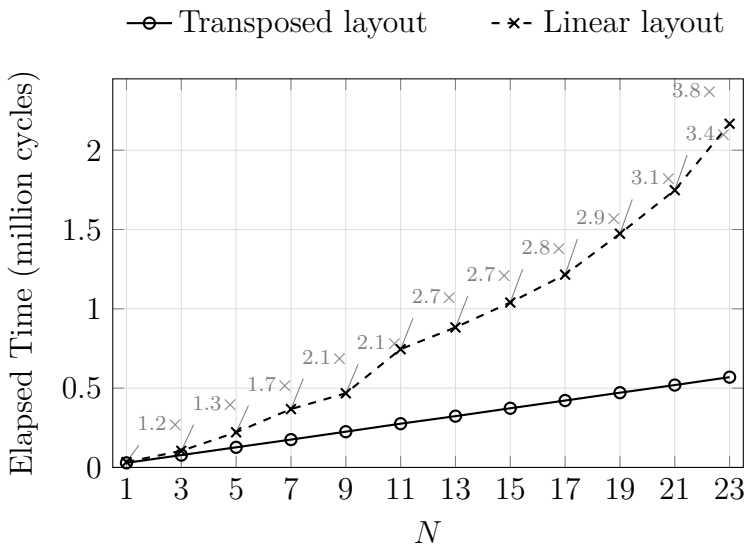


Figure 3.9: Elapsed kernel-execution times. Annotations indicate the slowdown ratio of the linear-layout kernel relative to the transposed-layout baseline.

3.4 Problem Statement

The analysis reveals that the performance degradation originates from a mismatch between the logical access structure of the computation and the linear physical memory layout used by global memory. This mismatch distributes the addresses requested by a warp across many cache sectors and lines, resulting in low intra-warp cache utilization.

Low cache-sector utilization causes each warp instruction to consume only a small fraction of the data fetched on its behalf. The remaining data must be accessed by subsequent instructions, making cache effectiveness dependent on sufficiently short reuse distances and successful retention of partially consumed sectors. Low cache-line utilization instead distributes the requested sectors across many cache lines, increasing cache-access wavefront generation and pressure on the memory subsystem. The resulting contention introduces queuing delays and throttle stalls, directly degrading performance and further increasing effective reuse distances. Poor intra-warp cache utilization therefore harms performance through both inefficient use of fetched cache data and increased memory-pipeline contention.

To address this mismatch, this work develops a memory-layout mapping mechanism that aligns the physical placement of data with the logical access structure of the computation, thereby improving intra-warp cache utilization without requiring manual restructuring of application data layouts. Higher cache-sector utilization reduces dependence on cross-instruction reuse and cache retention, while higher cache-line utilization consolidates cache accesses into fewer wavefronts. Together, these effects alleviate memory-system contention, shorten effective reuse distances, improve cache effectiveness, and ultimately enhance performance.

Chapter 4
Related Work

The motivation (Chapter 3) shows that the performance degradation caused by uncoalesced memory accesses arises from two closely related effects. First, low intra-warp cache-line utilization distributes the sectors requested by a warp across many cache lines, increasing cache-line occupancy, cache-access wavefront generation, and pressure on the memory subsystem. Second, low intra-warp cache-sector utilization leaves much of each fetched sector to be consumed by subsequent instructions, making cache effectiveness dependent on sufficiently short reuse distances. Instruction interleaving and memory-system backpressure may extend these distances, causing partially consumed sectors to be evicted before their remaining data are accessed. Prior work has addressed these effects by restructuring memory-access patterns, managing cache resources, staging data in software-managed memory, or reorganizing the physical placement of data.

These approaches intervene at different stages of memory access. Access-pattern transformations modify the computation or thread-to-data mapping so that warps generate more favorable address distributions. Cache-management techniques regulate memory requests after they have been generated, typically by controlling insertion, retention, bypassing, or scheduling. Scratchpad-based techniques explicitly stage data in software-managed scratchpad memory to exploit reuse and improve access regularity. Memory-layout techniques instead reorganize physical data placement so that the original computation achieves higher intra-warp cache utilization. This chapter reviews representative work in these directions and identifies their limitations.

4.1 Access-Pattern Transformations

Access-pattern transformations restructure the computation so that warp instructions generate more favorable address distributions. Rather than changing the memory hierarchy or physical data layout, these techniques modify the mapping between threads and data, the traversal order of the computation, or both. Such transformations can improve intra-warp cache utilization by concentrating the addresses requested by a warp within fewer cache sectors and cache lines.

Fauzia et al. [65] address uncoalesced accesses through thread remapping. Their framework dynamically identifies inefficient memory accesses in PTX code and applies polyhedral analysis to derive alternative thread-to-data mappings that reduce access strides. By permuting thread geometries, the approach can improve intra-warp cache utilization and, correspondingly, memory coalescing. However, the transformed accesses are not guar-

anteed to become unit stride. Their requested sectors may therefore remain distributed across multiple cache lines, while each fetched sector may still be only partially consumed by a warp instruction. The row-summation kernel in Listing 3.2, for example, cannot attain maximal utilization through thread remapping alone without changing the correspondence between threads and output elements. Similarly, Khan et al. [66] propose a script-based autotuning framework that explores alternative data-placement and thread-mapping strategies through transformation recipes. These transformations can improve cache utilization by changing which data are accessed concurrently by a warp, although their effectiveness depends on finding a suitable mapping and may require substantial restructuring of the original computation.

More general polyhedral frameworks, such as PPCG [67], attempt to bridge the gap between high-level programs and efficient GPU implementations. PPCG applies multilevel tiling and separates locality optimization from parallelization decisions, allowing thread-block configurations to be chosen independently of shared-memory and register allocations. By reorganizing loop nests and thread mappings, PPCG can improve temporal reuse and concentrate concurrent accesses within fewer cache sectors and lines. These benefits, however, arise through extensive transformation of the iteration space and execution mapping rather than through a change in the physical memory layout.

Collectively, access-pattern transformations improve memory behavior before requests enter the memory hierarchy. They may increase intra-warp cache-sector and cache-line utilization by changing which addresses are generated together by a warp. However, they generally require nontrivial changes to loop structure or thread mapping, and the transformed access distribution is not guaranteed to achieve high utilization at either cache granularity.

4.2 Hardware Cache-Management Techniques

Rather than modifying the address distribution generated by each warp, hardware cache-management techniques mitigate its consequences at runtime. These approaches cannot directly improve intra-warp cache-sector or cache-line utilization because the requested addresses have already been determined. Instead, they seek to preserve partially consumed cache data across long reuse distances, reduce cache pollution and thrashing, or limit the memory-system contention caused by large numbers of cache-line requests and wavefronts.

One prominent direction is locality-aware warp scheduling. Low cache-line utilization can cause a single warp instruction to occupy many cache lines and generate many concurrent cache requests, allowing the aggregate working set of resident warps to exceed the L1 cache capacity. Rogers et al. [38] propose *cache-conscious wavefront scheduling*, which detects lost intra-wavefront locality using a victim-tag array and dynamically throttles the number of active warps. By reducing inter-warp interference and limiting concur-

rent memory requests, the scheduler seeks to preserve cache residency while alleviating pressure on the memory subsystem.

Complementing scheduling-based approaches, Duong et al. [68] introduce *protecting distance*, a cache-management mechanism that retains cache lines for approximately the interval preceding their next reuse. The policy dynamically samples reuse-distance distributions and estimates a protection interval intended to improve cache effectiveness. For non-inclusive cache hierarchies, the design also bypasses the cache when no suitable victim line is available. Building on these concepts, Chen et al. [69] combine reuse-distance-aware cache bypassing with adaptive warp throttling to preserve useful cache contents while limiting pressure on on-chip interconnect resources.

Another line of work uses locality information to guide cache insertion and retention. Li et al. [70] propose a dynamic bypassing mechanism that separates tag and data storage, allowing hardware to identify accesses with poor locality. Access streams with long reuse distances can bypass the cache, preventing them from displacing data more likely to be reused. Similarly, Seshadri et al. [71] introduce the *evicted-address filter*, which uses a Bloom filter to distinguish high-reuse from low-reuse cache lines. By selectively retaining only a subset of a thrashing working set, the mechanism reduces cache pollution and conflict-induced thrashing. Koo et al. [72] further propose access-pattern-aware cache management, which uses an exemplary warp to predict the locality characteristics of individual load instructions. Based on these predictions, the hardware bypasses streaming accesses or pins frequently reused cache lines, reducing cache contention and improving cache effectiveness.

These techniques address the consequences of low intra-warp cache utilization after memory requests have been generated. In particular, retention and protection mechanisms can increase the probability that partially consumed sectors remain available for subsequent accesses, while warp throttling and bypassing can reduce the contention associated with requests distributed across many cache lines. However, they do not reduce the number of sectors or cache lines touched by the original warp instruction. Their benefits therefore rely on restricting concurrency, selectively discarding requests from the cache, or introducing additional hardware structures and control logic.

4.3 Scratchpad-Memory Strategies

Scratchpad-memory strategies improve memory behavior by explicitly staging data in software-managed shared memory rather than relying solely on hardware-managed caches. Shared memory acts as a programmer- or compiler-managed buffer through which global-memory accesses can be reorganized before the data are consumed. These techniques can therefore improve the intra-warp cache utilization of the global-memory transactions used for staging, even when the computation itself retains its original access structure.

CUDA-Lite [73] is a seminal example of this approach. Using programmer annotations, CUDA-Lite performs source-to-source transformations that automate data movement between global and shared memory while generating coalesced global-memory accesses. Shared memory may be used even when no data reuse exists, serving primarily as a staging buffer that converts non-unit-stride accesses into unit-stride transfers with higher intra-warp cache-line utilization. However, the data must first be loaded into shared memory and later consumed from it. The resulting data-movement and synchronization overheads can outweigh the benefit unless they are hidden through techniques such as double buffering [74], [75] or multistage pipelining [76].

Yang et al. [77] similarly use shared memory as a staging buffer to improve global-memory access efficiency. Their compiler-generated transformation reorganizes global-memory traffic through shared memory so that the staging loads and stores access fewer cache lines and achieve more favorable coalescing behavior.

The same principle appears in broader compilation frameworks. Khan et al. [66] combine thread remapping with shared-memory staging, allowing their autotuning framework to optimize both computation mapping and data movement. PPCG [67] likewise uses shared memory within its polyhedral code-generation framework to convert non-unit-stride global-memory accesses into more consolidated transfers while preserving the semantics of the original computation.

Collectively, scratchpad-memory strategies improve global-memory access efficiency by replacing the original access stream with explicit staging transfers that achieve higher intra-warp cache utilization. However, the improvement does not arise from changing the physical layout of the original data. Instead, it requires additional loads, stores, synchronization, and scratchpad capacity. Their effectiveness therefore depends on whether the reduction in cache-line requests and memory-pipeline pressure outweighs the overhead of staging the data through shared memory.

4.4 Memory-Layout Strategies

Memory-layout strategies improve memory behavior by changing how logical data structures are mapped to physical memory. Unlike access-pattern transformations, these approaches preserve the computation and its logical access structure. Instead, they reorganize data so that elements accessed together by a warp are placed within fewer cache sectors and cache lines, thereby improving intra-warp cache utilization.

Morton-order (Z-order) layouts [78] are a widely studied example. By recursively interleaving coordinate bits, Morton layouts preserve proximity across multiple dimensions and provide a compromise between row-major and column-major organizations. This multidimensional locality can increase both cache-sector and cache-line utilization for traversal patterns that would otherwise access widely separated locations under a linear

layout. Wise et al. [79] show that Morton-order arrays naturally support cache-oblivious algorithms [80] and can reduce page faults and translation lookaside buffer (TLB) misses throughout the memory hierarchy. Thiyagalingam et al. [81] further demonstrate that Morton layouts can avoid the pathological behavior of large-stride traversals, although their effectiveness depends on proper alignment of array bases to page boundaries.

Related work has also investigated block-based layouts in conjunction with tiling transformations. Park et al. [82] show that partitioning matrices into contiguous submatrices can reduce TLB misses and eliminate self-interference misses in direct-mapped caches. By placing the elements of each tile contiguously, block layouts reduce the number of cache lines occupied by accesses within the tile and improve the utilization of the fetched cache data across multiple levels of the memory hierarchy.

Memory-layout strategies directly address the physical organization that determines intra-warp cache utilization. They can improve sector utilization by placing more of the data consumed by a warp within each fetched sector and improve line utilization by consolidating the requested sectors within fewer cache lines. However, existing approaches often require explicit layout conversions and more complex address computations, whose overheads can offset the resulting gains.

4.5 Summary and Limitations

Existing approaches improve GPU memory efficiency through four complementary directions. Access-pattern transformations change the computation or thread-to-data mapping before requests reach the memory hierarchy. Hardware cache-management techniques mitigate inefficient request streams through warp scheduling, cache bypassing, and reuse-aware insertion or retention policies. Scratchpad-memory strategies reorganize global-memory traffic through explicit staging in software-managed shared memory. Memory-layout strategies instead change the physical placement of data to improve intra-warp cache utilization without altering the logical computation. These directions are largely orthogonal and can be combined to address distinct sources of memory-system inefficiency.

The approach proposed in this work belongs to the class of memory-layout strategies. Unlike software-managed layouts, it exploits the hardware-managed block-linear organization already provided by the texture subsystem. It therefore requires neither explicit layout conversion nor software implementation of layout-specific address computations. Compiler-guided analyses and transformations transparently identify eligible data and map them to this layout, allowing existing GPU programs to benefit without manual code modification.

By improving intra-warp cache-sector utilization, the approach proposed in this work allows each warp instruction to consume a larger fraction of the fetched sector data, reducing dependence on cross-instruction spatial reuse and cache retention. By improving intra-

warp cache-line utilization, it consolidates the requested sectors within fewer cache lines, reducing cache-line occupancy, cache-access wavefront generation, and memory-system contention. These effects shorten effective reuse distances and improve cache effectiveness and execution performance. Because the approach changes physical data placement rather than the logical access pattern or hardware cache policy, it remains complementary to access-pattern transformations, scratchpad-memory strategies, and hardware cache-management techniques.

Chapter 5
Texture Memory as a Layout-Mapping Mechanism

The performance degradation originates from a mismatch between the logical access structure of the computation and the linear physical layout of global memory. This mismatch distributes the addresses requested by a warp across many cache sectors and cache lines, resulting in low intra-warp cache utilization. Low cache-sector utilization increases dependence on retaining partially consumed sectors for subsequent accesses, while low cache-line utilization increases cache-line occupancy and cache-access wavefront generation. Together, these effects increase pressure throughout the memory system and ultimately lead to greater memory-system contention.

This chapter investigates the hardware-managed block-linear layout provided by the texture subsystem as a mechanism for addressing this mismatch. Unlike the conventional linear layout, which preserves contiguity primarily along one dimension, the block-linear layout groups elements from multiple logical dimensions within nearby physical locations. It can therefore consolidate the addresses requested by a warp into fewer cache sectors and cache lines, increasing both intra-warp cache-sector and cache-line utilization.

This observation establishes physical data placement as an alternative optimization dimension to access-pattern transformation. The logical coordinates generated by the computation need not change; instead, changing how those coordinates are mapped to physical memory can alter the cache sectors and cache lines accessed by each warp instruction. Improving intra-warp cache utilization therefore does not necessarily require restructuring the computation or remapping threads to data.

Texture memory exposes such a mapping through its block-linear organization (Section 2.3.3). The texture subsystem performs the corresponding address computation in hardware, allowing kernels to access multidimensional data without explicitly implementing the physical layout mapping. Higher cache-sector utilization allows each warp instruction to consume a larger fraction of the fetched sector data, reducing dependence on cross-instruction spatial reuse. Higher cache-line utilization consolidates the requested sectors within fewer cache lines, reducing cache occupancy, cache-access wavefront generation, and pressure on the downstream memory subsystem. These effects may also shorten effective reuse distances and improve cache effectiveness. The degree of improvement depends on how logical coordinates are mapped within the block-linear layout. This organization is governed by layout parameters that determine the dimensions and internal arrangement of its physical blocks. The remainder of this chapter characterizes these

parameters using the row-summation case study and analyzes how the resulting physical organization affects intra-warp cache utilization, cache-access wavefront generation, reuse distance, and execution performance.

5.1 Case Study on Texture Memory

To examine how the block-linear layout affects intra-warp cache utilization, we revisit the row-summation case study under the linear layout (Section 3.1.1). Each thread again accumulates one logical row across all input matrices, preserving both the computation and the thread-to-data mapping. The only difference is that the input matrices are allocated in texture memory and accessed through the texture subsystem. Each matrix is represented as a two-dimensional texture object at the base mipmap level, with each `FP32` matrix element stored as one texel. The terms *matrix element* and *texel* are therefore used interchangeably.

Listing 5.1: CUDA kernel for the row-summation case study using texture memory. Compared with Listing 3.1, the input matrices are represented as texture objects and accessed through texture-fetch operations.

```
__global__ void sumRows(float *__restrict__ v,
        cudaTextureObject_t m1 , ..., cudaTextureObject_t mN ) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    float vi = 0;
    for (int j = 0; j < K; j++) {
        vi += tex2D<float>(m1, j, i) + ... + tex2D<float>(mN, j, i) ;
    }
    v[i] = vi;
}
```

Although the kernel generates the same logical coordinates as the linear-layout case, texture memory maps those coordinates through a hardware-managed block-linear layout. Elements that are widely separated under the linear layout may therefore occupy nearby physical locations when they belong to the same block-linear region. The physical cache-sector and cache-line distribution of a warp instruction can consequently differ substantially even though its logical access structure remains unchanged. The resulting intra-warp cache utilization depends on the dimensions and internal organization of the block-linear regions. We therefore first characterize their cache-sector and cache-line organization using controlled microbenchmarks. We then analyze how this organization affects intra-warp cache-sector and cache-line utilization, cache-access wavefront generation, reuse distance, and performance in the row-summation case study.

5.2 Block-Linear Layout Characterization

The block-linear layout is hierarchically organized into groups of bytes (GOBs) and larger blocks, but the dimensions and internal arrangement of these structures are not publicly documented in sufficient detail for the architectures evaluated in this work. Because this physical organization determines how texture coordinates map to cache sectors and cache lines, we characterize the layout empirically using controlled microbenchmarks.

The experiments isolated individual layout properties while minimizing cache-capacity and contention effects. Throughout the characterization, we allocated a $58{,}368 \times 32$ two-dimensional texture of `FP32` texels, matching the dimensions of the input matrices in the row-summation case study. Each experiment issued only a small number of texture requests so that the observed behavior primarily reflects texel placement rather than interference within the memory hierarchy.

Because the L1 cache transfers data at cache-sector granularity but performs replacement at cache-line granularity, the characterization proceeds at both levels. We first determine which texels share cache sectors and then identify how those sectors are grouped into cache lines. These results provide the physical basis for evaluating intra-warp cache-sector and cache-line utilization under the block-linear layout.

5.2.1 Texture-Fetch Operation

The characterization microbenchmarks access the texture through the two-dimensional CUDA texture-fetch API. The same operation is later generated by the texture-access transformation described in Section 6.2.4. For a scalar `FP32` texture, the source-level operation has the following form:

```
float value = tex2D<float>(texObj, x, y);
```

where `texObj` is a texture-object handle and x and y identify the requested texel. The texture objects used in the characterization are configured with unnormalized coordinates, point filtering, and element-type read mode. Thus, the supplied coordinates directly identify a single texel, and each active thread requests one `FP32` value.

The source-level operation is lowered to a PTX texture instruction with the following representative form:

```
tex.2d.v4.f32.f32 {%f3, %f4, %f5, %f6}, [%rd1, {%f1, %f2}];
```

The 64-bit register `%rd1` contains the texture-object handle, while `%f1` and `%f2` contain the two floating-point texture coordinates. The `v4` designation indicates that the PTX instruction exposes four scalar destination components. For a scalar `FP32` texture, however, only the first component contains the requested texel value; the remaining components are not consumed by the program.

In the generated SASS, these four logical components are represented by two destination operands, each denoting a pair of consecutive registers. For example, destination operands `R1` and `R5` represent the physical-register pairs (`R1`, `R2`) and (`R5`, `R6`), respectively. The texture fetch can therefore be viewed as producing two 2-component destination tuples. When only the scalar texel value is required, the first component of the first tuple contains the result, while the remaining components are unused. The second destination tuple is consequently encoded using the zero register `RZ`, avoiding the allocation of registers for those unused outputs.

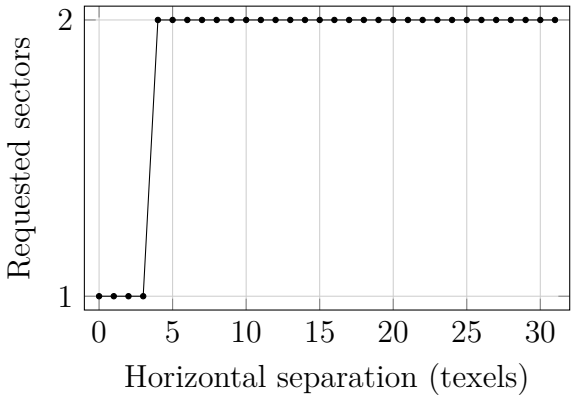
This paired-fetch behavior affects the sector count observed by probes placed near a sector boundary, but it does not affect the cache-sector structure inferred from the experiments. We therefore account for it during characterization but do not otherwise consider it in the remainder of this work.

5.2.2 Cache-Sector Organization

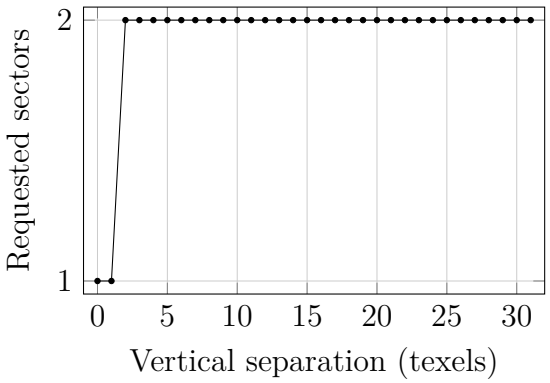
We first characterize the two-dimensional region covered by a cache-sector transfer. Each experiment issued one texture request containing two accesses: one to a fixed texel and another to a texel displaced by a controlled offset. The displacement was varied independently along the horizontal and vertical dimensions. If both texels reside in the same cache sector, the request accesses only one sector. Crossing a sector boundary increases the count to two. The number of requested sectors therefore reveals the spatial extent of a sector under the block-linear layout.

Figure 5.1 summarizes the characterization of the cache-sector organization. Figures 5.1 (a) and 5.1 (b) present the horizontal and vertical distance probes, respectively. Along the horizontal dimension, the number of requested sectors increased from one to two when the texel separation reached four, whereas the transition occurred at a separation of two texels in the vertical dimension. Figures 5.1 (c) and 5.1 (d) present the corresponding phase scans, in which the texel separation is fixed to one while the origin is shifted. In the horizontal scan, the number of requested sectors increased at origins $x = 2, 6, 10, \ldots$. These transitions occurred one texel before the physical sector boundaries because each scalar texture fetch is executed as a two-component fetch that also accesses the immediately following texel. For example, a probe originating at $x = 2$ logically requests texels 2 and 3, but the paired fetches extend the effective access footprint through texel 4, thereby reaching the next sector.

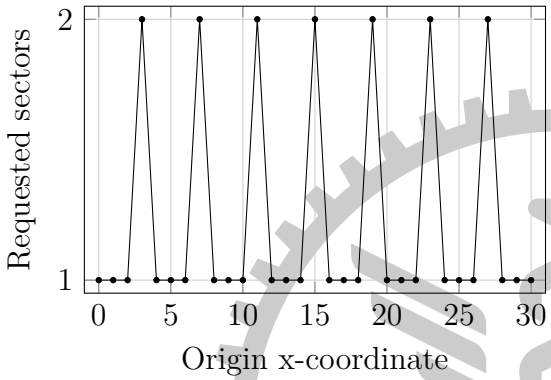
This prefetch-like behavior changes the observed access footprint but does not alter the underlying cache-sector organization. After accounting for the additional fetched component, the transitions still repeated every four texels and therefore confirmed a horizontal sector extent of four texels. In the vertical scan, transitions occurred at every



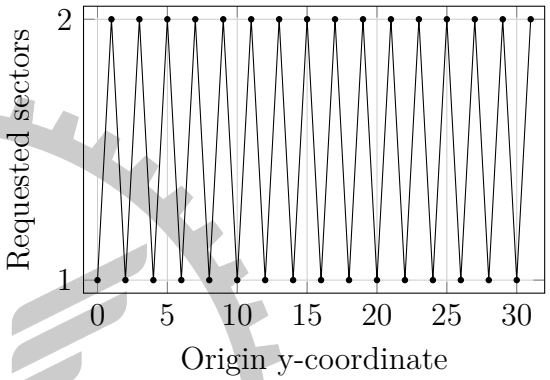
(a) Horizontal distance probe with origin $(0, 0)$.



(b) Vertical distance probe with origin $(0, 0)$.



(c) Horizontal phase scan with origin $(x, 0)$ and separation of one texel.



(d) Vertical phase scan with origin $(0, y)$ and separation of one texel.

Figure 5.1: Characterization of the cache-sector organization. The horizontal and vertical distance probes identify the sector boundaries, while the corresponding phase scans verify that the boundaries repeat periodically across the texture.

odd origin, $y = 1, 3, 5, \ldots$, confirming a vertical sector extent of two texels. Together, the phase scans verify that the sector boundaries repeat periodically across the texture.

These observations indicate that a cache sector covers a 4×2 region of `FP32` texels. The result further constrains the dimensions of the underlying block-linear hierarchy. Because texels are stored sequentially within the hierarchy, the horizontal extent of a GOB cannot exceed four texels; otherwise, the sector boundary would be crossed at a larger horizontal displacement. However, the experiment alone cannot uniquely determine the dimensions of a GOB, as smaller hierarchical units may exhibit the same sector behavior.

5.2.3 Cache-Line Organization

The preceding experiment identifies which texels share a sector transfer but does not reveal how sectors are grouped into cache lines for replacement. We therefore use selective eviction to characterize the cache-line organization. The experiment first accessed a target texel and a distant sentinel texel. A sequence of pressure accesses was then issued to

occupy cache capacity while minimizing interference with unrelated regions of the texture. Finally, the target and sentinel texels were accessed, and lookup misses were recorded.

The pressure sequence was adjusted until the target texel was consistently evicted while the distant sentinel remained resident. This established a selective conflict pattern that evicts the cache line containing the target texel without causing widespread cache eviction.[1] Because of the XYZ ordering, only sector regions that are horizontally or vertically adjacent can belong to the same cache line. The distant sentinel is therefore replaced by neighboring regions separated by the previously determined sector dimensions. If a neighboring region remains resident after the target region has been evicted, the two regions must reside in different cache lines. Conversely, if both regions are evicted together, they are inferred to reside within the same cache line.

When the neighboring region immediately adjacent to the target in the horizontal direction was tested, both the target and neighboring region were evicted together. The same behavior was observed for the vertically adjacent region. These observations indicate that the cache line extends across neighboring sector regions in both dimensions. The smallest cache-line organization consistent with these observations is therefore a 2×2 arrangement of the previously measured sector regions. Because each region occupies one 32-byte sector, the resulting cache line occupies four sectors, corresponding to 128 bytes.

5.2.4 Inferred Block-Linear Organization

Combining the two experiments yields the locality structure exposed by the block-linear layout. A cache sector spans a 4×2 region of `FP32` texels, while a cache line spans a 2×2 arrangement of such regions, corresponding to an 8×4 region of `FP32` texels. Logical coordinates falling within these regions are therefore mapped to the same sector or line despite spanning both texture dimensions.

Although this organization is consistent with a block-linear hierarchy in which sector- and line-sized regions align with lower-level layout structures, the experiments observed cache behavior rather than the hierarchy directly. We therefore interpret the results conservatively as the cache-sector and cache-line organization exposed to texture accesses.

Figure 5.2 applies the inferred organization to the 32×32 logical input sub-matrix accessed by the row-summation kernel. Rows correspond to the texels assigned to individual threads, while columns correspond to successive iterations of the summation loop. For a warp instruction accessing one logical column, every two adjacent threads address texels within the same cache sector, and every four adjacent threads address texels within the same cache line. The 32 addresses generated by the warp therefore access 16 sectors distributed across 8 cache lines, compared with 32 sectors in 32 lines for the linear layout.

[1] The pattern may be further exploited to characterize the address-to-set mapping of the cache. Such an analysis, however, is beyond the scope of this work.

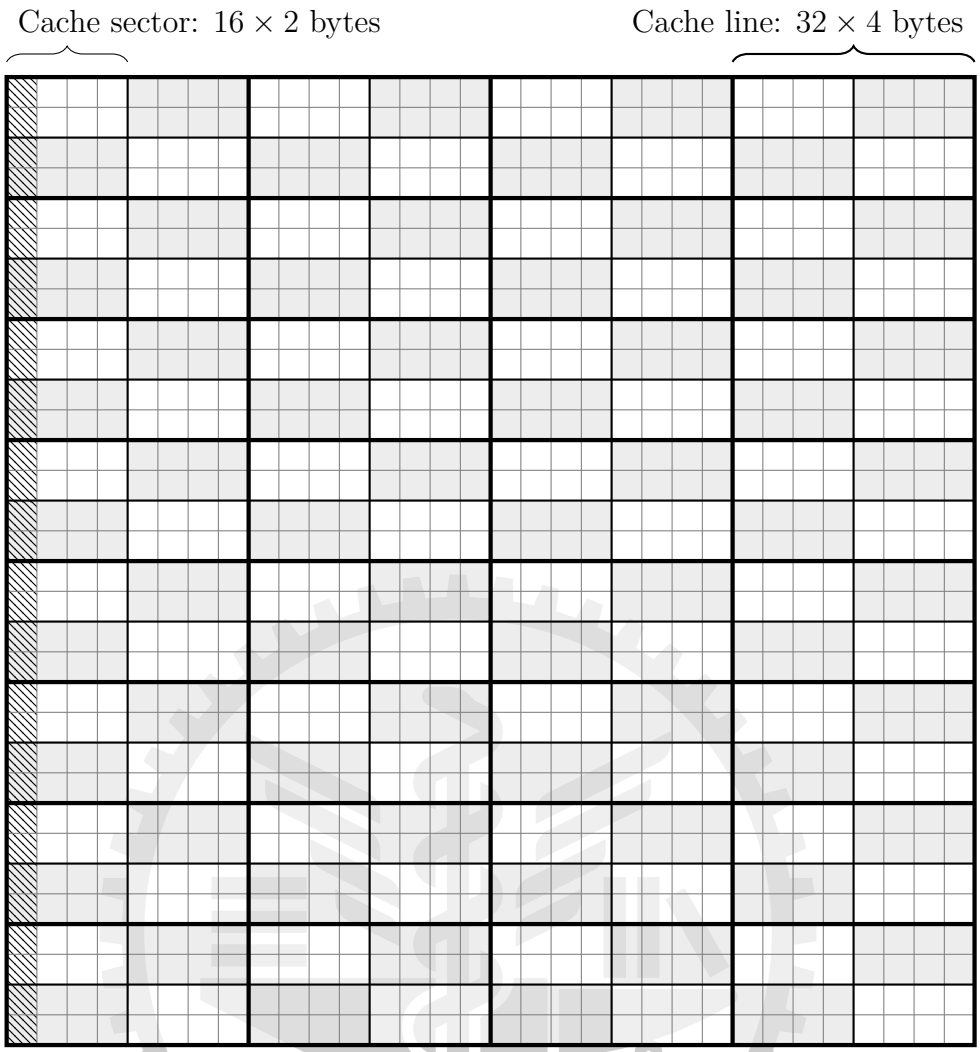


Figure 5.2: Cache-line organization of a 32×32 logical input sub-matrix accessed by a warp under the inferred block-linear layout. Thick and thin boundaries indicate cache lines and sectors, respectively. The alternating checkerboard pattern distinguishes adjacent sectors. Shaded elements indicate the matrix elements accessed by the warp during the first iteration of the summation loop.

Because each thread accesses one 4-byte texel, the corresponding intra-warp cache utilizations are

$$U(\text{sector}) = \frac{32 \times 4}{16 \times 32} = \frac{1}{4}, \qquad U(\text{line}) = \frac{32 \times 4}{8 \times 128} = \frac{1}{8}.$$

The block-linear layout thus doubles cache-sector utilization and quadruples cache-line utilization relative to the linear layout. Each warp instruction leaves less fetched sector data for subsequent instructions and occupies fewer cache lines, reducing both dependence on cross-instruction spatial reuse and cache-access wavefront generation.

5.3 Cache-Access Wavefront Reduction

For global-memory accesses, each requested cache line generates a separate cache-access wavefront (Section 3.3.1). The linear-layout row-summation kernel therefore generates 32 wavefronts per warp load, whereas the transposed-layout kernel generates only one. Figure 5.3 compares these implementations with the texture-memory variant. By increasing intra-warp cache-line utilization and reducing the number of requested cache lines from 32 to 8, the block-linear layout substantially reduced the total number of wavefronts. The texture-memory result consequently approached that of the transposed-layout kernel despite preserving the original logical access structure.

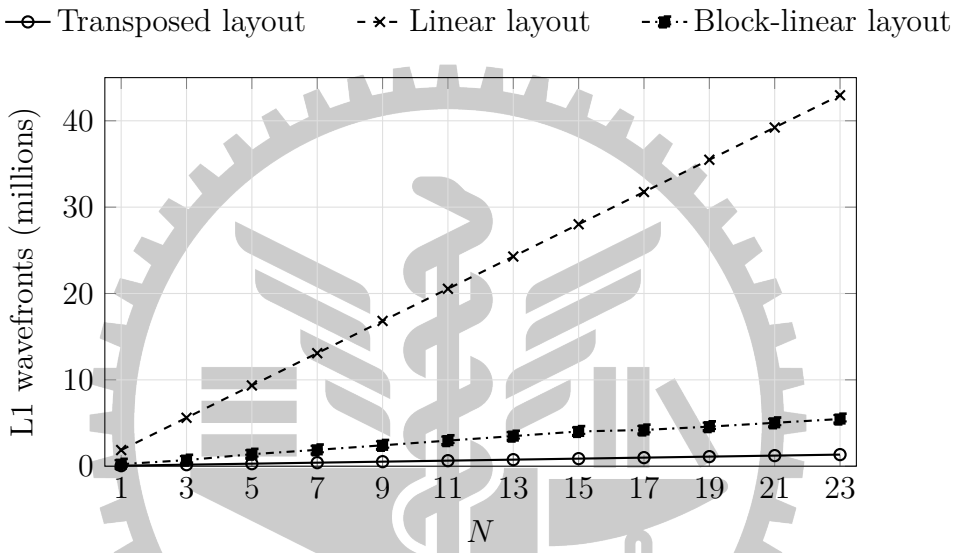


Figure 5.3: Number of cache-access wavefronts generated under three memory organizations. The transposed- and linear-layout global-memory curves are reproduced from Figure 3.5, with the texture-memory implementation added for comparison.

To understand this behavior, we investigate how the texture subsystem coalesces the accesses generated by a warp and subsequently forms cache-access wavefronts. As discussed for the memory-access pipeline (Section 2.6), request coalescing groups the thread-level accesses of a warp into memory-oriented requests, whereas each cache stage may process those requests as one or more wavefronts. Microbenchmarking revealed that the texture subsystem does not employ the same request-coalescing and wavefront-processing rules as the global-memory path. In particular, both horizontal and vertical traversals produced approximately 4 wavefronts per warp instruction despite exhibiting markedly different cache-line footprints.

The profiler does not expose the individual lane-level accesses before request coalescing. Nevertheless, targeted microbenchmarks indicated that the accesses generated by a warp instruction were first coalesced into 4 independent requests, each corresponding to a fixed group of 8 lanes. Cache-access wavefronts are then formed independently within each

coalesced request. The exact rule used to merge accesses within an 8-lane group could not be uniquely determined from the experiments and appears to depend on both the spatial arrangement of the requested texels and their assignment to lanes.

Although texture memory generated more wavefronts than the transposed global-memory case, it generated far fewer than the linear-layout case. The improvement arises from both the higher cache-line utilization provided by the block-linear layout and the wavefront-processing behavior of the texture subsystem. Together, these effects eliminate the 32-wavefront behavior of the linear layout, reducing cache-pipeline pressure and memory-system contention without modifying the logical computation.

5.4 Reuse Distance Reduction

The static instruction sequence and dynamic warp-scheduling order determine the order of logical memory accesses, whereas the physical layout determines the cache sectors and cache lines to which those accesses resolve. The block-linear layout therefore alters cache-granularity reuse distance while preserving the computation and logical access sequence.

Under the linear layout, each warp load accesses 32 distinct cache sectors. The block-linear layout reduces this footprint to 16 sectors by placing the texels requested by every two adjacent threads within the same sector. Equivalently, the intra-warp cache-sector utilization increases from 1/8 to 1/4. Because each instruction introduces fewer distinct sectors into the memory-access trace, fewer sectors intervene between successive accesses to a given sector. The block-linear layout therefore shortens the intrinsic cache-sector reuse distances associated with reuse across SFLS boundaries (Case 2).

The same effect is stronger at cache-line granularity. The block-linear layout reduces the footprint of each warp load from 32 to 8 cache lines, increasing intra-warp cache-line utilization from 1/32 to 1/8. Each instruction consequently introduces fewer distinct cache lines between successive accesses to the same line and occupies less cache capacity while awaiting reuse. The shorter reuse distances and smaller intervening footprint reduce the likelihood that partially consumed sectors and lines are evicted before their remaining data are accessed.

Figure 5.4 confirms the resulting improvement in cache retention. Compared with the linear-layout global-memory implementation, the texture-memory implementation maintained higher L1 hit rates as N increased. The improvement extended to the L2 cache, which continued to satisfy a substantial fraction of accesses after the L1 cache became ineffective. Fewer misses therefore propagated to DRAM, reducing lower-level sector traffic. Figure 5.5 quantifies this reduction by comparing the number of sectors accessed in the L1 cache and fetched from the L2 cache and DRAM under the two layouts.

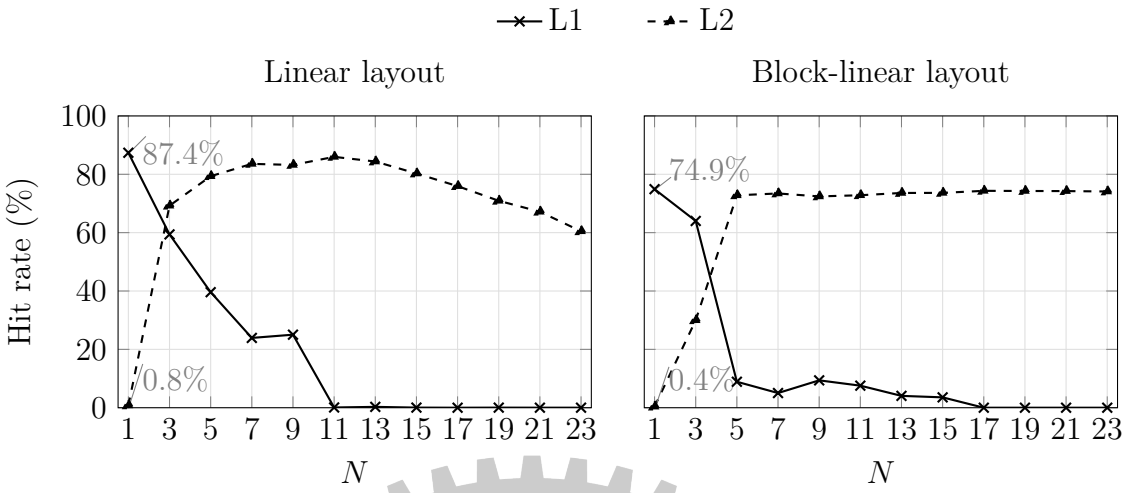


Figure 5.4: Cache hit rates under linear and block-linear layout. The linear-layout global-memory curves are reproduced from Figure 3.7, with the texture-memory implementation added for comparison.

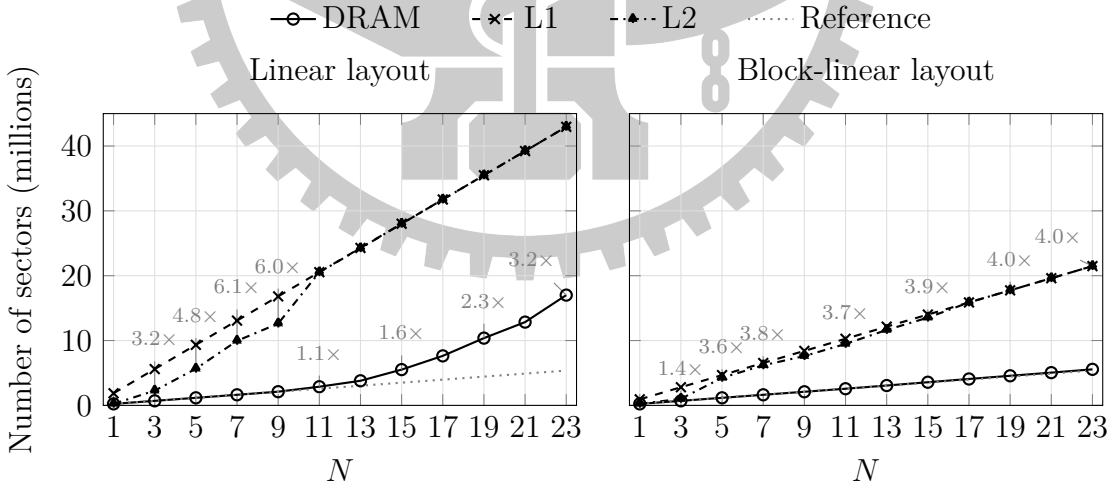


Figure 5.5: Number of sectors accessed in the L1 cache and fetched from the L2 cache and DRAM for the row-summation case study under linear and block-linear layouts. The linear-layout global-memory result is reproduced from Figure 3.8, with the texture-memory implementation added for comparison. Annotations show the ratio relative to the reference; ratios for the L1 count in the global-memory and texture-memory implementations are omitted as they are uniformly 8× and 4×, respectively.

The block-linear layout also shortens effective reuse distance by reducing cache-access wavefront generation and the resulting memory-system contention. Global-memory accesses are queued before entering the cache hierarchy, and excessive wavefront generation manifests as LG throttle stalls when the queue becomes saturated (Section 3.3.2). Texture memory follows the same principle through a dedicated texture queue, where contention is reflected by the corresponding *TEX throttle* stalls.

Figure 5.6 compares the normalized warp stall time of the linear-layout global-memory implementation with that of the texture-memory implementation. Stall time is normalized by the number of input matrices (N) to remove the linear increase in instruction count as the workload grows. Differences in instruction count between the implementations are considered separately later. The global-memory implementation was dominated by LG throttle stalls because each warp load generates 32 cache-access wavefronts. Texture memory reduced this count to approximately 4 wavefronts per instruction, lowering pressure on the texture pipeline. Its normalized throttle stall time therefore remained lower across the evaluated values of N. The reduction indicates that fewer stall-induced boundaries interrupt SFLSs, allowing fewer accesses from other warps to intervene before reuse and thereby shortening effective reuse distance.

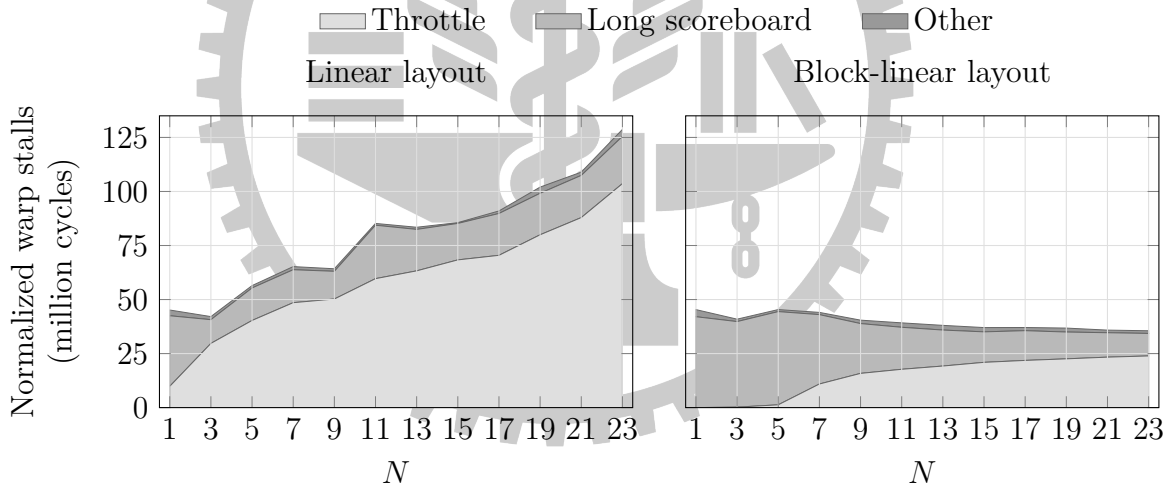


Figure 5.6: Normalized warp stall breakdown under linear and block-linear layouts. Warp stall time is normalized by the number of input matrices (N). The linear-layout global-memory result is reproduced from Figure 3.6, with the texture-memory implementation added for comparison. The throttle component denotes LG throttle for global memory and TEX throttle for texture memory.

The higher cache-sector utilization, smaller cache-line footprint, and lower wavefront count act together. Higher sector utilization reduces the amount of partially consumed data that must remain resident for subsequent accesses. Higher line utilization reduces both cache occupancy and the pipeline work required to serve each warp instruction. The resulting reductions in intrinsic and effective reuse distance improve cache retention and reduce traffic throughout the memory hierarchy.

Figure 5.7 reports the aggregate performance effect under four memory organizations. For the original row-summation access structure, texture memory reduced execution time across all evaluated working-set sizes relative to linear-layout global memory. Applying texture memory to the transposed access structure introduced moderate overhead, but this penalty remained considerably smaller than the degradation caused by executing the original access structure under the linear layout.

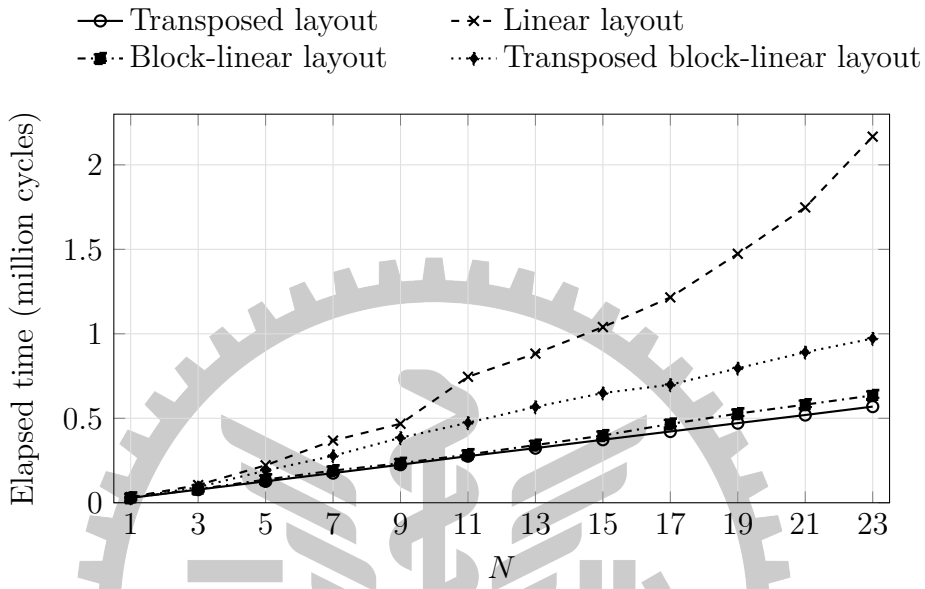


Figure 5.7: Execution time under four memory organizations. The transposed- and linear-layout global-memory results are reproduced from Figure 3.9, with the corresponding block-linear and transposed block-linear implementations added for comparison.

These results establish the block-linear layout as an effective layout-mapping mechanism. By increasing intra-warp cache-sector utilization, it reduces dependence on retaining partially consumed sectors and shortens intrinsic reuse distance. By increasing cache-line utilization and altering texture-pipeline wavefront formation, it reduces cache occupancy, wavefront generation, and throttle stalls, thereby shortening effective reuse distance. Together, these effects improve cache retention, reduce lower-level memory traffic, and narrow the performance gap between the original and transposed access structures without modifying the logical computation.

Chapter 6
AutoTex Compiler Framework

In this work, we propose *AutoTex*, a compiler-guided approach that addresses the mismatch (Chapter 3) between logical access structure and physical memory layout. AutoTex exploits the hardware-managed block-linear layout of texture memory (Chapter 5) to improve intra-warp cache utilization. Under the linear global-memory layout, logically related addresses requested by a warp may be distributed across many cache sectors and cache lines. AutoTex instead maps eligible multidimensional data objects to texture memory, allowing the block-linear layout to consolidate these addresses into fewer sectors and lines. The resulting organization can increase cache-sector utilization by allowing each instruction to consume more of the fetched sector data and increase cache-line utilization by reducing the number of cache lines occupied by its requests.

AutoTex is implemented as a collection of LLVM analysis and transformation passes integrated into the CUDA compilation pipeline (Section 2.8.1). Figure 6.1 illustrates the overall framework. The core framework targets read-only data through texture memory; support for writable data through surface memory is presented as an extension in Section 6.4.

Rather than replacing the existing compilation process, AutoTex extends the standard LLVM CUDA compilation pipeline in two ways. First, it introduces a launch-analysis phase that recovers provenance relationships among kernel arguments across launch sites. Second, it augments the device- and host-compilation phases with analyses and transformations that identify eligible data, rewrite device-side memory accesses, and modify host-side memory management and kernel launches. The transformed program then continues through the existing LLVM optimization and code-generation pipeline.

Accordingly, AutoTex comprises three phases, each responsible for a distinct aspect of the transformation. The ⓿ launch-analysis phase first analyzes the host program to recover the provenance relationship among kernel arguments. This information is unavailable during conventional device compilation because CUDA kernels are compiled independently of their launch sites. The recovered provenance information is therefore exported and supplied to the subsequent device-compilation phase.

During the ❶ device-compilation phase, AutoTex reconstructs multidimensional accesses through global-memory arguments and identifies those whose structure may produce low intra-warp cache utilization under the linear layout. The current implementation uses divergence in outer-dimensional indices as a structural indicator of such accesses rather than computing their exact cache-sector and cache-line footprints. These results are combined with the provenance mapping recovered during launch analysis so that all

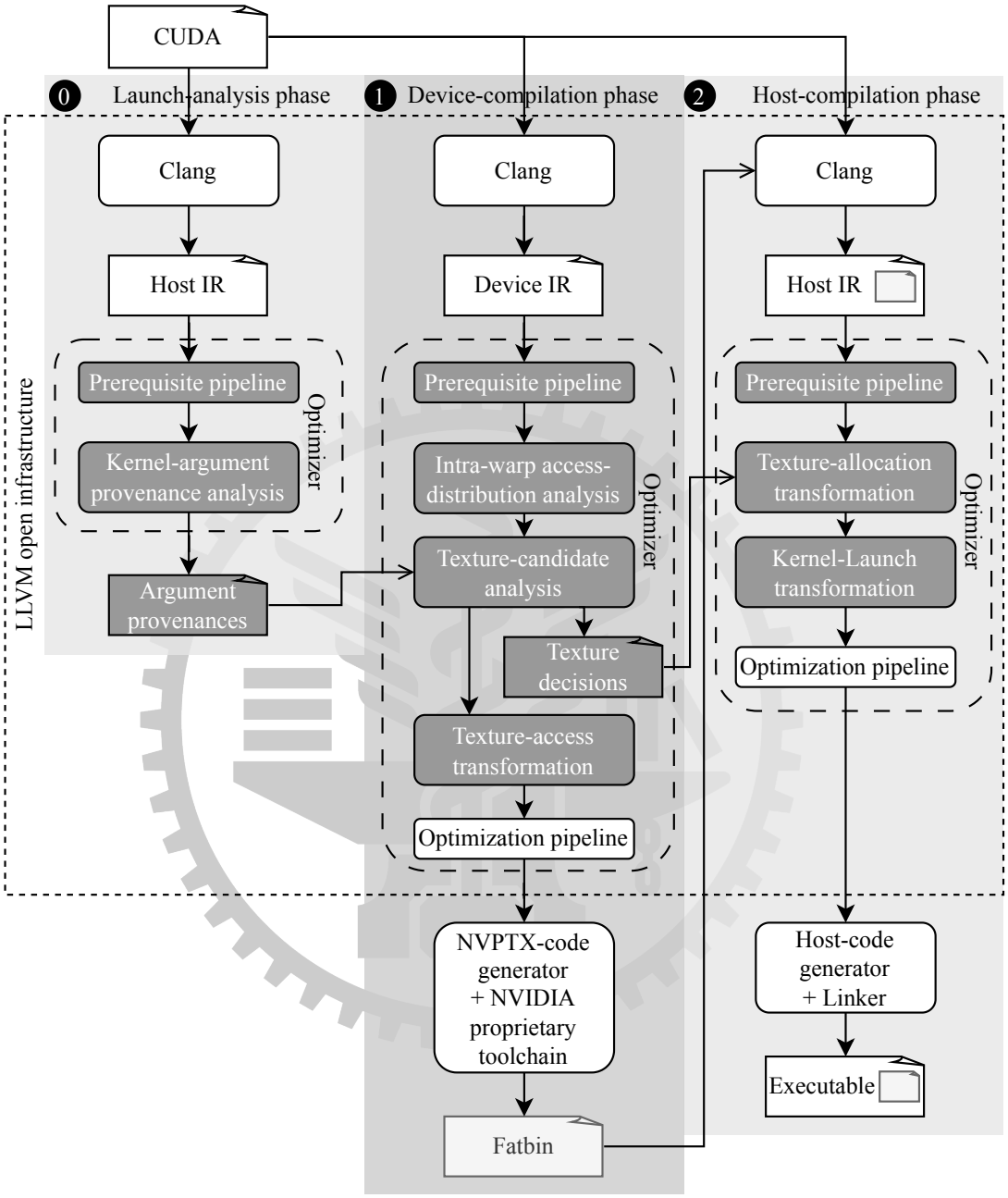


Figure 6.1: Compilation pipeline of the AutoTex framework. AutoTex augments the standard LLVM CUDA compilation pipeline in Figure 2.11 by introducing the launch-analysis phase and augmenting the device- and host-compilation phases with the analyses and transformations proposed in this work, which are denoted by dark-gray boxes. Filled arrows (→) indicate the normal compilation flow, whereas open arrows (⟶) represent information or binaries transferred across phases.

arguments referring to the same device-memory allocation are considered consistently. Provenance groups satisfying the semantic and dimensional requirements of texture memory are selected for transformation. The corresponding kernels are then rewritten to accept texture objects, and their global-memory loads are replaced by texture-fetch operations.

Finally, during the ❷ host compilation phase, AutoTex rewrites the host program according to the transformation decisions produced by the device phase. Device-memory allocations for selected arguments are replaced by texture-object allocations, the corresponding transfers are redirected to texture memory, and kernel stubs are rewritten to invoke the transformed kernels with texture objects instead of global-memory pointers.

The following sections present the design of each phase in detail. Section 6.1 describes the launch-analysis phase, which establishes cross-kernel argument provenance. Section 6.2 presents device-side access reconstruction, divergence analysis, candidate selection, and texture-access transformation. Section 6.3 describes the host-compilation phase, which allocates texture objects and rewrites kernel stubs to invoke the transformed kernels. Finally, Section 6.4 extends the framework to writable data through surface memory.

6.1 Launch-Analysis Phase

The launch-analysis phase establishes the correspondence between kernel arguments and the device-memory allocations from which they originate. This information is required because global-memory pointers and texture objects expose different access interfaces. Once a device-memory allocation is transformed into a texture object, every kernel accessing that allocation must likewise access it through the texture interface. Consequently, if multiple kernels receive arguments derived from the same device-memory allocation, they must agree on a single transformation decision. Otherwise, different kernels would interpret the same allocation using incompatible memory representations, resulting in an incorrect program.

This dependency cannot be resolved during conventional device compilation. In the LLVM CUDA compilation pipeline, device code is compiled before the host code responsible for launching the kernels. As a result, the device compiler has no visibility into kernel launch sites and cannot determine whether kernel arguments belonging to different kernels ultimately refer to the same device-memory allocation. To bridge this information gap, AutoTex introduces an additional launch-analysis phase that executes before device compilation and exports the provenance information for use during the subsequent device-compilation phase.

Before performing the analysis, the generated host IR is canonicalized using the memory-to-register pass. By promoting stack-allocated variables into SSA values, the pass eliminates unnecessary memory indirections and simplifies tracing the values passed to kernel launches. No further canonicalization is required because the provenance analysis

operates directly on the kernel-launch arguments and does not rely on complex control-flow or data-flow reasoning.

6.1.1 Kernel-Argument Provenance Analysis

The objective of the kernel-argument provenance analysis is to determine whether kernel arguments supplied to different kernel launches ultimately refer to the same device-memory allocation. We define the *provenance* of a kernel argument as the device-memory allocation from which the argument is derived. Internally, each provenance is represented by a *provenance key*, which uniquely identifies a device-memory allocation. Kernel arguments assigned the same provenance key are collected into a common provenance group and are subsequently treated as representing the same device-memory allocation.

Algorithm 6.1 summarizes the analysis. It first identifies the Clang-generated kernel stubs in the host module and then examines every direct call to each stub. For every kernel argument, the value supplied at the call site is mapped to a provenance key. When the argument value is produced by loading a pointer from a host variable, the storage location containing the pointer is used as the provenance key instead of the value produced by the load. Repeated loads from the same host variable are therefore recognized as referring to the same device-memory allocation. Otherwise, the argument value itself serves as the provenance key.[1] The resulting mapping associates each provenance key with the kernel arguments that refer to the corresponding allocation.

Algorithm 6.1 Kernel-argument provenance analysis

Require: Host module M
Ensure: Provenance-group mapping $\mathcal{P}$
  $\mathcal{P} \leftarrow \emptyset$
  $\mathcal{S} \leftarrow$ FindKernelStubs(M)
  for all kernel stubs $s \in \mathcal{S}$ **do**
    for all direct calls c to s **do**
      for $i \leftarrow 0$ to NumArguments(c) $- 1$ **do**
        $v \leftarrow$ Argument(c, i)
        if v is loaded from a pointer location p **then**
          $k \leftarrow p$
        else
          $k \leftarrow v$
        end if
        Insert($\mathcal{P}[k], (s, i)$)
      end for
    end for
  end for
  return $\mathcal{P}$

[1] This case typically arises when the pointer argument has already been materialized as an SSA value, such as the result of array-to-pointer decay represented by a `getelementptr` instruction.

The resulting provenance groups define an equivalence relation over kernel arguments. Two kernel arguments are provenance-equivalent when they are assigned the same provenance key. Because all members of a provenance group originate from the same device-memory allocation, the provenance group becomes the fundamental transformation unit throughout AutoTex. Every subsequent analysis and transformation therefore operates on provenance groups rather than individual kernel arguments, ensuring that each device-memory allocation is transformed consistently across all kernels.

The current analysis performs lightweight provenance tracing over direct argument values and identifiable pointer storage locations, which cover the kernel-launch patterns considered in this work. More complex relationships introduced through arbitrary pointer arithmetic, interprocedural propagation, or control-dependent assignments could be incorporated by extending the analysis with additional data-flow or alias information. Such extensions are orthogonal to the overall design of AutoTex.

6.2 Device-Compilation Phase

The device-compilation phase is responsible for identifying global-memory arguments whose accesses may benefit from the block-linear layout and transforming the corresponding kernels accordingly. Before performing the device analyses, the generated LLVM IR passes through a prerequisite pipeline that balances preservation of an analyzable representation of memory accesses with recovery of lightweight optimization opportunities. The device analyses then identify global-memory accesses with low intra-warp cache utilization, combine the results with the kernel-argument provenance recovered during the launch-analysis phase to determine which arguments should be transformed, and rewrite the selected accesses into texture fetches. The modified IR then continues through the original LLVM optimization and code-generation pipeline.

6.2.1 Pipeline Placement and Prerequisite Pipeline

The device-side analyses and transformations run before standard optimization in the LLVM device-compilation pipeline. Determining this placement involves balancing two competing objectives. On the one hand, the analyses require memory accesses to remain in a form that faithfully reflects the original multidimensional array accesses. On the other hand, the texture-access transformation replaces ordinary global-memory loads with target-specific texture intrinsics operating on opaque texture objects, after which certain target-independent memory optimizations are no longer directly applicable. The pipeline placement adopted in this work therefore represents a practical compromise between preserving an analyzable representation of memory accesses and retaining optimization opportunities prior to the transformation.

Executing the transformation after the standard optimization pipeline is undesirable because many optimizations aggressively restructure the IR. Loop transformations, value propagation, code motion, and other optimizations may substantially alter address computations and memory accesses, making the original logical array accesses more difficult to reconstruct. Because AutoTex must identify accesses whose logical structure produces low intra-warp cache utilization under the linear layout and determine whether they can benefit from the block-linear layout, preserving an analyzable representation of these accesses takes precedence over performing all target-independent optimizations beforehand.

Conversely, transforming the IR immediately after frontend code generation is likewise undesirable. Once global-memory loads have been rewritten into texture fetches, the accesses are no longer represented as ordinary pointer-based memory operations. Consequently, optimizations that eliminate redundant loads and stores, simplify address computations, or recognize opportunities such as vector-load formation can no longer operate on the transformed accesses. Although exploiting the block-linear layout generally provides greater performance benefits for the target workloads than these missed optimizations, unnecessary memory operations should nevertheless be eliminated before the transformation whenever possible.

The prerequisite pipeline is therefore introduced before the device analyses. Rather than executing the complete standard optimization pipeline, it applies only lightweight canonicalization and memory simplification passes that preserve the overall structure of the program while exposing a cleaner representation for the subsequent analyses. The resulting IR both approximates the memory accesses that would remain after local optimization and retains the structural information required by AutoTex to reconstruct multidimensional array accesses. Accordingly, the prerequisite pipeline consists of the LLVM passes summarized in Table 6.1.

The adopted prerequisite pipeline is a practical engineering choice rather than a globally optimal placement within the LLVM compilation pipeline. Other combinations of canonicalization and optimization passes may provide a similar or better balance between preserving analyzability for the AutoTex passes and applying standard optimizations before they are hindered by the texture-access transformation. Determining such an optimal placement is orthogonal to the proposed framework. The effectiveness of AutoTex stems from mapping eligible accesses with low intra-warp cache utilization to texture memory so that they can exploit the block-linear layout, rather than from any particular choice of prerequisite passes.

6.2.2 Intra-Warp Access-Distribution Analysis

The intra-warp access-distribution analysis identifies global-memory arguments whose logical access structure may produce low intra-warp cache utilization under the linear layout.

Table 6.1: Prerequisite pipeline executed before the device analyses. Passes are given in the order in which they are executed.

Pass	**Purpose**
Memory to register	Promotes stack-allocated variables into SSA form, eliminating unnecessary memory indirections and simplifying subsequent data-flow analyses.
Address-space inference	Resolves generic pointers into CUDA address spaces, allowing global-memory accesses to be identified directly from the IR.
Control-flow simplification	Simplifies trivial control-flow structures without substantially restructuring the program.
Early common-subexpression elimination	Eliminates redundant loads and common subexpressions while memory accesses are still represented as ordinary load instructions.
Dead-code elimination	Removes dead instructions exposed by the preceding simplification passes.
Dead-store elimination	Eliminates stores that are provably overwritten before being observed, reducing unnecessary memory operations.
Attributor	Infers function attributes, including whether pointer arguments are read-only, which are later used during texture-candidate selection.
Instruction combining	Canonicalizes pointer arithmetic and address computations, reducing equivalent representations of the same memory access.

Rather than attempting to determine the exact cache-sector and cache-line footprint of each memory instruction, the analysis decomposes the problem into three components. First, the global-memory access reconstruction recovers the multidimensional access associated with each global-memory reference. Second, the divergence analysis determines which index expressions may evaluate to different values across threads within a warp. Finally, the reconstructed accesses and divergence information are combined to determine whether the threads vary along a non-contiguous array dimension, indicating an access distribution that may benefit from the block-linear layout.

Global-Memory Access Reconstruction

The analysis begins by enumerating all accesses to global-memory pointer arguments. Owing to the preceding infer-address-spaces pass, accesses to global memory can be identified directly from their inferred address spaces. Each global-memory access is represented in LLVM IR by a chain of `getelementptr` instructions followed by a memory operation. The prerequisite instruction-combining pass canonicalizes these address computations when-

ever possible, reducing the number of equivalent address representations that the analysis must handle.

Rather than operating directly on LLVM pointer arithmetic, the analysis reconstructs each access into a logical memory-access representation. For every access, the analysis identifies the base global-memory address, reconstructs the multidimensional index expressions from the associated `getelementptr` instructions, and associates them with the corresponding load or store instruction. The reconstructed indices are organized according to the logical array dimensions, independent of the particular sequence of pointer-arithmetic instructions used to compute the address.

The current implementation derives these dimensions from the array types retained in the `getelementptr` instructions. It therefore requires a global-memory kernel parameter to be accessed through an array type rather than solely through a flat pointer and a linearized offset. Although LLVM provides a delinearization analysis for recovering multidimensional indices from linearized address expressions, the analysis relies on loop structures characteristic of CPU programs. CUDA kernels instead commonly distribute the iteration space across the thread hierarchy and form addresses from thread and block indices. The existing analysis therefore cannot recover the dimensional structure of these accesses, and supporting flat pointer-based accesses would require a GPU-aware delinearization mechanism.

The resulting representation groups all reconstructed accesses by their base global-memory address. Each reconstructed access records the logical index expressions, the number and ordering of dimensions, the access kind (load or store), and the associated memory instruction. Subsequent analyses can therefore reason about multidimensional array accesses rather than individual LLVM instructions. In particular, the divergence analysis examines the reconstructed index expressions to determine how accesses vary across threads within a warp, while the texture-access transformation reuses the representation to generate texture coordinates and rewrite the corresponding memory instructions.

Divergence Analysis

The intra-warp distribution of a memory access depends not only on its index expressions but also on which of those expressions vary across threads within a warp. AutoTex therefore employs the divergence analysis to classify SSA values as either *uniform* or *divergent*. The analysis is based on LLVM's generic divergence analysis, which follows the abstract-interpretation framework proposed by Rosemann et al. [83]

The divergence analysis is formulated as a forward may-analysis over the SSA graph. It begins by identifying *sources of divergence*, namely SSA values that are conservatively assumed to vary across threads within a warp. Starting from these sources, the analysis propagates divergence along SSA use-def relationships using a monotone worklist algo-

rithm. Whenever a divergent value is used to compute another SSA value, the derived value is likewise classified as divergent.

AutoTex uses a source-of-divergence policy tailored to intra-warp access-distribution analysis. Kernel arguments and block indices, such as `blockIdx.x`, are treated as uniform because all threads within a warp observe the same values. By contrast, `threadIdx.x` is treated as a divergence source because it distinguishes the threads considered by the current implementation. Atomic operations and memory-loaded values are also conservatively treated as divergence sources because they may vary across threads.

The current implementation considers only variation induced by `threadIdx.x`. A more complete analysis could incorporate `threadIdx.y` and `threadIdx.z` together with the thread-block dimensions recovered from kernel launch sites.[2] Such information would allow the analysis to model the joint distribution of all three thread-index dimensions and derive a more accurate intra-warp access distribution. The omission is a limitation of the current implementation rather than of the layout-mapping method or surrounding analysis framework.

Low-Utilization Access Classification

The final stage combines the reconstructed memory accesses with the divergence information to identify accesses that may exhibit low intra-warp cache utilization under the linear layout. For each reconstructed access, the analysis examines the divergence status of the index expression associated with every array dimension. Because a reconstructed index expression may comprise multiple SSA values, the divergence analysis is queried for each constituent value. An index expression is considered divergent if any of its constituent values is classified as divergent.

Let

$$A[i_0][i_1]\dots[i_{n-1}]$$

denote a reconstructed multidimensional access, where i_0 corresponds to the innermost array dimension. The current implementation classifies the access as potentially exhibiting low intra-warp cache utilization whenever any outer-dimensional index expression is divergent:

$$\frac{\exists\, k > 0,\ \text{divergent}(i_k)}{A[i_0][i_1]\dots[i_{n-1}] \text{ may exhibit low intra-warp cache utilization}}$$

This criterion follows from the physical organization of row-major arrays. Divergence confined to the innermost dimension generally distributes threads across nearby elements within the same row, whereas divergence in an outer dimension causes threads to access

[2] Warps are formed from consecutive linear thread indices; whether `threadIdx.y` and `threadIdx.z` vary within a warp therefore depends on the thread-block dimensions.

different rows or higher-dimensional regions. Under the linear layout, these regions are separated by the strides of the intervening dimensions and may therefore map to distinct cache sectors and cache lines, reducing intra-warp cache utilization.

The criterion is intentionally conservative and does not reconstruct the exact sector- or line-level distribution of a warp instruction. Such a distribution would additionally depend on the values and relationships of all varying thread-index dimensions, the thread-block dimensions, element size, alignment, and array extents. The current implementation instead uses outer-dimensional divergence as a structural indicator of accesses that may benefit from the multidimensional locality of the block-linear layout. This classification currently assumes four-byte elements, which correspond to the element types supported by AutoTex. Supporting elements of other sizes may require the classification to account explicitly for element width, because it affects how the addresses generated by a warp are distributed across cache sectors and cache lines.

The analysis records the classification of each reconstructed access and aggregates the results at the argument level. A global-memory argument is identified as a low-utilization candidate if any of its reconstructed accesses satisfies the criterion. These arguments are then forwarded to the texture-candidate analysis, which applies the remaining legality constraints for transformation.

6.2.3 Texture-Candidate Analysis

The texture-candidate analysis determines which provenance groups are eligible for transformation into texture objects. Although the intra-warp access-distribution analysis identifies arguments that may benefit from the block-linear layout, not every such argument is eligible. Eligibility also depends on the semantic, dimensional, and element-type constraints of texture memory and the current AutoTex implementation.

Algorithm 6.2 summarizes the texture-candidate analysis. It consumes the provenance-group mapping produced by the kernel-argument provenance analysis together with the arguments identified as low-utilization candidates by the intra-warp access-distribution analysis. For each such argument, the analysis consults the provenance mapping to obtain all kernel arguments originating from the same device-memory allocation. The analysis therefore retrieves the corresponding provenance group. Consequently, if one kernel accesses an allocation through a potentially low-utilization distribution, all provenance-equivalent arguments are selected together, even when their own accesses do not satisfy the classification criterion.

The resulting candidate groups are then filtered according to the semantic requirements of texture memory. First, every argument in a provenance group must be read-only. Because texture memory exposes a read-only access interface, a writable access through any argument in the group would violate the required semantics. Provenance groups

Algorithm 6.2 Texture-candidate analysis

Require: Low-utilization arguments $\mathcal{L}$, provenance-group mapping $\mathcal{P}$
Ensure: Set of texture-candidate groups $\mathcal{C}$

```
C ← ∅
for all arguments a ∈ L do
    G ← ProvenanceGroup(P, a)
    Insert(C, G)
end for
for all candidate groups G ∈ C do
    if ∃ a ∈ G : NumDimensions(a) > 3 then
        Remove(C, G)
    else if ∃ a ∈ G : ¬IsReadOnly(a) then
        Remove(C, G)
    else if ∃ a ∈ G : ¬IsSupportedElementType(a) then
        Remove(C, G)
    end if
end for
return C
```

containing writable accesses are therefore discarded. Second, CUDA texture memory supports up to three-dimensional arrays. Candidate groups containing accesses with more than three dimensions are likewise discarded.[3]

Finally, the current implementation supports only `float`, `int`, and `unsigned int` elements. CUDA texture memory supports additional basic integer and single-precision floating-point formats, including their one-, two-, and four-component vector variants, but these formats are not currently supported by AutoTex. Types that require multiple texture components, such as `double`, are likewise unsupported. User-defined structures cannot be transformed because texture memory supports only predefined scalar and vector channel formats rather than arbitrary aggregate element types. Candidate groups containing any unsupported element type are therefore discarded.

The remaining provenance groups constitute the final texture candidates. These transformation decisions are consumed by the texture-access transformation and exported to the host-compilation phase, ensuring that the device- and host-side transformations operate on the same allocations.

6.2.4 Texture-Access Transformation

The texture-access transformation rewrites selected kernels to access texture memory while preserving their original semantics. For each texture candidate identified by the

[3]One-dimensional textures retain a linear layout and therefore do not provide the block-linear organization exploited by AutoTex. No explicit filtering is required because one-dimensional accesses contain no outer-dimensional indices and therefore cannot satisfy the low-utilization criterion defined by the preceding access classification.

texture-candidate analysis, the corresponding global-memory argument is replaced by a texture object, and every access through that argument is redirected to texture fetches. Figure 6.2 illustrates the progression of the transformation using equivalent CUDA code for clarity. Because LLVM function signatures are immutable after function creation, the transformation constructs intermediate kernels while progressively updating the function signature and kernel body. The transformed kernel coexists with the original in the device module. During host-code transformation, calls within the current translation unit are redirected to the transformed kernel, whereas the original is retained for compatibility with external call sites not visible during compilation.

```
__global__ void kernel(
    float *glbPtr,
    int N, int x, int y) {
  ... = glbPtr[y * N + x];
}
```

(a) Original kernel.

```
__global__ void kernel.intmd(
    float *glbPtr,
    cudaTextureObject_t texObj,
    int N, int x, int y) {
  ... = glbPtr[y * N + x];
}
```

(b) Intermediate signature.

```
__global__ void kernel.intmd(
    float *glbPtr,
    cudaTextureObject_t texObj,
    int N, int x, int y) {
  ... = tex2D<float>(texObj, x, y);
}
```

(c) Access rewritten.

```
__global__ void kernel.final(
    cudaTextureObject_t texObj,
    int N, int x, int y) {
  ... = tex2D<float>(texObj, x, y);
}
```

(d) Final signature.

Figure 6.2: Progression of the texture-access transformation. Starting from the original kernel, the transformation first introduces texture-object parameters, then rewrites global-memory accesses as texture fetches, and finally removes obsolete pointer parameters from the kernel signature. Highlighted code denotes the modifications performed at each stage.

Kernel Signature Rewriting

The transformation begins by constructing an intermediate kernel whose parameter list is derived from the original kernel signature, as shown in Figure 6.2 (b). For each argument selected by the texture-candidate analysis, a texture-object parameter is inserted immediately after the corresponding global-memory pointer. The original pointer parameters are temporarily retained so that the copied kernel body preserves its original SSA use-def relationships during access rewriting.

After every memory access has been rewritten, the temporary pointer parameters become obsolete. The transformation therefore constructs the final kernel shown in Figure 6.2 (d), whose signature omits the redundant global-memory pointers while preserving

all remaining parameters. The resulting function is then registered as a CUDA kernel entry for device code generation.

Texture-Access Rewriting

The texture-access rewriting pass operates directly on the reconstructed accesses produced by the global-memory access reconstruction rather than rediscovering them from the LLVM IR. Each reconstructed access records the base argument, the reconstructed multidimensional index expressions, and the associated load instruction. The transformation therefore operates on logical array accesses instead of low-level pointer arithmetic.

The preceding texture-candidate analysis guarantees that every transformed argument satisfies the correctness requirements for texture memory. In particular, all accesses through a transformed argument are read-only, allowing each reconstructed load to be rewritten directly as a texture fetch.

For each reconstructed load, the associated multidimensional index expressions are reused directly as texture coordinates. Figure 6.2 (c) illustrates this rewriting using the CUDA texture-fetch interface for clarity. The implementation, however, replaces the load with the corresponding LLVM texture intrinsic.[4] LLVM exposes NVIDIA-specific texture operations through target-specific intrinsics, which are emitted directly using the reconstructed coordinates together and the texture-object parameter introduced during the kernel-signature transformation.

The selected intrinsic returns a four-component vector, corresponding to the vector-form texture instruction (Section 5.2.1). For a scalar `FP32` access, the transformation extracts the first component of this vector and uses it to replace all uses of the original scalar load. The remaining components are not exposed to the transformed program and therefore do not affect its semantics. Although the resulting machine instruction may fetch an additional neighboring component, this paired-fetch behavior changes only the physical access footprint and not the value returned for the rewritten scalar access.

Using the intrinsic also avoids an unnecessary coordinate conversion imposed by the CUDA texture-fetch API. Functions such as `tex2D` accept floating-point coordinates, requiring integer array indices to be converted before each texture fetch. In contrast, the selected LLVM texture intrinsic directly supports integer coordinates, allowing the original index expressions to be reused without integer-to-floating-point conversions. The value returned by the intrinsic replaces all uses of the original load instruction, after which the obsolete load and the address-computation instructions that become dead are removed.

[4]The rewritten access is illustrated using `tex2D<float>`, whereas the implementation emits the intrinsic `llvm.nvvm.tex.unified.2d.v4f32.s32`.

6.3 Host-Compilation Phase

The host-compilation phase materializes the transformation decisions produced during the device-compilation phase by rewriting the generated host LLVM IR. It replaces the original global-memory management with texture-object management while preserving the original program semantics.

Before transformation, the generated host IR is canonicalized using the memory-to-register pass. Promoting stack-allocated variables into SSA form simplifies the identification of kernel arguments, CUDA runtime calls, and launch stubs recovered during the launch-analysis phase. Since the subsequent transformations operate directly on these runtime invocations, no further canonicalization is required.

6.3.1 Texture-Allocation Transformation

The texture-allocation transformation rewrites the host-side memory management associated with each selected provenance group. Figure 6.3 illustrates the transformation for the row-summation case study.

Runtime Allocation Rewriting

For each selected provenance group, the transformation identifies the CUDA runtime calls responsible for allocating, initializing, and releasing the associated device-memory allocation. As shown in Figure 6.3 (b), the original global-memory allocation is replaced by the creation of a texture object backed by a CUDA array, host-to-device transfers are redirected to the underlying CUDA array, and deallocation is replaced by destruction of the texture object.

Runtime allocation requires information unavailable from the host program alone. In particular, the dimensions and element type of each transformed allocation are imported from the device-compilation phase, enabling construction of texture objects compatible with the multidimensional accesses generated during the texture-access transformation.

Wrapper Materialization

The rewritten runtime operations are implemented through a wrapper library that provides a uniform interface for creating, initializing, and destroying texture objects while shielding the compiler passes from the low-level CUDA texture API. Each transformed allocation is represented by a `TextureObject`, which encapsulates both the CUDA array and the corresponding texture-object handle.

Internally, the wrapper allocates the CUDA array, initializes the resource and texture descriptors, and configures the texture descriptor to use unnormalized coordinates, point

```
float *h_m1;
/* initialize h_m1 */
float *d_v;
float *d_m1;

cudaMalloc(&d_v, sizeof(float)*Q);
cudaMalloc(&d_m1,
    sizeof(float)*K*Q);
cudaMemcpy(d_m1, h_m1,
    sizeof(float)*K*Q,
    cudaMemcpyDefault);

dim3 blockDim{32};
dim3 gridDim{
    (Q + blockDim.x - 1)
    / blockDim.x
};

sumRows<<<gridDim, blockDim>>>(
    d_v, d_m1);

cudaFree(d_m1);
cudaFree(d_v);
```

(a) Original host code.

```
float *h_m1;
/* initialize h_m1 */
float *d_v;
TextureObject d_m1;

cudaMalloc(&d_v, sizeof(float)*Q);
malloc2DTextureObject(
    &d_m1, K, Q);
memcpy2DTextureObject(
    &d_m1, h_m1, K, Q);


dim3 blockDim{32};
dim3 gridDim{
    (Q + blockDim.x - 1)
    / blockDim.x
};

sumRows<<<gridDim, blockDim>>>(
    d_v, d_m1.tex);

freeTextureObject(&d_m1);
cudaFree(d_v);
```

(b) Host code after transformation.

Figure 6.3: Host-side transformation of the row-summation case study. The original host code launches the kernel in Listing 3.1, whereas the transformed host code launches the kernel in Listing 5.1. The transformation replaces global-memory management with texture-object management and redirects the kernel launch to pass the generated texture-object handle.

filtering, element-type read mode, and clamp addressing[5] in every dimension. These settings preserve the semantics of ordinary multidimensional array accesses.

6.3.2 Kernel-Launch Transformation

After texture objects have been introduced, the kernel-launch transformation rewrites launch sites to invoke the transformed kernels produced during the device-compilation phase. As illustrated in Figure 6.3 (b), the original global-memory argument `d_m1` is replaced by the corresponding texture-object handle `d_m1.tex`.

At the LLVM IR level, CUDA kernel launches are represented by host-side launch stubs. For each transformed kernel, the transformation constructs a new launch stub matching the signature of the transformed kernel generated during texture-access trans-

[5]The addressing mode affects only out-of-bounds accesses. Because these are already undefined for global-memory pointers, the choice does not affect the semantics of well-defined programs.

formation. Selected global-memory pointer parameters are replaced by texture-object parameters, while all other arguments and the launch configuration are preserved.

Each rewritten launch site invokes the new stub, supplying the texture object associated with the selected provenance group in place of the original device pointer. All remaining arguments are forwarded unchanged.

The new launch stub is registered with the CUDA runtime and associated with the transformed device function contained in the generated fat binary. The original launch stub may be retained when external launch sites outside the current compilation unit cannot be rewritten. After host-side transformation, the modified IR proceeds through standard LLVM optimization and code-generation pipeline to produce the final executable.

6.4 Extending AutoTex to Surface Memory

The design presented in the preceding sections targets texture memory, whose read-only access interface naturally matches immutable global-memory data. CUDA additionally provides surface memory, which exposes the same block-linear storage through a writable access interface. Supporting writable data therefore requires only localized modifications to the existing compiler framework rather than redesigning the compilation pipeline.

Figure 6.4 illustrates the extended framework. Compared with the framework presented in the preceding sections, the extension introduces a global-memory read-after-write analysis into the device-compilation phase and generalizes the texture-specific analyses and transformations into memory-oriented counterparts. The global-memory read-after-write analysis identifies global-memory arguments whose access behavior is incompatible with the coherence constraints of surface memory. The remaining candidate groups are then classified according to their access semantics, allowing the memory-access transformation to emit either texture or surface operations. On the host side, the memory-allocation transformation is extended to construct the corresponding CUDA memory objects, whereas the kernel-launch transformation remains unchanged.

6.4.1 Global-Memory Read-After-Write Analysis

The framework presented in the preceding sections relies on the read-only property of texture memory to guarantee that the transformed program preserves the semantics of the original global-memory accesses. Extending the framework to support surface memory permits writable accesses but introduces an additional correctness requirement. Under the coherence constraints of surface memory (Section 2.3.3), a transformed argument must not contain a read-after-write dependence that could observe a value written earlier in the same kernel invocation. Such arguments therefore remain in global memory.

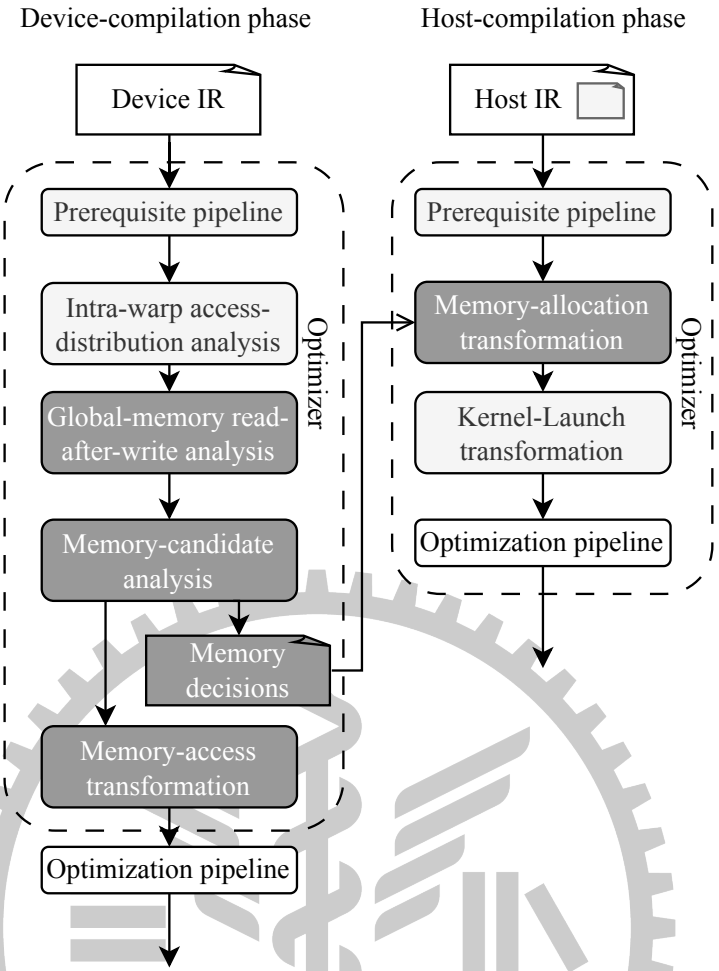


Figure 6.4: Extension of the AutoTex compilation pipeline to support both texture and surface memory. Relative to Figure 6.1, dark-gray boxes highlight the analyses and transformations that are added or modified by the extension.

The global-memory read-after-write analysis operates on the reconstructed global-memory accesses produced by the access reconstruction. For each global-memory argument, the analysis considers every ordered pair of reconstructed accesses and queries LLVM's dependence analysis to identify potential flow (read-after-write) dependences. The reported dependences are subsequently refined using control-flow, loop, and scalar-evolution information to eliminate false positives arising from the conservative nature of dependence analysis. In particular, access pairs that cannot execute in the required order or that can be proven not to overlap across loop iterations are discarded. If a flow dependence remains after refinement, the corresponding global-memory argument is conservatively classified as containing a read-after-write dependence.

Algorithm 6.3 summarizes the analysis. For each global-memory argument and its reconstructed accesses, the analysis iterates over every ordered pair of accesses. The dependence analysis is first used to identify candidate flow dependences. Each reported dependence is then refined to eliminate conservative false positives. The NeverExe-

Algorithm 6.3 Global-memory read-after-write analysis

Require: Mapping $\mathcal{A}$ from each global-memory argument to its reconstructed accesses
Ensure: Set $\mathcal{R}$ of global-memory arguments containing read-after-write dependences

```
R ← ∅
for all (a, A) ∈ A do
    for all (src, dst) ∈ A × A do
        if src = dst then
            continue
        end if
        dep ← DEPENDENCE(src, dst)
        if dep = ∅ or dep is not a flow dependence then
            continue
        end if
        if NEVEREXECUTESBEFORE(src, dst) then
            continue
        end if
        if NOLOOPCARRIEDOVERLAP(src, dst, dep) then
            continue
        end if
        R ← R ∪ {a}
        break
    end for
end for
return R
```

CUTESBEFORE predicate uses control-flow reachability together with dominator, post-dominator, and loop information to determine whether the source access can execute before the destination access. The NOLOOPCARRIEDOVERLAP predicate uses dependence distances together with scalar-evolution information to prove that the two accesses cannot overlap across loop iterations. If neither predicate disproves the reported dependence, the corresponding argument is recorded and no further access pairs for that argument are examined. The resulting set of arguments is subsequently used during memory-candidate analysis to exclude provenance groups that cannot be safely transformed.

6.4.2 Memory-Candidate Analysis

The memory-candidate analysis extends the texture-candidate analysis (Section 6.2.3) with two additional capabilities. It excludes provenance groups containing read-after-write dependences and classifies the remaining groups according to the CUDA memory interface they require. The analysis combines the provenance groups recovered during the launch-analysis phase with the results of the intra-warp access-distribution analysis and the global-memory read-after-write analysis to classify each provenance group uniformly.

Algorithm 6.4 extends the texture-candidate analysis by replacing the candidate set with a mapping from provenance groups to memory types. As before, every provenance

Algorithm 6.4 Memory-candidate analysis

Require: Low-utilization arguments $\mathcal{L}$, provenance-group mapping $\mathcal{P}$
Ensure: Mapping from candidate groups to memory types $\mathcal{M}$

```
M ← ∅
for all arguments a ∈ L do
    G ← ProvenanceGroup(P, a)
    M[G] ← Texture
end for
for all candidate groups G ∈ M do
    if ∃ a ∈ G : NumDimensions(a) > 3 then
        Remove(M, G)
    else if ∃ a ∈ G : ¬IsSupportedElementType(a) then
        Remove(M, G)
    else if ∃ a ∈ G : HasReadAfterWriteDependence(a) then
        Remove(M, G)
    else if ∃ a ∈ G : ¬IsReadOnly(a) then
        M[G] ← Surface
    end if
end for
return M
```

group containing an argument identified as a low-utilization candidate is initially considered a texture candidate. The analysis then discards unsupported provenance groups by applying the existing dimensionality and element-type constraints together with the read-after-write analysis. Finally, the remaining provenance groups are classified by their access semantics: groups whose arguments are all read-only remain texture candidates, whereas groups containing writable arguments are reclassified as surface candidates.

The resulting memory-type classification is preserved throughout the remainder of the compilation pipeline. The memory-access transformation uses it to emit either texture fetches or surface memory operations, while the host-compilation phase uses it to construct the corresponding CUDA memory objects and rewrite the kernel launches accordingly.

6.4.3 Memory-Access Transformation

The transformation follows the same procedure as the texture-access transformation (Section 6.2.4), except that the emitted memory operations depend on the assigned memory type. Instead of uniformly replacing global-memory accesses with texture fetches, the transformation consults the memory-type classification produced by the memory-candidate analysis and rewrites each selected provenance group according to its assigned CUDA memory interface. Texture candidates are rewritten to use texture fetches, whereas surface candidates are rewritten to use surface memory operations. All other accesses remain unchanged, thereby preserving the behavior of unsupported provenance groups.

Kernel Signature Rewriting

The kernel signature is derived from the original kernel signature by introducing additional parameters for every transformed argument. Texture candidates receive a texture-object parameter, whereas surface candidates receive a surface-object parameter. The original pointer arguments are temporarily retained during the rewriting process to facilitate cloning of the original kernel body and are removed after all transformed accesses have been rewritten. Arguments that are not selected by the memory-candidate analysis remain unchanged.

Memory-Access Rewriting

After the kernel body has been cloned, every reconstructed global-memory access belonging to a transformable provenance group is rewritten according to the assigned memory type. For texture candidates, global-memory loads are replaced with texture-fetch operations. For surface candidates, global-memory loads and stores are replaced with the corresponding surface read and surface write operations. Accesses belonging to provenance groups that remain in global memory are left unchanged. Finally, the obsolete pointer arguments introduced solely for the transformation are removed from the kernel signature, yielding the transformed kernel.

6.4.4 Memory-Allocation Transformation

The memory-allocation transformation extends the texture-allocation transformation (Section 6.3.1) by selecting the appropriate CUDA memory object for each transformed provenance group. The host-side rewriting procedure is identical to the texture-allocation transformation; the memory-type classification affects only the generated memory object and the associated allocation procedure. Texture candidates are allocated as read-only texture objects, whereas surface candidates are allocated with surface support to permit both reads and writes.

The kernel-launch transformation is likewise unchanged. Both texture and surface objects are represented as opaque 64-bit handles in the generated LLVM IR. Consequently, the transformed kernel signature and launch rewriting are identical for both memory types, differing only in the type of memory object supplied to the transformed kernel.

Chapter 7
Evaluation

AutoTex identifies multidimensional global-memory accesses that may benefit from a block-linear physical layout, maps eligible data objects to texture or surface memory, rewrites the corresponding device-side accesses, and adapts the associated host-side memory management and kernel launches. This chapter evaluates the performance impact of these transformations across three benchmark suites and two NVIDIA GPU platforms, considering both their direct effect on GPU kernel execution and their implications for complete application execution.

The evaluation addresses the following research question: How does AutoTex affect performance across different GPU architectures and execution bottlenecks? We first quantify the kernel-level performance impact of AutoTex and separate performance changes attributable to the memory-layout transformation from other effects introduced by the transformed memory interfaces. We then characterize the observed per-kernel variation according to each kernel's baseline execution bottleneck and examine the underlying architectural effects, including cache-access wavefront generation, memory-system stall behavior, cache effectiveness, and memory traffic. Finally, we evaluate application-level performance to determine how changes in individual kernel execution combine with the host-side costs introduced by the transformation. Together, these analyses establish the conditions under which AutoTex improves performance, explain the mechanisms responsible for the observed kernel-level outcomes, and determine the extent to which these effects translate into application-level improvements.

7.1 Experimental Methodology

The evaluation uses multiple GPU benchmark suites and two NVIDIA GPU platforms from different architectural generations. This section describes the hardware and software environments, benchmark suites, source-level preparation, experimental configurations, and measurement procedures used throughout the chapter.

7.1.1 Experimental Platforms

AutoTex is evaluated on two NVIDIA GPU platforms. *Ada Desktop* contains an NVIDIA GeForce RTX 4080 based on the Ada Lovelace architecture, whereas *Ampere Laptop* contains an NVIDIA GeForce RTX 3050 Laptop GPU based on the Ampere architecture. These platforms differ not only in their computational resources but also in the amount

Table 7.1: Experimental hardware and software platforms.

		Ada Desktop	**Ampere Laptop**
GPU	Model	NVIDIA GeForce RTX 4080	NVIDIA GeForce RTX 3050 Laptop
	Architecture	Ada Lovelace	Ampere
	SMs	76	16
	L1 cache	128 KiB	
	L2 cache	64 MiB	1.5 MiB
	Device memory[a]	15.6 GiB	3.7 GiB
	Texture limits (elements)	1D: 131,072 2D: 131,072 × 65,536 3D: 16,384 × 16,384 × 16,384	
CPU	Model	Intel® Core™ i7-12700KF	AMD Ryzen 7 5800H
	DRAM[b]	31.2 GiB	15.0 GiB
OS	Distribution	openSUSE Leap 16.0	Ubuntu 22.04.5 LTS
	Linux kernel	6.12.0	6.8.0
Compiler	LLVM	19.1.7	
CUDA	Driver	580.119.02	595.84
	Toolkit	12.9	12.6
	Driver API	13.0	13.2
	Runtime API	12.9	12.6
Profiler	Nsight Compute	2025.4.0.0	2026.1.1.0
	Nsight Systems	2025.5.2.266	2025.6.3.541

[a] `nvidia-smi --query-gpu=memory.free --format=csv,noheader`
[b] `grep MemTotal /proc/meminfo`

of last-level cache available relative to the number of SMs. Table 7.1 summarizes the complete hardware and software environments, including the supported texture dimensions and the versions of the CUDA toolkit and Nsight Compute used on each platform.

Ada Desktop provides 64 MiB of L2 cache shared among 76 SMs, corresponding to approximately 0.84 MiB per SM. Ampere Laptop provides 1.5 MiB shared among 16 SMs, corresponding to approximately 0.094 MiB per SM. Ampere Laptop therefore provides approximately one-ninth of the L2 capacity per SM available on Ada Desktop. Although the L2 cache is globally shared rather than statically partitioned among SMs, the per-SM capacity provides a useful indication of the relative cache resources available under high parallel load. Evaluating both systems allows us to examine whether improvements in intra-warp cache utilization remain beneficial under different cache-capacity and throughput conditions.

The maximum supported texture dimensions are identical on both platforms. These limits constrain the extent of each transformed array dimension independently of the available DRAM capacity.

The AutoTex compiler framework is implemented using LLVM 19.1.7. Each benchmark is compiled with the corresponding CUDA toolkit installed on its target platform. The standard LLVM `-O3` optimization pipeline is applied after the AutoTex analysis and transformation pipelines. Architectural metrics and kernel execution cycles are collected using the corresponding version of Nsight Compute.

7.1.2 Benchmark Suites

The evaluation uses PolyBench-ACC [84], Rodinia [85], and NPB-GPU [86] to assess AutoTex across applications with different computational structures and memory-access behaviors. Table 7.2 summarizes the number of benchmark programs and GPU kernels included from each suite, the number transformed by AutoTex, and the number excluded from the performance analysis because of imprecise candidate identification.

Table 7.2: Benchmark suites and transformed workloads used in the evaluation.

	PolyBench-ACC	**Rodinia**	**NPB-GPU**
Version	3.2[a]	3.1	`3f12d84`[b]
Benchmarks	21	26[c]	8
Transformed	9 (42.9%)	4 (15.4%)	3 (37.5%)
Kernels	47	68	106
Transformed	19 (40.4%)	8 (11.8%)	25 (23.6%)
Excluded	0	0	7

[a]The version of PolyBench/C [87] on which it is based.
[b]No formal release version; identified by repository revision.
[c]Only the `euler3d` variants are included from `cfd`.

We make a best effort to identify and include kernels whose memory accesses can be represented in a form supported by AutoTex. The reported transformed counts should therefore be interpreted as a summary of the evaluated workloads rather than as an exhaustive characterization of transformation applicability. In particular, we do not report a detailed distribution of rejection causes across the benchmark suites. The principal reasons that a kernel or memory object remains untransformed include read-after-write dependencies, accesses through unsupported custom data structures, irregular access patterns whose intra-warp distributions cannot be determined statically, and objects accessed only through one-dimensional indices.

PolyBench-ACC contains manually written CUDA implementations of polyhedral computations. Its kernels predominantly operate on regular multidimensional arrays, making it the primary suite used to evaluate the performance effects of AutoTex.

Rodinia contains heterogeneous applications drawn from several application domains and exhibits a broader range of control flow, synchronization, and memory-access behavior. It is included to evaluate AutoTex beyond regular polyhedral kernels.

NPB-GPU provides GPU implementations of the NAS Parallel Benchmarks [88]. The suite contains structured scientific workloads, including computational-fluid-dynamics kernels, that operate extensively on multidimensional arrays. Compared with PolyBench-ACC, these applications contain larger groups of cooperating kernels and therefore exercise the interprocedural and cross-kernel components of the AutoTex compiler framework.

A subset of the transformed NPB-GPU kernels reconstructs two- or three-dimensional indices from a flattened thread identifier, as illustrated in Listing 7.1. Because each reconstructed index is data-dependent on `threadIdx.x`, the divergence analysis (Section 6.2.2) conservatively classifies all of them as potentially divergent. In practice, the outer indices remain uniform within most warps and differ only when a warp crosses a row or plane boundary. The current analysis cannot represent this conditional uniformity and therefore overestimates the intra-warp distribution of the corresponding indices.

Listing 7.1: Reconstruction of three-dimensional indices from a flattened thread identifier.

```
int i_j_k, i, j, k;
i_j_k = blockIdx.x * blockDim.x + threadIdx.x;

i =  i_j_k % nx;
j = (i_j_k / nx) % ny;
k =  i_j_k / (nx * ny);
```

We exclude these seven kernels[1] from the performance analysis because their selection results from this known precision limitation rather than from the consistently low intra-warp cache utilization targeted by AutoTex. Including them would conflate the performance effect of the proposed memory-layout transformation with false-positive candidate identification. They remain included in the total and transformed kernel counts because the current framework does transform them, while the separate exclusion row in Table 7.2 makes their treatment in the performance results explicit. Recognizing flattened multidimensional index-reconstruction idioms and reasoning about their warp-local uniformity are left for future work.

[1]Three-dimensional reconstruction applies to `rhs_gpu_kernel_1`, `add_gpu_kernel`, `compute_rhs_gpu_kernel_1`, and `txinvr_gpu_kernel`; two-dimensional reconstruction applies to `setbv_gpu_kernel_1`, `setbv_gpu_kernel_2`, and `setbv_gpu_kernel_3`.

7.1.3 Benchmark Preparation and Experimental Configurations

Several benchmark modifications and controlled configurations are required to evaluate AutoTex consistently. First, the original benchmark sources must expose the dimensional structure of their array accesses so that the global-memory access reconstruction can analyze them. Second, equivalent non-aliasing information must be provided across the baseline and transformed configurations so that compiler optimizations enabled by stronger aliasing guarantees are not attributed to the memory transformation itself. Finally, additional memory-layout and access-interface configurations are constructed to isolate the effects of the hardware-managed block-linear layout and the texture and surface interfaces. The following subsections describe these preparations and configurations.

Multidimensional-Access Representation

The global-memory access reconstruction performed by AutoTex requires the dimensional structure of an array access to remain recoverable in the LLVM IR (Section 6.2.2). However, the original CUDA implementations in PolyBench-ACC, Rodinia, and NPB-GPU generally represent multidimensional arrays using flat pointers and manually linearized subscripts. For example, an access to $A[i][j]$ may appear as `A[i * N + j]`, leaving the dimensional structure implicit in integer arithmetic.

We therefore manually delinearize these accesses into equivalent multidimensional array accesses before applying AutoTex. This preparation changes only the representation of address computation; it does not change the logical access pattern, thread mapping, or numerical behavior of the benchmark. Array extents required by the resulting types are represented as compile-time constants.

Several NPB-GPU arrays originally use four-dimensional indices, whereas CUDA texture arrays support at most three dimensions. In the affected cases, one dimension represents a small, fixed number of components, such as the components of a state vector. We fold this component dimension into an adjacent dimension, producing an equivalent three-dimensional representation that preserves the original linear ordering of the elements.

Experimental Configurations

Replacing a global-memory access with a texture or surface access changes not only the memory interface but also the aliasing relationships visible in the transformed device IR. Before transformation, multiple kernel pointer arguments may conservatively be assumed to alias. After a selected allocation is accessed through a texture or surface object instead of its original global-memory pointer, the remaining global-memory pointers cannot alias the memory referenced by that object handle. The transformed configuration may therefore expose stronger non-aliasing information to subsequent compiler optimizations, independently of the change in physical memory layout.

Table 7.3: Experimental configurations and their memory-access and aliasing properties.

Configuration	Memory interface	Non-aliasing information
Global-May-Alias	Global	No
Global	Global	Yes
Texture	Texture for eligible read-only objects	Yes
Surface	Surface for eligible read-only objects	Yes
AutoTex	Texture for eligible read-only objects; surface for eligible writable objects	Yes

Table 7.3 summarizes the configurations and the controlled comparisons used throughout the evaluation. Two global-memory configurations are included to isolate the effect of alias information. The Global-May-Alias configuration preserves the original linear global-memory layout and requires the compiler to conservatively assume that eligible pointer arguments may alias. The Global configuration retains the same allocation and access interface but provides explicit non-aliasing information through the `__restrict__` qualifier. Comparing these two configurations therefore quantifies the performance effect of alias information alone.

The remaining configurations evaluate the effects of mapping eligible objects to block-linear storage through different CUDA access interfaces. All provide the same non-aliasing information as Global. The Texture configuration maps eligible read-only objects to texture memory, while eligible objects containing writable accesses remain in global memory. The Surface configuration maps the same eligible read-only objects to surface memory, allowing the texture and surface access interfaces to be compared while holding the transformed object set and block-linear layout constant. Finally, AutoTex maps eligible read-only objects to texture memory and eligible writable objects to surface memory, corresponding to the complete framework proposed in this work.

Together, these configurations allow the evaluation to separate performance changes due to compiler-visible alias information, block-linear storage, support for writable objects, and the selected CUDA access interface.

Problem-Size Selection

Several benchmarks provide configurable problem sizes. The selected sizes are determined by the available benchmark inputs, the need to provide sufficient grid-level parallelism, and the hardware limits of the target platform. Every allocation must fit within the available device-memory capacity. For transformed objects, each array dimension must also remain within the corresponding texture limit reported in Table 7.1. An allocation may therefore fit in device memory but remain unrepresentable as a texture or surface array because one of its dimensions exceeds the supported extent.

PolyBench-ACC problem sizes are selected independently for the two platforms because Ada Desktop and Ampere Laptop differ substantially in SM count, DRAM capacity, and computational resources. Rodinia uses the same benchmark inputs and runtime arguments on both platforms. For NPB-GPU, all benchmarks use Class C on the Ampere Laptop because it already provides sufficient grid-level parallelism for the principal measured kernels; larger classes would increase execution time without materially changing the evaluated behavior. On the Ada Desktop, IS likewise uses Class C because it already provides sufficient grid-level parallelism, whereas LU and SP use Class D to provide additional parallelism. NPB classes are defined separately for each benchmark, so the same class designation does not imply the same underlying problem dimensions.

Appendix A reports the selected problem sizes and the per-kernel theoretical occupancy, achieved occupancy, and waves per SM measured using Nsight Compute under the Global configuration. It also explains the interpretation of these metrics and identifies cases in which a kernel provides fewer than one wave per SM.

7.1.4 Performance Measurement

The evaluation reports both kernel execution time and application-level execution time because the device- and host-side transformations affect different portions of program execution. Kernel-level measurements isolate the performance effect on individual GPU kernels, whereas application-level measurements determine how these changes combine with the remaining runtime costs.

Kernel execution time is measured in GPU cycles using Nsight Compute (Section 2.9.2). Cache state and GPU clocks are controlled so that the baseline and transformed kernels are measured under comparable conditions. Measurements are collected separately for each kernel, allowing performance changes to be related to the applied transformation, the kernel's execution bottleneck, and the resulting architectural effects.

Application-level measurements are collected using Nsight Systems (Section 2.9.1). End-to-end measurements cover the measured application interval. The component analysis separately examines allocation and object creation, data transfer, kernel launch, kernel execution, and deallocation and object destruction. Unlike the Nsight Compute experiments, these measurements retain the natural cache state and clock behavior of each application execution. Together, the end-to-end and component-level measurements determine how changes in aggregate kernel-execution time combine with the host-side costs of memory management, object management, data transfer, and kernel launch associated with AutoTex.

Preliminary repetitions of the Nsight Compute experiments exhibited negligible variation, so one profiled execution is reported for each kernel and configuration. Application-

level timings collected using Nsight Systems exhibit greater run-to-run variation; each complete application is therefore executed five times, and the geometric mean is reported.

For a kernel, application, or runtime component with baseline execution time T_{base} and transformed execution time $T_{\text{transformed}}$, speedup is defined as

$$S_{\text{transformed/base}} = \frac{T_{\text{base}}}{T_{\text{transformed}}}.$$

Thus, $S_{\text{transformed/base}}$ denotes the speedup of the transformed configuration over the baseline configuration. A value greater than one indicates an improvement, whereas a value below one indicates a slowdown. Unless stated otherwise, each transformed configuration is compared with a baseline having the same source-level aliasing information, problem size, compiler optimization level, and platform-specific software environment.

7.2 Kernel-Level Performance Impact of AutoTex

This section evaluates the overall kernel-level performance impact of AutoTex on the two experimental platforms. Before attributing any performance difference to the proposed transformation, we first examine the effect of compiler-visible alias information and establish Global, which provides equivalent non-aliasing information, as the common baseline. We then compare the baseline, texture-only, and AutoTex configurations separately for kernels directly targeted by the candidate analysis and kernels transformed only because of provenance-group constraints. Finally, we compare texture and surface accesses to the same read-only objects to determine whether the choice of CUDA access interface materially affects performance when the underlying data use the same block-linear layout. Results are presented separately for the Ada Desktop and Ampere Laptop platforms. The resulting per-kernel performance variation is characterized in Section 7.3 using the baseline execution bottleneck and the resulting changes in memory-system behavior.

7.2.1 Establishing the Global-Memory Baseline

Replacing selected global-memory pointers with texture- or surface-object handles may expose stronger non-aliasing information to subsequent compiler optimizations. Comparing AutoTex directly with a global-memory implementation in which pointer arguments may alias could therefore conflate the effects of these optimizations with those of the block-linear memory transformation.

To quantify this effect, we compare Global-May-Alias, in which the compiler must conservatively assume that pointer arguments may alias, with Global, which provides source-level non-aliasing information while retaining the same global-memory allocation

and access interface. Figure 7.1 shows the per-kernel speedup of Global over Global-May-Alias.

Providing explicit non-aliasing information has a measurable and kernel-dependent effect on performance.[2] The effect was most pronounced in PolyBench-ACC, where the maximum speedup of Global over Global-May-Alias reached 2.26× on Ada Desktop and 6.20× on Ampere Laptop. The corresponding suite-level geometric means were 1.14× and 1.67×. By comparison, NPB-GPU remained close to parity, while Rodinia showed smaller aggregate improvements. Across all kernels, the overall geometric-mean speedup was 1.07× on Ada Desktop and 1.30× on Ampere Laptop.

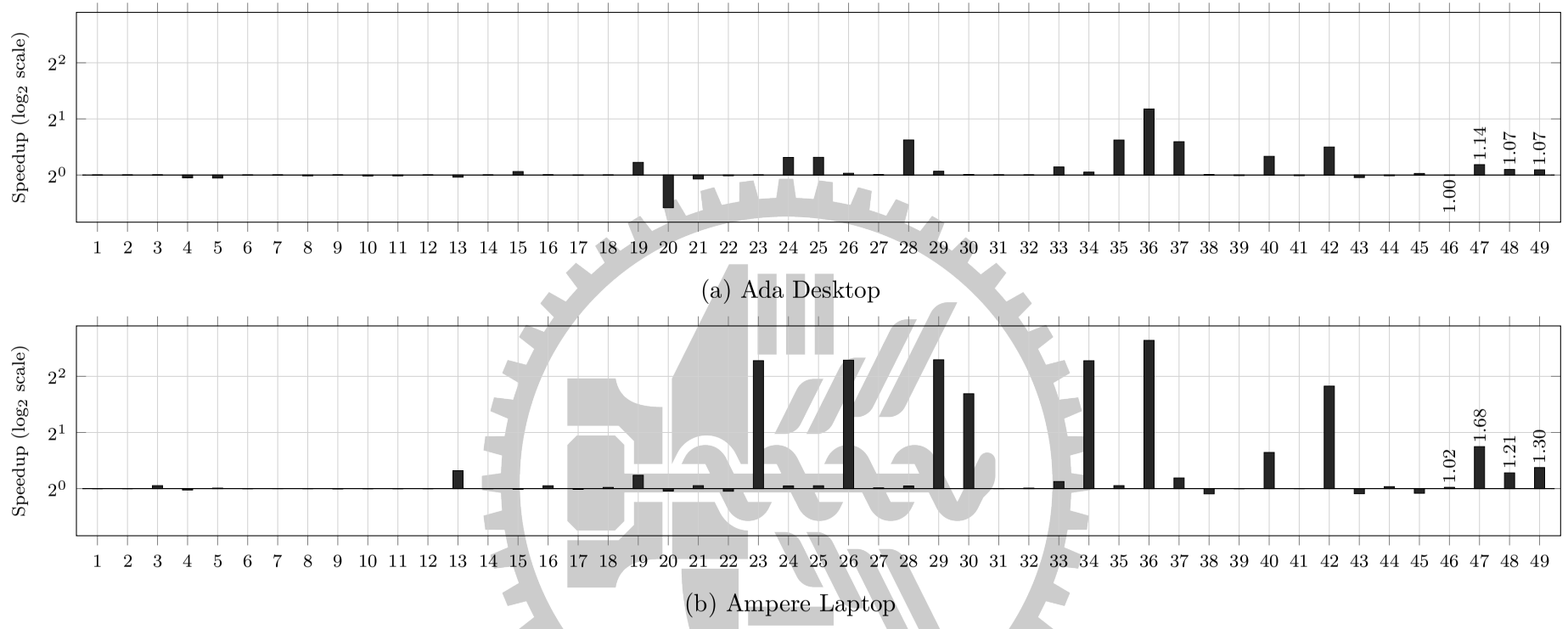
The pronounced gains in PolyBench-ACC arise from optimization opportunities that are inhibited when pointer arguments may alias. Several kernels update accumulation variables directly in memory rather than first loading them into temporary scalars. When aliasing cannot be ruled out, the compiler must conservatively preserve repeated loads and stores because another pointer access may refer to the same location. Explicit non-aliasing information allows the accumulated value to be promoted to a register and can enable loop-invariant code motion for computations that would otherwise remain inside the loop. These optimizations eliminate global-memory accesses and can therefore produce substantial speedups without changing the physical memory layout.

The comparison confirms that alias information is an independent source of performance variation and that Global-May-Alias would not provide a neutral baseline for evaluating AutoTex. We therefore use Global for all subsequent comparisons. Because Global provides the same source-level non-aliasing information as the transformed configurations, this baseline excludes gains attributable solely to register promotion, loop-invariant code motion, and related compiler optimizations, allowing the remaining performance differences to be associated more directly with the memory transformation.

7.2.2 Candidate-Containing and Compelled-Only Kernels

We next evaluate the kernel-level performance of AutoTex relative to the established global-memory baseline. The comparison includes three configurations: Global, which retains the original linear layout; Texture, which maps selected read-only provenance groups to texture memory; and AutoTex, which additionally maps selected writable provenance groups to surface memory. Both transformed configurations operate at the provenance-group level (Section 6.1.1), ensuring that all arguments referring to the same underlying allocation use a consistent memory representation. Performance is reported as speedup over the Global configuration.

[2] `adi_kernel3` on Ada Desktop is the only evaluated kernel for which Global exhibited a substantial slowdown relative to Global-May-Alias. Investigating this isolated regression is beyond the scope of this work.



1: IS/full_verify_gpu_kernel_1
2: IS/full_verify_gpu_kernel_2
3: LU/error_gpu_kernel
4: LU/jacld_blts_gpu_kernel
5: LU/jacu_buts_gpu_kernel
6: LU/pintgr_gpu_kernel_1
7: LU/pintgr_gpu_kernel_2
8: LU/pintgr_gpu_kernel_3
9: LU/rhs_gpu_kernel_2
10: LU/rhs_gpu_kernel_3
11: LU/rhs_gpu_kernel_4
12: LU/setiv_gpu_kernel
13: LU/ssor_gpu_kernel_2
14: SP/compute_rhs_gpu_kernel_2
15: SP/rhs_norm_gpu_kernel_1
16: SP/x_solve_gpu_kernel
17: SP/y_solve_gpu_kernel
18: SP/z_solve_gpu_kernel
19: adi/adi_kernel1
20: adi/adi_kernel3
21: adi/adi_kernel4
22: adi/adi_kernel6
23: atax/atax_kernel1
24: atax/atax_kernel2
25: bicg/bicg_kernel1
26: bicg/bicg_kernel2
27: gemver/gemver_kernel1
28: gemver/gemver_kernel2
29: gemver/gemver_kernel3
30: gesummv/gesummv_kernel
31: gramschmidt/gramschmidt_kernel1
32: gramschmidt/gramschmidt_kernel2
33: gramschmidt/gramschmidt_kernel3
34: mvt/mvt_kernel1
35: mvt/mvt_kernel2
36: syr2k/syr2k_kernel
37: syrk/syrk_kernel
38: gaussian/Fan1
39: gaussian/Fan2
40: huffman/pack2
41: huffman/vlc_encode_kernel_sm64huff
42: kmeans/invert_mapping
43: kmeans/kmeansPoint
44: leukocyte/GICOV_kernel
45: leukocyte/dilate_kernel
46: *npb-gpu gmean*
47: *polybench-acc gmean*
48: *rodinia gmean*
49: **Overall gmean**

Figure 7.1: Per-kernel speedup of GLOBAL over GLOBAL-MAY-ALIAS.

The results are reported separately for candidate-containing and compelled-only kernels. A *candidate-containing kernel* contains at least one argument directly selected by the low-utilization classification (Section 6.2.2). These kernels represent the accesses that AutoTex is specifically designed to improve. A *compelled-only kernel*, by contrast, contains no directly selected argument but is transformed because one or more of its arguments share an underlying allocation with a candidate argument used by another kernel. Such kernels may not exhibit the access pattern that motivated the transformation and therefore are not expected to obtain the same performance benefit.

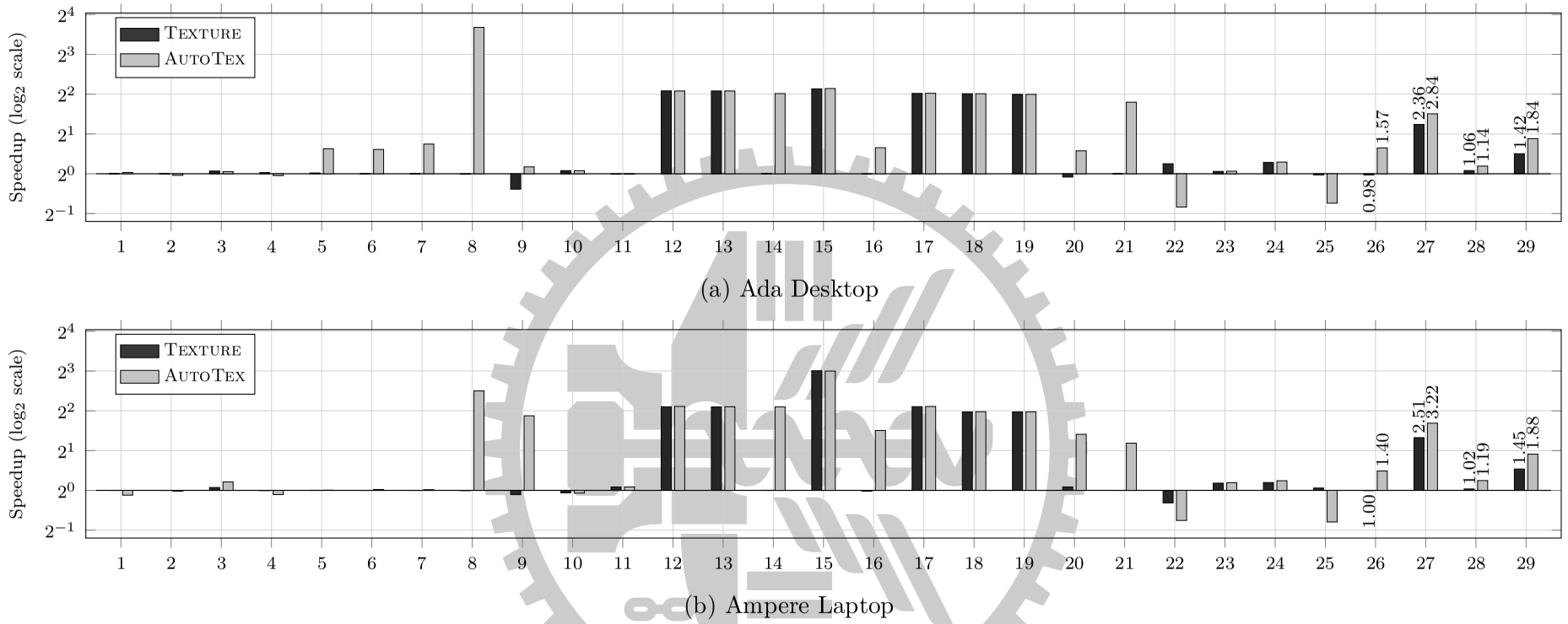
Separating the two groups distinguishes the direct performance effect of optimizing identified low-utilization accesses from the collateral effect of maintaining a consistent memory representation across kernels that share an allocation. For each group, we report per-kernel speedups and an aggregate geometric mean. The candidate-containing results characterize the optimization potential of AutoTex, whereas the compelled-only results quantify whether provenance-driven propagation introduces slowdowns or other unintended performance effects.

Figures 7.2 and 7.3 show a clear distinction between kernels containing directly selected arguments and those transformed only to preserve provenance-group consistency. For candidate-containing kernels, AutoTex achieved geometric-mean speedups of 1.84× on Ada Desktop and 1.88× on Ampere Laptop. The corresponding Texture speedups were 1.42× and 1.45×. The maximum AutoTex speedup reached 12.80× for `rhs_norm_gpu_kernel_1` (kernel 8) on Ada Desktop and 7.99× for `gesummv_kernel` (kernel 15) on Ampere Laptop. Although the individual results vary substantially across kernels, the similar aggregate speedups on the two platforms indicate that the targeted block-linear transformation remains effective across both architectures.

The advantage of AutoTex over Texture reflects its support for eligible writable objects. This effect was particularly pronounced in NPB-GPU: Texture remained close to parity, with suite-level geometric means of 0.98× and 1.00×, whereas AutoTex reached 1.57× and 1.40× on Ada Desktop and Ampere Laptop, respectively. When all selected objects of a candidate-containing kernel are writable, Texture leaves their accesses in global memory, producing performance close to the Global baseline.[3] AutoTex can instead map these objects to surface memory. For kernels containing only eligible read-only objects, the two configurations apply the same texture transformation and therefore generally exhibited similar performance.

Compelled-only kernels exhibited the opposite aggregate behavior. Because they contain no directly selected argument, their transformations are required by shared-allocation consistency rather than by an access pattern expected to benefit from block-linear storage. Their geometric-mean speedups under AutoTex were 0.90× and 0.91×, respectively,

[3]Small differences may remain because the configuration is compiled through the prerequisite AutoTex optimization pipeline even when no object is mapped to texture memory.



1: IS/full_verify_gpu_kernel_1
2: IS/full_verify_gpu_kernel_2
3: LU/jacld_blts_gpu_kernel
4: LU/jacu_buts_gpu_kernel
5: LU/pintgr_gpu_kernel_3
6: LU/rhs_gpu_kernel_3
7: LU/rhs_gpu_kernel_4
8: SP/rhs_norm_gpu_kernel_1
9: SP/x_solve_gpu_kernel
10: adi/adi_kernel1
11: adi/adi_kernel3
12: atax/atax_kernel1
13: bicg/bicg_kernel2
14: gemver/gemver_kernel3
15: gesummv/gesummv_kernel
16: gramschmidt/gramschmidt_kernel2
17: mvt/mvt_kernel1
18: syr2k/syr2k_kernel
19: syrk/syrk_kernel
20: gaussian/Fan1
21: gaussian/Fan2
22: huffman/pack2
23: kmeans/invert_mapping
24: kmeans/kmeansPoint
25: leukocyte/dilate_kernel
26: *npb-gpu gmean*
27: *polybench-acc gmean*
28: *rodinia gmean*
29: **Overall gmean**

Figure 7.2: Per-kernel speedup of Texture and AutoTex over Global for candidate-containing kernels.

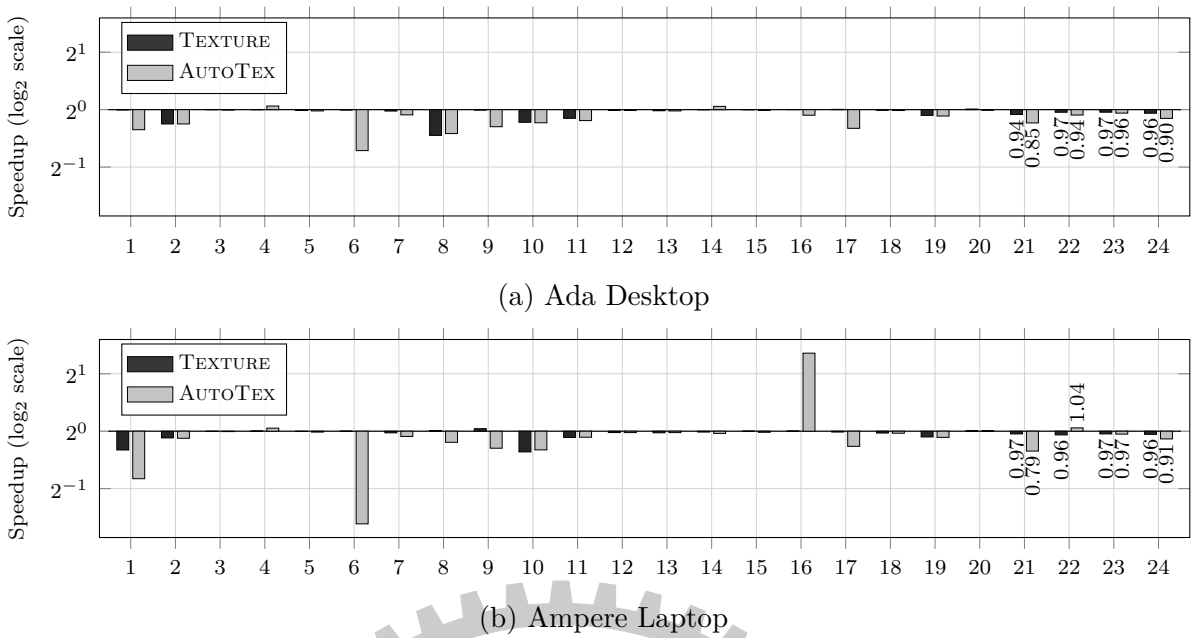


Figure 7.3: Per-kernel speedup of Texture and AutoTex over Global for compelled-only kernels.

compared with 0.96× on Ada Desktop and 0.96× on Ampere Laptop under Texture. The transformed memory interfaces therefore introduce a modest aggregate cost when the kernel itself lacks a targeted access, with the larger penalty under AutoTex indicating additional sensitivity to surface access for propagated writable objects.

Across both kernel groups, AutoTex achieved an overall geometric-mean speedup of 1.27× on Ada Desktop and 1.28× on Ampere Laptop when all kernels transformed by the framework are included. Excluding the seven NPB-GPU kernels identified as known false-positive selections increased the geometric-mean speedup to 1.34× and 1.36×, respectively. The all-kernel results reflect the performance achieved by AutoTex as implemented, whereas the filtered results isolate the performance of the memory-layout transformation among the kernels retained for the subsequent architectural characterization. In both cases, the gains of candidate-containing kernels outweigh the aggregate regressions among compelled-only kernels.

These geometric means weight kernels equally, however, and do not indicate how strongly either group affects complete benchmark execution. The application-level analysis in Section 7.4 therefore examines whether the slowdowns of compelled-only kernels

materially offset the gains of candidate-containing kernels according to their respective execution-time contributions.

7.2.3 Texture and Surface Access to Read-Only Data

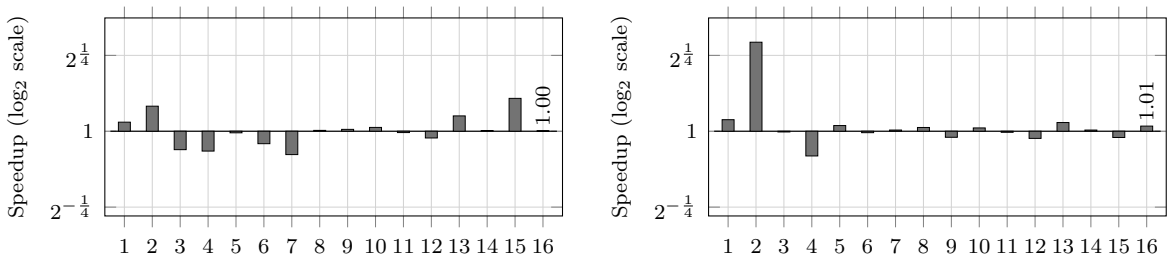
The block-linear organization used by CUDA arrays can be accessed through either texture or surface objects. Texture objects provide a read-only interface, whereas surface objects additionally permit writes. Although texture and surface accesses use different instructions and may traverse different processing stages, both operate on the same underlying block-linear storage (Section 2.3.3). We therefore examine whether the performance of transformed read-only objects is determined primarily by their physical layout or is sensitive to the selected access interface.

We isolate this distinction by comparing the TEXTURE and SURFACE configurations. Both transform the same set of read-only objects, use the same CUDA array allocations and block-linear organization, and preserve the same kernel coverage. They differ only in whether the transformed accesses are emitted as texture fetches or surface reads. The comparison is restricted to kernels for which the same read-only objects are transformed under both configurations, thereby excluding differences caused by candidate selection, allocation coverage, or writable data.

Figure 7.4 reports the per-kernel speedup of SURFACE over TEXTURE. Because the transformed object set and physical layout are unchanged, deviations from 1.00× reflect differences associated with the access interface and its processing path. The geometric-mean speedup was 1.00× on Ada Desktop and 1.01× on Ampere Laptop, indicating no systematic aggregate advantage for either interface. Most kernels remained close to parity, although individual kernels can favor either texture or surface access. In particular, `adi_kernel3` exhibited a pronounced surface advantage on Ampere Laptop, while other kernels showed smaller advantages for texture access.

The small aggregate difference between the two configurations, compared with the speedups obtained by mapping eligible objects from linear global memory to block-linear storage, indicates that the physical layout accounts for the primary performance effect. The remaining per-kernel variation shows, however, that the choice of access interface can have a secondary, kernel-dependent impact. Neither texture nor surface access is uniformly preferable for read-only data, and particular kernels may benefit from the distinct instruction and processing paths associated with either interface. Determining the architectural conditions under which one interface should be preferred would require a more detailed characterization of these paths and is left for future work.

AUTOTEX therefore uses texture access for eligible read-only objects and surface access when write support is required, without attempting to select between the two interfaces based on their kernel-specific performance.



(a) Ada Desktop

(b) Ampere Laptop

1: adi/adi_kernel1
2: adi/adi_kernel3
3: adi/adi_kernel4
4: adi/adi_kernel6
5: atax/atax_kernel1
6: atax/atax_kernel2
7: bicg/bicg_kernel1
8: bicg/bicg_kernel2
9: gesummv/gesummv_kernel
10: mvt/mvt_kernel1
11: mvt/mvt_kernel2
12: syr2k/syr2k_kernel
13: syrk/syrk_kernel
14: kmeans/invert_mapping
15: kmeans/kmeansPoint
16: **Overall gmean**

Figure 7.4: Per-kernel speedup of SURFACE over TEXTURE for kernels in which the same read-only objects are transformed under both configurations.

7.3 Performance Characterization

The preceding section establishes the kernel-level performance impact of AutoTex and separates the effects of alias information, provenance-driven transformation, and the selected CUDA access interface. The observed speedups nevertheless vary across kernels because improving the memory layout does not alleviate every execution bottleneck equally. A reduction in cache-access wavefronts and memory-system work may directly accelerate a latency- or memory-bound kernel, while having little effect when execution is constrained primarily by compute throughput.

This section characterizes that variation in two stages. We first classify candidate-containing kernels according to their dominant bottleneck under the GLOBAL baseline, including latency, compute, balanced, and resource-specific memory categories. This classification provides a coarse-grained view of the conditions under which AutoTex is most effective, but does not fully explain the behavior of individual kernels. We therefore then examine the architectural effects of the transformation, including changes in cache-access wavefronts, memory-system stalls, cache effectiveness, and memory traffic, to explain why kernels within the same bottleneck category may exhibit different performance outcomes.

7.3.1 Performance by Bottleneck Category

The overall results show that the performance impact of AutoTex varies substantially across kernels. To organize this variation, we classify each candidate-containing kernel according to its dominant execution bottleneck under the GLOBAL baseline. The classification is performed separately for each platform because the relative utilization of the compute and memory subsystems may differ across GPU architectures. Using the base-

Table 7.4: Execution-bottleneck classification used in the evaluation. The LSU, TEX, LTS, L1, L2, and DRAM categories refine the general memory-bound classification.

Category	Classification condition	Execution bottleneck
Latency	$C < 60\% \wedge M < 60\%$	Memory or instruction latency
Compute	$C - M \geq 10$ percentage points	Compute or instruction throughput
LSU	$M - C \geq 10$ percentage points; $M_{\mathrm{LSU}} = M_{\max}$	Load/store processing path
TEX	$M - C \geq 10$ percentage points; $M_{\mathrm{TEX}} = M_{\max}$	Texture processing path
LTS	$M - C \geq 10$ percentage points; $M_{\mathrm{LTS}} = M_{\max}$	LTS processing path
L1	$M - C \geq 10$ percentage points; $M_{\mathrm{L1}} = \max(M_{\mathrm{L1}}, M_{\mathrm{L2}}, M_{\mathrm{DRAM}})$	L1-cache throughput
L2	$M - C \geq 10$ percentage points; $M_{\mathrm{L2}} = \max(M_{\mathrm{L1}}, M_{\mathrm{L2}}, M_{\mathrm{DRAM}})$	L2-cache throughput
DRAM	$M - C \geq 10$ percentage points; $M_{\mathrm{DRAM}} = \max(M_{\mathrm{L1}}, M_{\mathrm{L2}}, M_{\mathrm{DRAM}})$	Off-chip memory throughput
Balanced	$\lvert C - M \rvert < 10$ percentage points after LSU-overlap refinement	No dominant compute or memory bottleneck

line configuration also ensures that the assigned category reflects the execution behavior preceding the memory-layout transformation.

The resulting categories provide a high-level basis for comparing the effectiveness of AutoTex across different execution regimes. They should not, however, be interpreted as complete explanations of individual kernel behavior, because kernels within the same category may differ in the extent to which the transformation changes wavefront generation, stall behavior, cache effectiveness, and memory traffic. These effects are examined in the subsequent subsections.

Table 7.4 summarizes the classification rules and the execution bottleneck represented by each category. The categorization follows the throughput-based interpretation used in Nsight Compute's Speed-of-Light analysis[4] to distinguish latency-bound kernels from kernels that approach a compute- or memory-throughput limit [90], [91]. Compute- and memory-bound kernels are throughput-bound: their performance is constrained by the throughput limits of the corresponding hardware resources [92]. By contrast, latency-bound kernels do not approach either throughput limit and are instead constrained by the latency of instruction execution or memory operations. Because this work specifically targets the memory subsystem, we further distinguish memory-bound kernels according to the dominant memory resource.

Let C and M denote the achieved compute and memory throughputs reported by Nsight Compute, respectively. A kernel is classified as *latency-bound* when both through-

[4]The Speed-of-Light analysis also incorporates roofline-model concepts [89].

puts are below 60%, indicating that neither subsystem approaches its throughput limit. When C exceeds M by at least 10 percentage points, the kernel is classified as *compute-bound*. Conversely, when M exceeds C by at least 10 percentage points, the kernel is classified as memory-bound and assigned a more specific category according to the dominant memory resource. Kernels whose compute and memory throughputs differ by less than 10 percentage points are classified as *balanced*.

For memory-bound kernels, the classifier first examines the detailed throughput associated with the LSU, TEX, and LTS processing paths. A kernel is assigned to the *LSU*, *TEX*, or *LTS* category when the corresponding throughput equals the dominant memory throughput. Otherwise, the classifier compares the L1-cache, L2-cache, and DRAM throughputs and assigns the kernel to the category corresponding to the largest of these values: *L1*, *L2*, or *DRAM*.

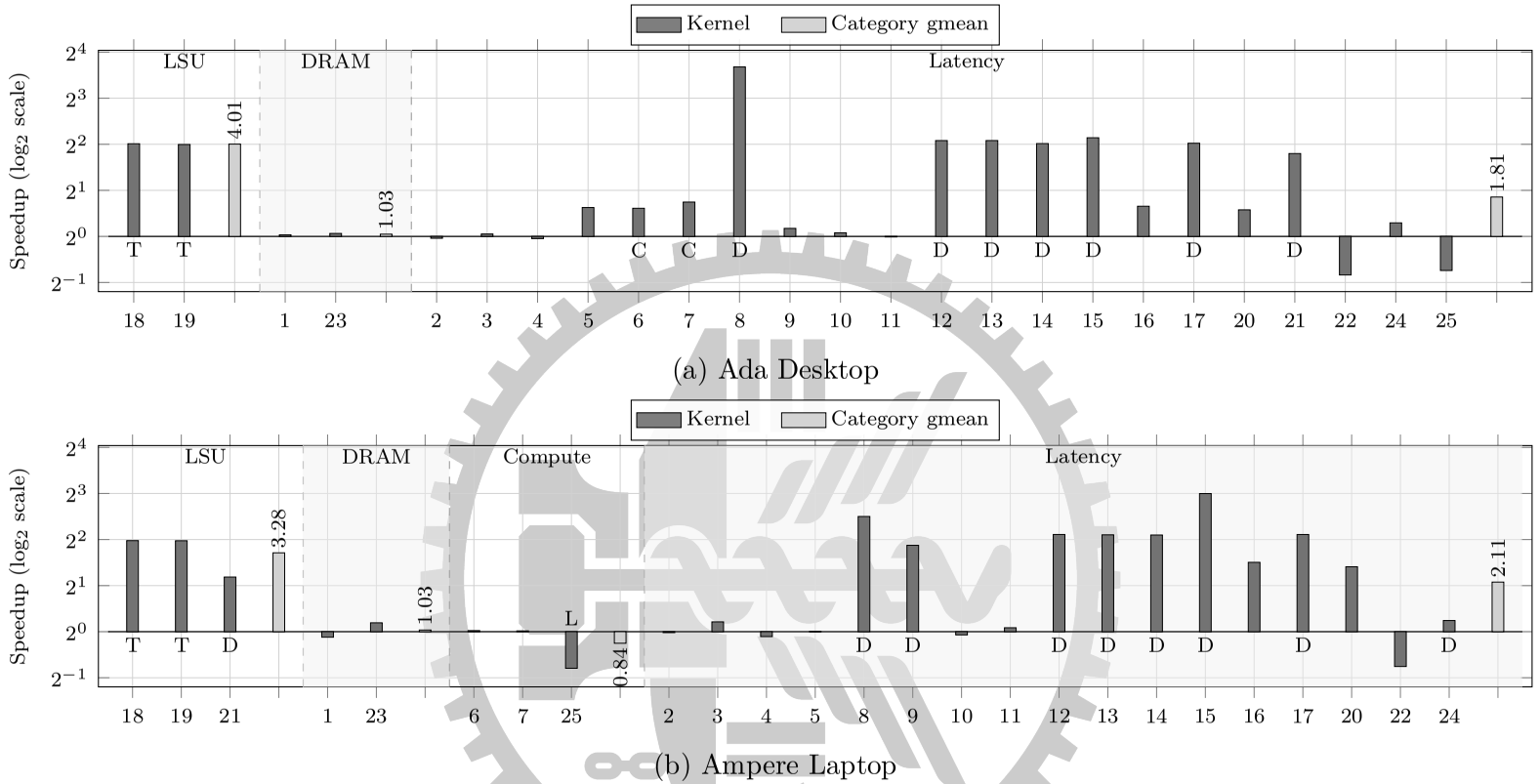
Balanced kernels require additional care because load/store activity may contribute to both the compute-side and memory-side throughput summaries [93]. When the dominant compute contributor is the LSU instruction pipeline and the dominant memory contributor is the LSU input-request path, the classifier treats them as overlapping indicators of the same activity.[5] The classifier removes both contributors and repeats the comparison using the next-highest compute and memory contributors. If neither side becomes dominant after this refinement, the kernel remains classified as balanced.

Figure 7.5 reports the speedup of AutoTex over Global for the resulting categories on both platforms. Only candidate-containing kernels are included because they contain at least one access directly selected by the compiler analysis; compelled-only kernels are excluded because their transformation follows from provenance-group consistency rather than from an access identified within the kernel itself. Each bar is additionally annotated when the bottleneck category under AutoTex differs from its Global category.

LSU-bound kernels obtained the largest and most consistent improvements, with geometric-mean speedups of 4.01× on Ada Desktop and 3.28× on Ampere Laptop. Most of these kernels transitioned from LSU-bound to TEX-bound after transformation. This shift is consistent with the replacement of global-memory instructions processed through the LSU path by texture or surface instructions processed through the TEX path. More importantly, the accompanying speedups indicate that moving the accesses away from the original LSU bottleneck substantially reduces the memory-processing pressure that limited the baseline kernels.

DRAM-bound kernels remained close to parity, with geometric-mean speedups of 1.03× on both platforms, and generally remained DRAM-bound after transformation. Their unchanged classification and limited speedups indicate that reducing work within

[5]Specifically, this refinement applies when the dominant contributors are `sm__inst_executed_pipe_lsu` on the compute side and `l1tex__lsuin_requests` on the memory side. These metrics may share the same underlying hardware counter.



1: IS/full_verify_gpu_kernel_1
2: IS/full_verify_gpu_kernel_2
3: LU/jacld_blts_gpu_kernel
4: LU/jacu_buts_gpu_kernel
5: LU/pintgr_gpu_kernel_3
6: LU/rhs_gpu_kernel_3
7: LU/rhs_gpu_kernel_4
8: SP/rhs_norm_gpu_kernel_1
9: SP/x_solve_gpu_kernel
10: adi/adi_kernel1
11: adi/adi_kernel3
12: atax/atax_kernel1
13: bicg/bicg_kernel2
14: gemver/gemver_kernel3
15: gesummv/gesummv_kernel
16: gramschmidt/gramschmidt_kernel2
17: mvt/mvt_kernel1
18: syr2k/syr2k_kernel
19: syrk/syrk_kernel
20: gaussian/Fan1
21: gaussian/Fan2
22: huffman/pack2
23: kmeans/invert_mapping
24: kmeans/kmeansPoint
25: leukocyte/dilate_kernel

Figure 7.5: Per-kernel speedup of AutoTex over Global for candidate-containing kernels grouped by their bottleneck category under Global. A letter near the 1× line denotes the bottleneck category under AutoTex when it differs from the baseline: C for compute, D for DRAM, L for latency, and T for TEX.

the cache-access pipeline does not remove the dominant cost when execution remains constrained by the rate at which data are supplied from off-chip memory.

Latency-bound kernels exhibited substantial but more varied improvements, reaching geometric-mean speedups of 1.81× on Ada Desktop and 2.11× on Ampere Laptop. Several faster kernels transitioned from latency-bound to DRAM-bound, indicating that the transformation sufficiently reduces the original latency-related limitation for off-chip memory throughput to become dominant. Other kernels remained latency-bound despite improving, showing that the broad classification alone cannot identify which latency source was reduced or which limitation remains. Their individual outcomes are therefore examined using wavefront, stall, cache, and traffic metrics in the following subsections.

Compute-bound kernels appeared only on Ampere Laptop. They remained compute-bound and close to 1.00×, apart from `dilate_kernel`, consistent with the expectation that reducing memory-system work provides little benefit while compute throughput continues to limit execution. The 0.84× category geometric mean was driven by `dilate_kernel`, which slowed down and transitioned from compute-bound to latency-bound. This transition indicates that the transformation introduces additional latency-related stalls that become the dominant execution limitation. The subsequent stall analysis identifies the specific source of this added waiting time.

7.3.2 Reduction in Cache-Access Wavefronts

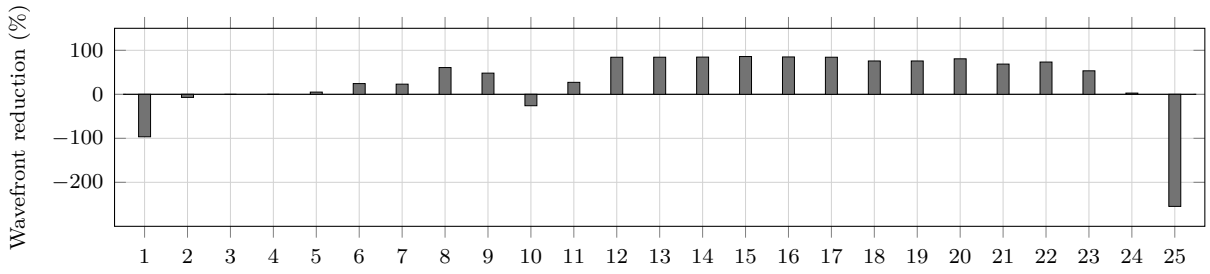
The bottleneck analysis indicates the execution conditions under which AutoTex improves performance, but does not directly establish how the transformation changes memory-system processing. The primary mechanism targeted by AutoTex includes the reduction of cache-access wavefronts generated by low-utilization global-memory instructions. By mapping multidimensional objects to a block-linear layout, the transformation places accesses from more threads within the same cache sectors and cache lines, allowing the memory subsystem to process each warp instruction using fewer wavefronts.

To quantify this effect, we compare the cache-access wavefronts generated by Global and AutoTex for the candidate-containing kernels. Let W_C denote the L1 wavefront count reported by the profiler for configuration C, including wavefronts processed through both the LSU and TEX paths. We report the reduction

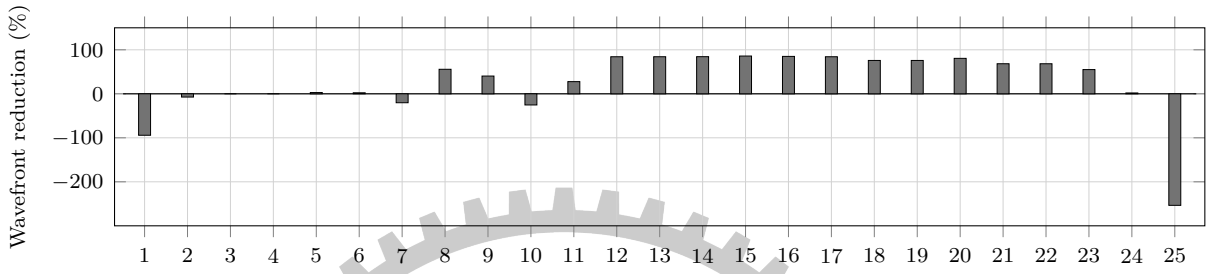
$$R_W = 1 - \frac{W_{\text{AutoTex}}}{W_{\text{Global}}},$$

as a percentage in Figure 7.6. Positive values indicate fewer wavefronts under AutoTex, whereas negative values indicate an increase.

AutoTex reduced total cache-access wavefronts for most candidate-containing kernels on both platforms. Many reductions fell between approximately 50% and 90%, indi-



(a) Ada Desktop



(b) Ampere Laptop

1: IS/full_verify_gpu_kernel_1
2: IS/full_verify_gpu_kernel_2
3: LU/jacld_blts_gpu_kernel
4: LU/jacu_buts_gpu_kernel
5: LU/pintgr_gpu_kernel_3
6: LU/rhs_gpu_kernel_3
7: LU/rhs_gpu_kernel_4
8: SP/rhs_norm_gpu_kernel_1
9: SP/x_solve_gpu_kernel
10: adi/adi_kernel1
11: adi/adi_kernel3
12: atax/atax_kernel1
13: bicg/bicg_kernel2
14: gemver/gemver_kernel3
15: gesummv/gesummv_kernel
16: gramschmidt/gramschmidt_kernel2
17: mvt/mvt_kernel1
18: syr2k/syr2k_kernel
19: syrk/syrk_kernel
20: gaussian/Fan1
21: gaussian/Fan2
22: huffman/pack2
23: kmeans/invert_mapping
24: kmeans/kmeansPoint
25: leukocyte/dilate_kernel

Figure 7.6: Per-kernel total cache-access wavefront reduction of AutoTex relative to Global for candidate-containing kernels.

cating that the block-linear layout places the addresses requested by a warp within fewer cache sectors and cache lines. The similar per-kernel reductions across the two platforms further suggest that this effect arises primarily from the transformed access pattern and physical layout rather than from platform-specific execution behavior.

The largest reductions occurred among kernels with substantial speedups in Figure 7.5. In particular, the strongly accelerated LSU-bound kernels presented substantially fewer wavefronts to the cache-access pipeline after transformation. Together with the throttle-stall reductions examined in the following subsection, this result shows that the reduced wavefront count alleviates contention along the memory-processing path. Several latency-bound kernels likewise combined large wavefront reductions with substantial speedups. The subsequent stall, cache, and traffic analyses determine how this reduction affects memory-related waiting and request propagation through the memory hierarchy.

Four kernels on Ada Desktop and five kernels on Ampere Laptop instead generated more wavefronts under AutoTex. Two occur in the NPB-GPU IS benchmark: the first

and second `full_verify` kernels. In both kernels, the reconstructed two-dimensional access has the form

```
key_buff2[idx / KEY_ROW_SIZE][idx % KEY_ROW_SIZE],
```

where `idx = blockIdx.x * blockDim.x + threadIdx.x` and `KEY_ROW_SIZE` = 8192. Because both index expressions depend on `threadIdx.x`, the divergence analysis conservatively classifies both as potentially divergent. In practice, the first index remains uniform within each warp, whereas adjacent threads request adjacent elements along the second dimension. The accesses therefore already concentrate the addresses requested by a warp within a small number of cache sectors and cache lines and do not exhibit the low intra-warp cache utilization inferred by the analysis.

Mapping these accesses to the TEX path increased wavefront generation because that path processes a warp through multiple lane groups (Section 5.3). The additional wavefronts contributed to the slowdown of the first `full_verify` kernel, whereas the second remained close to its baseline performance because other latency sources continued to dominate its execution. Smaller increases also appeared in `adi_kernel1` on both platforms and in `rhs_gpu_kernel_4` on Ampere Laptop.

The largest increase occurred in `dilate_kernel` (kernel 25). Its candidate index is derived through a more complex computation that the divergence analysis conservatively classifies as potentially divergent, although the resulting runtime index values appear to remain uniform within each warp. The transformation consequently replaces an access requiring few LSU-path wavefronts with a TEX-path access requiring more wavefronts. The added pipeline work introduces memory-related stalls, causing the kernel to transition from compute-bound to latency-bound and reducing its performance. Together, these cases expose a precision limitation of the current candidate analysis: dependence on a divergent value does not necessarily imply that the resulting addresses are widely distributed across a warp. Recognizing warp-local uniformity in reconstructed and computed indices would prevent these transformations, analogous to the imprecision responsible for the excluded kernels (Section 7.1.2).

7.3.3 Reduction in Memory-System Stalls

The preceding analysis shows that AutoTex reduces the cache-access wavefronts generated by most candidate-containing kernels. We next examine whether this reduction translates into less memory-system contention and shorter memory-related waiting time. Throttle stalls occur when a memory instruction cannot issue because the corresponding processing path lacks available resources, whereas long scoreboard stalls occur when a warp waits for an outstanding high-latency memory operation on which a subsequent instruction depends (Section 3.3.2).

Under GLOBAL, contention on the global-memory processing path appears as LG throttle stalls. Because AUTOTEX may access objects through global, texture, and surface interfaces, contention after transformation may instead appear as either LG throttle or TEX throttle stalls. We therefore aggregate the two as

$$T_{\text{throttle}} = T_{\text{LG throttle}} + T_{\text{TEX throttle}}.$$

This aggregation prevents memory-system work shifted from the LSU path to the TEX path from being misinterpreted as eliminated contention. Long scoreboard stalls are considered separately because they reflect waiting for outstanding memory operations rather than resource unavailability at instruction issue.

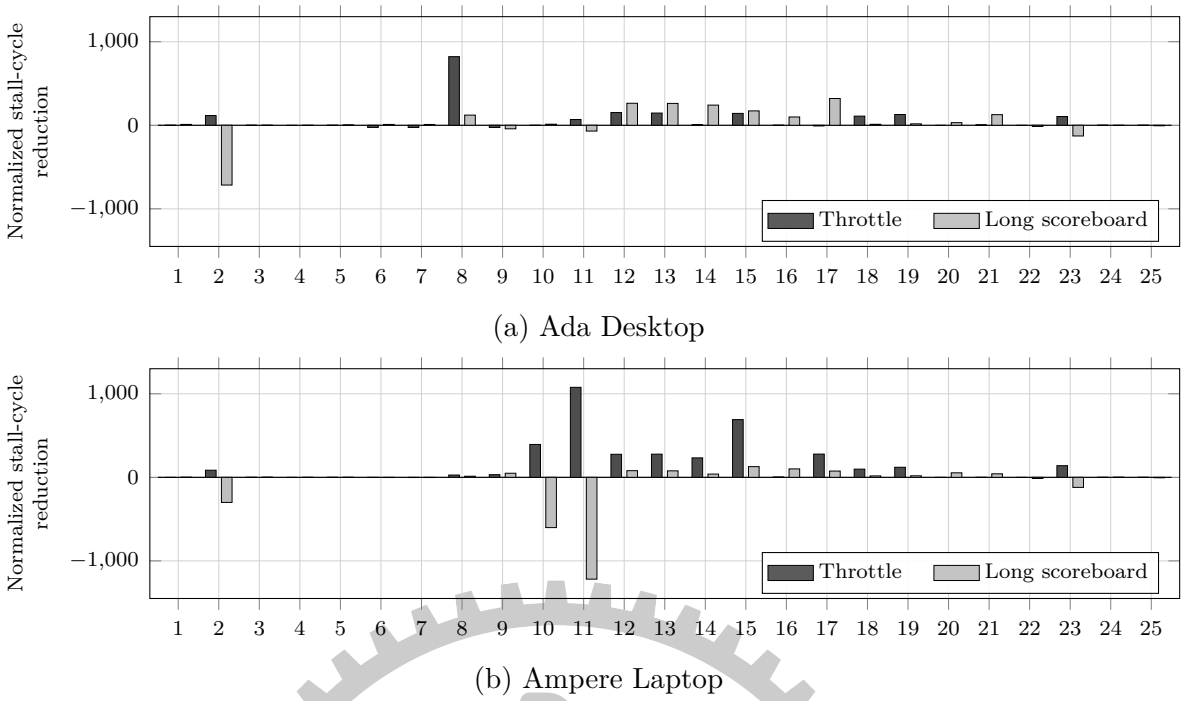
Figure 7.7 reports the change in these stalls for the candidate-containing kernels. For stall reason r, let $S_{C,r}$ denote the derived stall warp-cycle count under configuration C, and let I_{GLOBAL} denote the issuing warp-cycle count under GLOBAL. We report the normalized stall-cycle reduction

$$\Delta\rho_r = \frac{S_{\text{GLOBAL},r} - S_{\text{AUTOTEX},r}}{I_{\text{GLOBAL}}}.$$

Using the GLOBAL issuing warp-cycle count as a common denominator preserves the change in total stall waiting without allowing differences in issuing warp-cycle counts between the two configurations to affect the normalization. Positive values indicate stall warp-cycles eliminated by AUTOTEX, whereas negative values indicate additional waiting.

AUTOTEX reduced aggregate throttle stalls for 12 of the 25 candidate-containing kernels on Ada Desktop and 16 on Ampere Laptop. The reductions were particularly pronounced among the kernels that exhibited substantial wavefront reductions and performance improvements in the preceding analyses. This relationship is clearest for the LSU-bound kernels: reducing the number of wavefronts presented to the memory system decreases pressure on the memory-processing paths and allows more memory instructions to issue without being throttled. The cross-platform differences provide further evidence for this relationship. For example, kernels 8, 16, and 17 exhibited larger throttle-stall reductions on Ampere Laptop than on Ada Desktop, consistent with their larger performance improvements on Ampere in Figure 7.2.

The throttle and long-scoreboard components, however, should not be interpreted as independent sources of waiting. For a warp whose next instructions encounter both memory-pipeline pressure and dependencies on outstanding memory operations, the reported stall reason reflects the condition that currently prevents issue. Moving selected memory instructions from the LSU path to the TEX path changes this interaction because the two paths have separate instruction queues. When GLOBAL saturates the LSU path, a warp may encounter an LG throttle stall before reaching an instruction that depends on an outstanding memory operation. After transformation, the corresponding access may enter the TEX path without being throttled, allowing execution to advance until the



(a) Ada Desktop

(b) Ampere Laptop

1: IS/full_verify_gpu_kernel_1
2: IS/full_verify_gpu_kernel_2
3: LU/jacld_blts_gpu_kernel
4: LU/jacu_buts_gpu_kernel
5: LU/pintgr_gpu_kernel_3
6: LU/rhs_gpu_kernel_3
7: LU/rhs_gpu_kernel_4
8: SP/rhs_norm_gpu_kernel_1
9: SP/x_solve_gpu_kernel
10: adi/adi_kernel1
11: adi/adi_kernel3
12: atax/atax_kernel1
13: bicg/bicg_kernel2
14: gemver/gemver_kernel3
15: gesummv/gesummv_kernel
16: gramschmidt/gramschmidt_kernel2
17: mvt/mvt_kernel1
18: syr2k/syr2k_kernel
19: syrk/syrk_kernel
20: gaussian/Fan1
21: gaussian/Fan2
22: huffman/pack2
23: kmeans/invert_mapping
24: kmeans/kmeansPoint
25: leukocyte/dilate_kernel

Figure 7.7: Normalized reduction in memory-related warp stalls under AutoTex relative to Global for candidate-containing kernels. The throttle component combines LG throttle and TEX throttle stalls. Positive values indicate reduced waiting under AutoTex.

unresolved dependency instead produces a long scoreboard stall. Conversely, prolonged dependency waiting can reduce the rate at which instructions reach the memory queues and thereby reduce observed throttle stalls. A decrease in one stall reason can therefore expose an increase in the other without a comparable change in execution time.

This interaction is particularly visible in kernels 10 and 11, both of which are latency-bound. Under AutoTex, their throttle stalls decreased while their long scoreboard stalls increased by a comparable amount. The transformation shifts selected memory instructions from the LSU path to the TEX path, allowing them to enter a separate instruction queue rather than being blocked by LSU-path contention. Execution can therefore progress farther before encountering a dependency on an outstanding memory operation, causing waiting that previously appeared as a throttle stall to instead appear as a long scoreboard stall. Conversely, prolonged scoreboard waiting can reduce the rate at which subsequent memory instructions reach the issue queues. The similar magnitude of the two changes, together with the limited performance impact of the transformation,

indicates that much of the underlying memory-related waiting remains but is exposed through a different stall reason.

The remaining cases show that reductions in wavefront generation and throttle stalls are closely related but do not alone determine performance. Where both decrease substantially, the results provide direct evidence that AutoTex alleviates contention in the cache-access and memory-processing paths. Changes in long scoreboard stalls additionally reflect outstanding-memory latency and the altered exposure of dependencies after contention is removed. The following subsection therefore examines cache effectiveness and lower-level memory traffic to determine whether reduced memory-system work also decreases the amount of data propagated through the cache hierarchy.

7.3.4 Cache Effectiveness and Memory Traffic

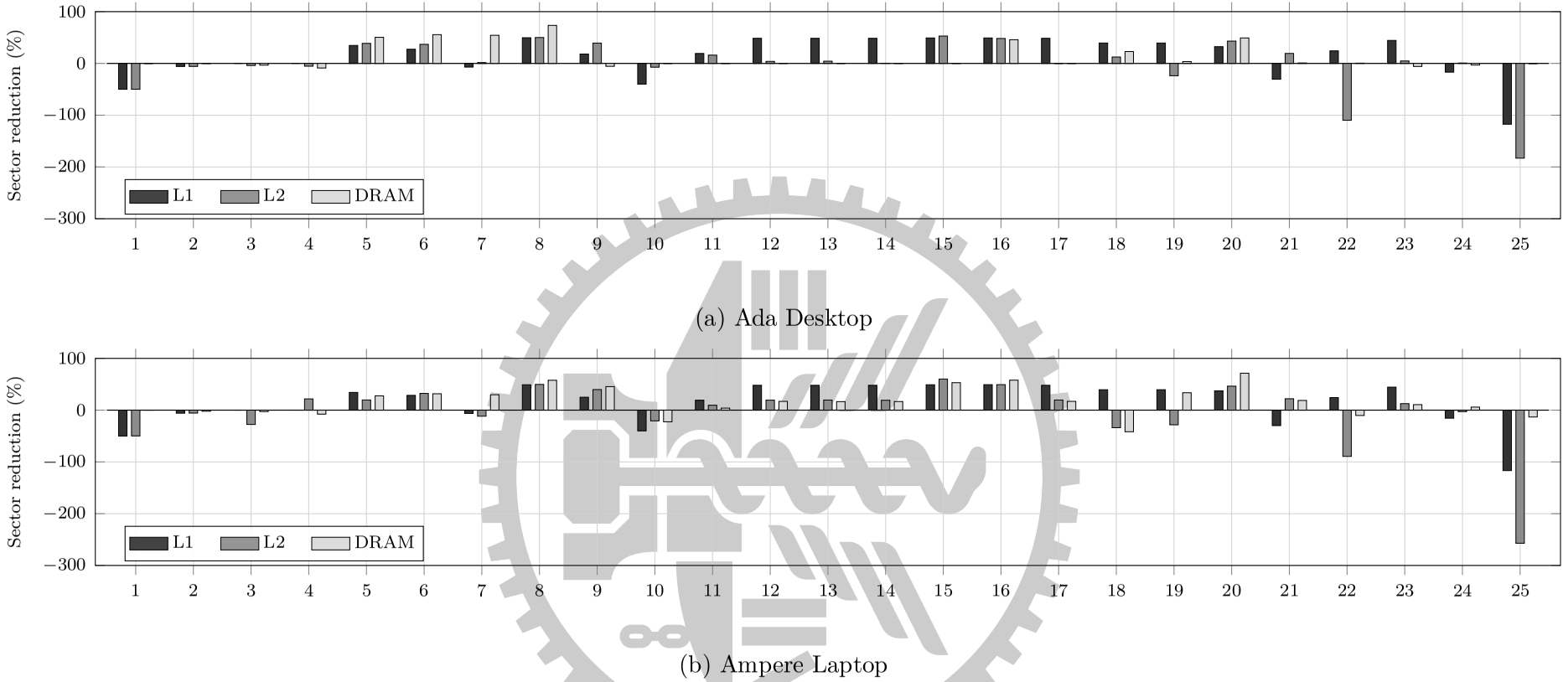
The preceding analyses show that AutoTex reduces cache-access wavefront generation and, for many kernels, the resulting memory-system stalls. We next examine how these changes affect the sector traffic processed throughout the memory hierarchy and where the remaining accesses are satisfied. Improving intra-warp cache utilization can directly reduce the number of L1 sectors requested by a warp. The resulting reduction in cache footprint may also reduce the traffic propagated to the L2 cache and DRAM, either because fewer requests are generated at the L1 cache or because reused data remain resident longer.

We first consider sector counts because they directly quantify the traffic processed at each level of the memory hierarchy. Let $N_{\ell,C}$ denote the sector count at hierarchy level ℓ under configuration C. We report the sector reduction

$$R_\ell = 1 - \frac{N_{\ell,\text{AutoTex}}}{N_{\ell,\text{Global}}},$$

as a percentage in Figure 7.8. At the L1 cache, $N_{\ell,C}$ counts the sectors accessed by the cache-access pipeline, whereas the L2 and DRAM counts represent sectors fetched from the corresponding lower levels. Positive values therefore indicate less sector traffic under AutoTex, whereas negative values indicate an increase.

Figure 7.8 shows that AutoTex reduced sector traffic for most candidate-containing kernels. The reduction was particularly consistent at the L1 cache and closely followed the wavefront reductions in Figure 7.6. These metrics capture related but distinct consequences of the block-linear layout: fewer wavefronts reduce the number of cache-access units required to process a warp instruction, whereas fewer L1 sectors show that the resulting accesses collectively touch less cache-sector data. Together, they provide direct evidence that the transformation improves intra-warp cache-sector utilization for the accesses identified correctly by the candidate analysis.



1: IS/full_verify_gpu_kernel_1
2: IS/full_verify_gpu_kernel_2
3: LU/jacld_blts_gpu_kernel
4: LU/jacu_buts_gpu_kernel
5: LU/pintgr_gpu_kernel_3
6: LU/rhs_gpu_kernel_3
7: LU/rhs_gpu_kernel_4
8: SP/rhs_norm_gpu_kernel_1
9: SP/x_solve_gpu_kernel
10: adi/adi_kernel1
11: adi/adi_kernel3
12: atax/atax_kernel1
13: bicg/bicg_kernel2
14: gemver/gemver_kernel3
15: gesummv/gesummv_kernel
16: gramschmidt/gramschmidt_kernel2
17: mvt/mvt_kernel1
18: syr2k/syr2k_kernel
19: syrk/syrk_kernel
20: gaussian/Fan1
21: gaussian/Fan2
22: huffman/pack2
23: kmeans/invert_mapping
24: kmeans/kmeansPoint
25: leukocyte/dilate_kernel

Figure 7.8: Memory-sector reduction of AutoTex relative to Global for candidate-containing kernels. Positive values indicate fewer sectors accessed in the L1 cache or fetched from the L2 cache and DRAM.

For many kernels, the reduction also extended to the L2 cache and DRAM, showing that less traffic propagated through the memory hierarchy. This effect can arise directly from the lower number of L1 sector requests and, where accompanied by increased cache hit rates, may additionally indicate improved cache retention. A smaller cache footprint reduces sensitivity to cache contention, allowing reused data to remain resident longer and avoiding some refetches from lower levels. Reduced lower-level traffic therefore provides an additional source of memory-system relief beyond the reduction in L1 processing pressure and can contribute to the throttle-stall reductions observed in Figure 7.7.

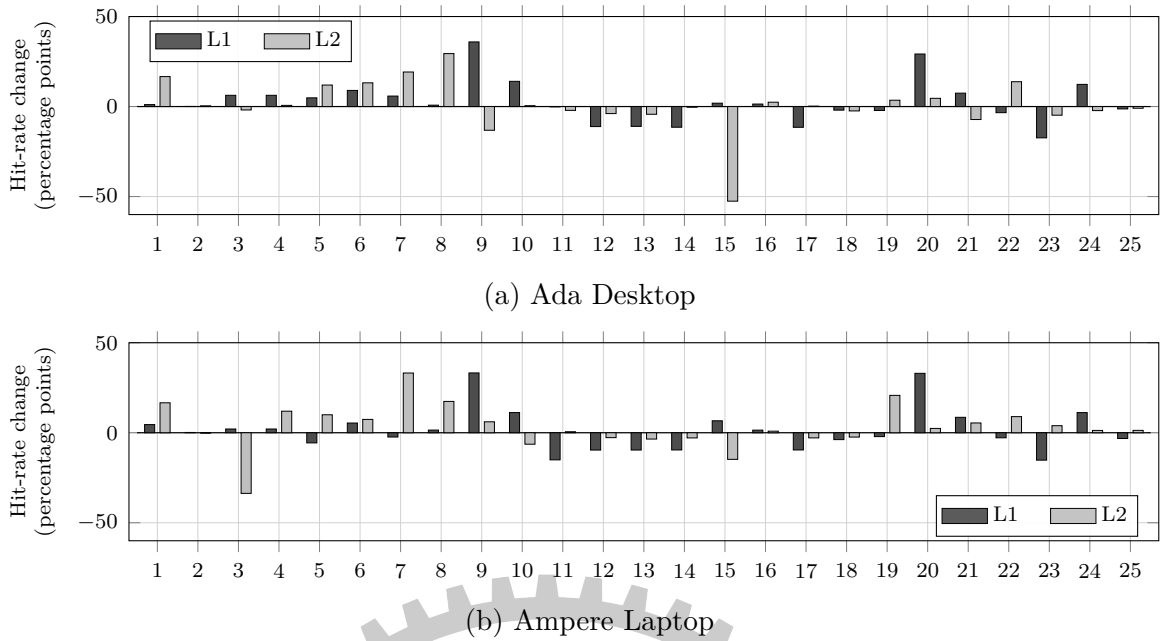
The reductions are not uniform across all hierarchy levels. In some kernels, fewer L1 sector accesses are accompanied by little change or a modest increase in L2 or DRAM traffic. Such behavior does not contradict the observed performance improvements. Reducing L1 sector processing can alleviate pressure on the cache-access and LSU processing paths even when the traffic reaching lower levels does not decrease. This distinction is particularly relevant to LSU-bound kernels, whose baseline limitation lies in processing memory requests rather than in the service rate of a lower memory level. For example, `syrk_kernel` substantially reduced its L1 sector accesses and improved performance despite comparatively small changes in lower-level traffic. Alleviating its baseline LSU bottleneck therefore does not require a corresponding reduction in DRAM demand.

More generally, reducing the total number of L1 accesses does not necessarily reduce the number of requests reaching lower levels because the transformation also changes which sectors are accessed and how they are reused before eviction. Conversely, lower-level traffic may decrease simply because fewer L1 requests are generated, without necessarily indicating improved cache retention. We therefore treat sector counts as the direct measure of traffic at each hierarchy level and use cache hit rates as supporting evidence for changes in retention.

The exceptions identified in the wavefront analysis are also visible in the sector counts. The first `full_verify` kernel (kernel 1) generated additional L1 traffic after transformation, consistent with its already concentrated baseline access pattern and increased wavefront generation. The most pronounced case was `dilate_kernel` (kernel 25), for which AutoTex substantially increased sector traffic throughout the hierarchy on both platforms. This result reinforces the earlier conclusion that its transformation does not improve intra-warp locality and instead introduces additional memory-system work, consistent with its increased wavefront generation, transition to a latency bottleneck, and performance regression.

Sector counts describe the amount of traffic processed at each level, but do not reveal what fraction of the remaining accesses is satisfied there. We therefore additionally examine the L1- and L2-cache hit rates as complementary measurements. For cache level ℓ, Figure 7.9 reports

$$\Delta H_\ell = H_{\ell,\text{AutoTex}} - H_{\ell,\text{Global}}$$



(a) Ada Desktop

(b) Ampere Laptop

1: IS/full_verify_gpu_kernel_1
2: IS/full_verify_gpu_kernel_2
3: LU/jacld_blts_gpu_kernel
4: LU/jacu_buts_gpu_kernel
5: LU/pintgr_gpu_kernel_3
6: LU/rhs_gpu_kernel_3
7: LU/rhs_gpu_kernel_4
8: SP/rhs_norm_gpu_kernel_1
9: SP/x_solve_gpu_kernel
10: adi/adi_kernel1
11: adi/adi_kernel3
12: atax/atax_kernel1
13: bicg/bicg_kernel2
14: gemver/gemver_kernel3
15: gesummv/gesummv_kernel
16: gramschmidt/ gramschmidt_kernel2
17: mvt/mvt_kernel1
18: syr2k/syr2k_kernel
19: syrk/syrk_kernel
20: gaussian/Fan1
21: gaussian/Fan2
22: huffman/pack2
23: kmeans/invert_mapping
24: kmeans/kmeansPoint
25: leukocyte/dilate_kernel

Figure 7.9: L1- and L2-cache hit-rate changes of AutoTex relative to Global for candidate-containing kernels. Changes are reported in percentage points.

in percentage points. Positive values indicate that a larger fraction of accesses hits at that cache level under AutoTex.

Unlike the sector reductions, the hit-rate changes do not exhibit a uniform relationship with performance. Both increases and decreases occur among kernels that improve under AutoTex, and the direction may differ between the L1 and L2 caches. This behavior follows from the fact that hit rate is a ratio whose underlying number of accesses also changes after transformation. Eliminating repeated accesses to partially consumed sectors, for example, can remove cache hits while still reducing the total number of sector accesses. Conversely, a higher hit rate does not by itself imply lower traffic if the transformed execution generates more requests.

The hit rates are therefore most useful for interpreting the cause of lower-level traffic changes. Where reduced L2 or DRAM sector counts are accompanied by increased upstream hit rates, the results support the interpretation that the smaller cache footprint improves retention and reduces refetching from lower levels. Where sector traffic decreases without a corresponding hit-rate increase, the reduction is more directly attributable to fewer requests being generated in the first place. Hit-rate decreases should therefore not

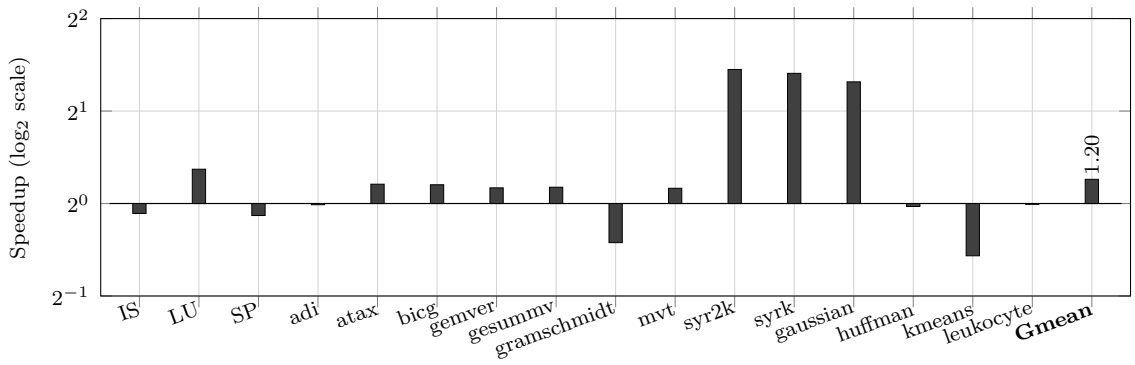
be interpreted as regressions in isolation; the sector counts determine whether more or less traffic is actually processed.

Taken together, the architectural measurements identify two related effects of the block-linear layout. First, improved intra-warp locality reduces the number of cache-access wavefronts and L1 sector accesses for most candidate-containing kernels, alleviating pressure on the cache-access and memory-processing paths and reducing throttle stalls. This effect can produce substantial gains for kernels whose baseline execution is limited by the LSU path even when lower-level traffic changes little. Second, for many kernels, the reduction extends to the L2 cache and DRAM, decreasing the traffic propagated through the memory hierarchy; favorable hit-rate changes in some of these cases further indicate improved cache retention and reduced refetching. The relative importance of these effects depends on the baseline bottleneck: kernels whose limiting resource is relieved can obtain substantial speedups, whereas kernels constrained by unaffected resources benefit less, and incorrectly selected accesses may instead introduce additional memory-system work. These effects collectively explain the major per-kernel performance differences observed in Figure 7.2.

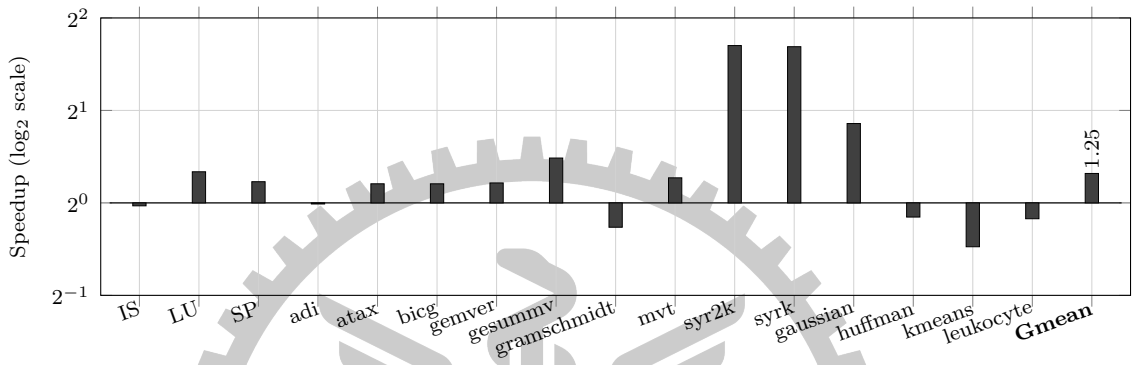
7.4 Application-Level Performance

The preceding sections evaluate the kernel-level performance of AutoTex and relate the observed speedups to changes in memory-system behavior. Kernel execution time alone, however, does not capture the complete application-level effect of the transformation. A benchmark may invoke multiple kernels whose performance changes differently under AutoTex, and the aggregate kernel-execution time depends on the execution-time contribution and invocation frequency of each kernel. Mapping an object from linear global memory to block-linear storage also changes the host-side operations required to manage that object. In addition to allocating and releasing the underlying storage, AutoTex may require different data-transfer APIs and the creation and destruction of texture or surface objects. The transformed kernel stubs must also marshal the corresponding access handles when issuing kernel launches. These factors may preserve, dilute, or completely offset the speedups observed for individual kernels.

This section therefore evaluates AutoTex at the application level. We first compare the end-to-end execution times of Global and AutoTex over the complete measured application interval. We then examine how the execution-time changes of individual kernels combine to determine the aggregate kernel-execution time of each benchmark. Finally, we analyze the baseline runtime shares and component-level changes associated with allocation and object creation, data transfer, kernel launch, kernel execution, and deallocation and object destruction. Together, these analyses determine whether the per-kernel improvements translate into end-to-end speedup and explain how the resulting



(a) Ada Desktop



(b) Ampere Laptop

Figure 7.10: Application-level speedup of AutoTex over Global.

performance is shaped by the distribution of work within each benchmark and by the host-side costs of block-linear storage.

The application-level analysis includes only workloads for which the benchmark execution and timing boundaries represent comparable end-to-end work across configurations. Initialization or verification performed outside the measured region is excluded consistently. Each benchmark is executed five times under each configuration, and the geometric mean of the five measured execution times is used to compute its application-level speedup. As in the kernel-level evaluation, Global serves as the baseline.

7.4.1 End-to-End Performance

Figure 7.10 presents the application-level speedup of AutoTex over Global. Because the measured interval includes both GPU kernel execution and the associated host-side CUDA operations, the results indicate whether the kernel-execution savings are sufficient to amortize any additional memory-management and object-management costs introduced by the transformation.

Overall, across the 16 evaluated benchmarks, AutoTex achieved application-level speedups with geometric means of 1.20× on the Ada Desktop and 1.25× on the Ampere Laptop. The largest improvements were obtained by `syr2k`, `syrk`, and `gaussian` on

both platforms, and `gesummv` also showed a pronounced gain on the Ampere Laptop. In these workloads, the reductions in kernel execution time were large enough relative to the remaining runtime components to produce pronounced end-to-end improvements.

The other benchmarks generally exhibited more modest changes. On the Ada Desktop, `LU`, `atax`, `bicg`, `gemver`, `gesummv`, and `mvt` obtained application-level improvements, whereas `adi` and `leukocyte` remained close to parity. The Ampere Laptop showed visible speedups for `LU`, `SP`, `atax`, `bicg`, `gemver`, `gesummv`, and `mvt`, while `IS` and `adi` remained close to parity. The smaller gap between the two platform-level geometric means therefore reflects a balance between stronger gains for several PolyBench-ACC workloads on Ampere and larger slowdowns in several non-PolyBench workloads.

A small number of workloads experienced net slowdowns. The most pronounced case was `kmeans`, which was slower under AutoTex on both platforms. `gramschmidt` also slowed down on both platforms, while `SP` and `IS` regressed on the Ada Desktop and `huffman` and `leukocyte` regressed on the Ampere Laptop. These results indicate that the kernel-level improvements are not always sufficient to determine the application-level outcome. The following subsections examine first how the changes in individual kernels combine within each benchmark and then how the resulting aggregate kernel time interacts with the remaining runtime components.

These results show that AutoTex provides a net application-level benefit on both platforms, but the end-to-end result cannot be inferred from an unweighted collection of per-kernel speedups. It first depends on how the changes in individual kernels combine according to their execution-time contributions and invocation frequencies. It then depends on the fraction of the complete application runtime occupied by aggregate kernel execution and on the changes in the remaining host- and device-side components. The following subsections examine these two levels in turn.

7.4.2 Per-Kernel Contribution to Application Performance

A benchmark may invoke multiple kernels whose performance responds differently to AutoTex. Some transformed kernels may execute faster, while others remain unchanged or slow down. The aggregate kernel-execution time therefore depends not only on the per-kernel speedups but also on the execution-time contribution and invocation frequency of each kernel.

For a benchmark containing kernels $k \in \mathcal{K}$, the total kernel-execution time under configuration C is

$$T_{\text{kernel},C} = \sum_{k \in \mathcal{K}} n_{k,C} T_{k,C},$$

where $n_{k,C}$ denotes the number of invocations of kernel k, and $T_{k,C}$ denotes its average execution time under configuration C. Because Global and AutoTex preserve the

application control flow, the invocation counts are expected to remain unchanged; differences in aggregate kernel time therefore arise from the execution-time changes of the individual kernels.

Figure 7.11 reports the contribution of each kernel to the total kernel-execution time under Global and AutoTex. The results distinguish kernels accelerated by the transformation from kernels that remained unchanged or became slower. This decomposition reveals whether a benchmark-level improvement is driven by one dominant kernel or by gains distributed across several kernels. It also identifies cases in which slowdowns in frequently invoked or long-running kernels outweigh improvements elsewhere.

A kernel can materially affect benchmark-level performance only when its execution-time change is large relative to its contribution to the baseline kernel time. Substantial speedups in kernels that occupy only a small fraction of execution may have little aggregate effect, whereas even a modest slowdown in a dominant kernel can cause the benchmark to regress. The benchmark-level kernel speedup should therefore be interpreted as a weighted outcome rather than as an unweighted summary of the per-kernel speedups.

The kernel-time distributions explain why the performance of individual transformed kernels does not directly determine the benchmark-level result. In several PolyBench-ACC workloads, a single transformed kernel dominated the original kernel-execution time. The reductions in this dominant component accounted for the pronounced aggregate kernel-time improvements of `gesummv`, `syr2k`, and `syrk` on both platforms. The improvements in `atax`, `bicg`, `mvt`, and `gaussian` similarly arose because the kernels accelerated by AutoTex constituted substantial fractions of their respective baseline kernel times.

Benchmarks containing several important kernels exhibited a weighted combination of their individual changes. For example, `LU`, `SP`, `adi`, and `gemver` each contain multiple transformed kernels with non-negligible baseline contributions. Their benchmark-level kernel performance was therefore determined by how the gains and regressions of these kernels combined according to their original execution-time contributions, rather than by a single uniformly dominant kernel. Kernels with negligible baseline contributions have little effect even when their individual speedups are large.

The kernel-time distributions distinguish these slowdown causes. In `gramschmidt`, the aggregate kernel-time increase on both platforms was driven primarily by the slowdown of its dominant transformed kernel, which is a compelled-only kernel. In `leukocyte`, kernels left unchanged by AutoTex accounted for most of the baseline kernel time, so the transformed kernels had only a limited effect on the aggregate result. On Ampere Laptop, however, the unchanged portion itself became slower under AutoTex. Because these kernels are not transformed, this regression is likely associated with the additional prerequisite passes applied before the AutoTex analysis and transformation passes.

By contrast, `kmeans` was dominated by `kmeansPoint`, whose aggregate kernel time was near parity on Ada Desktop but increased on Ampere Laptop. The severe end-to-end

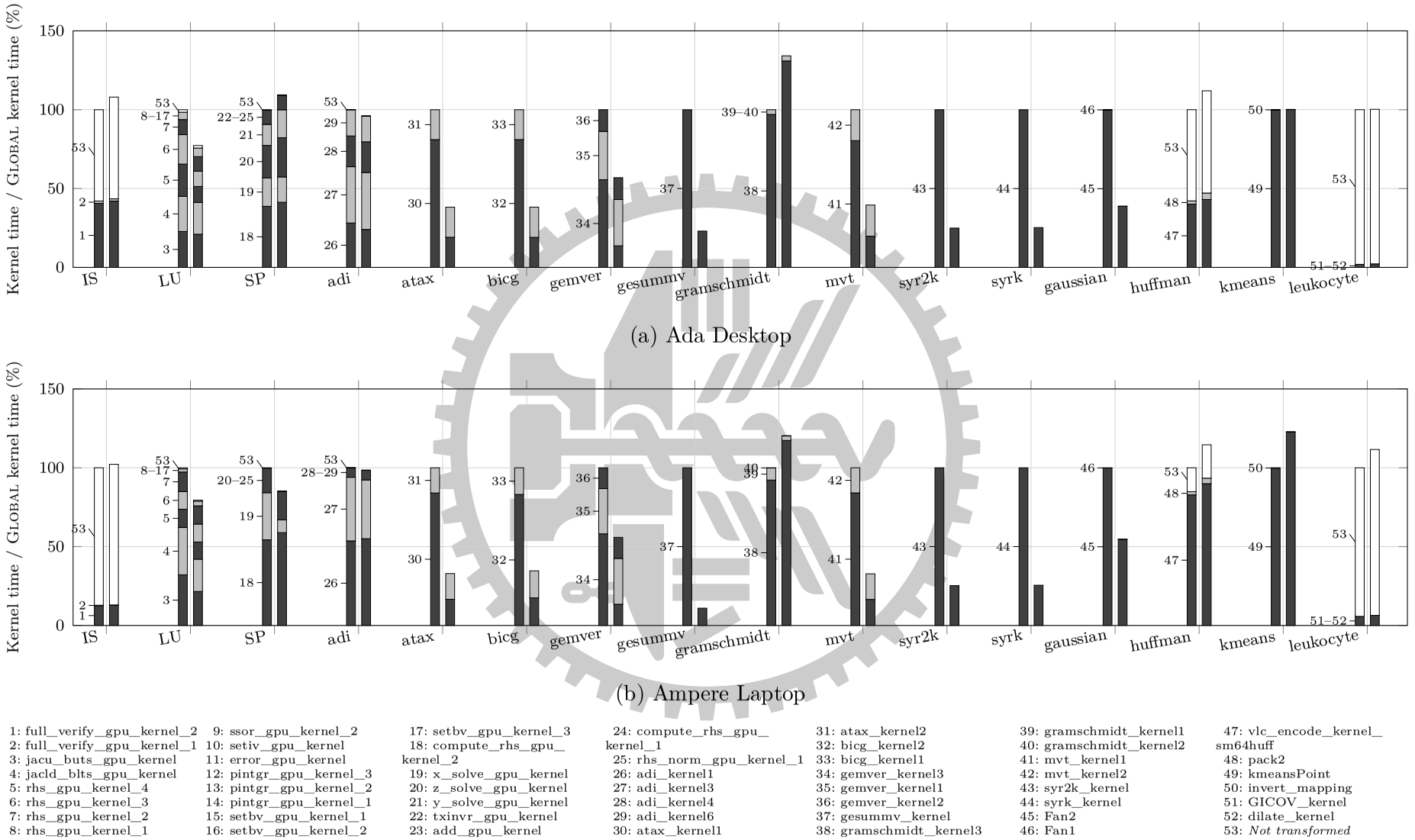


Figure 7.11: Per-kernel contribution to application-level kernel execution time. Each benchmark shows paired stacked bars for Global and AutoTex, normalized to the benchmark's total Global kernel-execution time. Small transformed-kernel contributions ($< 7\%$) are grouped by number range, while *Not transformed* combines kernels left unchanged by AutoTex.

slowdown of `kmeans` on both platforms therefore cannot be explained by kernel execution alone and depends on the non-kernel components examined in the following subsection.

7.4.3 Runtime Overhead Breakdown

To explain why the kernel-level gains do not translate directly into equivalent end-to-end speedups, we decompose the measured application time into the runtime components most relevant to the transformation. Table 7.5 defines these components and their corresponding Nsight Systems data sources.

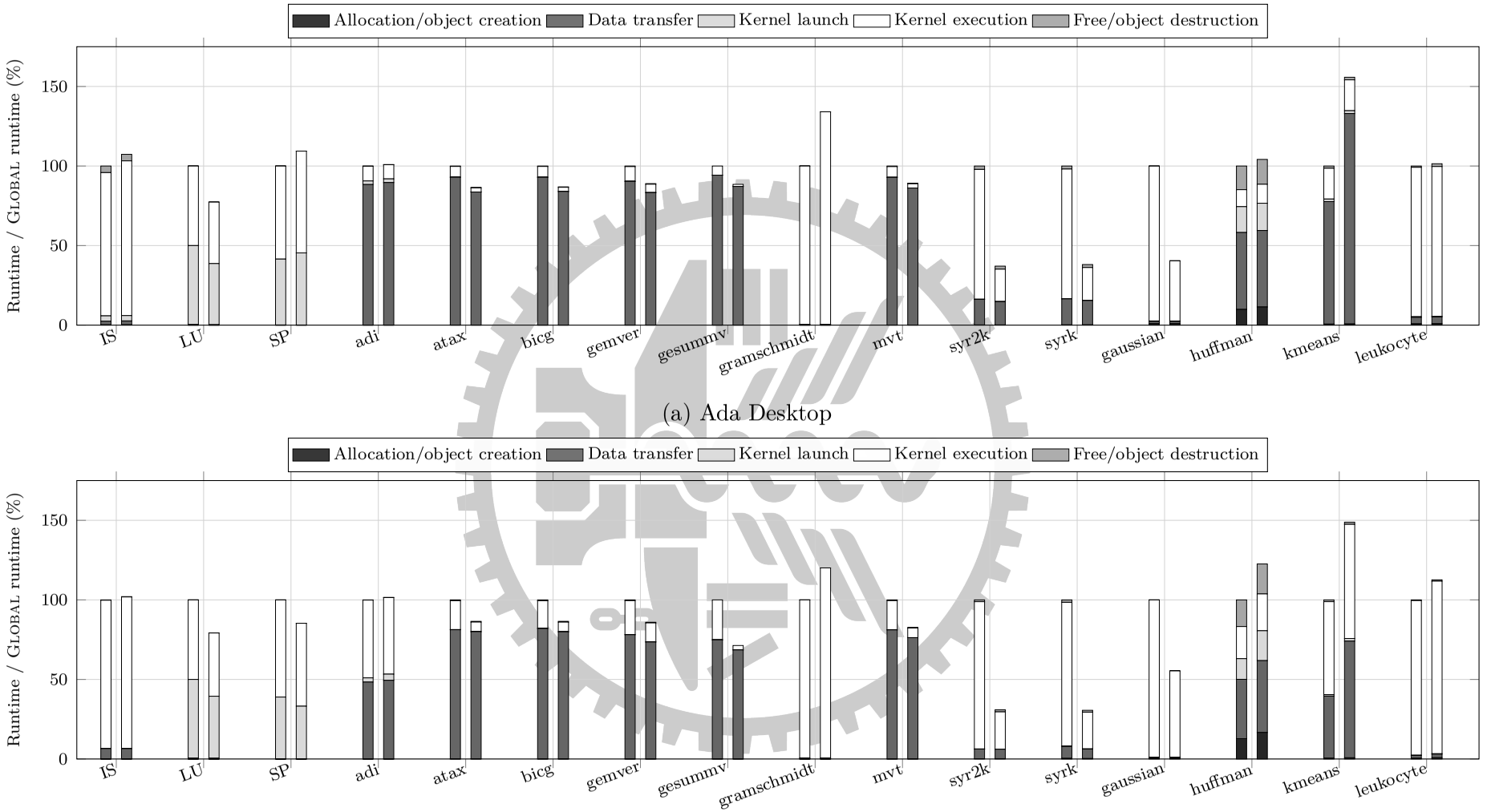
Table 7.5: CUDA Runtime APIs included in each host-side component of the application-level breakdown.

Runtime component	Included APIs
Allocation/object creation	`cudaMalloc`, `cudaMallocArray`, `cudaMalloc3DArray`, `cudaMallocPitch`, `cudaCreateTextureObject`, and `cudaCreateSurfaceObject`
Data transfer	All APIs whose names begin with `cudaMemcpy`, including `cudaMemcpy`, `cudaMemcpy2DToArray`, `cudaMemcpy2DFromArray`, and `cudaMemcpyToSymbol`
Kernel launch	`cudaLaunchKernel`
Free/object destruction	`cudaFree`, `cudaFreeArray`, `cudaDestroyTextureObject`, and `cudaDestroySurfaceObject`

The breakdown is constructed from the CUDA Runtime API and GPU kernel summary reports produced by Nsight Systems. The allocation and deallocation groups include both storage-management operations and the creation or destruction of the opaque access objects required by texture and surface memory. Data transfer includes all rows in the CUDA Runtime API summary whose API names begin with `cudaMemcpy`. Kernel launch represents the host-side invocation performed through `cudaLaunchKernel`, including launches issued by kernel stubs, whereas kernel execution represents the corresponding work performed on the GPU and is obtained from the `cuda_gpu_kern_sum` report. Rows outside the listed host-side groups are excluded from the component-level analysis.

A component materially affects end-to-end performance only when it occupies a meaningful share of the baseline runtime and changes appreciably under AutoTex. Figure 7.12 shows paired stacked bars for Global and AutoTex, with component times normalized to the Global runtime of each benchmark. The Global bars therefore show the baseline runtime distribution, while the corresponding AutoTex bars show how both the individual components and their combined runtime change under the transformation.

Kernel execution was the primary source of the application-level improvement. Across the evaluated benchmarks, its geometric-mean speedup reached 1.68× on the Ada Desktop and 1.77× on the Ampere Laptop. These improvements were larger than the correspond-



(a) Ada Desktop

(b) Ampere Laptop

Figure 7.12: Runtime-component distribution for Global and AutoTex. Each benchmark has paired stacked bars, with Global on the left and AutoTex on the right. All component times are normalized to the selected Global runtime of the same benchmark, so each Global bar sums to 100%.

ing application-level speedups of 1.20× and 1.25×, because kernel execution accounts for only part of the end-to-end runtime. The extent to which a kernel-execution speedup affects the final result therefore depends on the kernel-execution share shown in Figure 7.12.

The host-side components were smaller contributors than kernel execution, but they no longer remained neutral in every category. On the Ada Desktop, allocation/object creation, data transfer, kernel launch, and free/object destruction achieved component speedups of 0.94×, 1.02×, 1.03×, and 0.94×, respectively. On the Ampere Laptop, the corresponding values were 0.88×, 0.97×, 0.91×, and 0.99×. These results indicate that AutoTex introduces modest aggregate host-side overheads, especially for allocation/object creation and, on the Ampere Laptop, kernel launch, while individual workloads exhibit larger component-level changes.

The allocation and object-management results reflect the differences between the memory abstractions (Section 2.7.1) used by the two configurations. Under Global, a linear pointer identifies both the device allocation and the access handle used by the kernel. Under AutoTex, a transformed object may instead require a block-linear CUDA array and a separately created texture or surface object through which the kernel accesses the underlying storage. Deallocation likewise requires both destruction of the opaque object and release of its backing array. These additional operations can increase allocation/object-creation and free/object-destruction time for individual benchmarks; in the geometric mean, the clearest allocation overhead appears on the Ampere Laptop, while free/object destruction slows modestly on the Ada Desktop.

Data-transfer time can also change because the destination representation differs between the two configurations. Global transfers data to or from linear device allocations, whereas AutoTex may use array-specific copies to move data into or out of block-linear CUDA arrays. The observed transfer cost therefore depends on the amount and dimensionality of the transferred data, the source and destination representations, and the transfer APIs selected by the transformed host code. Nevertheless, the geometric-mean transfer speedup remained close to unity on both platforms, indicating that these changes do not impose a consistent transfer penalty across the evaluated workloads.

Kernel-launch time also changes modestly in aggregate. A CUDA launch is lowered to a host-side stub that marshals the execution configuration and kernel arguments before invoking the CUDA runtime (Section 2.8.3). AutoTex modifies these stubs to pass texture or surface object handles and to select the transformed kernels, but does not change the fundamental launch mechanism. The resulting kernel-launch component had geometric-mean speedups of 1.03× on the Ada Desktop and 0.91× on the Ampere Laptop.

The combination of per-kernel contributions, baseline runtime shares, and component-level changes explains the application results. `syr2k`, `syrk`, and `gaussian` obtained large reductions in aggregate kernel-execution time because their dominant kernels benefit substantially from the transformation. Kernel execution also occupies enough of their baseline

application runtime for these reductions to determine the end-to-end result. Workloads whose accelerated kernels make smaller contributions, or whose complete runtimes are dominated by data transfer and other host-side activity, preserve a smaller fraction of their per-kernel gains.

The slowdown cases arose from different combinations of these factors. In `gramschmidt`, a dominant transformed kernel became slower, increasing aggregate kernel time before host-side overhead is considered. In `kmeans`, the dominant kernel provided little or no aggregate kernel-time saving, while data transfer and other host-side components slowed down. In `leukocyte` on the Ampere Laptop, most kernel time remained in untransformed kernels, leaving only limited opportunity for AutoTex to compensate for the observed regression. Thus, application performance depends first on which kernels dominate the benchmark and then on how the resulting aggregate kernel-time change combines with the remainder of the runtime.

The difference between the two platforms follows the same relationship. The Ampere Laptop exhibited a larger geometric-mean kernel-execution speedup, 1.77× compared with 1.68×, but also showed larger aggregate slowdowns in allocation/object creation and kernel launch. On the Ada Desktop, the smaller kernel-execution improvement was accompanied by slowdowns in allocation/object creation and free/object destruction, but data transfer and kernel launch improved slightly in the geometric mean. These opposing effects left the two application-level geometric means closer than the kernel-execution speedups alone would suggest.

Together, the end-to-end, per-kernel, and component-level results show that AutoTex improves application performance primarily by reducing the execution time of kernels that make substantial contributions to their benchmarks. The resulting speedup is determined by the weighted combination of individual kernel changes, the share of baseline runtime occupied by aggregate kernel execution, and whether those savings are sufficient to amortize the memory- and object-management costs required by block-linear storage.

Chapter 8
Limitations and Future Work

The preceding chapters establish the applicability, implementation, and performance characteristics of AutoTex, while also revealing several constraints that limit its current scope or leave opportunities for further optimization. Some of these constraints arise from the CUDA mechanisms used to obtain and access block-linear storage and are therefore inherent to the proposed layout-mapping approach. Others result from the scope and precision of the current compiler analyses and transformations and may be relaxed through extensions to the framework. The evaluation further shows that, even when a transformation is legal and applicable, additional optimization decisions could improve how and when block-linear storage is employed.

This chapter discusses these limitations and the corresponding directions for future work. Section 8.1 examines constraints imposed by the underlying layout-mapping mechanism, including CUDA-array extent limits and compatibility with specialized memory-access mechanisms. Section 8.2 discusses limitations of the current compiler framework, including provenance analysis, access reconstruction, supported data types, interprocedural device-code analysis, intra-warp access-pattern reasoning, and irregular accesses. Finally, Section 8.3 considers opportunities to improve transformation profitability through performance-aware candidate selection, texture-versus-surface interface selection, and vectorized texture accesses.

8.1 Limitations of the Layout-Mapping Mechanism

AutoTex relies on CUDA arrays to obtain the hardware-managed block-linear physical layout (Chapter 5). This design allows AutoTex to improve intra-warp cache utilization without explicitly computing block-linear addresses or introducing software-managed layout conversions. The same design also inherits constraints imposed by CUDA arrays and by the hardware interfaces through which block-linear storage is accessed. These constraints are inherent to the underlying layout-mapping mechanism and therefore cannot be removed solely by extending the compiler analyses or transformations (Chapter 6). This section discusses the resulting limitations, focusing on the supported extent of block-linear allocations and their compatibility with specialized GPU memory-access mechanisms.

8.1.1 Texture Dimensionality and Extent

The applicability of AutoTex is constrained by the maximum dimensions supported by CUDA arrays. Although linear global-memory allocations are principally limited by the available device memory and address space, CUDA imposes explicit bounds on the extent of arrays used as block-linear storage. On the evaluated platforms, a two-dimensional CUDA array can contain at most 131,072 × 65,536 elements, while each dimension of a three-dimensional CUDA array is limited to 16,384 elements (Table 7.1). Consequently, a multidimensional object that exceeds the corresponding bound cannot be mapped directly to the block-linear representation used by AutoTex, even when its access pattern would otherwise make it a suitable transformation candidate.

This restriction is particularly relevant because the limit applies to the extent of each dimension rather than only to the total allocation size. An object may therefore occupy a feasible amount of device memory while still exceeding the CUDA-array limit along one dimension. In such a case, AutoTex cannot preserve the logical shape of the object in a single CUDA array and must leave the allocation in linear global memory. The restriction thus limits the range of problem sizes to which the proposed layout-mapping mechanism can be applied and, in the present evaluation, also constrains the problem sizes that can be used to assess its performance.

A possible extension is to partition an oversized logical object across multiple CUDA arrays and translate each logical access to the appropriate array and local coordinate. Such partitioning could relax the extent restriction without abandoning the block-linear representation. However, it would introduce additional address-selection logic, require multiple texture or surface objects, and potentially alter both the memory-access cost and the physical locality that the transformation is intended to improve. Determining suitable partitioning strategies and performance tradeoffs is therefore left for future work.

8.1.2 Specialized Memory-Access Compatibility

The block-linear representation used by AutoTex is accessed through texture or surface objects rather than through ordinary global-memory pointers. Although this representation enables hardware-managed address translation for multidimensional accesses, it is not directly compatible with specialized GPU mechanisms whose interfaces operate on linear global-memory addresses or require specific memory representations. As a result, an object mapped to a CUDA array by AutoTex cannot necessarily be consumed directly by such mechanisms without an additional representation change.

This limitation is particularly relevant to modern GPU workloads that rely on specialized data-movement and compute paths. For example, tensor-core computations are commonly supplied through memory-access sequences that stage data from global memory into registers or shared memory, while newer architectures additionally provide mecha-

nisms such as the TMA to transfer multidimensional regions between global and shared memory (Section 2.1.1). These mechanisms operate through their own addressing and transfer interfaces and do not directly consume texture or surface objects. Mapping an input object exclusively to the CUDA-array representation used by AutoTex can therefore prevent the same object from participating directly in these access paths.

The resulting restriction limits the composability of the proposed layout-mapping mechanism with workloads whose performance depends heavily on specialized memory movement and tensor-core execution, including many contemporary machine-learning kernels. Supporting such workloads would require either retaining an additional linear representation of the data or introducing explicit conversion between linear and block-linear representations. Both approaches would incur additional storage, data movement, or synchronization costs and could offset the locality benefits of the block-linear layout. This compatibility constraint therefore arises from the memory abstraction used by the layout-mapping mechanism rather than from the compiler framework itself (Chapter 6).

8.2 Limitations of the Compiler Framework

In addition to the constraints imposed by the underlying layout-mapping mechanism, the current implementation of AutoTex is limited by the scope and precision of its compiler analyses and transformations. These limitations do not arise inherently from the use of block-linear storage and may therefore be relaxed through extensions to the compiler framework (Chapter 6). In particular, the current implementation assumes that the provenance of transformed allocations can be recovered within a single translation unit, supports only a restricted set of reconstructable accesses and data types, and relies on simplified models when reasoning about device-function calls, intra-warp access distributions, and irregular memory accesses. The following subsections describe these limitations and outline corresponding directions for future work.

8.2.1 Whole-Program Provenance Analysis

Transforming an object from linear global memory to CUDA-array storage requires more than rewriting the individual kernel accesses identified as optimization candidates. Every host- and device-side use that depends on the representation of the same allocation must remain consistent with the selected memory type. AutoTex therefore tracks the provenance of kernel arguments back to their host-side allocations and groups arguments that originate from the same allocation (Section 6.1.1). When one argument in such a provenance group is transformed, the other uses of the same allocation must be considered together because they share the same underlying representation.

The current provenance analysis operates within a single translation unit. It can therefore follow allocations, pointer propagation, kernel-launch arguments, and associated memory-management operations only when the relevant code is available to the same compilation pipeline. If a pointer is passed through a function or module whose implementation is compiled separately, the analysis cannot in general determine whether the resulting value refers to an allocation that has been mapped to block-linear storage. Similarly, uses of an allocation that are hidden behind separately compiled host code cannot be rewritten consistently. AutoTex must therefore conservatively restrict transformation to objects whose relevant provenance can be established within the available translation unit.

This restriction limits the applicability of the current framework to programs whose memory allocation, propagation, and kernel-launch logic are sufficiently visible during compilation. In larger CUDA applications, however, these operations may be distributed across multiple source files, libraries, or abstraction layers. Extending AutoTex to such programs requires provenance information to be propagated across translation-unit boundaries so that all representation-dependent uses of a transformed allocation can be identified and handled consistently.

Future work could address this limitation through whole-program or link-time analysis. For example, the compiler could construct interprocedural provenance summaries for separately compiled functions and combine them during link-time optimization, allowing allocation identities and representation requirements to be propagated across module boundaries. Such an extension would preserve the provenance-group abstraction used by the current framework while broadening the scope over which groups can be recovered. More generally, a whole-program provenance analysis would allow AutoTex to reason about transformed allocations across larger software components without requiring the relevant host-side operations to reside in a single translation unit.

8.2.2 Access Reconstruction and Delinearization

AutoTex relies on access reconstruction to recover the logical multidimensional coordinates associated with global-memory accesses before determining whether an object is a suitable candidate for block-linear mapping (Section 6.2.2). The current implementation reconstructs accesses most reliably when the kernel interface preserves the multidimensional structure of the underlying object, such as through array-typed parameters whose dimensions remain visible in the LLVM IR. In these cases, the pointer-arithmetic expressions can be mapped back to the corresponding logical indices and reused directly when generating texture or surface accesses.

This approach is less effective for CUDA code that represents multidimensional objects through flattened pointers and explicitly linearizes coordinates in the source program. Af-

ter compilation, an access such as $A[i][j]$ may instead appear as a one-dimensional offset of the form $i \times N + j$, where the original dimensional structure is no longer encoded in the pointer type. Recovering the corresponding coordinates then requires the compiler to infer both the dimensional extents and the arithmetic relationship between the flattened offset and the original logical indices. The current implementation does not generally perform this delinearization and therefore cannot transform accesses whose multidimensional structure cannot be recovered directly from the available IR.

This limitation reduces the applicability of AutoTex to CUDA programs that expose sufficient structural information for access reconstruction. In practice, flattened pointer arithmetic is common in hand-written CUDA kernels and may also be introduced by source-to-source transformation or library interfaces. As observed in the evaluation, some otherwise relevant kernels must therefore be excluded when their accesses cannot be reconstructed with sufficient precision (Section 7.1.2).

Future work could extend the reconstruction analysis with delinearization techniques that recognize common CUDA indexing idioms and recover multidimensional coordinates from flattened address expressions. Such an analysis could combine symbolic expression matching with loop bounds, allocation sizes, and kernel-launch information to infer candidate dimensions and verify that a linear offset corresponds to a valid multidimensional mapping. More general approaches could also incorporate scalar-evolution or affine-analysis techniques to reason about loop-carried index expressions. Improving delinearization would broaden the range of CUDA programs that can be analyzed by AutoTex without changing the underlying layout-mapping mechanism.

8.2.3 Supported Data Types

The current implementation of AutoTex supports only data objects whose elements occupy four bytes. The texture- and surface-access rewriting currently handles only CUDA interfaces and LLVM intrinsics corresponding to four-byte elements (Section 6.2.3). Objects containing larger scalar or aggregate types therefore cannot be transformed, even when their access patterns and provenance otherwise satisfy the requirements of the framework.

This limitation excludes common element types such as double-precision floating-point values and user-defined structures. In these cases, preserving the original program semantics would require the compiler to reconstruct each logical element from one or more values returned by the underlying texture or surface access. The required transformation becomes more involved when the logical element spans multiple independently addressable components or when its in-memory representation contains fields with different sizes or alignments.

Future work could extend AutoTex to support additional scalar element sizes by exploiting the multi-component behavior of texture fetches characterized earlier in this work

(Section 5.2.1). In particular, an eight-byte value could potentially be retrieved as two four-byte components and reconstructed in registers before use. A similar approach may be applicable to other fixed-size element types, provided that the compiler can preserve their layout, alignment, and access semantics. Supporting arbitrary aggregate types would require more general decomposition and reconstruction of each element and may therefore require additional type- and layout-aware analysis.

Extending the supported data types would broaden the applicability of AutoTex without changing the underlying candidate-selection or provenance analyses. However, the additional instructions required to assemble or disassemble larger elements may introduce execution overhead, and their performance impact would need to be evaluated before such transformations are enabled generally.

8.2.4 Interprocedural Device-Code Analysis

The current implementation of AutoTex analyzes and rewrites memory accesses directly contained in CUDA kernels. Accesses performed within separately defined device functions are not considered for transformation, even when the accessed object is passed from a kernel argument that would otherwise be eligible for block-linear mapping. Consequently, a candidate object cannot be transformed when its relevant accesses cannot be fully analyzed and rewritten within the kernel itself.

Supporting device functions requires interprocedural analysis across the device-side call graph. In particular, AutoTex must propagate the provenance and logical multidimensional structure of kernel arguments through function parameters so that accesses in callees can be associated with the corresponding objects. The intra-warp access-distribution analysis must likewise propagate divergence information interprocedurally across function boundaries so that index expressions computed within device functions can be classified with respect to the calling kernel's thread indices. The transformation must then propagate the selected texture or surface access handle through the call chain and rewrite accesses in each affected callee consistently. These requirements extend both the intra-warp access-distribution analysis and the transformation beyond individual kernel boundaries (Sections 6.2.2 and 6.2.4).

Future work could extend AutoTex with interprocedural summaries that describe how device functions propagate pointer arguments and derive memory-access expressions from them. Such summaries could allow the intra-warp access-distribution analysis to propagate reconstructed access information and divergence properties between callers and callees, enabling candidate classification across device-function boundaries. Once an object is selected for transformation, the corresponding function signatures and call sites could then be rewritten to propagate texture or surface handles through the device-side call graph. This extension would allow AutoTex to support CUDA programs whose

memory-access logic is distributed across `__global__` and `__device__` functions rather than expressed entirely within individual kernels.

8.2.5 Intra-Warp Access-Pattern Analysis

The current intra-warp access-distribution analysis uses a simplified model of how thread-index expressions vary across a warp when identifying accesses with low intra-warp cache utilization (Section 6.2.2). In particular, the analysis models `threadIdx.x` as the source of intra-warp divergence but does not account for `threadIdx.y` or `threadIdx.z`. This model is sufficient when the threads of a warp vary only along the x dimension, but it becomes incomplete when the x-dimension of a thread block contains fewer than 32 threads. In such configurations, a warp can span multiple rows or planes of the thread block, causing `threadIdx.y` or `threadIdx.z` to vary within the same warp.

A more precise analysis could model the joint distribution of all three thread-index dimensions using the thread-block dimensions associated with each kernel launch. Because the launch analysis already identifies kernel launch sites, the corresponding block dimensions could be propagated to the device-side analysis and used to determine the values of `threadIdx.x`, `threadIdx.y`, and `threadIdx.z` for the lanes of each warp. This information would allow AutoTex to identify low-utilization access patterns that depend on the y or z thread dimensions and avoid relying on the assumption that intra-warp variation occurs exclusively along x.

The current analysis is also conservative when thread indices undergo arithmetic before being used in an access expression. Divergence indicates that an expression may differ among threads, but does not by itself describe the distribution of the resulting values. For example, arithmetic on `threadIdx.x` may change the stride, range, or grouping of the coordinates generated by a warp and therefore materially change the resulting cache-sector utilization. Without reasoning about these values, AutoTex cannot precisely characterize such access patterns and may fail to identify accesses for which block-linear mapping would be beneficial.

Future work could augment the divergence analysis with value-range or symbolic reasoning to track how arithmetic operations transform thread-index values across a warp. Combined with the recovered thread-block dimensions, such an analysis could derive a more precise approximation of the coordinates generated by the 32 lanes and use their resulting intra-warp distribution directly when estimating cache utilization. This extension would broaden the set of access patterns that AutoTex can identify while reducing the conservatism of the current candidate analysis.

8.2.6 Irregular Access Patterns

The current implementation of AutoTex relies on statically recoverable access expressions to determine whether a global-memory object exhibits low intra-warp cache utilization (Section 6.2.2). This requirement limits the framework when memory coordinates depend on values that cannot be resolved sufficiently at compile time. Examples include indirect indexing through another array, pointer chasing, and other data-dependent address computations. Although such accesses may still exhibit regular or locality-preserving behavior at runtime, their intra-warp distributions cannot generally be inferred from the static expressions available to the current analysis.

This limitation is particularly relevant because irregularity in the source-level expression does not necessarily imply irregularity in the addresses generated by a warp. An index array, for example, may contain values that form a regular stride or remain confined to a small multidimensional region for a particular workload. Without information about the runtime values, however, AutoTex cannot determine the resulting cache-sector utilization or establish that mapping the object to block-linear storage is likely to improve intra-warp locality. The current framework therefore restricts candidate identification to accesses whose relevant intra-warp distribution can be characterized statically.

Writable objects introduce an additional consideration because their read and write accesses must remain semantically consistent after transformation. AutoTex supports writable candidates through surface memory when the required access semantics can be established (Section 6.4.2), but irregular read or write locations remain subject to the same limitation: if their intra-warp distributions cannot be characterized, the framework cannot determine whether the object is a suitable candidate for block-linear mapping.

Future work could extend candidate identification with profile-guided or hybrid static–dynamic analysis. Runtime profiling could characterize the intra-warp address distributions of accesses that cannot be resolved statically and provide this information to the compiler when selecting objects for transformation. Alternatively, stronger value-range, dependence, or symbolic analyses could recover useful bounds or structural properties for some indirect accesses without requiring runtime information. Such extensions would broaden the applicability of AutoTex to workloads with data-dependent memory accesses while preserving the current requirement that transformations be applied only when their access behavior can be characterized with sufficient confidence.

8.3 Optimization Opportunities

Beyond relaxing the applicability limitations discussed above, AutoTex could also be extended to make more selective and performance-aware transformation decisions. The current framework primarily determines whether an object is eligible for block-linear mapping

based on its access structure, provenance, and memory semantics, but does not attempt to predict the resulting performance benefit before applying the transformation. The evaluation shows that the effectiveness of block-linear mapping depends on additional factors, including the baseline execution bottleneck, the contribution of transformed kernels to total execution time, and the behavior of other accesses that share the same allocation. This section discusses several opportunities to incorporate such information into the transformation process, including profitability-aware candidate selection, interface selection for read-only data, and vectorized texture accesses.

8.3.1 Performance-Aware Transformation Selection

The current implementation of AutoTex treats transformation eligibility and transformation profitability as separate concerns. Candidate identification determines whether an access exhibits low intra-warp cache utilization and whether the corresponding object satisfies the correctness requirements for block-linear mapping (Chapter 6). Once an eligible provenance group is identified, the framework applies the transformation without estimating whether the resulting reduction in cache-access wavefronts or memory traffic will translate into an execution-time improvement.

The evaluation demonstrates that eligibility alone does not guarantee a speedup. The benefit of improved intra-warp locality depends on the execution bottleneck of the affected kernel and on whether the memory-system behavior improved by AutoTex materially limits its performance (Section 7.3.1). Moreover, transformations are applied at the provenance-group granularity. An allocation selected because of one beneficial access may also be used by other kernels or arguments that do not independently benefit from block-linear storage. These compelled transformations can reduce the aggregate benefit of mapping the allocation even when the original candidate access improves.

A performance-aware selection mechanism could estimate the profitability of each candidate provenance group before transformation. Such a model could incorporate properties already available to the compiler, including the expected change in intra-warp cache utilization, the number and frequency of affected accesses, and the set of compelled uses associated with the same allocation. Profile-guided information could further incorporate baseline execution bottlenecks, kernel execution frequencies, and the relative contribution of each affected kernel to application execution time. Together, these factors could allow AutoTex to prioritize transformations whose expected benefits outweigh the costs imposed on other uses of the same object.

The profitability model could also account for host-side costs introduced by block-linear storage. As shown by the application-level evaluation, changes to allocation, data transfer, object management, and kernel-launch behavior can partially offset kernel-level improvements (Section 7.4). Incorporating these costs would allow AutoTex to select

transformations according to their expected end-to-end benefit rather than kernel performance alone. Such a model would complement the existing legality and candidate analyses by determining not only whether a transformation can be applied, but whether it is likely to be worthwhile.

8.3.2 Texture-versus-Surface Interface Selection

The current implementation of AutoTex selects the CUDA access interface according to the access semantics of each transformed object: read-only objects are accessed through texture objects, whereas writable objects through surface objects (Section 6.4.2). This policy provides a simple correspondence between program semantics and CUDA memory interfaces, but it does not consider whether a read-only object might execute more efficiently through the surface interface.

The evaluation shows that texture and surface accesses to the same block-linear storage can exhibit different performance even for read-only data (Section 7.2.3). Although neither interface consistently outperforms the other across all evaluated kernels, individual cases favor one interface over the other. Because the physical layout remains unchanged, these differences arise from the distinct access paths and processing stages associated with texture and surface instructions rather than from the block-linear organization itself.

Future work could therefore treat interface selection for read-only objects as a performance decision rather than deriving it solely from mutability. A compiler could select between texture and surface access based on static properties of the kernel, architectural characteristics, or profile-guided measurements. Such a selection mechanism could be incorporated into the broader performance-aware transformation process, allowing AutoTex to choose not only whether an object should use block-linear storage, but also which compatible access interface is expected to provide the better performance.

The present evaluation does not establish a general rule for predicting which interface is preferable in a given case. Further characterization of the texture and surface access paths, including their processing overheads and interactions with different execution bottlenecks, would therefore be required before interface selection could be automated reliably.

8.3.3 Vectorized Texture Accesses

The current implementation of AutoTex preserves the access granularity of the original program when rewriting global-memory loads as texture fetches. A scalar load is therefore transformed into a texture access that returns the corresponding scalar value, even though the underlying texture interface can return multiple adjacent components in a single fetch (Section 6.2.4). As a result, the present transformation does not explicitly exploit opportunities to combine neighboring accesses into wider texture fetches.

Future work could identify groups of scalar accesses that refer to adjacent elements and replace them with a single vectorized texture access when their alignment, ordering, and control-flow conditions permit. The returned components could then be distributed to the corresponding uses in the original program. Such a transformation could reduce the number of texture instructions issued for accesses that already exhibit suitable spatial locality and potentially amortize the instruction-processing overhead of the texture path across multiple values.

Vectorization would require additional analysis beyond the current per-access rewriting. In particular, AutoTex would need to determine whether neighboring accesses are executed together, whether they can be combined without changing observable memory semantics, and whether the wider fetch introduces unnecessary data movement. The benefit may also depend on the access pattern and execution bottleneck, because reducing the instruction count does not necessarily improve performance when the kernel is limited elsewhere. Vectorized texture access would therefore be most naturally combined with performance-aware transformation selection rather than applied uniformly.

The multi-component fetch behavior may also provide a mechanism for extending the supported data types discussed earlier in this chapter (Section 8.2.3). For example, two four-byte components could potentially be combined to reconstruct an eight-byte value. Although this use differs from vectorizing independent neighboring scalar accesses, both extensions rely on exposing and exploiting the wider return capability of the texture interface rather than treating every transformed access as an isolated four-byte operation.

Chapter 9
Conclusion

Modern GPU applications frequently operate on logically multidimensional data structures while storing those structures in linear global memory. When the access pattern of a warp does not align with this linear physical organization, the addresses generated by a single memory instruction may span many cache sectors and cache lines, resulting in poor intra-warp cache utilization. Such accesses not only consume fetched cache data inefficiently but also rely more heavily on cross-instruction reuse, making cache effectiveness more sensitive to inter-warp contention. This work investigated whether this mismatch can instead be addressed through the physical organization of memory, without restructuring the computation or explicitly reorganizing data in software. The central observation is that NVIDIA GPUs already provide a hardware-managed block-linear memory layout through CUDA arrays. By mapping eligible multidimensional data objects to this layout, the physical placement of nearby logical elements can be better aligned with the addresses requested by a warp.

Based on this observation, this work proposed the use of texture memory as a layout-mapping mechanism for improving intra-warp cache utilization. Unlike a blocked layout managed in software, the block-linear organization is established and addressed by existing GPU hardware, avoiding software-managed address computation for the transformed layout. For suitable access patterns, the resulting physical organization consolidates the data requested by a warp into fewer cache sectors and cache lines, thereby increasing the fraction of fetched data consumed by the requesting threads. This consolidation can reduce dependence on cross-instruction reuse, improve robustness to inter-warp cache contention, and reduce the number of cache-access wavefronts required to process a memory instruction. The proposed mechanism therefore targets the physical organization underlying inefficient cache access while remaining orthogonal to techniques that instead transform the computation or its access pattern.

To apply this mechanism transparently to existing CUDA programs, this work developed AutoTex, a compiler framework that identifies global-memory objects for which a block-linear layout may be beneficial and transforms their memory representation and accesses accordingly. AutoTex reconstructs the logical multidimensional structure of device-side memory accesses, analyzes their intra-warp access behavior, and tracks object provenance to preserve consistent layout decisions across related accesses. Eligible read-only and writable objects are mapped to texture and surface memory, respectively, while the corresponding device-side accesses and host-side memory-management operations are

transformed automatically. AutoTex therefore introduces hardware-managed block-linear storage without requiring source-level modification of the original CUDA program.

The evaluation across three benchmark suites and two NVIDIA GPU architectures demonstrates that AutoTex can improve kernel performance. Across all 52 transformed kernels, AutoTex achieved speedups with geometric means of 1.27× on the Ada Desktop and 1.28× on the Ampere Laptop, and individual kernels reached speedups of up to 12.80× and 7.99×, respectively. The accompanying architectural characterization shows that these improvements are consistent with the intended effect of the block-linear layout: consolidating warp accesses into fewer cache sectors and cache lines can reduce cache-access wavefront generation and memory-system pressure. The results therefore demonstrate that improving intra-warp cache utilization through physical layout transformation can translate into kernel-level performance gains.

Kernel-level improvements do not translate directly into proportional application-level speedups because the transformed memory representation introduces additional runtime costs. Across the 16 evaluated benchmarks, the kernel-execution component achieved speedups with geometric means of 1.68× on the Ada Desktop and 1.77× on the Ampere Laptop, while the corresponding end-to-end geometric-mean speedups were 1.20× and 1.25×. These results show that the kernel-level benefits of block-linear layout mapping can translate into meaningful application-level acceleration, although the realized improvement depends on whether the reduction in kernel execution time is sufficient to outweigh the associated runtime overhead.

Overall, this work establishes three complementary results. First, it identifies poor intra-warp cache utilization caused by a mismatch between logical multidimensional access structure and linear physical memory organization as a target for memory-layout optimization. Second, it demonstrates that the hardware-managed block-linear organization already provided by NVIDIA GPUs can serve as a practical layout-mapping mechanism for improving this utilization and reducing cache-access wavefront generation without software-managed address translation or additional hardware support. Third, it develops AutoTex as a compiler framework that applies this mechanism transparently to existing CUDA programs. More broadly, these results show that the physical layout of a GPU data object can be treated as a compiler-selectable optimization dimension in its own right. Specialized memory organizations need not be confined to the programming interfaces for which they were originally exposed; they can instead be repurposed by compiler analysis to better match physical data placement to the access structure of GPU computations. AutoTex demonstrates the viability of this approach by improving memory-system behavior and execution performance using existing hardware while remaining transparent to the source program.

References

[1] J. Montrym and H. Moreton, "The GeForce 6800," *IEEE Micro*, vol. 25, no. 2, pp. 41–51, 2005. DOI: 10.1109/MM.2005.37

[2] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008. DOI: 10.1109/MM.2008.31

[3] J. Krüger and R. Westermann, "Linear algebra operators for GPU implementation of numerical algorithms," *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 908–916, 2003. DOI: 10.1145/882262.882363

[4] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," *ACM Transactions on Graphics*, vol. 23, no. 3, pp. 777–786, 2004. DOI: 10.1145/1015706.1015800

[5] NVIDIA. "NVIDIA Ampere GA102 GPU architecture," Accessed: Apr. 7, 2026. [Online]. Available: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf

[6] NVIDIA. "NVIDIA Ada GPU architecture," Accessed: Apr. 7, 2026. [Online]. Available: https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf

[7] K. Zhang et al., "PipeWeave: Synergizing analytical and learning models for unified gpu performance prediction," in *2026 ACM/IEEE 53rd Annual International Symposium on Computer Architecture*, 2026, pp. 1734–1749. DOI: 10.1109/ISCA66397.2026.00126

[8] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*, 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017, ISBN: 0128119055.

[9] R. Soi, R. Yadav, F. Kjolstad, A. Aiken, M. M. Dehnavi, M. Garland, and M. Bauer, "Optimal software pipelining and warp specialization for tensor core GPUs," in *20th USENIX Symposium on Operating Systems Design and Implementation*, 2026, pp. 1875–1890.

[10] M. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966. DOI: 10.1109/PROC.1966.5273

[11] S. Marschner and P. Shirley, *Fundamentals of Computer Graphics, Fourth Edition*, 4th. USA: A. K. Peters, Ltd., 2016, ISBN: 1482229390.

[12] B. Fahs, E. T. Anderson, N. Barrow-Williams, S. Gadre, J. J. McCormack, B. S. Nordquist, N. R. Saxena, and L. V. Shah, "Technique for performing memory access operations via texture hardware," US9697006B2, Jul. 2017.

[13] M. Kenzel, B. Kerbl, D. Schmalstieg, and M. Steinberger, "A high-performance software graphics pipeline architecture for the GPU," *ACM Transactions on Graphics*, vol. 37, no. 4, pp. 1–15, 2018. DOI: 10.1145/3197517.3201374

[14] M. Doggett, "Texture caches," *IEEE Micro*, vol. 32, no. 3, pp. 136–141, 2012. DOI: 10.1109/MM.2012.44

[15] S. J. Heinrich, E. T. Anderson, J. A. Bolz, J. Dunaisky, R. Jandhyala, J. J. McCormack, A. L. Minkin, B. S. Nordquist, and P. Rao, "Load/store operations in texture hardware," US9595075B2, Mar. 2017.

[16] S. S. Muchnick, *Advanced compiler design and implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, ISBN: 1558603204.

[17] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006, ISBN: 0321486811.

[18] S. J. Heinrich, R. Jandhyala, and B.-O. Schneider, "Coalescing texture access and load/store operations," US20150046662A1, Feb. 2015.

[19] NVIDIA Corporation. "CUDA programming guide, release 13.3," Accessed: Jul. 7, 2026. [Online]. Available: `https://docs.nvidia.com/cuda/archive/13.3.0/cuda-programming-guide/index.html`

[20] W. E. Donovan, E. M. Kilgariff, K. M. Abdalla, and J. J. McCormack, "Block linear memory ordering of texture data," US7916149B1, Mar. 2011.

[21] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, *Dissecting the NVIDIA Volta GPU architecture via microbenchmarking*, 2018. arXiv: `1804.06826 [cs.DC]`.

[22] NVIDIA. "2. Profiling guide—Nsight Compute 13.2 documentation," Accessed: May 22, 2026. [Online]. Available: `https://archive.docs.nvidia.com/nsight-compute/2026.1/ProfilingGuide/index.html`

[23] J. Lee, Y. Ha, S. Lee, J. Woo, J. Lee, H. Jang, and Y. Kim, "GCoM: A detailed GPU core model for accurate analytical modeling of modern GPUs," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 424–436. DOI: `10.1145/3470496.3527384`

[24] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008, pp. 1–11. DOI: `10.1109/SC.2008.5214359`

[25] C. Nugteren, G.-J. van den Braak, H. Corporaal, and H. Bal, "A detailed GPU cache model based on reuse distance theory," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture*, 2014, pp. 37–48. DOI: `10.1109/hpca.2014.6835955`

[26] X. Mei and X. Chu, "Dissecting GPU memory hierarchy through microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 72–86, 2017. DOI: `10.1109/TPDS.2016.2549523`

[27] Z. Zhang, K. Cai, Y. Guo, F. Yao, and X. Gao, "Invalidate+Compare: A Timer-Free GPU cache attack primitive," in *33rd USENIX Security Symposium*, 2024, pp. 2101–2118.

[28] NVIDIA Corporation. "Parallel thread execution ISA version 9.2," Accessed: May 15, 2026. [Online]. Available: `https://docs.nvidia.com/cuda/archive/13.2.1/parallel-thread-execution/index.html`

[29] Z. Jin, C. Rocca, J. Kim, H. Kasan, M. Rhu, A. Bakhoda, T. M. Aamodt, and J. Kim, "Uncovering real GPU NoC characteristics: Implications on interconnect architecture," in *Proceedings of the 2024 57th IEEE/ACM International Symposium on Microarchitecture*, 2024, pp. 885–898. DOI: `10.1109/MICRO61859.2024.00070`

[30] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-Sim: An extensible simulation framework for validated GPU modeling," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, 2020, pp. 473–486. DOI: 10.1109/ISCA45697.2020.00047

[31] J. R. Nickolls, B. W. Coon, and M. C. Shebanow, "Instructions for managing a parallel cache hierarchy," US9639479B2, May 2017.

[32] T. M. Aamodt, W. W. L. Fung, and T. G. Rogers, *General-purpose Graphics Processor Architectures*. Morgan & Claypool Publishers, 2018, ISBN: 1627059237.

[33] A. L. Minkin, S. J. Heinrich, S. Rajeshwaran, B. W. Coon, C. McCarver, A. Rajendran, and S. G. Carlton, "Configurable cache for multiple clients," US8595425B2, Nov. 2013.

[34] A. L. Minkin, S. J. Heinrich, R. Selvanesan, C. McCarver, S. G. Carlton, M. Y. Siu, Y. Y. Tang, and R. J. Stoll, "Cache miss processing using a defer/replay mechanism," US8266383B1, Sep. 2012.

[35] M. D. Hill and A. J. Smith, "Evaluating associativity in CPU caches," *IEEE Transactions on Computers*, vol. 38, no. 12, pp. 1612–1630, 1989. DOI: 10.1109/12.40842

[36] Y. Solihin, *Fundamentals of Parallel Multicore Architecture*, 1st. Chapman & Hall/CRC, 2015, ISBN: 1482211181.

[37] Y. Zhong, X. Shen, and C. Ding, "Program locality analysis using reuse distance," *ACM Transactions on Programming Languages and Systems*, vol. 31, no. 6, pp. 1–39, 2009. DOI: 10.1145/1552309.1552310

[38] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 72–83. DOI: 10.1109/MICRO.2012.16

[39] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proceedings of the 8th Annual Symposium on Computer Architecture*, 1981, pp. 81–87. DOI: 10.5555/800052.801868

[40] M. Kiani and A. Rajabzadeh, "Efficient cache performance modeling in GPUs using reuse distance analysis," *ACM Transactions on Architecture and Code Optimization*, vol. 15, no. 4, pp. 1–24, 2018. DOI: 10.1145/3291051

[41] R. Huerta, M. A. Shoushtary, J.-L. Cruz, and A. Gonzalez, "Dissecting and modeling the architecture of modern GPU cores," in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*, 2025, pp. 369–384. DOI: 10.1145/3725843.3756041

[42] T. Tang, X. Yang, and Y. Lin, "Cache miss analysis for GPU programs based on stack distance profile," in *Proceedings of the 2011 31st International Conference on Distributed Computing Systems*, 2011, pp. 623–634. DOI: 10.1109/ICDCS.2011.16

[43] J. Xue and X. Vera, "Efficient and accurate analytical modeling of whole-program data cache behavior," *IEEE Transactions on Computers*, vol. 53, no. 5, pp. 547–566, 2004. DOI: 10.1109/TC.2004.1275296

[44] J.-C. Huang, J. H. Lee, H. Kim, and H.-H. S. Lee, "GPUMech: GPU performance modeling technique based on interval analysis," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 268–279. DOI: 10.1109/MICRO.2014.59

[45] L. Wang, M. Jahre, A. Adileho, and L. Eeckhout, "MDM: The GPU memory divergence model," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2020, pp. 1009–1021. DOI: 10.1109/MICRO50266.2020.00085

[46] G. Smith. "Re: Understanding tensor pipe throughput and throttle stalls." Post 4, NVIDIA Developer Forums, Accessed: Jul. 29, 2026. [Online]. Available: https://forums.developer.nvidia.com/t/understanding-tensor-pipe-throughput-and-throttle-stalls/355572/4

[47] G. Smith. "Re: How does the LSU (load/store unit) execute load/store instructions in the Ampere architecture?" Post 8, NVIDIA Developer Forums, Accessed: Jul. 28, 2026. [Online]. Available: https://forums.developer.nvidia.com/t/how-does-the-lsu-load-store-unit-execute-load-store-instructions-in-the-ampere-architecture/273699/8

[48] G. Smith. "Re: Global load and texture load on LSU traffic." Post 4, NVIDIA Developer Forums, Accessed: Jul. 29, 2026. [Online]. Available: https://forums.developer.nvidia.com/t/global-load-and-texture-load-on-lsu-traffic/351112/4

[49] NVIDIA and University of Illinois. "NVIDIA module 14 – efficient host-device data transfer," Accessed: Apr. 9, 2026. [Online]. Available: https://engineering.purdue.edu/~smidkiff/ece563/NVidiaGPUTeachingToolkit/Mod14DataXfer/Mod14DataXfer.pdf

[50] S. Boehm. "How to optimize a cuda matmul kernel for cublas-like performance: A worklog," Accessed: Jul. 6, 2026. [Online]. Available: https://siboehm.com/articles/22/CUDA-MMM

[51] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, "Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on GPUs," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 528–540. DOI: 10.1145/2749469.2750393

[52] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665

[53] NVIDIA Corporation. "NVIDIA CUDA compiler driver NVCC, release 13.3," Accessed: Jul. 7, 2026. [Online]. Available: https://docs.nvidia.com/cuda/archive/13.3.0/cuda-compiler-driver-nvcc/index.html

[54] J. Wu, A. Belevich, E. Bendersky, M. Heffernan, C. Leary, J. Pienaar, B. Roune, R. Springer, X. Weng, and R. Hundt, "gpucc: An open-source GPGPU compiler," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, 2016, pp. 105–116. DOI: 10.1145/2854038.2854041

[55] LLVM Project. "Compiling CUDA with clang," Accessed: Jul. 7, 2026. [Online]. Available: https://releases.llvm.org/19.1.0/docs/CompileCudaWithLLVM.html

[56] NVIDIA Corporation. "NVIDIA Nsight Systems," NVIDIA Developer, Accessed: Jul. 30, 2026. [Online]. Available: https://developer.nvidia.com/nsight-systems

[57] NVIDIA Corporation. "NVIDIA Nsight Compute," NVIDIA Developer, Accessed: Jul. 30, 2026. [Online]. Available: https://developer.nvidia.com/nsight-compute

[58] NVIDIA Corporation, *NVTX: NVIDIA tools extension library.*

[59] Y. Liang, X. Xie, G. Sun, and D. Chen, "An efficient compiler framework for cache bypassing on GPUs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 10, pp. 1677–1690, 2015. DOI: 10.1109/TCAD.2015.2424962

[60] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-aware warp scheduling," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 99–110. DOI: 10.1145/2540708.2540718

[61] J. Liu, J. Yang, and R. Melhem, "SAWS: Synchronization aware GPGPU warp scheduling for multiple independent warp schedulers," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 383–394. DOI: 10.1145/2830772.2830822

[62] Y. Liu, Z. Yu, L. Eeckhout, V. J. Reddi, Y. Luo, X. Wang, Z. Wang, and C. Xu, "Barrier-aware warp scheduling for throughput processors," in *Proceedings of the 2016 International Conference on Supercomputing*, 2016, pp. 1–12. DOI: 10.1145/2925426.2926267

[63] A. ElTantawy and T. M. Aamodt, "Warp scheduling for fine-grained synchronization," in *2018 IEEE International Symposium on High Performance Computer Architecture*, 2018, pp. 375–388. DOI: 10.1109/HPCA.2018.00040

[64] R. Ausavarungnirun, S. Ghose, O. Kayiran, G. H. Loh, C. R. Das, M. T. Kandemir, and O. Mutlu, "Exploiting inter-warp heterogeneity to improve GPGPU performance," in *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation*, 2015, pp. 25–38. DOI: 10.1109/PACT.2015.38

[65] N. Fauzia, L.-N. Pouchet, and P. Sadayappan, "Characterizing and enhancing global memory data coalescing on GPUs," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2015, pp. 12–22. DOI: 10.1109/CGO.2015.7054183

[66] M. Khan, P. Basu, G. Rudy, M. Hall, C. Chen, and J. Chame, "A script-based autotuning compiler system to generate high-performance CUDA code," *ACM Transactions on Architecture and Code Optimization*, vol. 9, no. 4, pp. 1–25, 2013. DOI: 10.1145/2400682.2400690

[67] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for CUDA," *ACM Transactions on Architecture and Code Optimization*, vol. 9, no. 4, pp. 1–23, 2013. DOI: 10.1145/2400682.2400713

[68] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving cache management policies using dynamic reuse distances," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 389–400. DOI: 10.1109/MICRO.2012.43

[69] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W.-M. Hwu, "Adaptive cache management for energy-efficient GPU computing," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 343–355. DOI: 10.1109/MICRO.2014.11

[70] C. Li, S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou, "Locality-driven dynamic GPU cache bypassing," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 67–77. DOI: 10.1145/2751205.2751237

[71] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry, "The evicted-address filter: A unified mechanism to address both cache pollution and thrashing," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, 2012, pp. 355–366. DOI: 10.1145/2370816.2370868

[72] G. Koo, Y. Oh, W. W. Ro, and M. Annavaram, "Access pattern-aware cache management for improving data utilization in GPU," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 307–319. DOI: 10.1145/3079856.3080239

[73] S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W.-m. W. Hwu, "CUDA-Lite: Reducing GPU programming complexity," in *Languages and Compilers for Parallel Computing*, 2008, pp. 1–15. DOI: 10.1007/978-3-540-89740-8_1

[74] K. Hong, G. Dai, J. Xu, Q. Mao, X. Li, J. Liu, K. Chen, Y. Dong, and Y. Wang, "FlashDecoding++: Faster large language model inference with asynchronization, flat GEMM optimization, and heuristics," in *Proceedings of Machine Learning and Systems*, 2024, pp. 148–161.

[75] M. Bauer, H. Cook, and B. Khailany, "CudaDMA: Optimizing GPU memory bandwidth via warp specialization," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–11. DOI: 10.1145/2063384.2063400

[76] G. Huang, Y. Bai, L. Liu, Y. Wang, B. Yu, Y. Ding, and Y. Xie, "ALCOP: Automatic load-compute pipelining in deep learning compiler for AI-GPUs," in *Proceedings of Machine Learning and Systems*, 2023, pp. 680–694.

[77] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU compiler for memory optimization and parallelism management," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010, pp. 86–97. DOI: 10.1145/1806596.1806606

[78] G. Morton, *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. International Business Machines Company, 1966.

[79] D. S. Wise, J. D. Frens, Y. Gu, and G. A. Alexander, "Language support for Morton-order matrices," in *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, 2001, pp. 24–33. DOI: 10.1145/379539.379559

[80] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," *ACM Transactions on Algorithms*, vol. 8, no. 1, pp. 1–22, 2012. DOI: 10.1145/2071379.2071383

[81] J. Thiyagalingam, O. Beckmann, and P. H. J. Kelly, "Improving the performance of Morton layout by array alignment and loop unrolling," in *Languages and Compilers for Parallel Computing*, 2004, pp. 241–257. DOI: 10.1007/978-3-540-24644-2_16

[82] N. Park, B. Hong, and V. K. Prasanna, "Tiling, block data layout, and memory hierarchy performance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 7, pp. 640–654, 2003. DOI: 10.1109/TPDS.2003.1214317

[83] J. Rosemann, S. Moll, and S. Hack, "An abstract interpretation for SPMD divergence on reducible control flow graphs," *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, pp. 1–31, 2021. DOI: 10.1145/3434312

[84] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to GPU codes," in *2012 Innovative Parallel Computing*, 2012, pp. 1–10. DOI: 10.1109/InPar.2012.6339595

[85] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, 2009, pp. 44–54. DOI: 10.1109/IISWC.2009.5306797

[86] G. Araujo, D. Griebler, D. A. Rockenbach, M. Danelutto, and L. G. Fernandes, "NAS parallel benchmarks with CUDA and beyond," *Software: Practice and Experience*, vol. 53, no. 1, pp. 53–80, 2023. DOI: 10.1002/spe.3056

[87] T. Yuki and L.-N. Pouchet. "PolyBench/C 4.2.1. "[Online]. Available: https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1/

[88] D. H. Bailey et al., "The NAS parallel benchmarks—summary and preliminary results," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, 1991, pp. 158–165. DOI: 10.1145/125826.125925

[89] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009. DOI: 10.1145/1498765.1498785

[90] G. Smith. "Re: Terminology used in Nsight Compute," Stack Overflow, Accessed: Jul. 31, 2026. [Online]. Available: https://stackoverflow.com/a/63430752

[91] R. Crovella. "Re: How to use Nsight Compute profiling results." Post 2, NVIDIA Developer Forums, Accessed: Jul. 31, 2026. [Online]. Available: https://forums.developer.nvidia.com/t/how-to-use-nsight-compute-profiling-results/107899/2

[92] V. Volkov, "Understanding latency hiding on GPUs," Ph.D. dissertation, EECS Department, University of California, Berkeley, Aug. 2016.

[93] G. Smith. "Re: Same SOL for memory and SM throughput." Post 2, NVIDIA Developer Forums, Accessed: Jul. 31, 2026. [Online]. Available: https://forums.developer.nvidia.com/t/same-sol-for-memory-and-sm-throughput/309092/2

Appendix A
Selected Benchmark Problem Sizes

This appendix documents the problem-size selection used in the benchmark evaluation of Chapter 7. Table A.1 reports the selected problem size and the corresponding per-kernel theoretical occupancy, achieved occupancy, and waves per SM on each platform. These profiling metrics are collected using Nsight Compute under the GLOBAL configuration. Table A.2 then summarizes the constraint or policy governing the selected problem size for each benchmark.

PolyBench-ACC problem sizes are selected independently for Ada Desktop and Ampere Laptop because the two platforms differ in device-memory capacity and computational resources. Rodinia uses the same benchmark inputs and runtime arguments on both platforms. For NPB-GPU, all benchmarks use Class C on the Ampere Laptop because it already provides sufficient grid-level parallelism for the principal measured kernels; larger classes would increase execution time without materially changing the evaluated behavior. On the Ada Desktop, IS likewise remains at Class C because it already provides sufficient grid-level parallelism, whereas LU and SP use Class D to provide additional parallelism. NPB classes are defined separately for each benchmark, so the same class designation does not imply the same underlying problem dimensions.

Theoretical occupancy denotes the maximum fraction of an SM's warp capacity that can be resident concurrently, as determined by the thread-block configuration and the kernel's register, shared-memory, warp, and block requirements. Achieved occupancy denotes the average fraction of warp capacity that is active during execution. A difference between the two may arise from insufficient grid size, load imbalance, tail effects, or kernels whose grids contain only a small number of thread blocks.

Waves per SM indicates the amount of grid-level parallelism provided by the launched grid relative to the number of thread blocks that can reside concurrently across the GPU. A value below one means that the grid contains fewer blocks than the GPU could accommodate concurrently for that kernel, whereas a value above one indicates that additional blocks remain after the first resident wave. Waves per SM is therefore used as the primary indicator of whether the selected problem size provides sufficient parallelism.

Some kernels nevertheless report fewer than one wave per SM. In several cases, the kernel's grid structure scales weakly or not at all with the overall benchmark size. A small number of kernels launch only a single thread, causing the reported value to round to 0.00. Increasing the benchmark problem size does not materially increase the parallelism of these kernels.

Table A.1: Selected benchmark problem sizes with per-kernel theoretical and achieved occupancy and waves per SM under GLOBAL.

Suite	Benchmark	Problem size	Kernel	Platform	Occupancy theoretical/achieved	Waves per SM
PolyBench-ACC	adi	$N = 35840$	`adi_kernel1`	Ada Desktop	100 % / 30.45 %	0.28
		$N = 17408$		Ampere Laptop	100 % / 64.79 %	0.71
PolyBench-ACC	adi	$N = 35840$	`adi_kernel3`	Ada Desktop	100 % / 30.02 %	0.28
		$N = 17408$		Ampere Laptop	100 % / 62.07 %	0.71
PolyBench-ACC	adi	$N = 35840$	`adi_kernel4`	Ada Desktop	100 % / 25.13 %	0.28
		$N = 17408$		Ampere Laptop	100 % / 63.84 %	0.71
PolyBench-ACC	adi	$N = 35840$	`adi_kernel6`	Ada Desktop	100 % / 24.38 %	0.28
		$N = 17408$		Ampere Laptop	100 % / 60.62 %	0.71
PolyBench-ACC	atax	$NX = 63488$; $NY = 63488$	`atax_kernel1`	Ada Desktop	100 % / 36.1 %	0.39
		$NX = 24576$; $NY = 24576$		Ampere Laptop	100 % / 70.64 %	1.00
PolyBench-ACC	atax	$NX = 63488$; $NY = 63488$	`atax_kernel2`	Ada Desktop	100 % / 38.49 %	0.39
		$NX = 24576$; $NY = 24576$		Ampere Laptop	100 % / 99.71 %	1.00
PolyBench-ACC	bicg	$NX = 63488$; $NY = 63488$	`bicg_kernel1`	Ada Desktop	100 % / 38.55 %	0.39
		$NX = 24576$; $NY = 24576$		Ampere Laptop	100 % / 99.61 %	1.00
PolyBench-ACC	bicg	$NX = 63488$; $NY = 63488$	`bicg_kernel2`	Ada Desktop	100 % / 37.43 %	0.39
		$NX = 24576$; $NY = 24576$		Ampere Laptop	100 % / 71.58 %	1.00
PolyBench-ACC	gemver	$N = 63488$	`gemver_kernel1`	Ada Desktop	100 % / 71.59 %	9,198.04
		$N = 24576$		Ampere Laptop	100 % / 73.23 %	24,576.00
PolyBench-ACC	gemver	$N = 63488$	`gemver_kernel2`	Ada Desktop	100 % / 28.04 %	0.28
		$N = 24576$		Ampere Laptop	100 % / 99.68 %	1.00

Table A.1: Selected benchmark problem sizes with per-kernel theoretical and achieved occupancy and waves per SM (continued).

Suite	Benchmark	Problem size	Kernel	Platform	Occupancy theoretical/achieved	Waves per SM
PolyBench-ACC	gemver	$N = 63488$	`gemver_kernel3`	Ada Desktop	100 %/ 28.02 %	0.28
		$N = 24576$		Ampere Laptop	100 %/ 71.34 %	1.00
PolyBench-ACC	gesummv	$N = 44032$	`gesummv_kernel`	Ada Desktop	100 %/ 29.09 %	0.30
		$N = 21504$		Ampere Laptop	100 %/ 66.32 %	0.88
PolyBench-ACC	gramschmidt	$NI = 35840$; $NJ = 35840$	`gramschmidt_kernel1`	Ada Desktop	100 %/ 2.09 %	0.00
		$NI = 17408$; $NJ = 17408$		Ampere Laptop	100 %/ 2.09 %	0.01
PolyBench-ACC	gramschmidt	$NI = 35840$; $NJ = 35840$	`gramschmidt_kernel2`	Ada Desktop	100 %/ 22.21 %	0.28
		$NI = 17408$; $NJ = 17408$		Ampere Laptop	100 %/ 52.6 %	0.71
PolyBench-ACC	gramschmidt	$NI = 35840$; $NJ = 35840$	`gramschmidt_kernel3`	Ada Desktop	100 %/ 27.93 %	0.28
		$NI = 17408$; $NJ = 17408$		Ampere Laptop	100 %/ 62.59 %	0.71
PolyBench-ACC	mvt	$N = 63488$	`mvt_kernel1`	Ada Desktop	100 %/ 27.97 %	0.28
		$N = 24576$		Ampere Laptop	100 %/ 70.58 %	1.00
PolyBench-ACC	mvt	$N = 63488$	`mvt_kernel2`	Ada Desktop	100 %/ 28.05 %	0.28
		$N = 24576$		Ampere Laptop	100 %/ 99.58 %	1.00
PolyBench-ACC	syr2k	$NI = 344$; $NJ = 65536$	`syr2k_kernel`	Ada Desktop	100 %/ 62.47 %	1.04
		$NI = 152$; $NJ = 65536$		Ampere Laptop	100 %/ 59.79 %	0.99
PolyBench-ACC	syrk	$NI = 344$; $NJ = 65536$	`syrk_kernel`	Ada Desktop	100 %/ 63.13 %	1.04
		$NI = 152$; $NJ = 65536$		Ampere Laptop	100 %/ 60.9 %	0.99
Rodinia	gaussian	`-s 8192`	`Fan1`	Ada Desktop	50 %/ 5.75 %	0.14
				Ampere Laptop	33.33 %/ 23.09 %	1.00

Table A.1: Selected benchmark problem sizes with per-kernel theoretical and achieved occupancy and waves per SM (continued).

Suite	Benchmark	Problem size	Kernel	Platform	Occupancy theoretical/achieved	Waves per SM
Rodinia	gaussian	`-s 8192`	`Fan2`	Ada Desktop	100 % / 92.49 %	574.88
				Ampere Laptop	100 % / 89.16 %	2,730.67
Rodinia	huffman	`test1024_H2...in`	`pack2`	Ada Desktop	50 % / 2.08 %	0.04
				Ampere Laptop	33.33 % / 8.05 %	0.25
Rodinia	huffman	`test1024_H2...in`	`vlc_encode_kernel_sm64huff`	Ada Desktop	100 % / 82.37 %	2.25
				Ampere Laptop	100 % / 93 %	10.67
Rodinia	kmeans	`65536_34f.txt`	`invert_mapping`	Ada Desktop	100 % / 47.98 %	0.56
				Ampere Laptop	100 % / 86.98 %	2.67
Rodinia	kmeans	`65536_34f.txt`	`kmeansPoint`	Ada Desktop	100 % / 38.81 %	0.56
				Ampere Laptop	100 % / 70.69 %	2.67
Rodinia	leukocyte	`testfile.avi`, 5 frames	`GICOV_kernel`	Ada Desktop	100 % / 91.85 %	0.98
				Ampere Laptop	100 % / 83.34 %	4.66
Rodinia	leukocyte	`testfile.avi`, 5 frames	`dilate_kernel`	Ada Desktop	100 % / 81.43 %	1.31
				Ampere Laptop	100 % / 78.64 %	6.22
NPB-GPU	IS	Class C	`full_verify_gpu_kernel_1`	Ada Desktop	100 % / 82.21 %	1,149.75
				Ampere Laptop	100 % / 79.94 %	5,461.33
NPB-GPU	IS	Class C	`full_verify_gpu_kernel_2`	Ada Desktop	100 % / 91.83 %	1,149.75
				Ampere Laptop	100 % / 92.22 %	5,461.33
NPB-GPU	LU	Class D	`error_gpu_kernel`	Ada Desktop	50 % / 49.67 %	90.37
		Class C		Ampere Laptop	33.33 % / 32.31 %	100.00

Table A.1: Selected benchmark problem sizes with per-kernel theoretical and achieved occupancy and waves per SM (continued).

Suite	Benchmark	Problem size	Kernel	Platform	Occupancy theoretical/achieved	Waves per SM
NPB-GPU	LU	Class D	`jacld_blts_gpu_kernel`	Ada Desktop	50 %/ 2.08 %	0.00
		Class C		Ampere Laptop	33.33 %/ 2.08 %	0.00
NPB-GPU	LU	Class D	`jacu_buts_gpu_kernel`	Ada Desktop	50 %/ 2.08 %	0.00
		Class C		Ampere Laptop	33.33 %/ 2.08 %	0.00
NPB-GPU	LU	Class D	`pintgr_gpu_kernel_1`	Ada Desktop	100 %/ 82.17 %	91.26
		Class C		Ampere Laptop	66.67 %/ 58.44 %	102.52
NPB-GPU	LU	Class D	`pintgr_gpu_kernel_2`	Ada Desktop	100 %/ 84.66 %	91.26
		Class C		Ampere Laptop	66.67 %/ 59.28 %	102.52
NPB-GPU	LU	Class D	`pintgr_gpu_kernel_3`	Ada Desktop	100 %/ 88.43 %	91.26
		Class C		Ampere Laptop	66.67 %/ 59.28 %	102.52
NPB-GPU	LU	Class D	`rhs_gpu_kernel_1`	Ada Desktop	50 %/ 49.2 %	1,163.61
		Class C		Ampere Laptop	33.33 %/ 31.69 %	518.99
NPB-GPU	LU	Class D	`rhs_gpu_kernel_2`	Ada Desktop	50 %/ 49.77 %	90.37
		Class C		Ampere Laptop	33.33 %/ 33.25 %	100.00
NPB-GPU	LU	Class D	`rhs_gpu_kernel_3`	Ada Desktop	50 %/ 49.9 %	90.37
		Class C		Ampere Laptop	33.33 %/ 33.26 %	100.00
NPB-GPU	LU	Class D	`rhs_gpu_kernel_4`	Ada Desktop	50 %/ 49.9 %	90.37
		Class C		Ampere Laptop	33.33 %/ 33.25 %	100.00
NPB-GPU	LU	Class D	`setbv_gpu_kernel_1`	Ada Desktop	50 %/ 42.99 %	2.85
		Class C		Ampere Laptop	33.33 %/ 29.36 %	3.21

Table A.1: Selected benchmark problem sizes with per-kernel theoretical and achieved occupancy and waves per SM (continued).

Suite	Benchmark	Problem size	Kernel	Platform	Occupancy theoretical/achieved	Waves per SM
NPB-GPU	LU	Class D	`setbv_gpu_kernel_2`	Ada Desktop	50 %/ 42.54 %	2.85
		Class C		Ampere Laptop	33.33 %/ 29.46 %	3.21
NPB-GPU	LU	Class D	`setbv_gpu_kernel_3`	Ada Desktop	50 %/ 42 %	2.85
		Class C		Ampere Laptop	33.33 %/ 29.38 %	3.21
NPB-GPU	LU	Class D	`setiv_gpu_kernel`	Ada Desktop	54.17 %/ 49.48 %	1,084.45
		Class C		Ampere Laptop	72.92 %/ 67.95 %	228.57
NPB-GPU	LU	Class D	`ssor_gpu_kernel_2`	Ada Desktop	66.67 %/ 36.32 %	2,168.89
		Class C		Ampere Laptop	66.67 %/ 33.89 %	1,600.00
NPB-GPU	SP	Class D	`add_gpu_kernel`	Ada Desktop	50 %/ 49.12 %	1,163.61
		Class C		Ampere Laptop	33.33 %/ 31.98 %	518.99
NPB-GPU	SP	Class D	`compute_rhs_gpu_kernel_1`	Ada Desktop	50 %/ 48.86 %	1,163.61
		Class C		Ampere Laptop	33.33 %/ 31.21 %	518.99
NPB-GPU	SP	Class D	`compute_rhs_gpu_kernel_2`	Ada Desktop	27.08 %/ 24.67 %	2,190.32
		Class C		Ampere Laptop	25 %/ 18.86 %	820.12
NPB-GPU	SP	Class D	`rhs_norm_gpu_kernel_1`	Ada Desktop	66.67 %/ 51.11 %	2,190.32
		Class C		Ampere Laptop	66.67 %/ 47.7 %	1,640.25
NPB-GPU	SP	Class D	`txinvr_gpu_kernel`	Ada Desktop	50 %/ 49.18 %	1,163.61
		Class C		Ampere Laptop	33.33 %/ 31.59 %	518.99
NPB-GPU	SP	Class D	`x_solve_gpu_kernel`	Ada Desktop	27.08 %/ 26.29 %	5.37
		Class C		Ampere Laptop	37.5 %/ 27.39 %	3.38

Table A.1: Selected benchmark problem sizes with per-kernel theoretical and achieved occupancy and waves per SM (continued).

Suite	Benchmark	Problem size	Kernel	Platform	Occupancy theoretical/achieved	Waves per SM
NPB-GPU	SP	Class D	`y_solve_gpu_kernel`	Ada Desktop	27.08 %/ 26.74 %	5.37
		Class C		Ampere Laptop	37.5 %/ 29.83 %	3.38
NPB-GPU	SP	Class D	`z_solve_gpu_kernel`	Ada Desktop	27.08 %/ 25.78 %	5.37
		Class C		Ampere Laptop	37.5 %/ 29.4 %	3.38

Table A.2: Problem-size selection constraints and policies for the evaluated benchmarks.

Suite	Benchmarks	Size-selection constraint or policy
PolyBench-ACC	adi, atax, bicg, gemver, gesummv, gramschmidt, mvt	device-memory capacity
PolyBench-ACC	syr2k, syrk	Texture dimensions and device-memory capacity
Rodinia	gaussian, leukocyte	Available suite input and runtime argument
Rodinia	huffman	Available suite input and device-memory capacity
Rodinia	kmeans	Available suite input and texture dimensions
NPB-GPU	IS, LU, SP	Sufficient grid-level parallelism