Hexcute: A Tile-based Programming Language with Automatic Layout and Task-Mapping Synthesis

Xiao Zhang[*]
University of Toronto
Toronto, Canada
zita.zhang@mail.utoronto.ca

Yaoyao Ding[*]
University of Toronto
Toronto, Canada
yaoyao@cs.toronto.edu

Yang Hu
CentML Inc.
Toronto, Canada
yang.hu@centml.ai

Gennady Pekhimenko[*]
University of Toronto
Toronto, Canada
pekhimenko@cs.toronto.edu

Abstract

Deep learning (DL) workloads mainly run on accelerators like GPUs. Recent DL quantization techniques demand a new matrix multiplication operator with mixed input data types, further complicating GPU optimization. Prior high-level compilers like Triton lack the expressiveness to implement key optimizations like fine-grained data pipelines and hardware-friendly memory layouts for these operators, while low-level programming models, such as Hidet, Graphene, and CUTLASS, require significant programming efforts. To balance expressiveness with engineering effort, we propose Hexcute, a tile-based programming language that exposes shared memory and register abstractions to enable fine-grained optimization for these operators. Additionally, Hexcute leverages task mapping to schedule the GPU program, and to reduce programming efforts, it automates layout and task mapping synthesis with a novel type-inference-based algorithm. Our evaluation shows that Hexcute generalizes to a wide range of DL operators, achieves 1.7-11.28× speedup over existing DL compilers for mixed-type operators, and brings up to 2.91× speedup in the end-to-end evaluation.

1 Introduction

Deep learning (DL) has demonstrated remarkable power across various applications, including image recognition [12], natural language processing [32], and autonomous driving [41]. Recently, large language models (LLMs) [1] have shown astonishing generalization capabilities, elevating the influence of DL to unprecedented levels. Deep neural networks (DNNs) are composed of tensor algebra operations, and both training and deployment require enormous computational and memory resources. To speed up these workloads, most DNNs run on accelerators, such as TPUs [16], GPUs [15], and NPUs [20]. Among these, GPUs are the most accessible accelerators for both research and industry use.

Programming efficient GPU programs is challenging due to the complex architectures and execution models of modern GPUs. GPUs consist of streaming multiprocessors (SMs) on NVIDIA devices or compute units (CUs) on AMD devices, both of which employ the Single Instruction Multiple Threads (SIMT) model [22]. Their memory hierarchy includes global memory (DRAM), a shared L2 cache, per-SM programmable L1 caches (shared memory), and fast register memory managed by threads. Memory access requires specific optimizations: global memory must be coalesced [4], and shared memory accesses must avoid bank conflicts [33]. Modern GPUs also integrate specialized units, such as NVIDIA's Tensor Cores or AMD's Matrix Cores, to accelerate deep learning. These units reshape the programming model by executing instructions collectively within thread groups (warps or wavefronts) rather than individually, complicating tensor programming by requiring coordinated thread behavior.

Various DL compilers [7, 40, 42, 44] are proposed to address the complexity of optimizing tensor programs. Most rely on loop transformations [2] to optimize tensor computations but struggle to express key techniques for Tensor Cores, such as software pipelining for latency hiding. To overcome this limitation, Hidet [6] proposes a task-mapping programming paradigm to express these fine-grained optimizations. It converts the scheduling of a GPU program into a problem of finding the optimal task mappings and efficient memory layouts. However, task mappings and layouts must be manually specified, which demands intimate knowledge of GPU architecture and considerable engineering effort. Furthermore, specifying task mappings is error-prone due to the interdependency of different task mappings, and inconsistent task mappings can cause the program to function improperly. Similarly, Graphene [11] adopts an approach that leverages a concept called Layout to represent task mappings.

Weight-only quantization [9, 18] for LLMs intensifies the complexity of tensor programming. While quantization compresses weights and accelerates computation, it introduces a new kind of matrix multiplication with mixed input types. Manually implementing these operators is challenging due to the proliferation of low-precision data types. In addition, existing compilers provide limited support for low-precision computations. For example, Triton [31] struggles to deliver

[*] Also with CentML Inc. and Vector Institute, Toronto, Canada

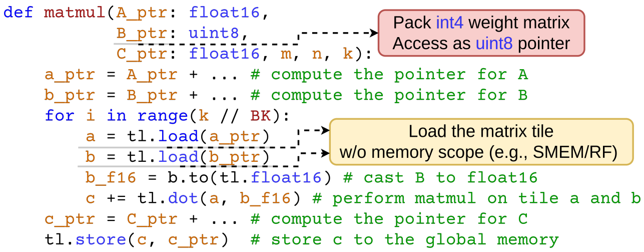
decent performance because its layout system cannot express the hardware-friendly shared memory layouts required for low-precision data types and lacks hardware abstractions, such as shared memory and register, to implement fine-grained data pipelines for these operators. Ladder [35] extends TVM with new primitives to support low-precision computation. However, TVM's loop-oriented transformations make it challenging to express these key techniques, often resulting in suboptimal performance.

To express the key optimizations for mixed-type operators and reduce the programming efforts of fine-grained programming models like Hidet [6] and Graphene [11], we propose *a tile-based programming language with shared memory and register abstractions*. This allows us to express the efficient data pipeline for low-precision computations. We leverage task mappings [6] to schedule GPU programs. To remove the need for manual task mapping specifications, we propose *a novel type system for automatically synthesizing task mappings and layouts of register and shared tensors*. In our design, the data distribution across threads is considered part of the tensor types. Then, we build constraints for these types and design a type-inference-based [10] algorithm to synthesize task mappings and layouts. This technique guarantees the correctness and performance of generated code.

We implement the proposed tile-based programming language, called **Hexcute**, to optimize a wide range of operators in DL training and inference, including mixed-type operations. We integrate Hexcute into vLLM [17] and demonstrate superior performance compared to existing state-of-the-art hand-written and compiler-generated kernels.

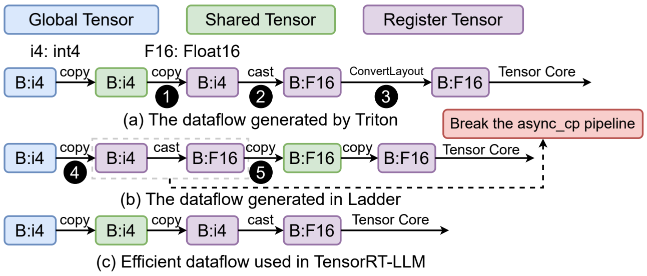
We conclude our contributions as follows:

- We propose a new tile-based programming language that exposes shared memory and registers abstraction, allowing us to express fine-grained optimizations for low-precision computations.
- We consider the data distribution across threads as part of tensor types. By building constraints on tensor types for each tile-level operation, we formalize the task mapping and layout synthesis problem as a constraint programming problem. We then design a type-inference-based algorithm to automatically synthesize them, which guarantees the correctness and performance of the generated GPU program.
- We implement Hexcute with the above ideas and integrate it into vLLM [17]. Our evaluation demonstrates that Hexcute generalizes to a wide range of DL operators, achieves 1.7-11.28× speedup over existing state-of-the-art DL compilers for mixed-type operators, and brings up to 2.9× in the end-to-end evaluation.



```
def matmul(A_ptr: float16,
           B_ptr: uint8,
           C_ptr: float16, m, n, k):
    a_ptr = A_ptr + ... # compute the pointer for A
    b_ptr = B_ptr + ... # compute the pointer for B
    for i in range(k // BK):
        a = tl.load(a_ptr)
        b = tl.load(b_ptr)
        b_f16 = b.to(tl.float16) # cast B to float16
        c += tl.dot(a, b_f16) # perform matmul on tile a and b
    c_ptr = C_ptr + ... # compute the pointer for C
    tl.store(c, c_ptr)  # store c to the global memory
```

Figure 1. A FP16×INT4 matmul written with Triton, where activation matrix A is of the float16 data type and weight matrix B consists of 4-bit integers.



(a) The dataflow generated by Triton

(b) The dataflow generated in Ladder

(c) Efficient dataflow used in TensorRT-LLM

Figure 2. The dataflow of mixed-type matmul kernels

2 Background and Motivation

2.1 Motivational Example: Mixed-Type Operation

Tensor Core instructions on modern GPUs have dramatically accelerated deep learning and reshaped the SIMT programming model [21]. Instead of uniform thread execution, multiple threads now work together on computations and data transfers, complicating GPU programming. Additionally, computational capacity has surged far faster than memory bandwidth. This imbalance demands sophisticated pipelining of data movement across memory hierarchies to hide latency [14, 38]. Recent weight-only quantization techniques for large language models (LLMs) further complicate tensor programming on GPUs. They introduce a new operator called mixed-type matmul, where weight and activation matrices use different data types. Manually supporting these new operations is becoming unsustainable due to the explosive combinations of data types and quantization schemes [5, 9, 18, 19, 28, 34, 36, 37].

2.2 High-level Programming Models without Sufficient Expressiveness

The challenge in mixed-type matmuls is expressing fine-grained software pipelining and hardware-friendly memory layouts. Existing compilers such as Triton [31] and Ladder [35] fail to address the challenge. Triton [31] is a high-level programming model that defines the behavior of an entire thread block and works on tiles instead of scalars. Figure 1 shows a FP16×INT4 matmul written in Triton. Triton-generated code is inefficient because it inserts a ConvertLayout

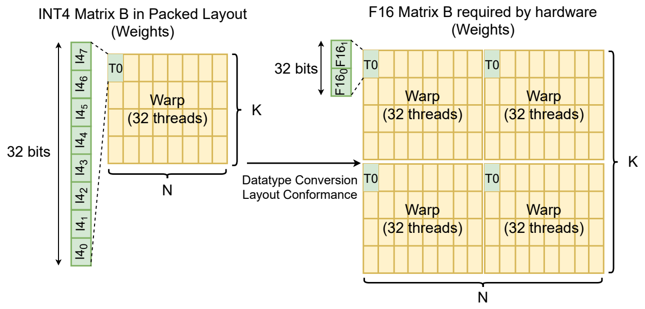


Figure 3. The INT4 packed layout designed in TensorRT-LLM ensures layout conformance when converting INT4 weights to F16 weights.

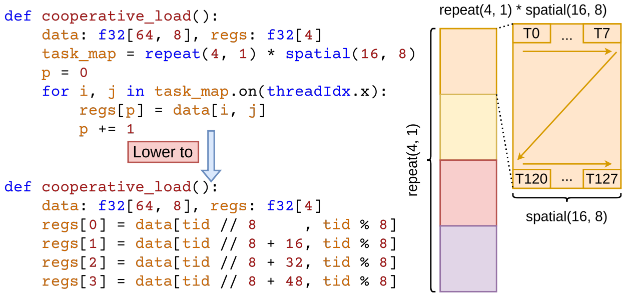
operation ❸ before feeding data to Tensor Core units, as shown in Figure 2 (a). This extra step exchanges register data through shared memory and forces synchronization within the thread block, degrading performance. Triton inserts ❸ after the type casting ❷ because Tensor Core dictates its inputs to be distributed across threads in a specific manner, while the INT4 weights loaded from ❶ do not meet this requirement. Figure 2 (b) shows the dataflow in Ladder, where the weight matrix is dequantized during its transfer from global to shared memory (from ❹ to ❺) The dataflow is also suboptimal because it breaks the asynchronous copy pipeline. Figure 2 (c) illustrates the efficient dataflow in TensorRT-LLM [25]. TensorRT-LLM designs a packed layout shown in Figure 3 that ensures layout conformity when converting the data type. As a result, the dataflow avoids adding extra layout conversion or breaking the asynchronous pipeline. Triton and Ladder generate inefficient pipelines because their layout systems lack the flexibility to express the packed layout and select suboptimal layouts with extra overheads. In addition, Triton does not expose hardware hierarchies like shared memory and register, which also hinders developers from expressing the dataflow explicitly. This limitation becomes severe in complex workloads like mixture-of-experts (MoE) layers [29] on Hopper architecture, where our evaluation shows Triton's approach incurs up to 10× higher latency increase. These shortcomings highlight the need for a more expressive programming model.

2.3 Programming Models with Fine-grained Control

Task Mapping. To express increasingly complex GPU optimizations, Hidet [6] introduces the task-mapping programming model, allowing users to define computational tasks via mapping functions. This model conceptualizes a GPU program as a sequence of thread-block-level operations, each specified by a task mapping. For example, Figure 4 (a) illustrates a `cooperative_load` function defined with constructs such as `repeat` and `spatial`. The task mapping converts thread indices to the task items processed by each thread, as displayed in the generated code at the bottom of Figure 4 (a).

Other works, such as Graphene [11], take a similar approach to optimizing tensor programs by leveraging a concept of **Layout**. This concept is also used in CuTe [24], a C++ tensor algebra library, to design sophisticated mappings.

Layout. Layouts like row-major and column-major usually define tensor storage in memory with two tuples: *shape*, specifying tensor dimensions, and *strides*, mapping tensor coordinates to memory addresses. CuTe generalizes the layout as a function mapping integers to integers and introduces hierarchical layouts for more complex representations. For instance, Figure 5 (a) illustrates a row-major-interleaved layout, `A`, that splits the row axis of a row-major layout into two sub-dimensions with different strides. This flexibility allows expressing the packed layout in Figure 3 as `((2,2,2,4),(8,)):((1,8,128,2),(16,))`. Layout accepts various input forms (e.g., one-dimensional, two-dimensional, or nested coordinates) and produces the same integer output as shown in Figure 5 (b). Figure 5 (c) illustrates the computation of `A(6)`, where a one-dimensional coordinate is decomposed into multi-dimensional coordinates and then combined with strides via a dot product to compute the final output.



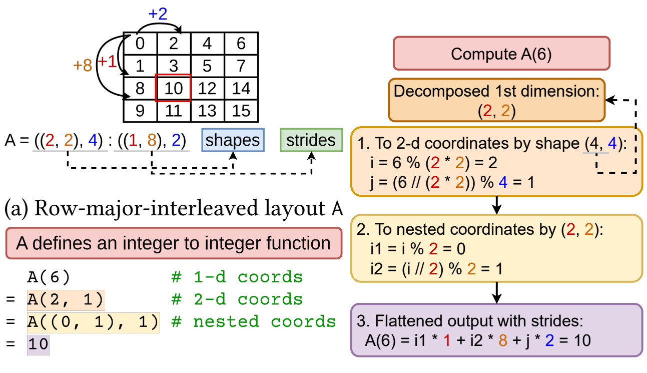
```
def cooperative_load():
    data: f32[64, 8], regs: f32[4]
    task_map = repeat(4, 1) * spatial(16, 8)
    p = 0
    for i, j in task_map.on(threadIdx.x):
        regs[p] = data[i, j]
        p += 1
```

```
def cooperative_load():
    data: f32[64, 8], regs: f32[4]
    regs[0] = data[tid // 8     , tid % 8]
    regs[1] = data[tid // 8 + 16, tid % 8]
    regs[2] = data[tid // 8 + 32, tid % 8]
    regs[3] = data[tid // 8 + 48, tid % 8]
```

(a) Load defined by task-mapping (b) The task-mapping

Figure 4. A thread block of 128 threads copies a 64×8 tile from shared memory to register files. Organized as a 16×8 grid, the threads copy the data in four iterations. Figure (a) shows the `cooperative_load` function implemented with task mapping constructs, while Figure (b) illustrates how task mapping defines the collective data movement.

Thread-Value Layout. CuTe introduces the *thread-value Layout* (TV layout), which determines how data in a tile is distributed across threads. A TV layout is defined as a function: $f : (\texttt{tid}, \texttt{vid}) \mapsto \texttt{flattened\_position}$, where `tid` is the thread ID and `vid` is the index of a value within the thread's local array. For example, Figure 6 (a) shows a TV layout `B` mapping eight threads, with each holding four values, to a 4 × 8 tile. Figure 6 (b) visualizes the data distribution of the tile with annotated thread and value IDs. Figure 6 (c) illustrates the function defined by `B`, where the numbers in the boxes represent the function's outputs, i.e., the column-major flattened indices within the tile. Finally, Figure 6 (d)



0	2	4	6
1	3	5	7
8	10	12	14
9	11	13	15

A = ((2, 2), 4) : ((1, 8), 2)

(a) Row-major-interleaved layout A

A defines an integer to integer function

```
  A(6)           # 1-d coords
= A(2, 1)        # 2-d coords
= A((0, 1), 1)   # nested coords
= 10
```

(b) Layout is a function

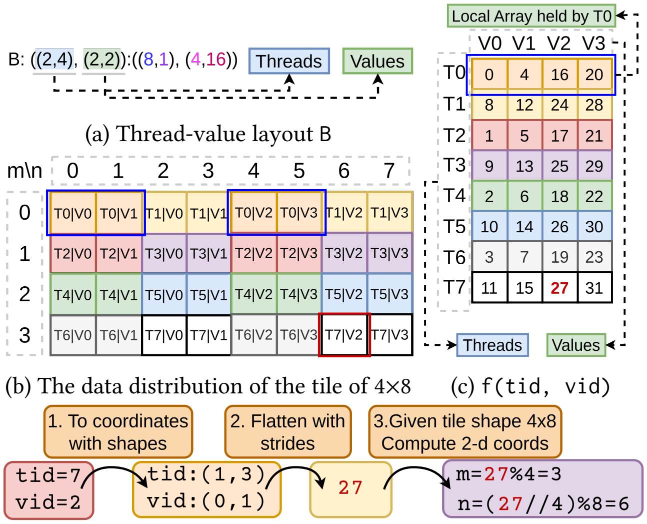
1. To 2-d coordinates by shape (4, 4):
   i = 6 % (2 * 2) = 2
   j = (6 // (2 * 2)) % 4 = 1

2. To nested coordinates by (2, 2):
   i1 = i % 2 = 0
   i2 = (i // 2) % 2 = 1

3. Flattened output with strides:
   A(6) = i1 * 1 + i2 * 8 + j * 2 = 10

(c) Compute A(6)

Figure 5. Example: row-major interleaved layout A. Figure (a) visualizes the function defined by A, with the integers in the box indicating the function's outputs.



B: ((2,4), (2,2)):((8,1), (4,16))

(a) Thread-value layout B

m\n	0	1	2	3	4	5	6	7
0	T0|V0	T0|V1	T1|V0	T1|V1	T0|V2	T0|V3	T1|V2	T1|V3
1	T2|V0	T2|V1	T3|V0	T3|V1	T2|V2	T2|V3	T3|V2	T3|V3
2	T4|V0	T4|V1	T5|V0	T5|V1	T4|V2	T4|V3	T5|V2	T5|V3
3	T6|V0	T6|V1	T7|V0	T7|V1	T6|V2	T6|V3	T7|V2	T7|V3

(b) The data distribution of the tile of 4×8

	V0	V1	V2	V3
T0	0	4	16	20
T1	8	12	24	28
T2	1	5	17	21
T3	9	13	25	29
T4	2	6	18	22
T5	10	14	26	30
T6	3	7	19	23
T7	11	15	27	31

(c) f(tid, vid)

1. To coordinates with shapes

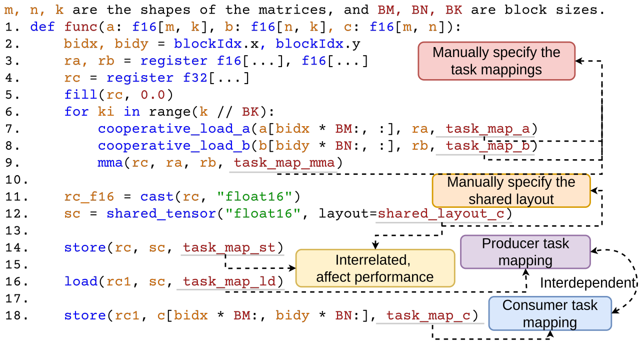
2. Flatten with strides

3.Given tile shape 4x8 Compute 2-d coords

```
tid=7    tid:(1,3)    27    m=27%4=3
vid=2    vid:(0,1)          n=(27//4)%8=6
```

(d) Computing f(7, 2)

Figure 6. Example: thread-value layout B


Manually specify the task mappings
Manually specify the shared layout
Interrelated, affect performance
Producer task mapping
Interdependent
Consumer task mapping

```
m, n, k are the shapes of the matrices, and BM, BN, BK are block sizes.
1. def func(a: f16[m, k], b: f16[n, k], c: f16[m, n]):
2.     bidx, bidy = blockIdx.x, blockIdx.y
3.     ra, rb = register f16[...], f16[...]
4.     rc = register f32[...]
5.     fill(rc, 0.0)
6.     for ki in range(k // BK):
7.         cooperative_load_a(a[bidx * BM:, :], ra, task_map_a)
8.         cooperative_load_b(b[bidy * BN:, :], rb, task_map_b)
9.         mma(rc, ra, rb, task_map_mma)
10.
11.    rc_f16 = cast(rc, "float16")
12.    sc = shared_tensor("float16", layout=shared_layout_c)
13.
14.    store(rc, sc, task_map_st)
15.
16.    load(rc1, sc, task_map_ld)
17.
18.    store(rc1, c[bidx * BM:, bidy * BN:], task_map_c)
```

Figure 7. Example: A GEMM kernel written with Hidet

demonstrates evaluating $f(7, 2)$: thread 7's second register corresponds to flattened position 27 in the tile, which is then converted to two-dimensional coordinates based on the tile shape. The TV layout, which defines the relationship between threads and their processed tasks, is functionally equivalent to task mapping.

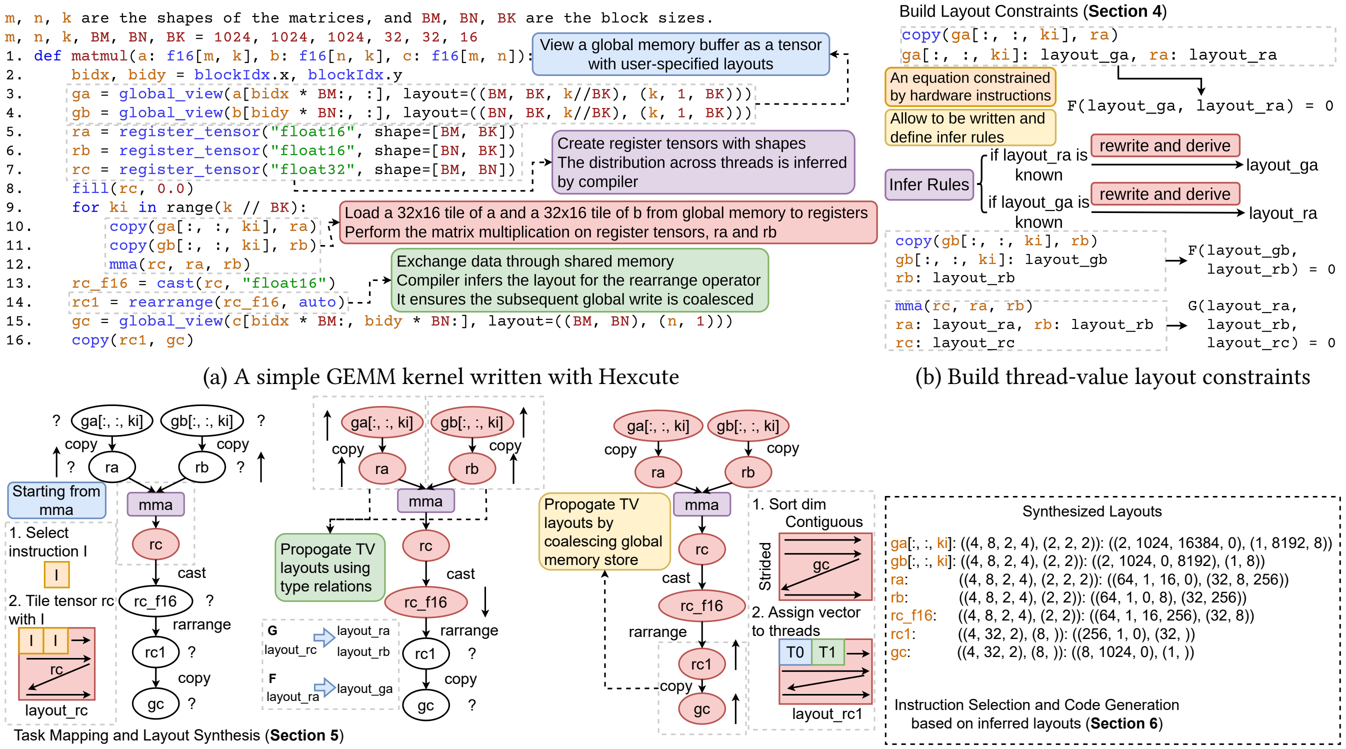
Layout Algebra. CuTe treats layouts as mathematical functions, enabling algebraic operations like composition and inversion. Composition $R = A \circ B$ is defined as $R(c) = A(B(c))$, where c is a coordinate in B's domain. A layout A is invertible if there exists a layout A^{-1} such that $A^{-1}(A(c)) = c$. For non-invertible layouts, we can define left and right inverses, L and R, which satisfy $(L \circ A)(c_1) = c_1$ and $(A \circ R)(c_2) = c_2$ for coordinates c_1 and c_2 in the corresponding domains.

Limitation. Despite the expressiveness, works like Hidet [6], Graphene [11], and CuTe [30], lack an automated mechanism for synthesizing layouts and task mappings. Programmers must manually specify these for each operation, which is error-prone and inefficient. For instance, Figure 7 illustrates a complete program written with the task-mapping programming model, where users manually define the task mapping for each block-level operation (e.g., **Lines** 7–9) and the shared memory layout (e.g., **Line** 12). Task mappings can be interdependent due to the operation producer-consumer relationship. For example, the `store` operation (**Line** 18) consumes data from the `load` operation (**Line** 16). The inconsistency of these two operations can lead to incorrect code. Additionally, the task mapping (**Line** 12) and the shared memory layout (**Line** 14) are interrelated and affect the performance together. Inefficient mappings and layouts can cause bank conflicts in shared memory accesses, degrading performance.

3 System Overview

3.1 Key Ideas

To bridge the gap between high-level programming models like Triton [31] and low-level programming models such as Hidet [6], Graphene [11], and CUTLASS [30], we propose a new tile-based programming language called Hexcute. Similar to Triton [31], Hexcute is based on Python and extends Hidet [6] with tile-level primitives. Hexcute *exposes hardware hierarchies, such as shared memory and registers, for explicit fine-grained pipeline orchestration*. To reduce the verbosity and error-proneness of low-level programming models, Hexcute *automates layout and task mapping synthesis using a type-inference-based algorithm*. Hexcute treats data distribution across threads as part of tensor types and represents them with thread-value layouts. Hexcute then constructs type relations (constraints) of layouts through algebraic operations (*Section* 4), which forms a constraint programming problem. Hexcute solves the constraint system through a type-inference-based algorithm in *Section* 5. After inferring thread-value layouts, Hexcute reuses the constraints to select optimal instructions and generate Hidet IR (*Section* 6). The



```
m, n, k are the shapes of the matrices, and BM, BN, BK are the block sizes.
m, n, k, BM, BN, BK = 1024, 1024, 1024, 32, 32, 16
1.  def matmul(a: f16[m, k], b: f16[n, k], c: f16[m, n]):
2.      bidx, bidy = blockIdx.x, blockIdx.y
3.      ga = global_view(a[bidx * BM:, :], layout=((BM, BK, k//BK), (k, 1, BK)))
4.      gb = global_view(b[bidy * BN:, :], layout=((BN, BK, k//BK), (k, 1, BK)))
5.      ra = register_tensor("float16", shape=[BM, BK])
6.      rb = register_tensor("float16", shape=[BN, BK])
7.      rc = register_tensor("float32", shape=[BM, BN])
8.      fill(rc, 0.0)
9.      for ki in range(k // BK):
10.         copy(ga[:, :, ki], ra)
11.         copy(gb[:, :, ki], rb)
12.         mma(rc, ra, rb)
13.     rc_f16 = cast(rc, "float16")
14.     rc1 = rearrange(rc_f16, auto)
15.     gc = global_view(c[bidx * BM:, bidy * BN:], layout=((BM, BN), (n, 1)))
16.     copy(rc1, gc)
```

View a global memory buffer as a tensor with user-specified layouts
Create register tensors with shapes
The distribution across threads is inferred by compiler
Load a 32x16 tile of a and a 32x16 tile of b from global memory to registers
Perform the matrix multiplication on register tensors, ra and rb
Exchange data through shared memory
Compiler infers the layout for the rearrange operator
It ensures the subsequent global write is coalesced

(a) A simple GEMM kernel written with Hexcute

Build Layout Constraints (**Section 4**)

```
copy(ga[:, :, ki], ra)
ga[:, :, ki]: layout_ga, ra: layout_ra
                              F(layout_ga, layout_ra) = 0

copy(gb[:, :, ki], rb)
gb[:, :, ki]: layout_gb          →  F(layout_gb,
rb: layout_rb                          layout_rb) = 0

mma(rc, ra, rb)                     G(layout_ra,
ra: layout_ra, rb: layout_rb     →    layout_rb,
rc: layout_rc                          layout_rc) = 0
```

(b) Build thread-value layout constraints

(c) Task-mapping and layout synthesis

Synthesized Layouts

```
ga[:, :, ki]: ((4, 8, 2, 4), (2, 2, 2)): ((2, 1024, 16384, 0), (1, 8192, 8))
gb[:, :, ki]: ((4, 8, 2, 4), (2, 2)): ((2, 1024, 0, 8192), (1, 8))
ra:           ((4, 8, 2, 4), (2, 2, 2)): ((64, 1, 16, 0), (32, 8, 256))
rb:           ((4, 8, 2, 4), (2, 2)): ((64, 1, 0, 8), (32, 256))
rc_f16:       ((4, 8, 2, 4), (2, 2)): ((64, 1, 16, 256), (32, 8))
rc1:          ((4, 32, 2), (8, )): ((256, 1, 0), (32, ))
gc:           ((4, 32, 2), (8, )): ((8, 1024, 0), (1, ))
```

Instruction Selection and Code Generation based on inferred layouts (Section 6)

(d) Instruction selection and code generation

Figure 8. Hexcute System Overview

generated IR is subsequently compiled into efficient GPU code using existing compiler passes.

3.2 Example: A Simple GEMM Kernel

Figure 8 (a) illustrates a simple GEMM kernel that multiplies FP16 matrices a, b, and c using tile-level primitives in Hexcute. In this kernel, the matrix multiplication is tiled so that each thread block is responsible for computing a $BM \times BN$ tile of the matrix c.

Lines 3-4 define global memory tensors with user-specified layouts. Specifically, **Line** 3 interprets a sub-tensor from matrix a as a three-dimensional tensor of shape $(BM, BK, k/BK)$ with strides $(k, 1, BK)$. Similarly, **Line** 4 reshapes a sub-tensor of b into shape $(BN, BK, k/BK)$. The decomposition converts the tensor into an iterator, which allows efficient address calculation in the following loop. **Lines** 5-7 create register tensors ra, rb, and rc, with the compiler automatically inferring the data distribution for them. In **Line** 8, the accumulator rc is initialized to zero.

The loop in **Lines** 9-12 iterates over k/BK tiles, loading tiles of a and b into registers (**Lines** 10-11) and performing matrix multiply-accumulate (mma) operations. After the loop, the FP32 accumulator rc is cast to FP16 in **Line** 14 and rearranged via shared memory in **Line** 15 to ensure coalesced global memory writes. Here, the rearrange operator is annotated with the auto keyword. This hints to the compiler to determine the optimal register tensor layout to ensure coalescing the subsequent store in **Line** 16. Finally, **Lines** 15–16 store the results into matrix c.

```
# Create a shared tensor with the specified shapes
sa = shared_tensor("float16", shape=[BM, BK])

# Create a shared tensor with the specified layout
sa = shared_tensor("float16", layout=((BM, BK), (BK, 1)))

# Create a shared tensor of 4-bit integers
sa = shared_tensor("uint4b", shape=[BM, BK])
```

Figure 9. Create tensors in shared memory

Hexcute supports two strategies for creating shared memory tensors, as shown in Figure 9: either by explicitly declaring the shape or specifying the layout. *We choose CuTe-layout [24] to represent the shared memory layout so that Hexcute can express the packed layout in Figure 3.* In both cases, the compiler automatically resolves shared memory bank conflicts through swizzling [26]. Low-precision tensors (e.g., 4-bit integers) are supported as first-class citizens (Figure 9). Internally, register tensors use *thread-value layouts* to represent distributed data across threads. Table 1 summarizes the tile-level operation semantics, and Appendix A includes a formal definition of the operator syntax.

Table 1. Hexcute tile-level primitive semantics

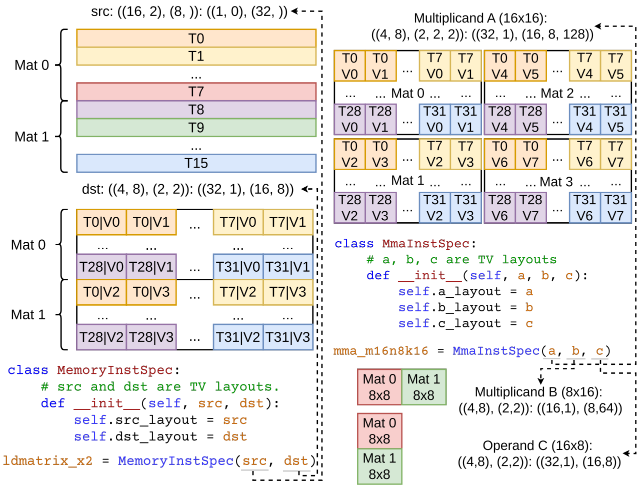
Primitives	Semantics
`global_view(b, l)`	View a global memory buffer `b` as a tensor with layout `l`
`register_tensor(t, s)`	Create a tensor with data type `t` and shape `s` in registers
`shared_tensor(t, s)` or `shared_tensor(t, l)`	Create a tensor with data type `t` and shape `s` or layout `l` in shared memory
`copy(a, b)`	Copy data from tensor `a` to tensor `b`
`mma(c, a, b)`	Perform matmul on tensors `a`, `b`, and `c`
`cast(a, t)`	Cast the tensor `a` to another data type `t`
`rearrange(a, l)`	Redistribute register tensor across the threads to layout `l` via shared memory
`elementwise(...)`	Elementwise operations on tensors
`reduce(a, d)`	Reduce the tensor `a` along the dimension `d`

3.3 Task Mapping and Layout Synthesis

Hexcute provides a programming model that allows fine-grained control comparable to Hidet[6], Graphene[11], and CUTLASS[30], while removing the need for error-prone specifications. Unlike the above works, Hexcute automatically infers task mappings for tile-level operations and layouts for shared/register tensors. The key insight is that *the thread-value layout represents data distribution across threads and can become part of the tensor types*. Hexcute then builds type constraints (relations) on the thread-value layouts, as detailed in Section 4.2. The thread-value layout constraint is conceptually an equation of thread-value layouts, which can be rewritten to define infer rules, as illustrated in Figure 8 (b). For example, the constraint of the copy in **Line** 9 in Figure 8 (a) is formalized as: F (layout_ga, layout_ra) = 0, where `layout_ga` and `layout_ra` denote the thread-value layouts of `ga[:, :, ki]` and `ra`.

Hexcute employs a type-inference-based algorithm to synthesize distributed register/shared memory layouts and task mappings. In the GEMM example, the compiler starts from the `mma` operation. It selects a Tensor Core instruction I to tile the tensor `rc`, which determines the thread-value layout `layout_rc`, as illustrated on the left side of Figure 8 (c). Using the constraint G of the `mma` operation, it derives the thread-value layouts for `ra` and `rb`. Next, applying the constraint F from the `copy` operation, it determines the layouts for `ga[:, :, ki]` and `gb[:, :, ki]`, as illustrated in the middle of Figure 8 (c). Thus, the compiler incrementally propagates the thread-value layouts.

For the `rearrange` operator, which exchanges data through shared memory, no thread-value layout constraints apply to its output `rc1`. However, since `rc1` is eventually stored in global memory, the optimal layout can be determined by coalescing the global memory write in **Line** 18, as illustrated on the right side of Figure 8 (c). After inferring all layouts, the algorithm synthesizes task mappings, generates low-level IR, and compiles the IR into GPU code via existing passes.



```
class MemoryInstSpec:
    # src and dst are TV layouts.
    def __init__(self, src, dst):
        self.src_layout = src
        self.dst_layout = dst

ldmatrix_x2 = MemoryInstSpec(src, dst)
```

```
class MmaInstSpec:
    # a, b, c are TV layouts
    def __init__(self, a, b, c):
        self.a_layout = a
        self.b_layout = b
        self.c_layout = c

mma_m16n8k16 = MmaInstSpec(a, b, c)
```

(a) `ldmatrix` instruction

(b) `mma` instruction

Figure 10. Thread-value layouts specify the semantics of the memory and compute instructions on modern GPUs.

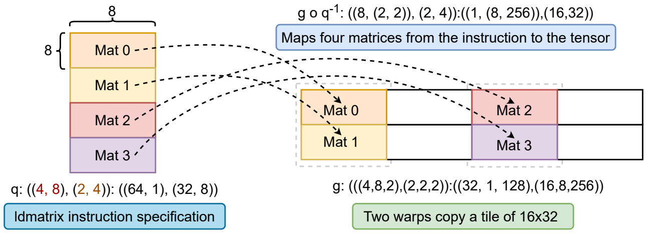

8
8
Mat 0
Mat 1
Mat 2
Mat 3
q: ((4, 8), (2, 4)): ((64, 1), (32, 8))
ldmatrix instruction specification
g o q-1: ((8, (2, 2)), (2, 4)):((1, (8, 256)),(16,32))
Maps four matrices from the instruction to the tensor
Mat 0
Mat 1
Mat 2
Mat 3
g: (((4,8,2),(2,2,2)):((32, 1, 128),(16,8,256))
Two warps copy a tile of 16x32

Figure 11. Composite function $g \circ q^{-1}$

4 Thread-Value Layout Constraints

This section details the construction of thread-value layout constraints. We start by formalizing the behavior of memory and computing instructions.

4.1 Instruction Abstraction

Modern GPU instructions such as `ldmatrix` and `mma` execute memory and compute tasks with coordinate thread groups. We formalize their behavior using *thread-value layouts*.

For example, in the `ldmatrix` instruction in Figure 10 (a), 32 threads collectively load two 8×8 matrices from shared memory. Each thread is responsible for loading one row in the 16×8 matrix, after which the data is redistributed across threads so that each thread holds 4 elements. This behavior can be modeled by two layouts: the layout, `src`, which maps thread-value pairs, (t, v), to logical shared memory addresses, and the layout, `dst`, which describes the distribution of output data across threads. Similarly, thread-value layouts specify the semantics of Tensor Core instructions. For example, The instruction `mma.sync.aligned.m16n8k16.row.col` executes small matrix multiplications with operands distributed across threads in a warp. As shown in Figure 10 (b), a thread-value layout defines the distribution of multiplicand A; the same approach applies to operands B and C. In summary, thread-value layouts fully capture the semantics of these instructions, allowing us to abstract their behavior as shown in the code examples in Figure 10 (a) and (b).

4.2 Thread-Value Layout Constraints

To automate layout and task-mapping synthesis, we first build constraints on the thread-value layout. We illustrate the derivation using a `copy(a, b)` operation.

Composite Mapping Functions. Let f and g denote the thread-value layouts of tensors `a` and `b`, respectively. These layouts are functions that map a thread ID t and a value index v to a logical coordinate in the tensor tile:

$$f : (t, v) \mapsto (m_T, n_T), \quad g : (t, v) \mapsto (m_T, n_T)$$

where m_T and n_T represent the row and column coordinates within the tensor tile.

Suppose a memory instruction I (used to implement the `copy` operation) is specified by its own input and output layouts, denoted by p and q, respectively. These layouts map the same thread-value pair to coordinates in the instruction tile. For instance, we might have:

$$p : (t, v) \mapsto (m_I, m_I), \quad q : (t, v) \mapsto (m_I, n_I)$$

where m_I and n_I denote the row and column coordinates within the instruction tile.

By taking the inverse functions p^{-1} and q^{-1}, we can convert instruction tile coordinates back to a thread-value pair. Composing these inverses with the tensor layouts gives us the composite mappings: $f \circ p^{-1}$ and $g \circ q^{-1}$, which map coordinates from the instruction tile to the corresponding positions in the input and output tensor tiles, respectively. Since the `copy` operation transfers an identical logical region from the input tensor to the output tensor, these composite mappings must be equal: $f \circ p^{-1} = g \circ q^{-1}$.

An Intuitive Example of $g \circ q^{-1}$. We consider the example of an `ldmatrix` instruction with an output layout defined as `q: ((4,8),(2,4)):((64,1),(32,8))`. Suppose the output tensor `b` has a layout defined as `g: ((4,8,2),(2,2,2)) :((32,1,128),(16,8,256))`, representing two warps holding a 16×32 tile. In this scenario, the mapping function $g \circ q^{-1}$ maps four 8×8 matrices produced by `ldmatrix` to four distinct locations in the output tensor, as shown in Figure 11.

Generalized Constraints. Figure 12 summarizes the constraints for key operations. Figure (a) formalizes the constraint for the `copy` operation. For the `mma` operation, we similarly construct the composite functions to bridge the instruction and the data tensors, as shown in Figure (b). Since the tensors `a`, `b`, and `c` have different coordinate spaces (i.e., $\mathbb{Z}^{[0,M)} \times \mathbb{Z}^{[0,K)}$ for `a`, $\mathbb{Z}^{[0,N)} \times \mathbb{Z}^{[0,K)}$ for `b`, and $\mathbb{Z}^{[0,M)} \times \mathbb{Z}^{[0,N)}$ for `c`, where M, N, and K denote the matrix dimensions), we define helper functions μ_* and η_* to connect the composite mappings. For example, η_M maps the instruction row coordinate m_I into the two-dimensional space: $\eta_M : m_I \mapsto (m_I, 0)$, and the projection function μ_M extracts the M dimension from the data tensor coordinate pair (m_T, n_T): $\mu_M : (m_T, n_T) \mapsto m_T$. A detailed discussion about the constraints for `mma` is provided in Appendix B. For the `elementwise` operation, all tensors must have the same thread-value layouts, as shown in Figure (c). Lastly, the constraint for the `reduce` operation is provided in Figure (d), where a projection function $\eta : \mathbb{Z}^d \rightarrow \mathbb{Z}^{d-1}$ collapses the reduced dimension (for example, $\mu : (m, n) \mapsto m$ for a reduction along n dimension).

(a) copy(a, b)
Tensor `a` and `b` have thread-value layouts f and g.
Instrucion I is represented by thread-value layouts p and q for its input and output operands. I can execute the `copy` operation.

tp/tp-copy

$$f \circ p^{-1} = g \circ q^{-1}$$

(b) mma(a, b, c)
Tensor `a`, `b`, and `c` have thread-value layouts f_A, f_B and f_C.
Instruction I is represented by thread-value layouts p_A, p_B and p_C.
I can execute the `mma` operation.

tp/tp-mma

$$\mu_M \circ \left(f_C \circ p_C^{-1}\right) \circ \eta_M = \tilde{\mu}_M \circ \left(f_A \circ p_A^{-1}\right) \circ \tilde{\eta}_M$$
$$\mu_N \circ \left(f_C \circ p_C^{-1}\right) \circ \eta_N = \tilde{\mu}_N \circ \left(f_B \circ p_B^{-1}\right) \circ \tilde{\eta}_N$$
$$\mu_K \circ \left(f_A \circ p_A^{-1}\right) \circ \eta_K = \tilde{\mu}_K \circ \left(f_B \circ p_B^{-1}\right) \circ \tilde{\eta}_K$$

(c) elementwise(a1, a2, a3, ..., an)
Tensors `a1, ..., an` have thread-value layouts $f_1, f_2, \cdots, f_n$

tp/tp-elem

$$f_1 = f_2 = \cdots = f_n$$

(d) b = reduce(a)
Tensors `a` and `b` have thread-value layouts f and g.

tp/tp-reduce

$\mu \circ f = g$, μ collapses the reduced dimension.

Figure 12. The thread-value layout constraits for `copy`, `mma`, `elementwise`, and `reduce` operations.

5 Task Mapping and Layout Synthesis

5.1 Thread-value Layout Synthesis

Building on the thread-value layout constraints in Figure 12, we design Algorithm 1 to synthesize thread-value layouts.

Graph Partition. The algorithm first constructs a directed acyclic graph (DAG) of tile-level operations and partitions it into connected subgraphs separated by shared memory reads and writes (**Line** 1). For each subgraph, the algorithm selects anchor operators and initializes the thread-value layouts for the inputs and outputs. It then builds thread value layout constraints and propagates the layouts based on the constraints detailed in Figure 12. In subgraphs containing `mma` operations, these operations are chosen as anchors due to their critical impact on kernel performance. If `mma` operations are absent, the algorithm selects the `copy` operation that transfers the most data as the anchor. We choose this design because non-mma operations are typically memory-bound, which makes the copy operation the most critical to kernel performance. In addition, this is feasible because every connected component includes at least one copy operation that reads or writes data; otherwise, the subgraph would be removed during dead code elimination.

Initialization. For an mma anchor, the algorithm selects the fastest Tensor Core instruction available on the target GPU and tiles the matrix C with the instruction, which materializes the layout for C (**Lines** 8-9). It then solves the layouts for A and B using the constraints in Figure 12 (b) (**Lines** 10-11). For a copy anchor, the layout is constructed by coalescing memory accesses (**Line** 15). The process begins by sorting the memory layout dimensions by their strides. The vector size for the ld/st instruction is then determined by analyzing the divisibility of the strides. Finally, the thread-value layout is constructed such that consecutive threads access contiguous vectors in memory, thereby coalescing the memory access.

Layout Solving. The algorithm maintains a set of remaining constraints, $\mathscr{C}$, and a ready list of constraints, Rq for each subgraph (**Lines** 3-5). A constraint is added to Rq when only one thread-value layout variable remains unknown. At that point, we rewrite the constraint so that the unknown variable appears on the left-hand side of the equation (**Line** 22). For example, if the constraint for a copy operation is ready and the layout g is known, we select an available instruction I with input and output layouts p and q, and rewrite the equation from Figure 12 as $f = g \circ q^{-1} \circ p$. Using this equation, we derive the unknown layout f by applying the algebraic operations of layouts. As new layouts become known, we update the ready list accordingly. This process repeats until all constraints in $\mathscr{C}$ have been resolved.

```
...
1. reg_qk = mma(reg_q, reg_k)
2. reg_qk_sum = reduce_sum(reg_qk, axis=1)
3. reg_qk1 = reg_qk / reg_qk_sum
4. reg_qk2 = cast(reg_qk1, "float16")
5. reg_o = mma(reg_qk2, reg_v)
...
```

Figure 13. Example: two matmul with a reduce in between

Algorithm 1 Thread-Value Layout Synthesis Algorithm

Input: A directed acyclic graph $G = (V, E)$ of tile-level operations. V denotes the operators, and E represents the tensors.
Output: Thread-value layouts $L_1, L_2, \ldots, L_n$ for all tensors in E

```
 1: Partition the graph $G$ into connected components $\mathscr{S} = S_1, S_2, \ldots, S_n$
 2: for $S_i$ in $\mathscr{S}$ do
 3:     $\mathscr{C} \leftarrow$ BuildConstraints($S_i$)        ▷ The set of the constraints in $S_i$
 4:     $\mathscr{L} \leftarrow \{\}$                          ▷ The set of synthesized layouts in $S_i$
 5:     $Rq \leftarrow \{\}$                    ▷ The constraints in $S_i$ that are ready to solve
 6:     if $S_i$ includes an mma operation then
 7:         for mma operation in $S_i$ do
 8:             Select the fastest tensor core instruction $I$ on the target GPU.
 9:             Tile matrix $C$ with instruction $I$ and materialize layout $L_C$.
10:             Solve layouts $L_A$ and $L_B$ with constraints of mma.
11:             Update $L_A, L_B, L_C$ in $\mathscr{L}$.
12:         end for
13:     else
14:         Pick an anchor copy operation $Ac$.
15:         Materialize layout, $L$, by coalescing the memory access.
16:         Update $L$ in $\mathscr{L}$.
17:     end if
18:     UpdateReadyQueue($Rq, \mathscr{C}, \mathscr{L}$)
19:     while $\mathscr{C}$ is not empty do
20:         while $Rq$ is not empty do
21:             $C \leftarrow$ Dequeue($Rq$)
22:             $L \leftarrow$ Solve($C, \mathscr{L}$)
23:             Update $L$ in $\mathscr{L}$.
24:             Remove $C$ from $\mathscr{C}$.
25:         end while
26:         UpdateReadyQueue($Rq, \mathscr{C}, \mathscr{L}$)
27:     end while
28: end for
29: return
```

Conflict Handling. For kernels with multiple GEMM operations (e.g., FlashAttention [3]), the algorithm can propagate thread-value layouts starting from central mma operations. Without loss of generality, consider an example with two matrix multiplications and a reduction in between, as shown in Figure 13. First, the compiler materializes the thread-value layouts for the mma operations at **Line** 1 and **Line** 3. Then, the constraints for the reduce_sum and cast operations at **Line** 2 and **Line** 4 become ready to solve. Once these constraints are resolved, and the thread-value layouts of reg_qk1, reg_qk, and reg_qk_sum are fully determined, the compiler verifies their logical consistency, ensuring the register tensors are distributed across the threads in the same way. If the consistency condition is not met, the compiler injects a rearrange operator on reg_qk1 to produce a valid program. However, this approach incurs data exchange through shared memory, which may introduce additional overhead. To circumvent this overhead, Hexcute allows users to annotate the task mapping for the mma operations and ensure consistency.

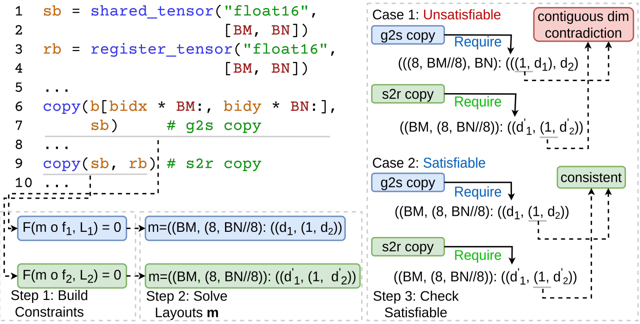
Expanding Search Tree. In cases where multiple valid instructions exist for a single copy operation, Algorithm 1 is extended with depth-first search (DFS) and backtracking. This extension transforms the search space into a tree, with each leaf node representing a valid program. We then build an analytical latency model to estimate the latency of each leaf program, select the one with the lowest predicted latency, and generate the lower-level IR.

5.2 Shared Memory Layout Synthesis

For shared memory synthesis, we represent the shared memory layout M as the composition, $M = S \circ m$, where $m : \mathbb{Z} \to \mathbb{Z}$ is a memory layout function subject to instruction alignment requirements and $S : \mathbb{Z} \to \mathbb{Z}$ is a swizzle function that permutes shared memory addresses to eliminate bank conflicts. The goal is to find the satisfied functions, m and S.

Alignment Requirement. We first consider synthesizing the memory layout function m. As a concrete example, suppose a `copy` operation loads a `float16` tile of size 4×64 from shared memory to registers. Assume a 16-byte aligned `ld` instruction is selected, and the thread-value layout f inferred by Algorithm 1 is: $f = ((8,),(8,4)) : ((32,),(4,1))$. Composing layout f with memory layout m yields the addresses assigned to each thread. Since alignment requirements dictate addresses must be multiples of 16 bytes (8 `float16` elements), we construct the constraint: $m \circ f = ((8,),(8,4)) : ((d_1,),(1,d_2))$, where d_1 and d_2 are undetermined strides. By multiplying by the inverse f^{-1}, we obtain a candidate layout: $m = (4,8,8) : (d_2,1,d_1)$. Based on the observation, the compiler traverses the program and applies this approach to each `copy` operation involving the same shared memory tensor. By comparing candidate layouts from all operations, the compiler synthesizes a unified memory layout m satisfying all alignment constraints. A detailed discussion about the example and the general case is provided in Appendix C.

For example, Figure 14 illustrates a code snippet with shared memory read/write operations, where `BM` and `BN` represent known block sizes. The algorithm constructs alignment constraints for `g2s` and `s2r` copy operations (**Lines 6 and 9**). Solving these constraints partially determines the memory layout strides, as depicted in the middle of Figure 14. In the example, a dimension with shape 8 and stride 1 indicates the 8 elements are contiguous in memory, ensuring 16-byte alignment. Finally, the compiler checks layout consistency by verifying stride satisfiability across operations. The top-right case in Figure 14 is unsatisfiable due to conflicting contiguous dimensions. Conversely, the bottom-right scenario remains satisfiable. If all constraints are met, the compiler finalizes unknown strides d_1 and d_2 to complete the synthesis of the shared memory layout.

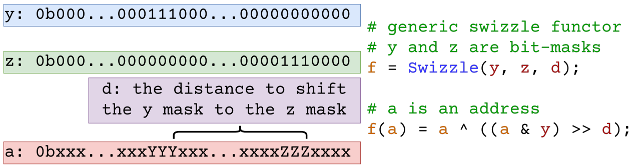


```
1   sb = shared_tensor("float16",
2                      [BM, BN])
3   rb = register_tensor("float16",
4                        [BM, BN])
5   ...
6   copy(b[bidx * BM:, bidy * BN:],
7        sb)     # g2s copy
8   ...
9   copy(sb, rb) # s2r copy
10  ...
```

Case 1: Unsatisfiable
contiguous dim contradiction
g2s copy
Require
(((8, BM//8), BN): (((1, d1), d2)
s2r copy
Require
((BM, (8, BN//8)): ((d'1, (1, d'2))
Case 2: Satisfiable
consistent
g2s copy
Require
((BM, (8, BN//8): ((d1, (1, d2))
s2r copy
Require
((BM, (8, BN//8)): ((d'1, (1, d'2))
F(m o f1, L1) = 0
F(m o f2, L2) = 0
m=((BM, (8, BN//8): ((d1, (1, d2))
m=((BM, (8, BN//8)): ((d'1, (1, d'2))
Step 1: Build Constraints
Step 2: Solve Layouts m
Step 3: Check Satisfiable

Figure 14. Example: shared memory synthesis.

Eliminate Bank-Conflicts. To eliminate the bank conflict in the shared memory access, we leverage the generic swizzle function defined in CuTe [24], which is shown in Figure 15.



```
# generic swizzle functor
# y and z are bit-masks
f = Swizzle(y, z, d);

# a is an address
f(a) = a ^ ((a & y) >> d);
```

y: 0b000...000111000...00000000000
z: 0b000...000000000...00001110000
d: the distance to shift
the y mask to the z mask
a: 0bxxx...xxxYYYxxx...xxxxZZZxxxx

Figure 15. Generic swizzle functor

We design a compiler pass that traverses the program and identifies all `copy` operations that access the same shared memory buffer. The pass then enumerates the applicable swizzle functions and selects the one most effectively reducing bank conflicts. This is done by analyzing the memory addresses accessed by the threads within a warp and choosing the swizzle function that minimizes overlapping accesses to the same bank.

6 Code Generation

After synthesizing thread-value layouts, shared memory layouts, and task mappings, we generate the Hidet IR for the tile-level primitives. The code generation process consists of two steps: instruction selection and loop generation. Instruction selection leverages the thread-value layout constraints established in Figure 12 to find the best instruction. For each tile-level operation, we generate a loop that repeatedly executes the instruction. The loop structure is determined by analyzing the thread-value layout of the tensor operands.

7 Evaluation

We implement Hexcute in Python with about 10K lines of code. Built on top of Hidet, we introduce new tile-level primitives and implement compiler passes to lower these primitives to low-level Hidet IR. We propose a novel type system to guarantee the correctness and performance of GPU programs. We leverage the layout algebra of CuTe to build type constraints, but unlike CuTe, Hexcute's code generation directly emits pure C code without depending on any C++ templates in CuTe [30]. While Hexcute increases the compilation time compared to Hidet, as the DFS enumerates all kernel variations with different instructions, we employ the hardware-centric searching space [6] to keep compilation time acceptable. In the following sections, we demonstrate the effectiveness of Hexcute via extensive benchmarks.

7.1 Experimental Setup

The experiments are conducted on three NVIDIA GPUs, RTX4090, A100, and H100, representing the Ampere, Ada, and Hopper architectures. To avoid experiment fluctuations, we lock the GPUs' frequencies and clear the L2 cache before running all the microbenchmarks. Unless otherwise specified, we use float32 as the accumulator for `mma` instruction.

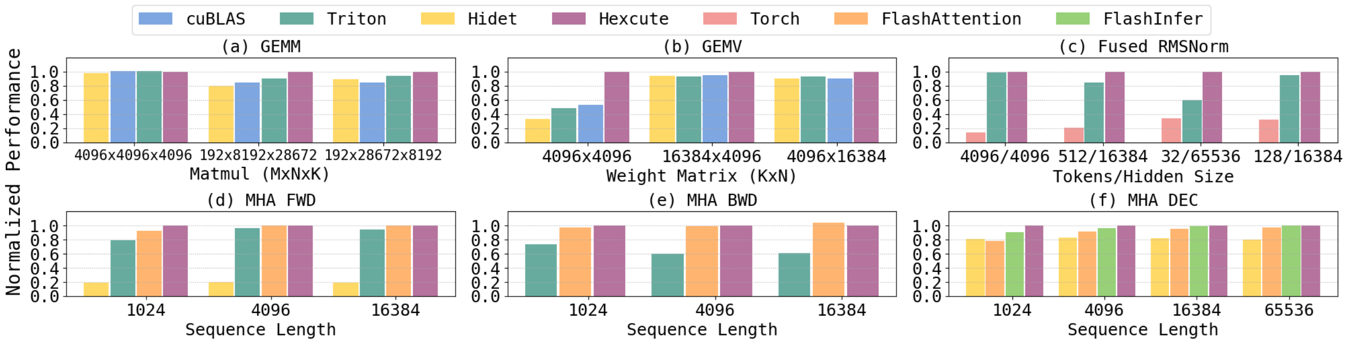


Figure 16. Operator comparison to existing compilers and hand-written kernel libraries on the RTX4090 GPU. The performance of all systems is normalized by Hexcute (higher is better). Hexcute brings up to 1.86 × to the best existing system.

7.2 Operator Benchmark

To demonstrate Hexcute's generality across a wide range of DL operators, we evaluate its performance on six operator benchmarks. Figure 16 compares Hexcute against compilers such as Hidet [6] (v0.6.0.dev) and Triton [31] (v3.0.0), as well as established hand-written kernel libraries. Our kernel library baseline includes cuBLAS [23] (v12.4), FlashAttention [3] (v2.6.3), and FlashInfer [39] (v0.0.3). We summarize the results as follows:

GEMM is the key machine-learning operator on GPUs. This experiment evaluates if Hexcute can achieve performance comparable to cuBLAS, the most optimized GPU GEMM library. The first matmul is compute-bound, and Hexcute perfectly matches cuBLAS performance. In the other two matmuls, where a tall and a flat matrix B are selected, tuning tile sizes allows Hexcute to produce kernels that exceed the performance of Triton, Hidet, and cuBLAS.

GEMV, short for general matrix-vector multiplication, is widely used in large language model (LLM) serving systems [13]. Due to its long-tail property, this operator is often not well-optimized. Hexcute implements the GEMV layer using the `reduce` operator and, in the memory-latency-bound case, Hexcute outperforms other kernel providers by 46.4%.

RMSNorm is a layer normalization widely used in language models. Hexcute exploits parallelism in the hidden dimension and applies inter-thread-block reduction, which cannot be done by Triton, and achieves up to 1.67× speedup over it.

MHA FWD, the multi-head attention forward layer, is evaluated under compute-bound conditions. Hidet uses manually designed layouts and task mappings that are not fully optimized. In contrast, Hexcute automatically generates efficient ones, matching the performance of FlashAttention, the state-of-the-art attention library.

MHA BWD, the multi-head attention backward layer, is essential for training language models. In our experiment, Hexcute matches the performance of FlashAttention, which is the best backward kernel available. To optimize this operator, developers must balance caching the Q, K, and V tensors in shared memory or registers. However, Triton cannot do this because it does not expose these hardware hierarchies. By carefully orchestrating pipelining, Hexcute outperforms Triton's solution by up to 40.3%.

MHA DEC is the attention layer used during decoding, a key part of LLM serving, and is often memory-bound. Hexcute matches the performance of top hand-written libraries like FlashAttention and FlashInfer. By splitting the KV cache loading and using compiler-generated optimal task mappings and layouts, Hexcute achieves 20% lower latency than Hidet.

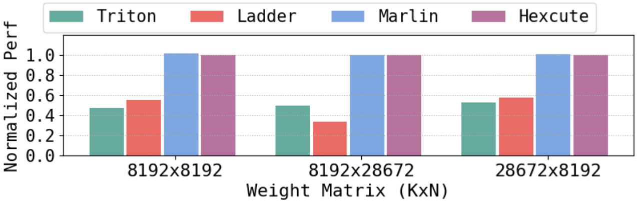


Figure 17. Performance comparison of an FP16×INT4 matmul kernel on RTX4090. The performance is normalized by Hexcute (higher is better). Hexcute achieves up to 2.9× and 2.1× speedups over Triton and Ladder, respectively.

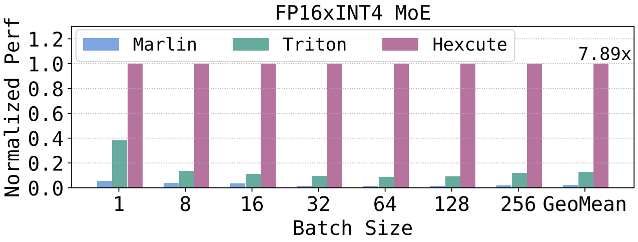


Figure 18. Performance comparison of Hexcute, Triton, and Marlin of an MoE layer with 256 experts on H100. The performance is normalized by Hexcute (higher is better). Hexcute brings an average speedup of 7.89× over Triton.

7.3 Case Studies

To demonstrate that Hexcute generates efficient code for low-precision computations, we compare Hexcute against compiler baselines, including Ladder (v0.0.1.dev15) and Triton (v3.0.0), as well as against Marlin [8] (commit 2f6d7c), the best mixed-type GEMM library.

Mixed-type GEMM. Figure 17 compares Hexcute to Triton, Ladder, and Marlin on a FP16×INT4 matmul kernel on RTX4090. The results show that Hexcute brings up to 2.9× and 2.1× speedups over Triton and Ladder, respectively, while matching the performance of Marlin. We analyze these improvements in detail. Triton and Ladder's layout systems cannot express the packed layout in Figure 3, so they select suboptimal layouts that introduce extra overheads, as shown in Figure 2 (a) and (b). In contrast, Hexcute exposes shared memory and register abstractions to explicitly express the optimal data pipeline shown in Figure 2 (c). Moreover, our layout synthesis algorithm finds the layout that satisfies the constraints without incurring additional shuffles through shared memory or registers. For example, Hexcute discovers a shared layout `((8,2,8),(2,64)):((4,2,2048),(1,32))` , which allows loading a INT4 tile of 128×128 with `ldmatrix`.

Mixed-type MoE Layer. Figure 18 compares the performance of a mixed-type MoE layer on the H100. Marlin [8] designs the efficient layout and data pipeline to hide the cost of dequantization, but it is written by hand and hard to adapt to complex scenarios like MoE. So, Marlin implements the MoE layer by launching a separate kernel for each expert, introducing significant kernel launch overhead. Triton fuses all experts into a single kernel to remove the launch overhead, but its layout selection and data pipeline are suboptimal. In contrast, Hexcute combines the benefits of both approaches, achieving an average speedup of 7.89× over Triton.

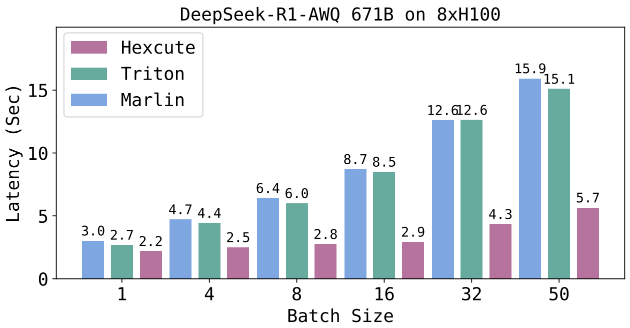


Figure 19. Integrating Hexcute-generated MoE kernels into vLLM brings up to 2.91× speedup to the best existing system.

7.4 End-to-end Benchmark

To evaluate whether Hexcute-generated kernels are useful in practice, we integrated our mixed-type MoE kernels into vLLM [17] (v0.7.3) and evaluated the end-to-end performance of the DeepSeek-R1-AWQ model on 8×H100 GPUs. As shown in Figure 19, Hexcute reduces latency by 16%–65% compared to the best existing compilers and hand-written kernels. In this experiment, latency is measured as the time required to generate 100 output tokens from 100 input tokens.

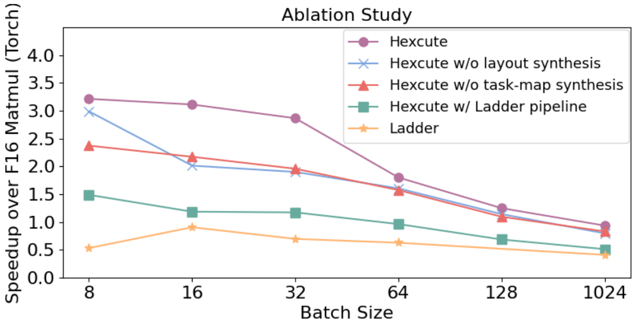


Figure 20. The ablation study on the A100 GPU.

7.5 Ablation Study

We conduct an ablation study on an A100 GPU using a linear layer from Llama 3.1 (70B) with float16 activations and int4 weights (n=28672, k=8192). The experiment compares Hexcute's optimized kernel with kernels generated by disabling individual compiler passes, as shown in Figure 20. On average, performance drops by 19.3%, 52.5%, and 21.2% when shared memory layout synthesis, task-mapping synthesis, and Hexcute's optimized software pipeline (replaced by Ladder's) are disabled, respectively. Notably, even when replacing Hexcute's optimal pipelining with Ladder's, Hexcute still outperforms Ladder, demonstrating the effectiveness of its task mapping and layout synthesis. These results highlight the importance of Hexcute's compiler optimizations.

8 Related Work

Most existing DL compilers [2, 7, 27, 40, 42–44] focus on operators with mainstream data types like FP16 and FP32, providing limited support for low-precision computations. Ladder [35] extends TVM [2] to optimize mixed-type operators. However, Ladder [35] leverages TVM's loop-oriented schedule primitives, which makes it hard to express some key optimizations like software pipelining and fine-grained dataflow. In contrast, Hexcute proposes a tile-based programming language that explicitly exposes shared memory and register abstractions, which allows us to implement these optimizations for mixed-type operators. Hidet [6] employs the task mapping programming model to schedule the tensor programs. Hexcute adopts a similar approach but simplifies the programming efforts by automating the synthesis of task mappings and layouts with a type-inference-based algorithm. Triton [31] is another tile-based programming language, but it lacks the exposure of shared memory and registers, limiting its applicability to low-precision optimizations. Graphene [11] leverages the Layout concept to represent the GPU optimizations such as tiling and bank-conflict elimination, but it does not support low-precision computations and requires manual specifications for the optimal layouts. Specialized libraries such as CUTLASS [30] and Marlin [8] demonstrate efficient low-precision kernels (e.g., FP16×INT4 matrix multiplication) through hand-tuned templates or manual implementations. However, these solutions demand considerable engineering efforts and are hard to generalize to broader low-precision computations.

9 Conclusions

This paper presents Hexcute, a tile-based programming language that exposes hardware hierarchies, enabling fine-grained optimizations for mixed-type operations. To reduce programming effort, Hexcute automates layout synthesis with a type-inference-based algorithm. Experiments show that Hexcute generalizes across a wide range of DL operators and achieves competitive performance. Moreover, Hexcute generates efficient code for low-precision computations, achieves 1.7-11.28× speedup over existing compilers for operators, and brings up to 2.91× speedup in the end-to-end evaluation.

References

[1] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D 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 Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901.

[2] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *USENIX Symposium on Operating Systems Design and Implementation.*

[3] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems*.

[4] Jack W Davidson and Sanjay Jinturkar. 1994. Memory access coalescing: A technique for eliminating redundant memory accesses. *Acm Sigplan Notices* 29, 6 (1994), 186–195.

[5] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. 2025. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437

[6] Yaoyao Ding, Cody Hao Yu, Bojian Zheng, Yizhi Liu, Yida Wang, and Gennady Pekhimenko. 2023. Hidet: Task-Mapping Programming Paradigm for Deep Learning Tensor Programs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 370–384. https://doi.org/10.1145/3575693.3575702

[7] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, and Tianqi Chen. 2022. TensorIR: An Abstraction for Automatic Tensorized Program Optimization. arXiv:2207.04296 [cs.LG]

[8] Elias Frantar and Dan Alistarh. 2024. Marlin: a fast 4-bit inference kernel for medium batchsizes. https://github.com/IST-DASLab/marlin.

[9] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2023. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. arXiv:2210.17323 [cs.LG]

[10] Ronald Garcia and Matteo Cimini. 2015. Principal Type Schemes for Gradual Programs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 303–315. https://doi.org/10.1145/2676726.2676992

[11] Bastian Hagedorn, Bin Fan, Hanfeng Chen, Cris Cecka, Michael Garland, and Vinod Grover. 2023. 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 302–313. https://doi.org/10.1145/3582016.3582018

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[13] Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, kangdi chen, Yuhan Dong, and Yu Wang. 2024. FlashDecoding++: Faster Large Language Model Inference with Asynchronization, Flat GEMM Optimization, and Heuristics. In *Proceedings of Machine Learning and Systems*, P. Gibbons, G. Pekhimenko, and C. De Sa (Eds.), Vol. 6. 148–161. https://proceedings.mlsys.org/paper_files/paper/2024/file/

5321b1dabcd2be188d796c21b733e8c7-Paper-Conference.pdf

[14] Guyue Huang, Yang Bai, Liu Liu, Yuke Wang, Bei Yu, Yufei Ding, and Yuan Xie. 2023. ALCOP: Automatic Load-Compute Pipelining in Deep Learning Compiler for AI-GPUs. In *Proceedings of Machine Learning and Systems*, D. Song, M. Carbin, and T. Chen (Eds.), Vol. 5. Curan, 680–694. https://proceedings.mlsys.org/paper_files/paper/2023/file/d6cde2c1b161daa31be560d062cf2251-Paper-mlsys2023.pdf

[15] Qihang Huang, Zhiyi Huang, Paul Werstein, and Martin Purvis. 2008. GPU as a General Purpose Computing Resource. In *2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies*. 151–158. https://doi.org/10.1109/PDCAT.2008.38

[16] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. 2023. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 82, 14 pages. https://doi.org/10.1145/3579371.3589350

[17] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (, Koblenz, Germany,) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. https://doi.org/10.1145/3600006.3613165

[18] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, Chuang Gan, and Song Han. 2023. AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration. arXiv:2306.00978 [cs.CL]

[19] Yujun Lin, Haotian Tang, Shang Yang, Zhekai Zhang, Guangxuan Xiao, Chuang Gan, and Song Han. 2024. QServe: W4A8KV4 Quantization and System Co-design for Efficient LLM Serving. arXiv:2405.04532 [cs.CL] https://arxiv.org/abs/2405.04532

[20] Eric Mahurin. 2023. Qualocmm® Hexagon™ NPU. In *2023 IEEE Hot Chips 35 Symposium (HCS)*. 1–19. https://doi.org/10.1109/HCS59251.2023.10254715

[21] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. 2018. NVIDIA Tensor Core Programmability, Performance & Precision . In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE Computer Society, Los Alamitos, CA, USA, 522–531. https://doi.org/10.1109/IPDPSW.2018.00091

[22] John Nickolls and William J. Dally. 2010. The GPU Computing Era. *IEEE Micro* 30, 2 (2010), 56–69. https://doi.org/10.1109/MM.2010.41

[23] NVIDIA. [n. d.]. *cuBLAS*. https://docs.nvidia.com/cuda/cublas/

[24] NVIDIA. 2024. *CUTLASS - CuTe documentation*. https://github.com/NVIDIA/cutlass/tree/main/media/docs/cute

[25] NVIDIA. 2024. *TensorRT-LLM*.

[26] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. 2019. Swizzle Inventor: Data Movement Synthesis for GPU Kernels. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 65–78. https://doi.org/10.1145/3297858.3304059

[27] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *SIGPLAN Not.* 48, 6 (jun 2013), 519–530. https://doi.org/10.1145/2499370.2462176

[28] Bita Darvish Rouhani, Ritchie Zhao, Ankit More, Mathew Hall, Alireza Khodamoradi, Summer Deng, Dhruv Choudhary, Marius Cornea, Eric Dellinger, Kristof Denolf, Stosic Dusan, Venmugil Elango, Maximilian Golub, Alexander Heinecke, Phil James-Roxby, Dharmesh Jani, Gaurav Kolhe, Martin Langhammer, Ada Li, Levi Melnick, Maral Mesmakhosroshahi, Andres Rodriguez, Michael Schulte, Rasoul Shafipour, Lei Shao, Michael Siu, Pradeep Dubey, Paulius Micikevicius, Maxim Naumov, Colin Verrilli, Ralph Wittig, Doug Burger, and Eric Chung. 2023. Microscaling Data Formats for Deep Learning. arXiv:2310.10537 [cs.LG] https://arxiv.org/abs/2310.10537

[29] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. arXiv:1701.06538 [cs.LG] https://arxiv.org/abs/1701.06538

[30] 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. 2024. *CUTLASS*.

[31] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 10–19. https://doi.org/10.1145/3315508.3329973

[32] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[33] Andreas Tretter, Georgia Giannopoulou, Matthias Baer, and Lothar Thiele. 2017. Minimising access conflicts on shared multi-bank memory. *ACM Transactions on Embedded Computing Systems (TECS)* 16, 5s (2017), 1–20.

[34] Hongyu Wang, Shuming Ma, Li Dong, Shaohan Huang, Huaijie Wang, Lingxiao Ma, Fan Yang, Ruiping Wang, Yi Wu, and Furu Wei. 2023. BitNet: Scaling 1-bit Transformers for Large Language Models. arXiv:2310.11453 [cs.CL] https://arxiv.org/abs/2310.11453

[35] Lei Wang, Lingxiao Ma, Shijie Cao, Quanlu Zhang, Jilong Xue, Yining Shi, Ningxin Zheng, Ziming Miao, Fan Yang, Ting Cao, Yuqing Yang, and Mao Yang. 2024. Ladder: Enabling Efficient Low-Precision Deep Learning Computing through Hardware-aware Tensor Transformation. In *OSDI 2024*.

[36] Haojun Xia, Zhen Zheng, Xiaoxia Wu, Shiyang Chen, Zhewei Yao, Stephen Youn, Arash Bakhtiari, Michael Wyatt, Donglin Zhuang, Zhongzhu Zhou, Olatunji Ruwase, Yuxiong He, and Shuaiwen Leon Song. 2024. Quant-LLM: Accelerating the Serving of Large Language Models via FP6-Centric Algorithm-System Co-Design on Modern GPUs. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 699–713. https://www.usenix.org/conference/atc24/presentation/xia

[37] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. SmoothQuant: accurate and efficient post-training quantization for large language models. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) *(ICML'23)*. JMLR.org, Article 1585, 13 pages.

[38] Da Yan, Wei Wang, and Xiaowen Chu. 2020. Demystifying Tensor Cores to Optimize Half-Precision Matrix Multiply. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 634–643. https://doi.org/10.1109/IPDPS47924.2020.00071

[39] Zihao Ye, Lequn Chen, Ruihang Lai, Yilong Zhao, Size Zheng, Junru Shao, Bohan Hou, Hongyi Jin, Yifei Zuo, Liangsheng Yin, Tianqi Chen, and Luis Ceze. 2024. FlashInfer: Kernel Library for LLM Serving. https://github.com/flashinfer-ai/flashinfer.

[40] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning. In *Proceedings of the 28th ACM International Conference*

on Architectural Support for Programming Languages and Operating Systems, Volume 3 (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 660–678. https://doi.org/10.1145/3582016.3582047

[41] Ekim Yurtsever, Jacob Lambert, Alexander Carballo, and Kazuya Takeda. 2020. A survey of autonomous driving: Common practices and emerging technologies. *IEEE access* 8 (2020), 58443–58469.

[42] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 49, 17 pages.

[43] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. 2022. AMOS: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang (Eds.). ACM, 874–887. https://doi.org/10.1145/3470496.3527440

[44] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 859–873. https://doi.org/10.1145/3373376.3378508

Appendix

A Tile-level Operation Specifications

```
scope       ::=  Global | Shared | Register
type        ::=  float16 | bfloat16 | float32 | ...
             |   int8 | int32 | uint8 | uint32 | ...
             |   int1 | int2 | int4 | uint1 | uint2 | uint4
             |   float8_e5m2 | float8_e4m3
             |   ...
tuple_var   ::=  int | ’(’ int ’,’ ... ’,’ int ’)’
             |   ’(’ tuple_var ’,’ ... ’,’ tuple_var ’)’
tuple       ::=  ’(’ tuple_var ’,’ ... ’,’ tuple_var ’)’
shape       ::=  ’(’ tuple ’)’
layout      ::=  ’(’ tuple ’:’ tuple ’)’
operator    ::=  global_view ’(’ buffer ’,’ layout ’)’
             |   register_tensor ’(’ type ’,’ shape ’)’
             |   shared_tensor ’(’ type ’,’ shape ’)’
             |   shared_tensor ’(’ type ’,’ layout ’)’
             |   copy ’(’ tensor ’,’ tensor ’)’
             |   mma ’(’ tensor ’,’ tensor ’,’ tensor ’)’
             |   cast ’(’ tensor ’,’ type ’)’
             |   rearrange ’(’ tensor ’,’ layout ’)’
             |   elementwise ’(’ tensor ’,’ ... ’,’ tensor ’)’
             |   reduce ’(’ tensor ’,’ int ’)’
expr        ::=  tensor
             |   operator
             |   expr ’+’ expr
             |   expr ’-’ expr
             |   expr ’/’ expr
             |   ...
```

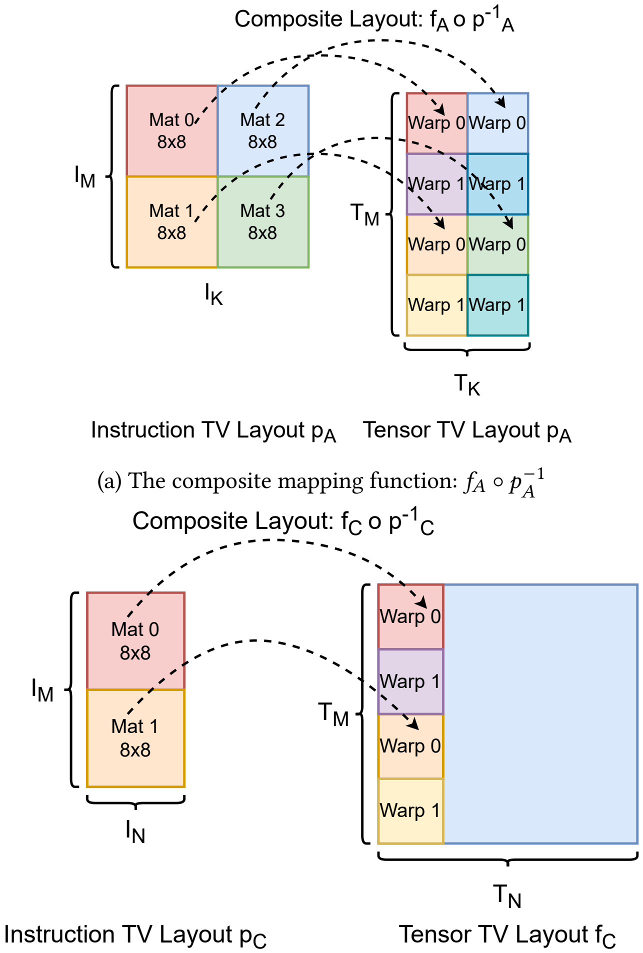
Figure 21. Syntax of the tile-level operators in Hexcute

This section details the syntax of tile-level operations in Hexcute, which is illustrated in Figure 21. Hexcute defines memory scopes, such as `Global`, `Shared`, and `Register`, to explicitly represent a tensor's memory space. In addition, Hexcute supports low-precision data types such as `int1|2|4`, `uint1|2|4`, and specialized floating-point formats like `float8_e5m2`.

Building on CuTe's methodology, Hexcute employs recursively defined integer tuples to specify hierarchical shapes and layouts, which enables the construction of complex memory layouts and thread-value layouts. Layouts are expressed as colon-separated pairs of integer tuples (`(tuple:tuple)`). Moreover, Hexcute supports primitive tensor operations in tensor programming, including data movement (`copy`), matrix multiplication (`mma`), type conversion (`cast`), layout transformation via shared memory (`rearrange`), elementwise operations, and reductions, which can be combined with arithmetic expressions to specify computations flexibly.

B Thread-value Layout Constraints for Mma Operation

In this section, we present a detailed discussion of the thread-value layout constraints for the `mma` operation.



(a) The composite mapping function: $f_A \circ p_A^{-1}$

(b) The composite mapping function: $f_C \circ p_C^{-1}$

Figure 22. Consider an `mma` operation where the operand tensors `A`, `B`, and `C` have thread-value layouts f_A, f_B, and f_C, respectively. The Tensor Core instruction `mma.sync.aligned.m16n8k16.row.col` is used to execute the operation, with its operands A, B, and C having thread-value layouts p_A, p_B, and p_C, respectively. Figure (a) shows that the composite mapping function $f_A \circ p_A^{-1}$ maps the four matrices from the instruction tile `A` to distinct positions in tensor A, while Figure (b) illustrates that $f_C \circ p_C^{-1}$ maps the two matrices in the instruction tile `C` to different positions in tensor C.

An Intuitive Example. Consider an `mma` operation where the operand tensors `A`, `B`, and `C` have thread-value layouts f_A, f_B, and f_C, respectively. Suppose that the Tensor Core instruction `mma.sync.aligned.m16n8k16.row.col`, represented by thread-value layouts p_A, p_B, and p_C for operands A, B, and C, is selected to execute the `mma` operation. Similar to the copy operation, we can relate the thread-value layouts of the instruction to those of the tensors. For example, Figure 22 (a) illustrates the composite mapping function $f_A \circ p_A^{-1}$ maps the four matrices from the instruction tile `A` to distinct positions in tensor A. Similarly, Figure 22 (b) shows the composite mapping function $f_C \circ p_C^{-1}$ maps the two matrices in the instruction tile `C` to different positions in tensor C. Since the instruction operands `A` and `C` have the same dimension I_M and tensors A and C share the same dimension T_M, the restrictions of the mapping functions $f_A \circ p_A^{-1}$ and $f_C \circ p_C^{-1}$ to the M dimension must yield the same mapping from I_M to T_M. Based on this observation, we formalize the constraints of the `mma` operation as stated in Theorem 1.

$$\begin{tikzcd}
I_M \times I_K \arrow[d, "f_A \circ p_A^{-1}"'] & I_M \arrow[l, "\tilde{\eta}_M"'] \arrow[r, "\eta_M"] \arrow[d] & I_M \times I_N \arrow[d, "f_C \circ p_C^{-1}"] \\
T_M \times T_K \arrow[r, "\tilde{\mu}_M"] & T_M & T_M \times T_N \arrow[l, "\mu_M"']
\end{tikzcd}$$

$$\begin{tikzcd}
I_N \times I_K \arrow[d, "f_B \circ p_B^{-1}"'] & I_N \arrow[l, "\tilde{\eta}_N"'] \arrow[r, "\eta_N"] \arrow[d] & I_M \times I_N \arrow[d, "f_C \circ p_C^{-1}"] \\
T_N \times T_K \arrow[r, "\tilde{\mu}_N"] & T_N & T_M \times T_N \arrow[l, "\mu_N"']
\end{tikzcd}$$

$$\begin{tikzcd}
I_M \times I_K \arrow[d, "f_A \circ p_A^{-1}"'] & I_K \arrow[l, "\tilde{\eta}_K"'] \arrow[r, "\eta_K"] \arrow[d] & I_N \times I_K \arrow[d, "f_B \circ p_B^{-1}"] \\
T_M \times T_K \arrow[r, "\tilde{\mu}_K"] & T_K & T_M \times T_K \arrow[l, "\mu_K"']
\end{tikzcd}$$

Figure 23. The thread-value constraints for matrices `A`, `B`, and `C` in the `mma` operation represented using commutative diagrams. The projection functions, μ_M, $\tilde{\mu}_M$, μ_N, $\tilde{\mu}_N$, μ_K, and $\tilde{\mu}_K$, and the natural embedding functions, η_M, $\tilde{\eta}_M$, η_N, $\tilde{\eta}_N$, η_K, and $\tilde{\eta}_K$, are defined in Theorem 1. The Cartesian product $T_M \times T_N$, $T_M \times T_K$, and $T_N \times T_K$, represent the coordinate spaces of matrices, `A`, `B`, and `C` in the `Mma` operation, while the Cartesian product $I_M \times I_N$, $I_M \times I_K$, and $I_N \times I_K$, represent the coordinate spaces of A, B, and C operands in instruction I.

Theorem 1. *Consider a* `mma` *operation where the tensors* `A`, `B`, *and* `C` *have the thread-value layouts f_A, f_B, and f_C, respectively. Suppose we use a Tensor Core instruction I, represented by the thread-value layouts p_A, p_B, and p_C for the A, B, and C operands, to execute this operation. Let the projection be defined as follows:*

$$\begin{aligned}
\mu_M &: (m_T, k_T) \mapsto m_T, & \tilde{\mu}_M &: (m_T, n_T) \mapsto m_T \\
\mu_N &: (n_T, k_T) \mapsto n_T, & \tilde{\mu}_N &: (m_T, n_T) \mapsto n_T \\
\mu_K &: (m_T, k_T) \mapsto k_T, & \tilde{\mu}_K &: (n_T, k_T) \mapsto k_T
\end{aligned}$$

, and the natural embedding functions are defined:

$$\begin{aligned}
\eta_M &: m_I \mapsto (m_I, 0) \in I_M \times I_N, & \tilde{\eta}_M &: m_I \mapsto (m_I, 0) \in I_M \times I_K \\
\eta_N &: n_I \mapsto (0, n_I) \in I_M \times I_N, & \tilde{\eta}_N &: n_I \mapsto (n_I, 0) \in I_N \times I_K \\
\eta_K &: k_I \mapsto (0, k_I) \in I_M \times I_K, & \tilde{\eta}_K &: k_I \mapsto (0, k_I) \in I_N \times I_K
\end{aligned}$$

where m_T, n_T, and k_T are the coordinates within tensors of the `mma` *operation, and m_I, n_I, and k_I are the coordinates within the operands of the instruction I. The Cartesian product $I_M \times I_N$, $I_M \times I_K$, and $I_N \times I_K$, represent the coordinate spaces of A, B, and C operands of instruction I. Then, the following consistency equations must hold:*

$$\begin{aligned}
\mu_M \circ \left(f_C \circ p_C^{-1}\right) \circ \eta_M &= \tilde{\mu}_M \circ \left(f_A \circ p_A^{-1}\right) \circ \tilde{\eta}_M \\
\mu_N \circ \left(f_C \circ p_C^{-1}\right) \circ \eta_N &= \tilde{\mu}_N \circ \left(f_B \circ p_B^{-1}\right) \circ \tilde{\eta}_N \\
\mu_K \circ \left(f_A \circ p_A^{-1}\right) \circ \eta_K &= \tilde{\mu}_K \circ \left(f_B \circ p_B^{-1}\right) \circ \tilde{\eta}_K
\end{aligned}$$

Proof. In the `mma` operation, the thread-value layouts f_A, f_B, and f_C of the matrix `A`, `B`, and `C` map thread ID and value ID to the logical positions within the tensors:

$$\begin{aligned}
f_A &: (t, v) \mapsto (m_T, k_T) \\
f_B &: (t, v) \mapsto (n_T, k_T) \\
f_C &: (t, v) \mapsto (m_T, n_T)
\end{aligned}$$

Similarly, for the Tensor Core instruction I, represented by the thread-value layouts p_A, p_B, and p_C for the A, B, and C operands, these layouts map thread ID and value ID to the operand coordinates:

$$\begin{aligned}
p_A &: (t, v) \mapsto (m_I, k_I) \\
p_B &: (t, v) \mapsto (n_I, k_I) \\
p_C &: (t, v) \mapsto (m_I, n_I)
\end{aligned}$$

To relate the tensor layouts with the instruction layouts, we construct composite functions from the instruction operands to the matrix tensors:

$$\begin{aligned}
f_A \circ p_A^{-1} &: (m_I, k_I) \mapsto (m_T, k_T) \\
f_B \circ p_B^{-1} &: (n_I, k_I) \mapsto (n_T, k_T) \\
f_C \circ p_C^{-1} &: (m_I, n_I) \mapsto (m_T, n_T)
\end{aligned}$$

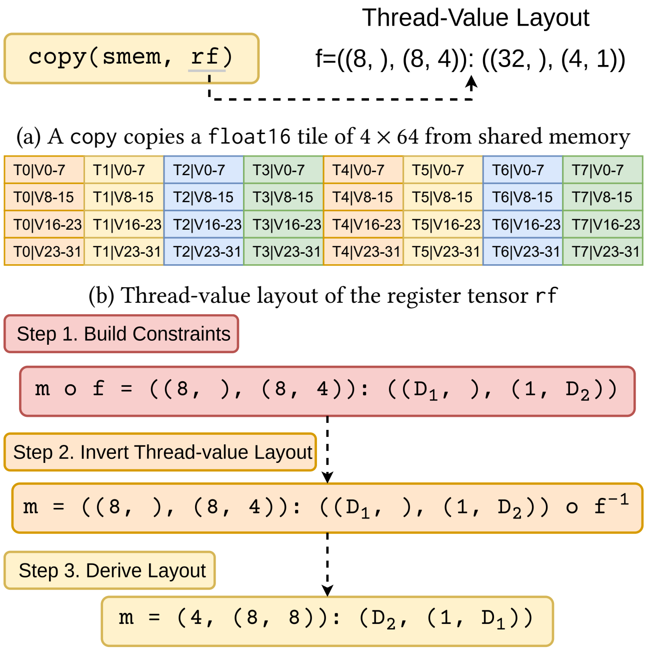
Now, let's consider the first equation in Theorem 1. To relate the mapping functions $f_A \circ p_A^{-1}$ (for matrix `A`) and $f_C \circ p_C^{-1}$ (for matrix `C`), we construct the following mappings by composing the projection and embedding functions:

$$\begin{aligned}
\mu_M \circ \left(f_C \circ p_C^{-1}\right) \circ \eta_M &: m_I \mapsto m_T \\
\tilde{\mu}_M \circ \left(f_A \circ p_A^{-1}\right) \circ \tilde{\eta}_M &: m_I \mapsto m_T
\end{aligned}$$

Since both functions map from the instruction operands'(A and C) m-coordinate to the matrix tensors'(`A` and `C`) m-coordinate, these functions must be identical. This relationship is represented by a commutative diagram (top of Figure 23). Similarly, we can relate the mappings for matrix `B` and matrix `C`(the second equation in Theorem 1) and for matrix `A` and matrix `B` (the third equation in Theorem 1). These relationships are represented with communicative diagrams, as shown in the middle and bottom of Figure 23. □

C Shared Memory Layout Synthesis

In this section, we present a detailed discussion of the memory layout constraint derivation. We begin with the example in Section 5.2 to illustrate the core concepts.



Thread-Value Layout
copy(smem, rf)
f=((8,), (8, 4)): ((32,), (4, 1))

(a) A copy copies a `float16` tile of 4×64 from shared memory

T0|V0-7	T1|V0-7	T2|V0-7	T3|V0-7	T4|V0-7	T5|V0-7	T6|V0-7	T7|V0-7
T0|V8-15	T1|V8-15	T2|V8-15	T3|V8-15	T4|V8-15	T5|V8-15	T6|V8-15	T7|V8-15
T0|V16-23	T1|V16-23	T2|V16-23	T3|V16-23	T4|V16-23	T5|V16-23	T6|V16-23	T7|V16-23
T0|V23-31	T1|V23-31	T2|V23-31	T3|V23-31	T4|V23-31	T5|V23-31	T6|V23-31	T7|V23-31

(b) Thread-value layout of the register tensor `rf`

Step 1. Build Constraints
m o f = ((8,), (8, 4)): ((D_1,), (1, D_2))
Step 2. Invert Thread-value Layout
m = ((8,), (8, 4)): ((D_1,), (1, D_2)) o f^{-1}
Step 3. Derive Layout
m = (4, (8, 8)): (D_2, (1, D_1))

(c) Infer the shared memory layout

Figure 24. Example: shared memory layout constraints

The Concrete Example. In the example in Seciont 5.2, we have a `copy` operation that loads a `float16` tile of size 4×64 from shared memory into the register files. In addition, we assume a 16-byte `ld` instruction is selected to implement the `copy` operation, and the thread-value layout f of the register tensor inferred by Algorithm 1 is

$$f = ((8,), (8, 4)) : ((32,), (4, 1)).$$

Figure 24 visualizes the data distribution of the register tensor: 8 threads each access 8×4 elements.

The `ld` instruction requires 16-byte alignment. Since each `float16` element occupies 2 bytes, the 16-byte alignment mandates that the innermost 8 elements (16 bytes / 2 bytes per element) accessed by each thread must be contiguous in shared memory. In addition, the composition of the shared memory layout m and the thread-value layout f yields the address assigned to each thread. Thus, the composite function $m \circ f$ must satisfy:

$$m \circ f = ((8,), (8, 4)) : ((d_1,), (1, d_2))$$

where d_1 and d_2 are some undetermined strides.

To solve for m, we composite the right inverse f^{-1} with both sides of the above constraints:

$$\begin{aligned} m &= ((8,), (8, 4)) : ((d_1,), (1, d_2)) \circ f^{-1} \\ &= (4, 8, 8) : (d_2, 1, d_1) \end{aligned}$$

This equation implies that the layout m should have a shape of $(4, 8, 8)$, with the middle dimension (size 8) being contiguous (stride 1), while the remaining strides (d_1, d_2) are undetermined free strides. Based on this observation, we can apply the same approach to every `copy` operation corresponding to a shared memory tensor. By comparing the candidate memory layouts, we can verify that the alignment requirements are satisfiable and then materialize a shared memory layout that meets all constraints.

General Case. Consider a `copy` operation where the input and output tensors have thread-value layouts f and g, respectively, and each data element in the tile occupies K bytes. Without loss of generality, assume the input tensor resides in shared memory and that the instruction I executing the copy requires an alignment of L bytes. Composing the thread-value layout f with the memory layout m assigns a memory address to each thread within the tile. Since every address must be a multiple of L bytes, we require:

$$\begin{aligned} m \circ f = \big((t_0, \ldots, t_m), (L/K, v_1, \ldots, v_n)\big) : \\ \big((d_0, \ldots, d_m), (1, l_1, \ldots, l_n)\big), \end{aligned}$$

where $t_0, \ldots, t_m$ and $v_1, \ldots, v_n$ denote the thread and value shapes, respectively, and $d_0, \ldots, d_m$ and $l_1, \ldots, l_n$ denote the corresponding thread and value strides.

Thus, the layout m can be given as:

$$\begin{aligned} m = \big((t_0, \ldots, t_m), (L/K, v_1, \ldots, v_n)\big) : \\ \big((d_0, \ldots, d_m), (1, l_1, \ldots, l_n)\big) \circ f^{-1}. \end{aligned}$$

Here, $d_0, \ldots, d_m$ and $l_1, \ldots, l_n$ are free strides. We apply this process for all copy operations and verify the satisfiability by comparing the known strides of the candidate layouts. If the shared memory layouts are satisfiable, we materialize the unknown strides and synthesize a unified shared memory layout that meets all the alignment requirements, as described in Section 5.2.