Towards Effortless High-Performance Kernel Development for LLM Workloads

Jinqi(Kathryn) Chen

CMU-CS-25-130

August 2025

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Tianqi Chen, Chair
Zhihao Jia

Submitted in partial fulfillment of the requirements for the degree of Masters degree in Computer Science.

Keywords: Large Language Models, LLM Inference, Machine Learning Compiler, High-Performance Computing, Distributed Systems, Programming Model

To my dear friends and family.

Abstract

Recent advances in large language models (LLMs) have pushed GPU hardware to its limits, requiring highly optimized kernels for compute- and bandwidth-intensive operations such as matrix multiplication, attention, and inter-GPU communication. However, achieving state-of-the-art efficiency often demands deep low-level expertise, slowing development and limiting accessibility.

This thesis presents TIR+, a multi-level compiler framework that unifies high-level productivity and low-level optimization within a single compilation and runtime infrastructure. TIR+ spans from a Python-based tiling DSL, enabling rapid kernel prototyping, to a hardware-centric intermediate representation (IR), offering fine-grained control over memory, parallelism, and specialized instructions. Between these extremes, it provides optimized tensor libraries and reusable primitives. Crucially, TIR+ is distributed-aware, supporting multi-GPU execution with built-in communication management and compute–communication overlap. We demonstrate the capability of TIR+ through key LLM kernels, such as GEMM, attention, and fused compute–communication kernels. Among these cases, TIR+ matches the state-of-the-art performance with significantly less development effort than hand-tuned CUDA, demonstrating a unified and scalable path toward hardware-aware kernel optimization for current and future AI workloads.

Acknowledgments

I owe my deepest gratitude to my advisor, Tianqi Chen, whose guidance has shaped not only my research but also who I am as a person. From the very beginning, he encouraged me to follow the path I truly wanted, even when it was uncertain or ambitious, and he believed in me more than I believed in myself. He showed me what research truly means: not merely solving problems, but asking the right questions, remaining curious and rigorous, and daring to pursue bold ideas. Beyond academic research, he taught me lessons that I will carry with me for the rest of my life. I feel incredibly fortunate to have met him, to have worked under his guidance, and to have grown both intellectually and personally through this journey.

I am also grateful to Zhihao Jia, for his thoughtful feedback on this thesis and on our project, which helped sharpen and strengthen my work. I would also like to thank Todd Mowry, whose class on Optimizing Compilers for Modern Architectures was one of the most enjoyable classes I've ever taken.

To my wonderful labmates — Bohan Hou, Hongyi Jin, Ruihang Lai, Guanjie Wang, Derrick Yang — thank you for your encouragement, support, and mentorship. You made long days lighter and reminded me that research is never a solitary effort. I am equally thankful for external collaborators — Zihao Ye, Vinod Grover, Yaxing Cai — who brought insights to this project. I also want to thank the Apache TVM, MLC-LLM, and FlashInfer open-source communities for making this research possible.

My deepest gratitude goes to my family and close friends. To my family — thank you for your love, patience, and belief in me, and your support has been the foundation of everything I have achieved. To my dear friends, Coco Dong and Tony Tao, thank you for standing by me through the hardest times.

Finally, I would like to acknowledge Rezz, the Canadian DJ and producer. Her dedication to her art have always reminded me to stay true to my own.

Contents

1 Introduction **1**

2 Background **3**
2.1 Challenges in High-Performance Kernel Development . . . 3
2.2 Existing DSL Compiler and Template-based Solutions . . . 3
2.3 Key LLM Workloads . . . 4

3 System Architecture and Design **5**
3.1 Overall Compiler Stack Architecture . . . 5
3.2 Hardware Abstraction Layer . . . 6
3.3 Operator Library Layer . . . 8

4 Supported Infrastructure **9**
4.1 TIR+ Intermediate Representation (IR) Structure . . . 9
4.1.1 Programming Granularity and Execution Scope . . . 9
4.1.2 Tensor Layout . . . 10
4.1.3 Parser/Printer, FFI, Transformations . . . 12
4.2 First Class Support for Distributed Execution . . . 13

5 Operator Scheduler **15**
5.1 Overview of Operator Scheduling and Lowering . . . 15
5.2 Event Tensor Abstraction . . . 15
5.2.1 Motivation . . . 15
5.2.2 Implementation . . . 17
5.3 Operator Scheduling Examples . . . 18

6 Experimental Evaluation **21**
6.1 Experimental Setup and Benchmark Methodology . . . 21
6.2 Kernel Performance . . . 21
6.2.1 Memory-Bound Kernels . . . 21
6.2.2 GEMM Kernels . . . 22
6.2.3 Attention Kernels . . . 23
6.2.4 Communication-Computation Overlap Kernels . . . 24
6.3 Discussion . . . 26

7 Future Directions **27**

8 Conclusion **29**

Bibliography **31**

List of Figures

6.1 Speedup of TIR+ vs. FlashInfer on RMSNorm (Single B200 GPU) 22
6.2 Speedup of TIR+ vs. cuBLAS on GEMM (Single B200 GPU) 23
6.3 Speedup of TIR+ vs. FlashInfer on BatchDecode (Single B200 GPU) 24
6.4 Speedup of TIR+ vs. Baselines on AllGather+GEMM (8×B200 GPUs) 25
6.5 Speedup of TIR+ vs. Baselines on GEMM+ReduceScatter (8×B200 GPUs) . . . 25
6.6 Timeline of Execution for Overlapped GEMM+ReduceScatter 26

List of Tables

4.1 Host and device-side NVSHMEM APIs integrated into TIR+. 13

Chapter 1

Introduction

Recent advances in large language models (LLMs) have raised unprecedented demands on GPU computing. Training and inference now involve trillions of floating-point operations, massive data movement through hierarchical memory systems, and execution across multiple accelerators connected by high-speed interconnects. To meet these demands, developers must rely on highly optimized kernels that saturate tensor cores, maximize memory bandwidth, and minimize communication overhead. Yet achieving such performance often requires low-level expertise in CUDA or PTX, careful orchestration of warps and memory layouts, and substantial manual tuning. While this can yield kernels that approach peak hardware efficiency, the engineering cost is immense, and every new model or hardware generation exacerbates the burden.

Meanwhile, high-level approaches have emerged to make kernel development more productive. These systems abstract away many hardware details and allow developers to express tiled computation in a concise manner. However, they often limit the degree of control over scheduling, which can hinder performance in complex or irregular workloads. The gap between productivity and performance is particularly evident in distributed execution, where overlapping inter-GPU communication with computation is crucial for scaling efficiency but rarely integrated into the kernel programming model itself. As LLMs continue to grow, the lack of a unified solution that reconciles expressiveness and performance has become a fundamental limitation.

This thesis presents **TIR+**, a multi-level compiler framework designed to bridge this gap. TIR+ combines a hardware-level intermediate representation (IR), which exposes fine-grained control over memory, parallelism, and specialized instructions, with an operator library layer that provides reusable primitives. These primitives can be explicitly scheduled by developers or left partially specified for the compiler to infer, balancing control with convenience. A central design principle is providing first-class distributed support: TIR+ integrates NVSHMEM primitives directly into the IR, enabling fused kernels that pipeline communication with computation at tile granularity.

The main contributions of this thesis are:

- **A multi-level compiler stack** that unifies high-level productivity with low-level control, enabling developers to work at the most natural level of abstraction while retaining performance portability.
- **An operator library layer** with reusable primitives, designed to support both fully speci-

fied and partially specified schedules.

- **A compositional IR design** with explicit abstractions for execution scopes, tensor layouts, and synchronization events, making complex scheduling analyzable and portable.
- **First-class distributed support** through NVSHMEM integration, allowing fused communication–computation kernels at tile granularity.
- **An evaluation on key LLM kernels** including GEMM, attention, normalization, and fused communication+compute, which shows that TIR+ achieves state-of-the-art performance with significantly less development effort compared to hand-tuned CUDA.

Together, these contributions establish TIR+ as a unified path toward effortless high-performance kernel development for LLM workloads, offering both the accessibility of high-level abstractions and the efficiency of low-level control.

Chapter 2

Background

2.1 Challenges in High-Performance Kernel Development

Developing high-performance GPU kernels has long required substantial manual effort. At the low level, CUDA and PTX expose full control over threads, memory hierarchy, and synchronization strategies, but achieving peak efficiency demands deep hardware knowledge and careful tuning. Developers must manage coalesced memory accesses, shared memory reuse, warp-level coordination, and tensor-core instructions — details that are highly error-prone and architecture-dependent. Hand-tuned kernels can reach near-peak performance, but the engineering cost is high, and optimizations often need to be adapted in each new GPU generation.

In contrast, high-level DSLs such as Triton[10] reduce this burden by providing a productive tile-based programming model in Python. Many low-level optimizations are handled automatically, making it feasible to prototype custom kernels with far less effort. However, this abstraction comes with trade-offs: Triton hides certain scheduling decisions from the user, provides limited control at warp or multi-SM scopes, and may fall short of expert-tuned performance in complex or irregular workloads. This tension between productivity and performance remains central to GPU kernel development.

2.2 Existing DSL Compiler and Template-based Solutions

To address the challenges of GPU kernel development, researchers have proposed a range of compilers, DSLs, and libraries. Each strikes a different balance between abstraction and control, exposing distinct trade-offs between usability and peak performance. Below, we review representative solutions and highlight their design trade-offs, motivating the unified approach of this thesis.

On the compiler side, Triton[10] introduces a Python-based DSL for tile-level programming. It automates many block-local optimizations and enables concise kernels with performance close to cuBLAS. However, it leaves block partitioning and cross-block coordination to the user, limiting expressiveness beyond single-block optimization. TileLang[11] takes a similar approach but separates algorithm description from scheduling more explicitly. Kernels are expressed as tiled computations with scheduling directives as annotations. This improves clarity and intro-

duces stronger support for pipelining and thread hierarchy, though its generality across irregular or distributed workloads is still uncertain.

On the templates side, CUTLASS[9] provides C++ templates for hierarchical tiling and warp-level primitives, primarily for GEMM and convolution. It achieves performance close to expert-tuned code, but the effective use requires deep understanding of its abstractions. ThunderKittens[7] is a C++ embedded DSL organized around GPU-oriented abstractions such as fixed-size tiles and overlap patterns. This design maps efficiently to tensor cores and achieves strong results on GEMM and attention. However, its reliance on fixed patterns limits flexibility, and tuning remains necessary at the C++ level.

Taken together, existing systems span a spectrum: some emphasize productivity through high-level abstractions, while others stress efficiency by exposing low-level building blocks. Each approach delivers value in specific contexts, but none fully resolves the broader challenge of reconciling ease of use with consistently high performance across diverse workloads. This motivates the unified framework of TIR+, which integrates tile-level abstractions, library-like components, and hardware-specific primitives within a single system.

2.3 Key LLM Workloads

Large Language Model (LLM) workloads bring together some of the most demanding kernel challenges in modern AI, which has spurred intense interest in optimizing a variety of GPU operations. They amplify these challenges by combining compute-intensive and memory-bound operations with distributed execution.

In LLM training and inference settings, GEMM remains the core building block, dominating FLOPs in attention projections and feed-forward layers; it requires saturating tensor cores for maximum throughput. Attention kernels add complexity by mixing matrix multiplications, softmax, and cache-aware memory access, making them highly sensitive to bandwidth and latency. Normalization layers such as RMSNorm are memory-bound yet ubiquitous, becoming bottlenecks in both training and inference. Beyond single-device kernels, fused communication-compute kernels (e.g., GEMM + reduce-scatter, all-gather + GEMM) are critical in distributed training, where overlapping data movement with computation determines scaling efficiency.

These kernels, spanning GEMM, attention, normalization, and communication overlap, are representative of the performance-critical operations in LLMs. They form the focus of our experimental evaluations in Section 6, allowing us to assess whether a unified compiler can deliver competitive performance across diverse regimes.

Chapter 3

System Architecture and Design

3.1 Overall Compiler Stack Architecture

The design of TIR+ is motivated by the dual challenges of expressiveness and performance in modern machine learning workloads. Kernel developers are often forced to navigate a spectrum of abstractions: on one end, low-level programming interfaces such as CUDA/PTX or hardware-native ISAs provide maximal control at the cost of a high engineering burden; on the other end, high-level tile-based DSLs such as Triton enable rapid development but frequently fall short in extracting the full performance potential of the hardware. TIR+ aims to unify the existing fragmented landscape by offering a multi-layered compiler stack that exposes multiple programming models within a single framework.

The compiler is built on top of TVM, leveraging its mature infrastructure for code generation and optimization, while extending its design to address the aforementioned challenges we identified. Importantly, TIR+ supports both NVIDIA GPUs and AWS Trainium, demonstrating its portability across heterogeneous accelerators. Our system design is guided by three principles:

- **T0**: Compiler optimizations that universally improve performance should be seamlessly incorporated into the programming model.
- **T1**: The trade-off between performance and engineering effort is central to kernel development, and the programming model must allow developers to make informed choices along this spectrum.
- **T2**: Distributed execution is a first-class concern; kernels must be able to target multi-device clusters, with explicit control over communication and computation.

To realize these principles, TIR+ introduces a spectrum of abstractions that can interoperate and compose seamlessly. Higher-level layers compile down to lower-level layers, ensuring that optimizations propagate through the stack. This multi-layered approach enables users to write code at the granularity most suitable for their application, ranging from hardware-level intrinsics to high-level operator library, while relying on the compiler to bridge gaps with automatic optimization. Concretely, the architecture of TIR+ is organized into the following major layers:

- **Hardware Abstraction Layer (Section 3.2)**. This is the lowest layer that faithfully reflects the underlying hardware ISA (e.g., CUDA/PTX, NKI). It exposes native primitives,

provides raw memory buffers in different address spaces, and serves as the foundation for all higher abstractions.

- **Operator Library Layer (Section 3.3)**. Built on top of the hardware abstraction, this layer packages highly optimized routines (e.g., GEMM, copy, reductions) into reusable components. By drawing inspiration from libraries such as CUTLASS, CuTe, and CUB, it provides building blocks for efficient operator implementations.

Together, these layers form a continuum of programming models. A developer may write kernels directly in CUDA for full control or employ library routines for reusable performance primitives. Crucially, these models interoperate seamlessly: higher-level operator libraries can be lowered to hardware abstractions. This hierarchical design provides both flexibility and performance portability, enabling TIR+ to serve as a compiler for today's GPUs and NPUs while remaining adaptable to future hardware.

3.2 Hardware Abstraction Layer

The Hardware Abstraction Layer serves as the foundation of TIR+. Its primary goal is to establish the speed-of-light performance of kernels by exposing their full implementation details. At this level, every aspect of execution is explicitly specified: data layout, tiling strategy, memory movement, thread binding, and pipelining. By targeting hardware-native instructions (CUDA/PTX, NKI etc.), this layer ensures that the performance of each kernel is well-understood and maximized. Because the higher layers ultimately lower into the hardware layer, guaranteeing the state-of-the-art efficiency is essential.

When designing this layer, TIR+ draws on the recurring structures that appear in almost all kernel libraries and compilers. By providing these shared components, the system sets a reusable foundation upon which higher abstractions can be built. The following elements form the core of this layer:

1. **Primitive Expressions and Statements**. Kernels at this level are represented in terms of simple IR constructs—loops, conditionals, and arithmetic expressions. This form is close to TVM's TIR, enabling the reuse of its infrastructure. By grounding hardware-level kernels in this canonical representation, TIR+ ensures that transformations, analysis, and scheduling can be applied consistently before code generation.
2. **Buffers**. All higher-level tensors eventually map to buffers located in a particular memory space: global, shared, or register. Modern accelerators also expose structured on-chip memories, such as NVIDIA's Tensor Memory or Trainium's SRAM. Modeling these explicitly allows the compiler to reason about their placement, movement, and reuse. The buffer abstraction not only unifies these diverse storage spaces under a single representation, but also enables compiler-level memory planning, such as coordinating shared memory reuse across tiles on the same SM, which is a particularly important problem in megakernels where multiple stages compete for limited on-chip resources.
3. **Native Operations**. At the lowest level, kernels are expressed in terms of hardware instructions. On NVIDIA GPUs, this includes examples such as *mma.sync* for tensor core matrix multiply–accumulate and *ldmatrix* for warp-cooperative loads. These are exposed

```
descA = T.local_cell("uint64")
descB = T.local_cell("uint64")

for ko in T.serial(PIPE_CYCLE):
    for ks in T.unroll(PIPELINE_DEPTH):
        stage = ko * PIPELINE_DEPTH + ks
        for ki in T.unroll(BLK_K // MMA_K):
            T.ptx.tcgen05.encode_matrix_descriptor(
                T.address_of(descA),
                A_smem.ptr_to([ks, warp_id, 0, ki * MMA_K]),
                ldo=1,
                sdo=8 * BLK_K * F16_BYTES // F128_BYTES,
                swizzle=SWIZZLE
            )
            T.ptx.tcgen05.encode_matrix_descriptor(
                T.address_of(descB),
                B_smem.ptr_to([ks, 0, ki * MMA_K]),
                ldo=1,
                sdo=8 * BLK_K * F16_BYTES // F128_BYTES,
                swizzle=SWIZZLE
            )
            if stage == 0 and ki == 0:
                T.ptx.tcgen05.mma(
                    "float32", a_type, b_type,
                    warp_id * MMA_N, descA, descB,
                    descI, False, CTA_GROUP, False
                )
            else:
                T.ptx.tcgen05.mma(
                    "float32", a_type, b_type,
                    warp_id * MMA_N, descA, descB,
                    descI, False, CTA_GROUP, True
                )
```

Listing 1: Example of Hardware Layer

through PTX, a virtual instruction set that is later lowered to native machine code (SASS). On AWS Trainium, the analogous interface is NKI (Neuron Kernel Interface). Abstracting such native ops ensures that TIR+ remains capable of targeting the latest hardware generations (such as sm_100a for Blackwell) while retaining portability.

4. **Code Generation**. The final step is to lower this IR to target-specific code. On NVIDIA GPUs this means emitting PTX or CUDA C++; on Trainium, this means generating NKI-compatible instructions. The codegen stage provides a bridge between abstract IR and executable artifacts, guaranteeing that every low-level optimization (e.g., instruction scheduling, unrolling, register allocation) is faithfully realized.

As shown in the example 1, the surrounding structure of the kernel is written using TIR's primitive constructs such as for loops with `T.unroll`, as well as basic arithmetic on loop indices. Memory is represented via buffers, which includes operations like `T.local_cell("uint64")`,

which allocates a scalar register to hold a matrix descriptor, and instructions such as `T.address_of(...)` and `buf.ptr_to(...)` which compute explicit addresses within a shared or local memory tile. The native operations are embedded directly through short, descriptive instruction which reflects the underlying hardware instruction used. `T.ptx.tcgen05.encode_matrix_descriptor` configures how a matrix tile is laid out in shared memory, while `T.ptx.tcgen05.mma` issues a tensor core fused multiply-add instruction from the tcgen05 family of the Blackwell architecture.

Ultimately, these constructs are lowered through code generation, translating the IR into corresponding CUDA/PTX intrinsics (e.g., `cuTensorMapEncodeTiled`, `tcgen05.mma`). This process ensures that the compiler's representation of iteration, memory, and instructions is faithfully realized on the hardware.

3.3 Operator Library Layer

A central challenge in GPU programming is determining how operations are mapped onto the hardware's execution hierarchy. The ThunderKittens framework [7] approaches this problem using a template-based library that requires developers to specify scheduling decisions directly in C++ templates. While this approach ensures scheduling predictability, it limits flexibility and shifts a significant portion of the burden to the user. In contrast, TIR+ treats scheduling as a first-class abstraction within the compiler. The goal is to capture scheduling information at varying levels of completeness and let the compiler resolve the rest.

At a high level, our programming model supports fully, partially, and non-specified programs. A fully-specified program specifies all key decisions upfront—such as tensor layout, memory space, parallelization strategy, and pipeline overlapping. It relies on the user to provide every decision with maximum efficiency in mind. Conversely, partially and non-specified programs provide only incomplete information, leaving degrees of freedom for the compiler to optimize. For instance, a kernel may specify that a matrix multiplication must be broken into tiles of size 128×128, but it defers the decision of how these tiles are distributed across warps or how shared memory reuse is arranged. This design lets programmers express their intent while entrusting the compiler to fill in the details, which is crucial for portability across different architectures.

To support this spectrum of capabilities, TIR+ introduces a small set of scheduling primitives. These include abstractions for execution scopes (thread, warp, warpgroup, block), pipeline constructs for overlapping producers and consumers, and memory-planning constructs for scratchpad reuse. Crucially, these primitives are designed to be composable: they can be provided explicitly by the user or inferred automatically by the compiler. This hybrid approach allows us to unify the template-driven style of ThunderKittens with the automation expected from a compiler's Intermediate Representation (IR).

In combined, this layer provides a clear path toward portable, high-performance kernels that can be expressed at the higher tile level and lowered seamlessly to the hardware level. More examples will be discussed in Section 5.3.

Chapter 4

Supported Infrastructure

4.1 TIR+ Intermediate Representation (IR) Structure

4.1.1 Programming Granularity and Execution Scope

A recurring challenge in GPU programming is that code must execute across a hierarchy of hardware scopes—like threads, warps, cooperative thread arrays (CTAs). Existing frameworks often expose only partial control over this hierarchy. For instance, Triton [10] adopts a kernel–CTA decomposition, letting users express per-block parallelism but leaving warp-level structure implicit. Similarly, Graphene [4] formalizes specification decomposition from kernel to CTA, warp, and thread, but primarily as a scheduling abstraction rather than as a first-class IR construct. Meanwhile, libraries such as CUB [6] provide warp- and CTA-collective primitives, but these appear as external function calls rather than being semantically integrated with the compiler's representation of program structure.

TIR+ addresses this gap by bringing execution granularity as the first-class construct of the IR. We observe two common patterns across existing compilers and DSLs. First, code blocks run at a sub-scope of their parent, mirroring the hierarchical refinement of kernel $\rightarrow$ CTA $\rightarrow$ warp $\rightarrow$ thread. Second, a code block is executed cooperatively by a group of threads within the parent scope, akin to CUB's warp- or CTA-wide collectives. TIR+ generalizes both patterns by allowing any code block to be explicitly annotated with its execution scope, thereby making the scope hierarchy explicit in the IR.

In implementation, the parent–child relation is represented as a tree-like structure. For example, a kernel may spawn CTAs using `T.cta_id`, each CTA may contain multiple warp groups identified by `T.warpgroup_id`, and each warp within a warp group can be further refined with `T.warp_id()` and so on. In addition to strict nesting, TIR+ allows sub-scopes to 'leap' and bind directly to an ancestor (e.g., `T.thread_id(..., parent="cta")`), allowing flexible hierarchies. Within each scope, blocks of code can either decompose into finer scopes or invoke collective operations at their current scope. The snippet below illustrates this design in TIR+:

In the example 2, the kernel launches eight CTAs; each CTA is subdivided into three warp groups, and each warp group further contains four warps of 32 threads each. The program partitions the global matrix A into eight tiles, with each CTA collectively loading one tile into shared memory using all its threads. This representation makes the execution hierarchy explicit—kernel,

```
with T.kernel():
    bx, by = T.cta_id([2, 4], parent="kernel")
    wg_id = T.warpgroup_id([3], parent="cta")
    warp_id_in_wg = T.warp_id([4], parent="warpgroup")
    lane_id = T.thread_id([32], parent="warp")
    with T.cta():
        acc = T.alloc_buffer(
            [384,],
            dtype="float16",
            scope="shared.dyn",
        )
        with T.thread():
            acc[wg_id * 128 + warp_id_in_wg * 32 + lane_id] =
                A[bx, by, wg_id * 128 + warp_id_in_wg * 32 + lane_id]
```

Listing 2: Example of the Hierarchy of Execution Scopes

CTA, warp group, warp, and thread—while abstracting away low-level launch details. By embedding scope declarations directly in the IR, TIR+ captures the parallel hierarchy in a structured and uniform way.

The usefulness of this design lies in making the parallel hierarchy explicit and compositional. Developers gain precise control over granularity, while the compiler benefits from a structured representation that enables analysis, verification, and transformation. Because scopes are explicit IR nodes rather than implicit scheduling conventions, compiler passes can reason about scope boundaries when performing optimizations such as synchronization insertion, shared memory planning, or scope-specific fusion. Moreover, the tree-structured parent–child hierarchy naturally extends to future GPU architectures that introduce new intermediate scopes (e.g., thread block clusters in Blackwell architectures) or distributed communication across devices. By making execution granularity explicit, TIR+ unifies existing decomposition and collective patterns within a single IR framework, bridging the gap between low-level CUDA programming and higher-level kernel DSLs.

4.1.2 Tensor Layout

In TIR+, tensor layouts are represented as first-class objects that map logical tensor coordinates onto memory and thread axes. This abstraction is critical for expressing how data is distributed across shared memory, registers, and cooperative thread groups, without directly exposing hardware-specific indexing.

We introduce **Axe**, a unified layout system designed to capture both intra-kernel tiling and inter-device sharding within a single algebraic framework. Drawing from GSPMD [12], Axe generalizes common strategies: sharded (D), replicated (R), and a third owner-only (O) pattern, where a partition is exclusively owned by one device or thread group. The O pattern arises naturally in distributed training (e.g., after reducing along a device axis) and is equally relevant within kernels, where certain partitions must remain private to a warp or warp group.

Axe also treats memory and thread hierarchies symmetrically: logical coordinates can map

```
A_layout = T.ComposeLayout(
    T.SwizzleLayout(3, 3, 3, swizzle_inner=True),
    T.TileLayout(
        shard=(
            (PIPELINE_DEPTH, NUM_CONSUMER, BLK_M, BLK_K),
            (NUM_CONSUMER * BLK_M * BLK_K, BLK_M * BLK_K, BLK_K, 1),
        )
    ),
)
B_layout = T.ComposeLayout(
    T.SwizzleLayout(3, 3, 3, swizzle_inner=True),
    T.TileLayout(
        shard=(
            (PIPELINE_DEPTH, BLK_N, BLK_K), (BLK_N * BLK_K, BLK_K, 1),
        )
    ),
)
```

Listing 3: Example of Composed Layout

to memory axes (global, shared, registers), thread axes (warp IDs, lane IDs), or hybrids of both. Unlike CuTe and Triton, which model layouts as surjective maps from hardware resources to tensor indices, Axe defines them as functions from logical coordinates to hardware axes. This choice allows it to express exclusive ownership (O) patterns, where some logical partitions map to no hardware resources, while still covering the standard replicated (R) and sharded (D) cases.

In practice, layouts are constructed from composable primitives such as swizzles and tiles. In example 3, it shows the operand layouts in a GEMM kernel. Here, `T.ComposeLayout` combines a swizzle with a multi-axis tiling strategy. By construction, the layout object encodes how threads in a CTA collectively cover a tile of A or B, making these mappings explicit in the IR.

Layouts also support hierarchical tiling across warps and threads. Example 4 demonstrates how layouts are composed across multiple granularities: a warp-level layout (`warp_layout`) is combined with thread-level atomic tiles (`atom`) and then repeated and extended into accumulator tiles (`acc_layout`). The resulting buffer is allocated with a precise logical layout that dictates both memory addressing and thread cooperation.

Embedding layouts into the IR (Intermediate Representation) is particularly valuable for scheduling. At the operator library layer, scheduling decisions such as double-buffering, epilogue fusion, and collective reductions rely on layout information. The compiler can directly query how data is partitioned across threads, whether memory accesses are coalesced, and how tiles map to shared memory to make scheduling decisions. By providing layouts as algebraic IR objects rather than as implicit indexing schemes, TIR+ makes these scheduling choices analyzable and composable, thereby ensuring both efficiency and portability.

```
atom = T.TileLayout(shard=([1, 2], [2, 1]))
warp_layout = T.TileLayout(
    shard=([8, 4], [(4, "laneid"), (1, "laneid")])
)
warp_atom = atom.tile(warp_layout, (8, 4), (1, 2))
tile = T.TileLayout(shard=([2, NUM_COL // 8], [1, 2]))
acc_layout = warp_atom.tile(tile, (2, NUM_COL // 8), (8, 8))

acc = T.alloc_buffer(
    [2, NUM_COL // 4],
    dtype=dtype,
    scope="local",
    logical_scope="thread",
    layout=atom.tile(tile, (2, NUM_COL // 8), (1, 2)),
)
```

Listing 4: Example of Hierarchical Tiling Layout

4.1.3 Parser/Printer, FFI, Transformations

To support productive development on top of this Intermediate Representation (IR), our compiler includes several infrastructure components that simplify the construction and transformation of TIR+ programs. These components handle the translation from user code to IR, enable flexible interactions between high-level Python and low-level C++ code, and provide utilities for analyzing or modifying the IR through specialized passes.

A lightweight Python DSL and parser allow developers to express kernels in a high-level syntax, which is automatically translated into IR nodes of TIR+. A corresponding IR printer converts optimized programs back into a readable, Python-like form, making it easy to inspect transformations.

Integration between Python and C++ is achieved through a packed-function FFI (Foreign Function Interface). This interface allows Python code to construct IR nodes, invoke compiler passes, and launch kernels, while C++ routines handle performance-critical logic. As a result, developers can remain in Python for productivity without losing access to efficient low-level implementations.

For analysis and optimization, TIR+ leverages the visitor/mutator framework from TIR. Visitors systematically traverse the IR for checks or collecting statistics, while mutators implement rewrites such as loop transformations or memory optimizations. The arithmetic analyzer simplifies index expressions and proves bounds, ensuring that transformations produce efficient and correct code.

Together, these components form the backbone of TIR+, providing a high-level frontend, a robust bridge, and a transformation framework that supports both developer productivity and advanced compiler optimizations.

Table 4.1: Host and device-side NVSHMEM APIs integrated into TIR+.

Host APIs	
`nvshmemx_cumodule_init`	Register CUDA module for device-side NVSHMEM
`nvshmemx_barrier_all_on_stream`	Stream-ordered barrier across PEs
`nvshmem_malloc`	Collective symmetric allocation
`nvshmem_ptr`	Translate symmetric object to a process-local pointer
Device APIs	
`nvshmem_getmem_nbi`	Nonblocking one-sided read
`nvshmem_putmem_nbi`	Nonblocking one-sided write
`nvshmem_putmem_signal_nbi`	Ordered write followed by remote signal publish
`nvshmem_signal_op`	Atomic update of a symmetric signal
`nvshmem_wait_until`	Wait for condition on a symmetric variable
`nvshmem_fence`	Order prior NVSHMEM ops before subsequent ones
`nvshmem_quiet`	Wait for completion of outstanding NVSHMEM ops
`nvshmem_barrier_all`	Device-side barrier across all PEs
`nvshmem_ptr`	Translate symmetric object to a device pointer

4.2 First Class Support for Distributed Execution

TIR+ brings first-class support for distributed execution. We integrate a distributed runtime, a set of IR-level abstractions, and NVSHMEM backend to enable compute–communication overlap at tile granularity.

We build on top of TVM's Disco runtime to manage multi-worker execution and object placement. Disco follows a controller–worker model and exposes three kinds of session backends to form a cluster: a ThreadedSession runs an in-process thread pool; a ProcessSession launches multiple worker processes on a single node (for multi-GPU servers); and a SocketSession connects workers across multiple nodes over network sockets. Distributed references (DRef) name per-worker objects uniformly (e.g., GPU NDArrays and compiled modules), and session methods provide coarse-grain collectives (broadcast, scatter, all-gather, all-reduce). In TIR+, the compiled artifacts (functions and buffers) are materialized as DRefs, and data motion across workers is explicit in the generated host code.

TIR+ distributed execution is backended by NVSHMEM. The code generator and build system emit necessary NVSHMEM link flags and runtime hooks, so that compiled TIR+ modules are NVSHMEM-ready at load time. We add host- and device-side NVSHMEM support (see Table 4.1) to enable GPU-initiated, one-sided communication and signaling within a kernel.

- **Host side**. We enable device-side NVSHMEM for a CUDA module at load time, allocate symmetric memory and resolve local pointers on remote symmetric buffers, and insert stream-ordered barriers at synchronization points during execution. These calls are issued on the same CUDA streams as the scheduled kernels to preserve ordering with respect to compute.
- **Device side**. TIR+ provides intrinsics that lower to NVSHMEM device operations for nonblocking one-sided transfers (put/get), write-with-notify, lightweight signal/wait, and

scoped ordering and completion. This enables a single kernel to pipeline communication with computation.

This design treats distributed execution as a first-class abstraction: Disco provides process and cluster management with DRefs; the IR captures necessary instructions; and NVSHMEM supplies device-initiated transfers and signals. This results in the communication-heavy kernels—distributed GEMMs and MoE layers in particular—compile into single, pipelined kernels that fully utilize the interconnect, rather than sequences of compute and bulk collectives separated by idle time.

For example, in GEMM + ReduceScatter, each device accumulates partial results for its output shard. As soon as a subtile completes, the producer issues a nonblocking one-sided write into the owner's symmetric buffer and publishes a device-side signal; on the owner, a per-tile arrival counter advances, and when arrivals reach the world size, the kernel performs the reduction and epilogue in place. We implement this using a persistent kernel with a work queue: CTAs fetch GEMM tiles and handle communication and signaling upon finish, and fetch epilogue reduction tiles when they're ready. This design sustains tile-granularity overlap, keeping the interconnect busy while tensor cores execute and thereby reducing latency.

Chapter 5

Operator Scheduler

5.1 Overview of Operator Scheduling and Lowering

Library-level operators are lowered in TIR+ through a schedule registry. Each operator (e.g., `Tp.copy_async`, `Tp.sum`) is associated with one or more device-specific implementations (e.g., CUDA TMA copies, warp-level reductions), registered under a specific target kind. During lowering, the compiler consults this registry: if the operator's call site matches the preconditions of a registered implementation, the corresponding TIR schedule is substituted; otherwise, the call is left intact to be handled by generic rules or later passes. This design cleanly separates operator definition from its scheduling: the front-end code can freely use library operators without embedding hardware-specific knowledge, while the lowering pass dispatches to tuned implementations when applicable.

This registry-based adaptive dispatch also provides a natural way to accommodate varying levels of specification in the programs. When details such as tensor layout, memory space, or pipelines are fully determined, the dispatcher can directly substitute the most specialized implementation. When only partial information is given—for example, a tiling scheme without an explicit parallelization strategy—the compiler can still resolve the operator by analyzing buffer properties, event usage, and scope annotations, filling in missing details automatically. Even when no scheduling information is provided beyond the operator call itself, generic implementations ensure correctness while leaving room for later optimizations. Section 5.3 presents case studies that illustrate operator scheduling in practice.

5.2 Event Tensor Abstraction

5.2.1 Motivation

Modern GPU programming offers a wide range of synchronization primitives — from `bar.sync` for warpgroup coordination, to `mbarrier` and `tcgen05.wait` for TMA and tensor-core operations, and NVSHMEM signals for inter-GPU communication. While these mechanisms are powerful, they expose highly specialized semantics tied to particular hardware pipelines or memory scopes. This tight coupling makes it difficult to compose synchronization across dif-

```
with T.cta():
    event = Tp.alloc_semaphore_event_tensor(
        EventImpl.kTMALoad,
        state=[mbarrier, phase, tx_cnt],
    )
    event[0].init(1)

    for stage in range(n):
        Tp.copy_async(A_smem[*r_smem], A[*r_gmem(stage)], event[0])
        event[0].commit()
        event[0].wait()
```

Listing 5: Example of the Event Tensor Init/Commit/Wait APIs

ferent operators or reuse synchronization logic beyond the exact producer-consumer patterns assumed by existing APIs.

For example, a typical copy pipeline encapsulates synchronization into a fixed producer-consumer model with cyclic buffers. This design works well for staged pipelines such as global-to-shared copies, but it becomes restrictive in several scenarios.

1. **Beyond Cyclic Buffers**. Many kernels require synchronization patterns that are not cyclic-buffered pipelines. For instance, serializing two warpgroups such that one must load data into registers before the other begins execution cannot be expressed without specific workarounds. The pipeline abstraction forces everything into the mold of a staged buffer, even when the underlying hardware primitive (e.g., `mbarrier`) does not require this structure.
2. **Decoupling Synchronization with Operator Dispatch**. In the current design, the synchronization logic is tightly bound to specific operators like async copy schedules. This prevents reuse in other contexts (e.g., GEMM tile scheduling or NVSHMEM transfers), even though the hardware synchronization mechanism (`mbarrier`, `bar.sync`, etc.) is conceptually orthogonal to the operator. Therefore, programmers cannot flexibly combine different operator patterns while reusing the same synchronization mechanism.
3. **Hardware fragmentation**. Each backend synchronization method comes with distinct semantics and usage rules (e.g., transaction counters for mbarrier, group counters for bulk async, phase flips for tcgen05, semaphores for Trainium). Without a unifying abstraction, programmers must reason about these details in every kernel.

What we would like instead is a unified abstraction for events, one that (1) captures the shared notion of "the completion of an operator" independent of its implementation, and (2) decouples synchronization from operator dispatch in the operator library layer. By lifting events into a first-class tensor-like abstraction, Event Tensor, TIR+ provides a composable mechanism to bind operators to events, signal completion (commit), and enforce ordering (wait). Each Event Tensor can then be lowered to the appropriate backend primitive (`mbarrier`, `bar.sync`, NVSHMEM signal, or even software semaphore) depending on its usage context. There are two key benefits of this design:

- **G0: Expressiveness**. Programmers can express both standard patterns (async pipelines,

cyclic buffering) and irregular patterns (serializing warpgroups, inter-device signaling) using the same abstraction.

- **G1: Composability**. The synchronization logic is decoupled from the operators, enabling flexible reuse across diverse kernels (copy, GEMM, communication) while still mapping efficiently to hardware-specific primitives.

5.2.2 Implementation

An Event Tensor in TIR+ is defined as a tensor of event counters. Each element in an Event Tensor corresponds to a specific tile's synchronization event. Conceptually, an Event Tensor element holds a wait count equal to the number of producer tasks that must complete before the event is considered "fulfilled." The TIR+ dialect provides intrinsic operations to manage these events, typically expressed as `init`, `commit`, and `wait` actions on Event Tensors.

- `event.init()`. When an Event Tensor is created, each element's counter is set to the expected number of incoming signals. For example, declaring `E = ETensor((n,), wait_count=4)` creates a 1-D Event Tensor E of length n where each element expects 4 signals (i.e., there are 4 producer tasks per index). This wait count encodes the dependency fan-in for each consumer task (in this case, perhaps a tile has 4 sub-tasks producing partial results that must all complete before the consumer is released).
- `event.commit()`. A producer task "commits" to an Event Tensor by notifying its completion and decrements the event's counter. Multiple producers may commit to the same Event Tensor element; once the counter reaches zero, the event is fully signaled. In the operator library layer, these commits can be inserted automatically by the compiler. For instance, if a device function is annotated to produce an event, the compiler will generate a notify at the end of that task. Internally, the compiler lowers this to the appropriate device-side notify calls.
- `event.wait()`. A consumer task issues a wait on an Event Tensor element before it begins execution. Conceptually, this blocks until the event's counter has reached zero, meaning that all expected producers have committed. In the operator library layer, the waits can be inserted at the start of the consumer task automatically by the compiler, and be lowered to appropriate device-side intrinsics, such as a lightweight busy-wait loop on the counter. Once the counter reaches zero, the event is fully signaled and the dependent tasks may safely proceed.

Example 5 illustrates this workflow in an asynchronous TMA copy from global to shared setting: an Event Tensor is allocated with type `kTMALoad` and initialized with a wait count. Each asynchronous copy stage signals completion through `commit()`, and the corresponding consumer waits via `wait()` before proceeding. More on asynchronous copy operator scheduling will be discussed in Section 5.3.

In summary, the Event Tensor abstraction in TIR+ provides a simple yet powerful mechanism for operator scheduling inside GPU kernels. It allows the compiler to express fine-grained dependencies (even with dynamic shapes) in a natural tensor index manner, and to generate GPU code that uses event counters to synchronize tasks efficiently. This leads to correct and highly

parallel execution of fused operators: tasks start as soon as their inputs are ready, and multiple stages of computation can overlap without sacrificing program correctness. By integrating event-based synchronization at the IR level, the abstraction extends naturally to crucial use cases such as persistent megakernels, which orchestrate complex computations across the entire GPU.

5.3 Operator Scheduling Examples

As shown in example 6 and 7, an operator call in TIR+ is automatically lowered into an intricate mix of instructions, memory operations, and synchronization code that modern GPUs demand.

Scheduling and dispatching a library-level operator such as `Tp.sum` or `Tp.copy_async` involves first validating and then generating code. Validation ensures the operator is well-formed for the target: checking legality of data movement (e.g., only global↔shared transfers), verifying layouts are compatible with hardware features like TMA, and confirming that axes and swizzles can be mapped consistently.

Once validated, the schedule generates the implementation by assigning execution scopes across CTA, warp, or thread levels; if necessary, allocating shared memory and registers with layouts that preserve coalescing, bank efficiency, and reuse; wiring synchronization through event tensors that manage counters, barriers, or semaphores; and constructing producer–consumer pipelines that enable overlap, such as double-buffered copy $\rightarrow$ transform $\rightarrow$ mma. Host-side initialization code may also be emitted, for example encoding tensor maps before kernel launch. Schedules further provide choices for cache hints, tiling strategies, and fallbacks (e.g., vectorized `cp.async`) when constraints fail, ensuring robustness across targets.

In practice, a single high-level operator expands into a substantial amount of low-level device and host code—barriers, descriptors, allocations, and synchronization—that would otherwise be hand-written, making scheduling both a correctness mechanism and enhanced developer productivity.

```
def sum_before():
    A_smem = T.alloc_shared((32, 32), "float16",
    ↪  layout=T.TileLayout(shard=([32, 32], [(32, "m"), (1, "m")])))
    B_smem = T.alloc_shared((32,), "float16",
    ↪  layout=T.TileLayout(shard=([32], [(1, "m")])))
    Tp.sum(B_smem[0:32], A_smem[0:32, 0:32], [-1], False)

def sum_after():
    A_smem = T.alloc_shared((1024,), "float16")
    B_smem = T.alloc_shared((32,), "float16")
    for tid_x in T.thread_binding(32, thread="threadIdx.x"):
        thread_data = T.allocate([1], "float16", "local")
        for step in range(32):
            if step + tid_x // 32 < 32:
                thread_data_1 = T.Buffer((1,), "float16", data=thread_data,
                ↪  scope="local")
                thread_data_1[0] = T.float16(0.0)
                for t in range(1):
                    if t * 32 + tid_x % 32 < 32:
                        thread_data_1[0] = thread_data_1[0] + A_smem[step *
                        ↪  32 + tid_x]
                mask: T.uint32 = T.tvm_warp_activemask()
                thread_data_1[0] = thread_data_1[0] +
                ↪  T.tvm_warp_shuffle_xor(mask, thread_data_1[0], 1, 32,
                ↪  32)
                thread_data_1[0] = thread_data_1[0] +
                ↪  T.tvm_warp_shuffle_xor(mask, thread_data_1[0], 2, 32,
                ↪  32)
                thread_data_1[0] = thread_data_1[0] +
                ↪  T.tvm_warp_shuffle_xor(mask, thread_data_1[0], 4, 32,
                ↪  32)
                thread_data_1[0] = thread_data_1[0] +
                ↪  T.tvm_warp_shuffle_xor(mask, thread_data_1[0], 8, 32,
                ↪  32)
                thread_data_1[0] = thread_data_1[0] +
                ↪  T.tvm_warp_shuffle_xor(mask, thread_data_1[0], 16, 32,
                ↪  32)
                if tid_x % 32 == 0:
                    B_smem[step] = thread_data_1[0]
    T.tvm_storage_sync("shared", T.bool(False), -1)
```

Listing 6: Example of Scheduling Reduction Operator

```
def tma_load_before(ks):
    mma2tma_bar.wait(ks, phase[0])
    Tp.copy_async(A_smem[ks, :, :], A[m_start: m_start + BLK_M, k_start:
    ↪  k_start + BLK_K], evt=tma2trans_event[ks])
    Tp.copy_async(B_smem[ks, :, :], B[n_start: n_start + BLK_N, k_start:
    ↪  k_start + BLK_K], evt=tma2trans_event[ks])
    tma2trans_event[ks].commit()

def tma_load_after(ks):
    T.ptx_mbarrier_try_wait(T.tvm_access_ptr(T.type_annotation("uint64"),
    ↪  buf.data, 18 + ks, 6 - ks, 3), T.bitwise_xor(1, phase[0]))
    if T.ptx_elect_sync(T.int64(4294967295)):
        with T.thread():
            tx_cnt[0] = tx_cnt[0] + 16384
            for lvs_0, lvs_1 in T.grid(1, 1):
                T.ptx_cp_async_bulk_tensor_global_to_cluster(2,
                ↪  T.tvm_access_ptr(T.type_annotation("float8_e4m3fn"),
                ↪  buf.data, 1024 + T.shift_left(T.bitwise_xor(ks * 1024,
                ↪  T.shift_right(T.bitwise_and(ks * 1024, 56), 3)), 4),
                ↪  98304 - T.shift_left(T.bitwise_xor(ks * 1024,
                ↪  T.shift_right(T.bitwise_and(ks * 1024, 56), 3)), 4), 3),
                ↪  T.tvm_access_ptr(T.type_annotation("uint64"), buf.data,
                ↪  6 + ks, 6 - ks, 3), A_tensormap_1, stage[0, 0] * 128 +
                ↪  lvs_1 * 128, (tile_scheduler_m_idx[0, 0] * 2 +
                ↪  clusterCtaIdx_x) * 128, 0, 1, "")
            tx_cnt[0] = tx_cnt[0] + 14336
            for lvs_0, lvs_1 in T.grid(1, 1):
                T.ptx_cp_async_bulk_tensor_global_to_cluster(2,
                ↪  T.tvm_access_ptr(T.type_annotation("float8_e4m3fn"),
                ↪  buf.data, 99328 + T.shift_left(T.bitwise_xor(ks * 896,
                ↪  T.shift_right(T.bitwise_and(ks * 896, 56), 3)), 4),
                ↪  86016 - T.shift_left(T.bitwise_xor(ks * 896,
                ↪  T.shift_right(T.bitwise_and(ks * 896, 56), 3)), 4), 3),
                ↪  T.tvm_access_ptr(T.type_annotation("uint64"), buf.data,
                ↪  6 + ks, 6 - ks, 3), B_tensormap_1, stage[0, 0] * 128 +
                ↪  lvs_1 * 128, (tile_scheduler_n_idx[0, 0] * 2 +
                ↪  clusterCtaIdx_x) * 112, 0, 1, "")
    if T.ptx_elect_sync(T.int64(4294967295)):
        with T.thread():
            T.ptx_mbarrier_arrive_expect_tx(T.tvm_access_ptr(T.type_annotat⌋
            ↪  ion("uint64"), buf.data, 6 + ks, 6 - ks, 3), tx_cnt[0])
            tx_cnt[0] = 0
```

Listing 7: Example of Scheduling Asynchronous Copy Operator

Chapter 6

Experimental Evaluation

6.1 Experimental Setup and Benchmark Methodology

All experiments are conducted on NVIDIA Blackwell B200 GPUs. We evaluate both single-GPU and multi-GPU settings. For distributed experiments, we use all eight GPUs within a single node, which are fully interconnected via NVLink.

At the current stage of development, TIR+ provides full support at the hardware abstraction layer and partial support at the operator library layer. Our experiments use the available operators where applicable and rely primarily on the hardware layer, which is sufficient to evaluate representative kernels from LLM workloads.

We evaluate kernels across a broad coverage of tensor shapes drawn directly from popular LLM architectures. Specifically, we include shapes from LLaMA[3], Qwen[13], Gemma[8], Mixtral[5], and GPT[2] models. These shapes reflect diverse scaling strategies in sequence length, hidden dimension, and intermediate size, which in turn evaluates different aspects of kernel performance such as memory bandwidth usage, tensor core utilization, and inter-GPU communication. The exact tensor configurations used in each experiment are disclosed in the corresponding subsections of Section 6.2.

For profiling and measurement on single-GPU kernels, we use the Proton profiler from Triton[10]; on multi-GPU kernels, we rely on CUDA event timing for lightweight measurements and NVIDIA Nsight Systems (nsys) for full end-to-end timelines. In addition, we have a built-in fine-grained profiler in TIR+, which we use to analyze scheduling behavior in later sections.

6.2 Kernel Performance

6.2.1 Memory-Bound Kernels

Many kernels in LLM workloads, such as positional encodings (e.g., RoPE), normalization layers (LayerNorm, RMSNorm), activations, and Softmax, are memory-bound. These kernels exhibit low arithmetic intensity and are primarily limited by memory bandwidth. Here, we choose RMSNorm as a representative kernel. RMSNorm operates on input tensors of shape (num_rows, hidden_size), where num_rows = batch_size $\times$ sequence_length. For each row, it computes a nor-

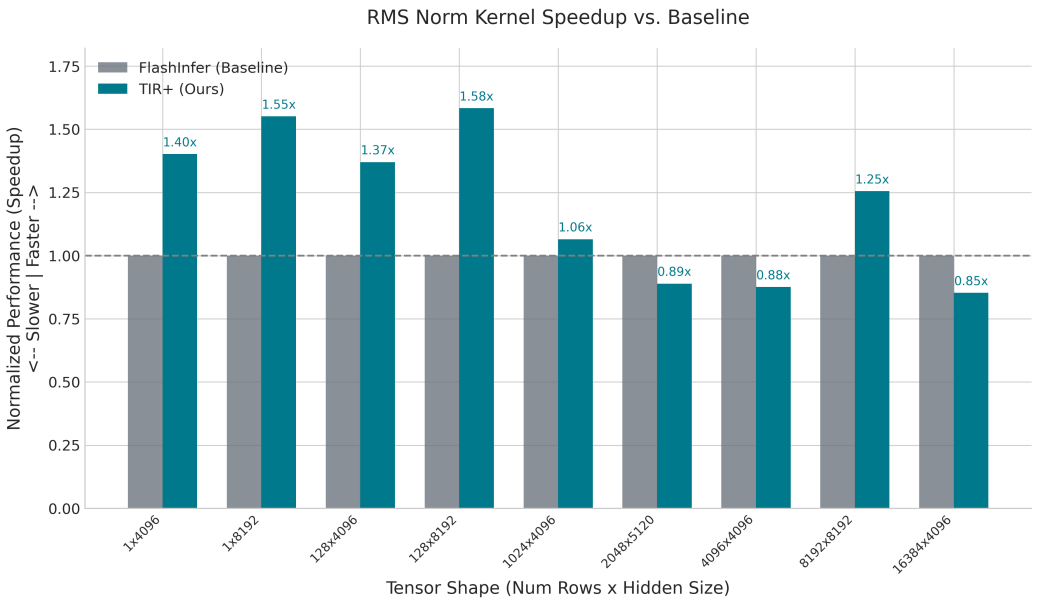

Figure 6.1: Speedup of TIR+ vs. FlashInfer on RMSNorm (Single B200 GPU)

malization factor via a reduction over the hidden dimension followed by element-wise scaling. All experiments use float16 inputs and outputs.

We evaluate a spectrum of shapes that correspond to diverse LLM scenarios. The small-row cases such as (1×4096) mimic latency-sensitive autoregressive decoding. The larger-row cases such as (4096×4096) and (16384×4096) correspond to prefill or large-batch training workloads. This range captures both latency-critical and throughput-critical regimes. We compare the performance of our TIR+ RMSNorm kernel against a highly optimized implementation from the FlashInfer[14] library.

Figure 6.1 shows the performance comparison of RMSNorm kernel between TIR+ and FlashInfer across various tensor shapes on a single B200 GPU. The speedup is calculated as FlashInfer Latency / TIR+ Latency. Results indicate that TIR+ kernel has superior performance in latency-bound scenarios such as autoregressive decoding, and can achieve up to $1.58\times$ speedup. In large-batch training and prefill situations, however, FlashInfer kernels have slightly superior performance, likely due to heuristic optimizations such as multi-row per block scheduling.

6.2.2 GEMM Kernels

We evaluate two GEMM kernel variants in TIR+, distinguished by the number of warpgroups dedicated to the consumer stage, which performs matrix multiplications on Tensor Cores using `mma` instructions. The one-consumer-group version adopts a more balanced design: it assigns comparable resources to data fetching via TMA and to computation, making it effective when load and compute times are similar or when resources such as registers and shared memory are constrained. In contrast, the two-consumer-group version prioritizes computation, dedicating additional warp groups to the consumer stage to maximize Tensor Core throughput.

Both kernels are benchmarked against cuBLAS on a single NVIDIA B200 GPU across a

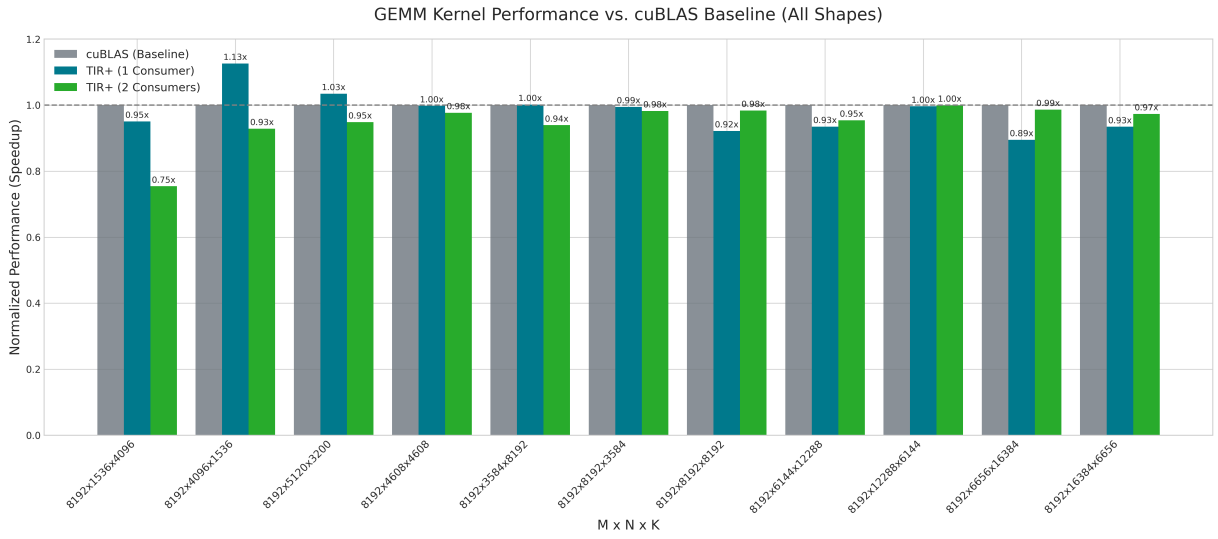

Figure 6.2: Speedup of TIR+ vs. cuBLAS on GEMM (Single B200 GPU)

diverse set of matrix shapes derived from major LLM architectures, covering a wide range of M, N, and K dimensions. Both input and output data use float16.

The results are summarized in Figure 6.2. The results demonstrate that kernels generated by TIR+ are highly competitive with the industry-standard cuBLAS library. Across the majority of tested shapes, TIR+ achieves over 98% of cuBLAS performance, and in several configurations, it matches the baseline almost exactly.

The analysis reveals that the optimal scheduling strategy is highly dependent on the matrix dimensions. Notably, our one-consumer variant surpasses the performance of cuBLAS on specific shapes with small K. For instance, on a matrix of shape M=8192, N=4096, K=1536, the one-consumer kernel is 12.6% faster than cuBLAS, and on M=8192, N=5120, K=3200, it achieves a 3.4% speedup.

The results indicate that the one-consumer variant is well-suited for workloads with a small K dimension and moderate M, N, where balanced data movement and compute reduce synchronization overhead. The two-consumer variant, by contrast, excels in cases that are strongly compute-bound, such as extremely large K, large N, or large balanced dimensions, where saturating Tensor Cores becomes the dominant factor.

6.2.3 Attention Kernels

We evaluate the performance of our attention kernel on the BatchDecode operation, a critical component in LLM inference that computes attention for a batch of new query tokens against a cached history of keys and values. This operation often utilizes Grouped-Query Attention (GQA)[1] to balance computational cost and model quality. We compare our TIR+ generated kernel against a highly optimized implementation from FlashInfer[14], a state-of-the-art library for LLM inference kernels. All experiments are conducted with float16 precision.

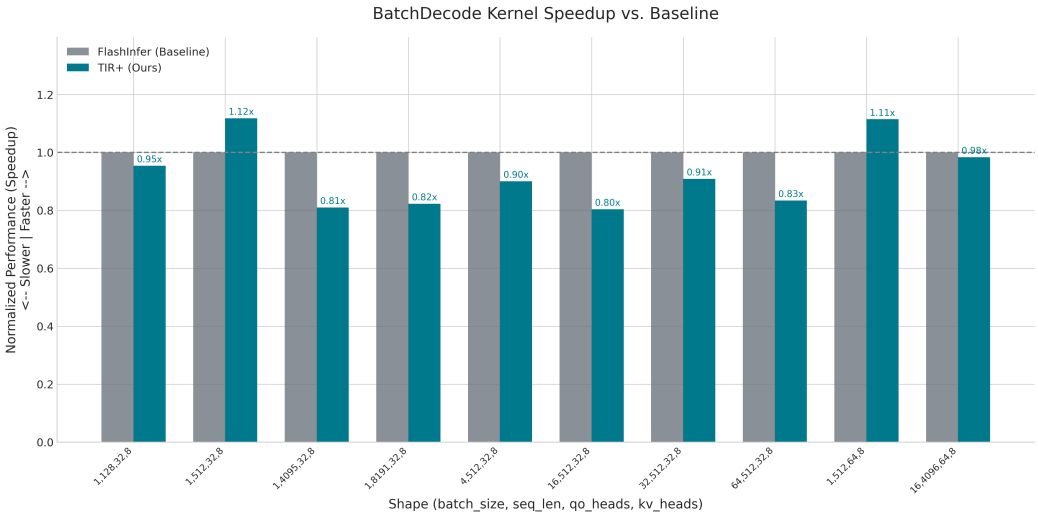

Figure 6.3: Speedup of TIR+ vs. FlashInfer on BatchDecode (Single B200 GPU)

The results, summarized in Figure 6.3, show that TIR+ generates highly competitive kernels, particularly in latency-sensitive scenarios. At small batch size and a moderate sequence length of 512, TIR+ achieves a speedup of up to 1.12× over FlashInfer. However, for very long sequences (e.g., 4095 and 8191) and large batch sizes, FlashInfer's autotuning strategy gives it a performance advantage of approximately 18-23% faster than TIR+ kernels. This suggests that FlashInfer's implementation is better tuned for maximizing parallelism in throughput-oriented workloads. More adaptive scheduling within TIR+ could further improve performance across both latency and throughput-critical regimes, which is left for future work.

6.2.4 Communication-Computation Overlap Kernels

A key advantage of TIR+ is its first-class support for distributed execution, enabling the creation of single, fused kernels that overlap communication and computation at a fine-grained level. This approach is particularly effective for Tensor Parallelism in LLM workloads, where tensors are often split across multiple GPUs. We evaluate two common distributed patterns: fused All-Gather+GEMM and GEMM+ReduceScatter.

To achieve this fine-grained overlap, TIR+ implements these operations as a single, persistent kernel that utilizes a dynamic scheduler. CTAs fetch tasks from a shared work queue, allowing for flexible orchestration of compute and communication. For instance, in our GEMM+ReduceScatter kernel, producer CTAs compute local GEMM tiles and issue non-blocking writes to the destination GPU. Concurrently, consumer CTAs on each GPU monitor arrival counters; a consumer begins the reduction and epilogue for a specific output tile only after receiving the partial results from all peer GPUs. This ensures that computation on one tile can proceed while communication for other tiles is still in flight.

We compare against multiple open-source baselines: Async-TP in PyTorch for tensor-parallel decomposition, Triton-Distributed [15] for overlap-aware scheduling, and cuBLAS+NCCL as the baseline for non-overlapping execution. All evaluations are on 8×B200 GPUs.

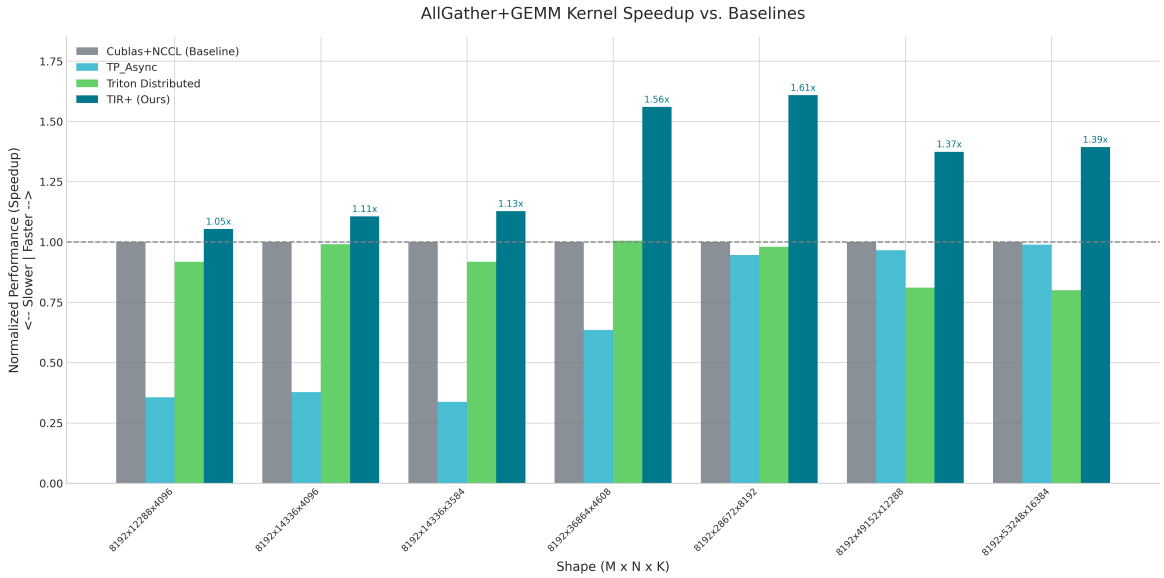

Figure 6.4: Speedup of TIR+ vs. Baselines on AllGather+GEMM (8×B200 GPUs)

Figure 6.5: Speedup of TIR+ vs. Baselines on GEMM+ReduceScatter (8×B200 GPUs)

As shown in Figure 6.4, this fine-grained overlap results in substantial performance gains for AllGather+GEMM. The TIR+ kernel consistently outperforms the standard cuBLAS+NCCL baseline, achieving a speedup ranging from 5% to 38%. The benefit of overlap is especially pronounced in configurations with large N and K dimensions, such as the (M=8192, N=28672, K=8192) shape, where TIR+ is 1.6× faster than the sequential baseline and 1.64× faster than Triton Distributed. This demonstrates the efficiency of tile-based pipelining in hiding communication latency and maximizing hardware utilization.

The results of GEMM+ReduceScatter are presented in Figure 6.5. Across all tested model

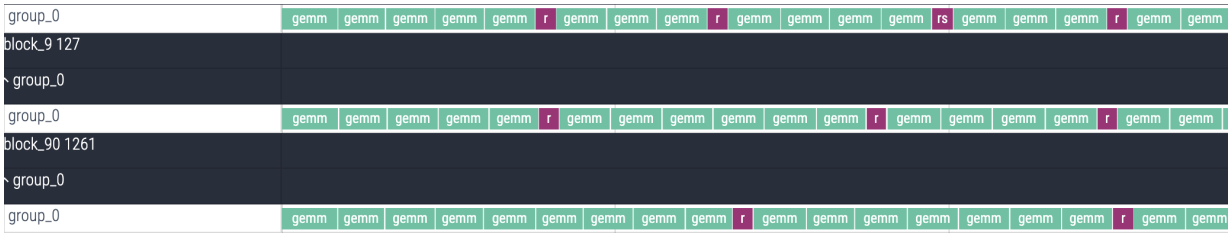

Figure 6.6: Timeline of Execution for Overlapped GEMM+ReduceScatter

configurations, the TIR+ kernel delivers a stable and significant speedup of 22-40% over the cuBLAS+NCCL baseline. This consistent improvement is a direct result of our scheduling strategy's ability to eliminate hardware idle time and driving simultaneous utilization of both communication bandwidth and Tensor Core computation.

The effectiveness of our fusion strategy is visually confirmed by the execution timeline in Figure 6.6. The TIR+ kernel exhibits a continuous stream of operations. The timeline shows GEMM compute tiles (gemm) tightly interleaved with communication and reduction primitives (r, rs) at a granular level, with the dynamic scheduling overhead between a compute and communication tile being a minimal 1-2 µs. This demonstrates true hardware-level overlap within a single kernel, eliminating synchronization overhead and maximizing GPU utilization to reduce end-to-end latency.

6.3 Discussion

The evaluation shows that TIR+ is both expressive and performant across diverse LLM workloads. Its abstractions, such as explicit scheduling, operator library, and first-class distributed primitives, enable concise implementation of kernels ranging from memory-bound, to compute-bound, and distributed fused kernels. The TIR+ kernels are typically around 100-300 lines of code, compared to the thousands of CUDA code for an implementation with equivalent performance. The ability to achieve high performance across these categories demonstrates that TIR+ effectively captures low-level hardware details while remaining flexible for optimization. Performance results further validate this design: GEMM kernels are consistently on par with cuBLAS and occasionally surpass it, RMSNorm matches or outperforms FlashInfer in memory-bound settings, and distributed kernels deliver large speedups over existing baselines.

Meanwhile, the experiments highlight auto-scheduling as a critical direction for future work. Optimal schedules often vary with tensor shapes and workload regimes, as seen in attention kernels where FlashInfer remains more effective in certain throughput-oriented cases. Integrating automated scheduling and tuning strategies into TIR+ would allow the compiler to adapt dynamically to different workload characteristics, ensuring that performance remains competitive across the full spectrum of scenarios.

Chapter 7

Future Directions

Looking ahead, TIR+ can be extended along several directions.

A first direction is to further improve **kernel performance**. While current implementations are competitive, additional gains are possible through topology-aware and workload-adaptive scheduling. Attention workloads in particular vary widely in shape and granularity, and autotuning tailored for these cases could reliably close the gap between compiler schedules and expert tuning.

Another direction is to raise the level of abstraction through a **tile-based DSL**. A Python front end, similar in spirit to Triton or TileLang, would allow developers to express kernels directly in terms of tiles. Such programs could then be lowered through the operator library into efficient hardware-level code. This additional layer would improve productivity while retaining performance portability.

Extending TIR+ with **megakernel compilation** is especially promising. Persistent kernels that fuse computation and communication have demonstrated large latency reductions in recent LLM systems. TIR+ could incorporate cross-operator fusion, intra-kernel scheduling, and event-tensor–based overlap to support pipelines such as GEMM + reduce-scatter, attention + all-gather, or entire decoder blocks within a single launch. This approach also extends naturally to irregular workloads such as Mixture-of-Experts, sparse attention, and dynamic routing, where fine-grained scheduling and communication–computation overlap are critical for scalability.

Finally, **AI-assisted kernel generation** represents a longer-term opportunity. Recent work has shown that large language models can synthesize and refine GPU kernels when guided by performance feedback. The layered design of TIR+ makes it attractive: models could propose high-level operator compositions or tile specifications, which the compiler then lowers and optimizes, with feedback loops refining scheduling choices such as tile sizes or pipeline depths. Such an integration could make high-performance kernel development increasingly automated and accessible.

Chapter 8

Conclusion

The thesis presents TIR+, a multi-level compiler that unifies productivity and performance for GPU kernel development in large language model (LLM) workloads. By exposing fine-grained hardware control alongside reusable operator libraries and distributed primitives, TIR+ makes it possible to achieve state-of-the-art efficiency without the prohibitive engineering cost of hand-tuned CUDA.

The impact of this design is twofold. Practically, TIR+ lowers the barrier for writing optimized kernels: developers can express complex normalization, GEMM, attention, and fused communication–computation kernels in hundreds rather than thousands of lines, while still matching or surpassing expert baselines. This enables faster iteration, quicker adaptation to new hardware, and broader accessibility of high-performance kernel development. For systems researchers, TIR+ demonstrates that compiler abstractions can directly capture modern GPU execution patterns, including tile-level scheduling, tensor layouts, and inter-GPU communication, offering a framework that is both analyzable and portable.

Empirical results validate this impact. TIR+ matches cuBLAS on GEMM, outperforms FlashInfer in latency-sensitive RMSNorm, and delivers up to 40% speedups for fused distributed kernels. At the same time, certain cases remain where expert-tuned implementations retain an advantage, such as throughput-oriented attention. These gaps highlight opportunities for future optimization, particularly through adaptive scheduling and automated tuning.

More broadly, the layered design of TIR+ opens new directions for kernel development. By serving as a foundation for higher-level DSLs, megakernel fusion, or even AI-assisted scheduling, TIR+ creates a pathway toward more automated and accessible performance engineering. The broader significance is clear: by making high-performance kernels easier to build, TIR+ can accelerate innovation in AI systems, reduce time-to-solution for new models, and democratize access to efficient use of cutting-edge hardware.

Bibliography

[1] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023. URL https://arxiv.org/abs/2305.13245. 6.2.3

[2] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020. URL https://arxiv.org/abs/2005.14165. 6.1

[3] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova,

Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Ning Zhang, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vítor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippos Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai

Wu, Kam Hou U, Karan Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Miao Liu, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Summer Deng, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaojian Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yu Zhao, Yuchen Hao, Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, Zhiwei Zhao, and Zhiyu Ma. The llama 3 herd of models, 2024. URL `https://arxiv.org/abs/2407.21783`. 6.1

[4] Bastian Hagedorn, Bin Fan, Hanfeng Chen, Cris Cecka, Michael Garland, and Vinod Grover. Graphene: An ir for optimized tensor computations on gpus. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 302–313, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399180. doi: 10.1145/3582016.3582018. URL `https://doi.org/10.1145/3582016.3582018`. 4.1.1

[5] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang,

Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts, 2024. URL https://arxiv.org/abs/2401.04088. 6.1

[6] NVIDIA Corporation. CUB: Cuda unbound – cuda core compute libraries. https://nvidia.github.io/cccl/cub/. Accessed: 2025-08-19. 4.1.1

[7] Benjamin F. Spector, Simran Arora, Aaryan Singhal, Daniel Y. Fu, and Christopher Ré. Thunderkittens: Simple, fast, and adorable ai kernels, 2024. URL https://arxiv.org/abs/2410.20399. 2.2, 3.3

[8] Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, Johan Ferret, Peter Liu, Pouya Tafti, Abe Friesen, Michelle Casbon, Sabela Ramos, Ravin Kumar, Charline Le Lan, Sammy Jerome, Anton Tsitsulin, Nino Vieillard, Piotr Stanczyk, Sertan Girgin, Nikola Momchev, Matt Hoffman, Shantanu Thakoor, Jean-Bastien Grill, Behnam Neyshabur, Olivier Bachem, Alanna Walton, Aliaksei Severyn, Alicia Parrish, Aliya Ahmad, Allen Hutchison, Alvin Abdagic, Amanda Carl, Amy Shen, Andy Brock, Andy Coenen, Anthony Laforge, Antonia Paterson, Ben Bastian, Bilal Piot, Bo Wu, Brandon Royal, Charlie Chen, Chintu Kumar, Chris Perry, Chris Welty, Christopher A. Choquette-Choo, Danila Sinopalnikov, David Weinberger, Dimple Vijaykumar, Dominika Rogozińska, Dustin Herbison, Elisa Bandy, Emma Wang, Eric Noland, Erica Moreira, Evan Senter, Evgenii Eltyshev, Francesco Visin, Gabriel Rasskin, Gary Wei, Glenn Cameron, Gus Martins, Hadi Hashemi, Hanna Klimczak-Plucińska, Harleen Batra, Harsh Dhand, Ivan Nardini, Jacinda Mein, Jack Zhou, James Svensson, Jeff Stanway, Jetha Chan, Jin Peng Zhou, Joana Carrasqueira, Joana Iljazi, Jocelyn Becker, Joe Fernandez, Joost van Amersfoort, Josh Gordon, Josh Lipschultz, Josh Newlan, Ju yeong Ji, Kareem Mohamed, Kartikeya Badola, Kat Black, Katie Millican, Keelin McDonell, Kelvin Nguyen, Kiranbir Sodhia, Kish Greene, Lars Lowe Sjoesund, Lauren Usui, Laurent Sifre, Lena Heuermann, Leticia Lago, Lilly McNealus, Livio Baldini Soares, Logan Kilpatrick, Lucas Dixon, Luciano Martins, Machel Reid, Manvinder Singh, Mark Iverson, Martin Görner, Mat Velloso, Mateo Wirth, Matt Davidow, Matt Miller, Matthew Rahtz, Matthew Watson, Meg Risdal, Mehran Kazemi, Michael Moynihan, Ming Zhang, Minsuk Kahng, Minwoo Park, Mofi Rahman, Mohit Khatwani, Natalie Dao, Nenshad Bardoliwalla, Nesh Devanathan, Neta Dumai, Nilay Chauhan, Oscar Wahltinez, Pankil Botarda, Parker Barnes, Paul Barham, Paul Michel, Pengchong Jin, Petko Georgiev, Phil Culliton, Pradeep Kuppala, Ramona Comanescu, Ramona Merhej, Reena Jana, Reza Ardeshir Rokni, Rishabh Agarwal, Ryan Mullins, Samaneh Saadat, Sara Mc Carthy, Sarah Cogan, Sarah Perrin, Sébastien M. R. Arnold, Sebastian Krause, Shengyang Dai, Shruti Garg, Shruti Sheth, Sue Ronstrom, Susan Chan, Timothy Jordan, Ting Yu, Tom Eccles, Tom Hennigan, Tomas Kocisky, Tulsee Doshi, Vihan Jain, Vikas Yadav, Vilobh Meshram, Vishal Dharmadhikari, Warren Barkley, Wei Wei, Wenming Ye, Woohyun Han, Woosuk Kwon, Xiang Xu, Zhe Shen, Zhitao Gong, Zichuan Wei, Victor Cotruta, Phoebe Kirk, Anand Rao, Minh Giang, Ludovic Peran, Tris Warkentin, Eli Collins, Joelle Barral, Zoubin Ghahramani, Raia Hadsell, D. Sculley, Jeanine Banks, Anca Dragan, Slav Petrov, Oriol Vinyals, Jeff Dean, Demis Hassabis, Koray Kavukcuoglu, Clement Farabet, Elena Buchatskaya, Se-

bastian Borgeaud, Noah Fiedel, Armand Joulin, Kathleen Kenealy, Robert Dadashi, and Alek Andreev. Gemma 2: Improving open language models at a practical size, 2024. URL https://arxiv.org/abs/2408.00118. 6.1

[9] Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. CUTLASS, January 2023. URL https://github.com/NVIDIA/cutlass. 2.2

[10] Philippe Tillet, Hsiang-Tsung Kung, and David D. Cox. Triton: an intermediate language and compiler for tiled neural network computations. *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2019. URL https://api.semanticscholar.org/CorpusID:184488182. 2.1, 2.2, 4.1.1, 6.1

[11] Lei Wang, Yu Cheng, Yining Shi, Zhengju Tang, Zhiwen Mo, Wenhao Xie, Lingxiao Ma, Yuqing Xia, Jilong Xue, Fan Yang, and Zhi Yang. Tilelang: A composable tiled programming model for ai systems, 2025. URL https://arxiv.org/abs/2504.17577. 2.2

[12] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, Ruoming Pang, Noam Shazeer, Shibo Wang, Tao Wang, Yonghui Wu, and Zhifeng Chen. Gspmd: General and scalable parallelization for ml computation graphs, 2021. URL https://arxiv.org/abs/2105.04663. 4.1.2

[13] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388. 6.1

[14] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. Flashinfer: Efficient and customizable attention engine for llm inference serving, 2025. URL https://arxiv.org/abs/2501.01005. 6.2.1, 6.2.3

[15] Size Zheng, Wenlei Bao, Qi Hou, Xuegui Zheng, Jin Fang, Chenhui Huang, Tianqi Li, Haojie Duanmu, Renze Chen, Ruifan Xu, Yifan Guo, Ningxin Zheng, Ziheng Jiang, Xinyi Di, Dongyang Wang, Jianxi Ye, Haibin Lin, Li-Wen Chang, Liqiang Lu, Yun Liang, Jidong Zhai, and Xin Liu. Triton-distributed: Programming overlapping kernels on distributed ai systems with the triton compiler, 2025. URL https://arxiv.org/abs/2504.19442. 6.2.4