Hoppolyta: Polyhedral Kernel Generation Meets Hopper Architecture

Aravind Acharya
NVIDIA
India
aravinda@nvidia.com

Somashekaracharya Gunasagara Bhaskaracharya
NVIDIA
India
sbhaskaracha@nvidia.com

Evghenii Gaburov
NVIDIA
USA
egaburov@nvidia.com

Bin Fan
NVIDIA
USA
binf@nvidia.com

Alexander Collins
NVIDIA
UK
acollins@nvidia.com

Bastian Hagedorn
NVIDIA
Germany
bhagedorn@nvidia.com

Vinod Grover
NVIDIA
USA
vgrover@nvidia.com

Abstract

Modern deep learning workloads require increasingly efficient tensor operations on GPU accelerators. NVIDIA's Hopper architecture introduces specialized tensor cores with advanced features such as asynchronous warpgroup matrix multiply-accumulate instructions and bulk copy operations that present new kernel generation challenges and optimization opportunities.

We present Hoppolyta, an automatic polyhedral kernel generator that targets Hopper tensor cores for efficient execution of operator graphs consisting of matrix multiplications, convolutions, pointwise, reshape and permute operations. Our approach combines polyhedral compilation techniques with Hopper-specific optimizations, leveraging asynchronous bulk copy instructions and warpgroup operations to maximize tensor core utilization. The system takes high-level Python DSL specifications and automatically generates optimized CUDA kernels. Our experimental evaluation demonstrates geometric mean speedups of **2.33**× and **1.35**× on subgraphs from inference models in Torchvision and Hugging Face Transformers networks. Furthermore, on these networks, we observe an end-to-end performance improvement of **1.43**× and **1.41**× respectively over the PyTorch (eager) baseline when using kernels generated by Hoppolyta.

CCS Concepts

• **Software and its engineering → Source code generation**.

Keywords

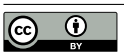
Polyhedral Compilation, NVIDIA Hopper Tensor Cores, Fused Kernel Generation, Flash Attention

1 Introduction

The NVIDIA Hopper architecture [23] introduces fundamental changes that distinguish it from previous tensor core generations including Ampere, Turing and Volta. Hopper tensor cores support asynchronous warpgroup matrix multiply-accumulate (WGMMA) instructions that operate across four contiguous warps (128 threads). This warpgroup approach enables cooperative execution across multiple warps, significantly increasing computational throughput compared to previous warp-level operations. Hopper also features asynchronous tensor memory accelerator (TMA), implemented through `cp.async.bulk.tensor` instructions, for non-blocking global-to-shared memory transfers. Together, TMA and WGMMA enable overlap between data movement and computation, but they also require explicit producer-consumer management through specialized barriers. The warpgroup execution model further complicates thread mapping and shared memory management by requiring coordination across four cooperating warps. This deviates from the traditional synchronous warp-level execution model of previous NVIDIA tensor core generations.

Polyhedral compilation provides a mathematical framework for analyzing and transforming loop nests by representing iteration spaces and data dependencies as integer polyhedra [3, 31]. This approach enables systematic reasoning about complex loop transformations. The application of polyhedral techniques for GPU code generation [17, 32] has been an active area of research, driven by the need to address complex memory hierarchies and massive parallelism through specialized transformations. Recent approaches targeting tensor core architectures, in particular the Volta and Ampere architectures [4, 5, 19], have demonstrated effective polyhedral approaches for generating high performance matrix multiplication kernels. These approaches establish multi-level tiling strategies that map computations across GPU memory hierarchies and employed reuse analysis to guide memory promotion. However, existing approaches target warp-level MMA operations and synchronous memory transfers, making them inadequate for Hopper. The transition to warpgroup operations and asynchronous memory transfers requires new implementation strategies for automatic kernel generation.

Modern deep learning workloads require advanced kernel fusion and support for operations like convolutions and attention mechanisms. Convolution implementation through implicit GEMM introduces data layout transformation challenges, particularly for



This work is licensed under a Creative Commons Attribution 4.0 International License.
PACT '26, Chicago, IL, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2915-7/2026/10
https://doi.org/10.1145/3838684.3846869

```
for (int i = 0; i < M; i++)
  for (int j = 0; j < N; j++)
    for (int k = 0; k < K; k++)
/* S */ C[i][j] += A[i][k] * B[j][k];
```

Figure 1: Canonical MNK loop-nest for transposed matrix multiplication.

image tensor replication in shared memory. Padding, stride and dilation parameters combined with asynchronous memory operations create complex dependence patterns.

In this work, we present Hoppolyta, a polyhedral compiler for Hopper tensor cores with the following contributions:

- An automatic approach relying on polyhedral analysis and transformations, guided by heuristics, to efficiently target Hopper tensor cores.
- A comprehensive implicit GEMM based convolution kernel generation approach that leverages the GEMM kernel generation infrastructure.
- Fusion of pointwise epilogues and layout transformations (e.g, reshape, permute) with GEMM and convolution operations.
- Automatic fusion of multi-head attention based on the flash attention algorithm, employing warp specialization.
- We evaluate Hoppolyta on graphs corresponding to computational patterns from models in Torchvision and HuggingFace Transformers networks demonstrating speedups of 2.33× and 1.35× respectively over PyTorch (eager).
- We observe end-to-end performance improvements of 1.43× and 1.41× respectively over PyTorch (eager).

After introducing the background in Section 2, we present GEMM and convolution code generation in Sections 3 and 4, followed by operator fusion in Section 5. The Hoppolyta compiler pipeline is presented in Section 6. The details of our experiments are provided in Section 7. We discuss the related work in Section 8, and conclude in Section 9.

2 Background

This section provides background on polyhedral compilation and key instructions that target NVIDIA Hopper tensor cores.

2.1 Polyhedral Compilation

The polyhedral model provides a mathematical abstraction for reasoning about loop-nests by representing them using integer polyhedra. For example, consider the *transposed matrix multiplication* example shown in Figure 1, where a matrix A is multiplied with the transpose of the matrix B to produce the output matrix C. The iteration domain of the statement S is given by:

$$\mathcal{I}_S = \{S[i,j,k] \mid 0 \le i < M, 0 \le j < N, 0 \le k < K\}. \quad (1)$$

In the polyhedral model, data dependences in a loop-nest are captured through affine relations that connect source and target iterations in the corresponding iteration domains. For example, the flow dependence carried by the k-dimension of the matrix multiplication is given by:

$$\{S[i,j,k] \rightarrow S[i,j,k-1] \mid 0 \le i < M, 0 \le j < N, 1 \le k < K\}. \quad (2)$$

The mathematical foundation of polyhedral compilation enables precise modeling of complex loop transformations including tiling, fusion, distribution and interchange as affine mappings (quasi-affine in some cases) from iteration domains to a multi-dimensional integer space. The composability of affine transformations makes it well-suited for applying sequences of optimizations in a systematic manner while preserving semantic equivalence. Algorithms for schedule synthesis and transformations are readily available in the Integer Set Library (ISL) [31]. Tools like ISL, CLooG [3], Omega [7] enable code generation from polyhedral schedules.

2.2 Hopper Tensor Cores

The NVIDIA Hopper architecture introduces significant improvements to tensor core capabilities through specialized hardware units and instruction sets that enable efficient cooperative computation and asynchronous data movement.

2.2.1 Warpgroup Matrix Operations. Hopper introduces asynchronous warpgroup matrix multiply-accumulate (WGMMA) instructions [24] that enable multi-warp cooperation on matrix operations. Each warp-group consists of 4 contiguous warps (128 threads) that collectively execute the `wgmma.mma_async` instruction, which requires operand B in shared memory while operand A can reside in either registers or shared memory, with `wgmma.commit_group` and `wgmma.wait_group` signaling operation start and completion. Matrix descriptors [26] are used to describe the operands of the WGMMA operation. WGMMA operates on various datatypes. Though Hoppolyta supports float16, bfloat16 and int8 datatypes, for brevity, we will only consider float16 datatypes with float32 accumulation in the rest of this paper. Float16 WGMMA instructions are of shape $64 \times n \times 16$, where n is any multiple of 8 between 8 and 256, all of which are supported by Hoppolyta.

2.2.2 Tensor Memory Accelerator and Data Movement. The Tensor Memory Accelerator (TMA) relies on asynchronous bulk copy instructions (`cp.async.bulk.tensor`) [25] to accelerate tensor data movement. These instructions operate in multiple modes and on tensors of different dimensionalities. TMA supports tiled mode and im2col mode operations for 2D and 4D tensors respectively. The im2col mode can be used for convolutions modeled as implicit GEMM (see Section 4). TMA operates through tensor maps [22] that describe multi-dimensional tensor layouts, dimension sizes, strides, box sizes and swizzling patterns to facilitate efficient data movement. Built-in swizzling mechanisms automatically rearrange data layouts in shared memory to minimize bank conflicts and maximize memory throughput based on tensor map configurations. TMA operations integrate with mbarrier synchronization primitives to coordinate data movement with computation, enabling high-bandwidth transfers with minimal impact on compute resources.

3 GEMM Kernel Generation

This section describes our approach for generating generalized matrix-matrix multiplication (GEMM) kernels targeting Hopper

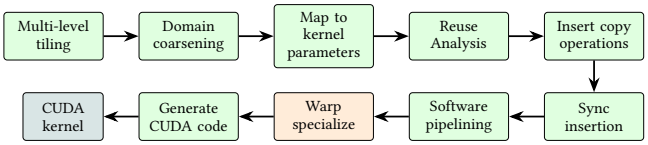


Figure 2: GEMM code generation pipeline.

```
DOMAIN: "[M, N, K] -> {S[i, j, k]:0 <= i < M and 0 <= j < N and 0 <= k < K}"
  SCHEDULE: // Block level tiling (128×128×64)
    [{ S[i, j, k]->[floor(i/128), floor(j/128), floor(k/64)] }]
  SCHEDULE: // Warpgroup level tiling (64×128×64)
    [{ S[i, j, k]->[floor(i/64)-2*floor(i/128),
          floor(j/128)-2*floor(j/128), 0, i - 64*floor(i/64), j -
          128*floor(j/128), k - 64*floor(k/64)] }]
```

Figure 3: Block and Warpgroup level tiling.

```
DOMAIN: "[M, N, K]->{ S[i, j, k]:0 <= i < M and 0 <= j < N and 0 <= k < K
  and 64*floor(i/64) = i and 128*floor(j/128) = j and 16*floor(k/16) = k }"
  SCHEDULE:
    [{ S[i, j, k]->[floor(i/128), floor(j/128), floor(k/64),
          floor(i/64)-2*floor(i/128), floor(j/128)-2*floor(j/128),
          0, i - 64*floor(i/64), j - 128*floor(j/128), k -
          64*floor(k/64)] }]
```

Figure 4: Domain coarsening.

tensor cores. A GEMM kernel on Hopper consists of asynchronous bulk copy instructions that load input tensors from global to shared memory, a `wgmma.mma_async` operation on the shared memory tiles and an epilogue store operation that stores the result of the wgmma operation to global memory. Tensor maps provide metadata for these operations. We employ an *AsyncRef* data structure to manage the asynchronous data movement and synchronization. The accumulator tile is stored to global memory via shared memory, with tensor elements reordered to enable vectorized stores. The entire kernel generation pipeline in Hoppolyta is summarized in Figure 2 and the remainder of this section describes these individual steps.

3.1 Multi-level Tiling

Consider the operation $C = matmul(A, B)$, where A is an $M \times K$ matrix, B is a $K \times N$ matrix and C is an $M \times N$ matrix. In order to target the `wgmma.mma_async` instruction which is the basic operation for the Hopper tensor core, we employ a 2-level tiling strategy where the first level is the thread block level and the second level is the warp-group level. The schedule tree transformations illustrating tiling with tile sizes of $128 \times 128 \times 64$ and $64 \times 128 \times 64$ respectively are shown in Figure 3. Note that each thread block can contain multiple warp-groups. The iterations within each warp-group are further tiled to fit the shape of the `wgmma.mma_async` instruction via domain coarsening. The domain coarsening transformation shown in Figure 4 ensures that each instance of the matmul operation now executes a wgmma operation of shape $64 \times 128 \times 16$ as opposed to the scalar multiply and accumulate operation shown in Figure 1. These transformations are correct because the canonical GEMM dependence in Equation 2 is non negative in every dimension.

```
DOMAIN: "[M, N, K, cwgIdx_y, cwgIdx_x, bid0, bid1] ->
  { wgmma[i, j, k] :cwgIdx_x = 0 and 128*bid0 < M and 128*bid1 < N and
    i >= 64*cwgIdx_y + 128*bid0 and i >= 128*bid0 and 0 <= i <= 127 + 128*bid0
    and i <= 63 + 64*cwgIdx_y + 128*bid0 and j >= 128*bid1 and
    0 <= j <= 127 + 128*bid1 and k >= 0 and 64*floor((k)/64) < K }"
  SCHEDULE:
    [{wgmma[i, j, k] -> [(bid0), (bid1), floor((k)/64), cwgIdx_y]}
```

Figure 5: Schedule tree after mapping to kernel parameters.

3.2 Schedule Dimensions to Kernel Parameters

The outermost parallel dimensions are mapped to kernel parameters `bid0` and `bid1` respectively representing the block indices of the grid. Similarly, the parallel dimensions corresponding to warp-group level tile iterators can be mapped to kernel parameters `cwgIdx_y` and `cwgIdx_x` respectively, which represent the warp-group indices. These warpgroup indices are computed based on the linear index of the thread within the thread block. In the case of batched GEMMs, the batches are completely parallelized along the z-dimension of the grid. The resulting schedule tree after mapping these dimensions to kernel parameters is shown in Figure 5.

3.3 Reuse Analysis

We analyze tensor access relations to identify data tiles that are reused at thread block and warp-group levels. In the running example, tensor A exhibits reuse along dimension j at the thread block level and warp-group level. The data tiles that are reused at the thread block level are referred to as shared memory data tiles as they can be promoted to reside in shared memory of the GPU memory hierarchy. Given a (partial) schedule and the access relations, our data reuse analysis determines the depth at which the data tiles are reused along with the bounds of the data tile. Although our approach builds upon the work of Bhaskaracharya et al. [4], reuse analysis can also be implemented using PPCG [32] or MLIR's *affine-data-copy-generate* transform [20] with minor modifications.

3.4 AsyncRef Abstraction

Our `AsyncRef` data structure, based on Chen et al. [8], encapsulates several key components required for asynchronous bulk copy operations as shown in Figure 6. These include (1) shared memory addresses for tiles corresponding to the two input tensors loaded for `wgmma.mma_async` operations, (2) barrier fields namely `mbar_full` and `mbar_empty` that coordinate the producer and consumer stages of asynchronous bulk copy instructions and (3) the pipeline depth parameter indicating the number of `cp.bulk.async` instructions that can be issued before waiting on the first one to complete.

The constructor initializes the `AsyncRef` instance by accepting the starting address of the mbarrier array (with `mbar_full` preceding `mbar_empty`), the shared memory addresses for data tiles of both input tensors and their corresponding tensor maps. Producer and consumer stage parities are initialized appropriately during construction.

The `put()` method performs a try-wait operation to verify the empty state, then initiates TMA loads for both data tiles using tile offsets provided by the calling kernel. The method concludes by inverting the parity in preparation for the subsequent pipeline phase. Analogously, the `get()` method executes a try-wait operation

```
template <class T> class AsyncRef {
  public:
    __device__ __forceinline__ AsyncRef(int64_t *mbar_, T
          *a_sh_, T *b_sh_, const CUtensorMap *tma_desc_A_,
          const CUtensorMap *tma_desc_B_, uint32_t depth_)
      : mbar_full(mbar_), mbar_empty(mbar_ + depth_),
            a_sh(a_sh_), b_sh(b_sh_), tma_desc_A(tma_desc_A_),
            tma_desc_B(tma_desc_B_), prod_parity(1),
            cons_parity(0), depth(depth_), prod_i(0),
            cons_i(0), consumed_i(0)
    { } // constructor
    __device__ __forceinline__ void put( const uint32_t offA1,
          const uint32_t offA0, const uint32_t tA1, const
          uint32_t tA0, const uint32_t boxA0, const uint32_t
          offB1, const uint32_t offB0, const uint32_t tB1,
          const uint32_t tB0, const uint32_t boxB0) {
      const uint32_t tx_count = (tA1 * tA0 + tB1 * tB0) *
            sizeof(T);
      mbarrier_try_wait(&mbar_empty[prod_i], prod_parity);
      mbarrier_expect_tx(&mbar_full[prod_i], tx_count);
#pragma unroll for (int i = 0; i < (tA0 / boxA0); ++i) //
       tma() for TMA load of A tile when buffer empty
      tma(a_sh + prod_i * tA1 * tA0 + tA1 * boxA0 * i, 0,
            tma_desc_A, offA1, offA0 + boxA0 * i,
            &mbar_full[prod_i]);
#pragma unroll for (int i = 0; i < (tB0 / boxB0); ++i) //
       tma() for TMA load of B tile when buffer empty
      tma(b_sh + prod_i * tB1 * tB0 + tB1 * boxB0 * i, 0,
            tma_desc_B, offB1, offB0 + boxB0 * i,
            &mbar_full[prod_i]);
      mbarrier_arrive(&mbar_full[prod_i]);
      prod_i = (prod_i == (depth - 1) ? 0 : prod_i + 1);
      prod_parity = ((prod_i == 0) ? (prod_parity ^ 1) :
            prod_parity);
    }
    __device__ __forceinline__ uint32_t get() {
      mbarrier_try_wait(&mbar_full[cons_i], cons_parity); //
            wait for buffer to be full, update index and parity
      uint32_t retVal = cons_i;
      cons_i = (cons_i == (depth-1) ? 0 : cons_i + 1);
      cons_parity = ((cons_i == 0) ? (cons_parity ^ 1) :
            cons_parity);
      return retVal;
    }
    __device__ __forceinline__ void consumed() {
      mbarrier_arrive(&mbar_empty[consumed_i]);
      consumed_i = (consumed_i == (depth-1) ? 0 : consumed_i+1);
    } };
template <const int N> __device__ __forceinline__ void
       wgmma_wait_group(AsyncRef &aref) {
  asm volatile("wgmma.wait_group.sync.aligned %0;" ::"n"(N));
  aref.consumed();
}
```

Figure 6: AsyncRef class for asynchronous bulk copy operations and synchronization.

to confirm the full state, ensuring that shared memory data tiles are available for subsequent wgmma operations.

The `consumed()` method handles the completion of tile consumption by `wgmma.mma_async` operations. Once the shared memory tiles are fully consumed (synchronized through the `mbarrier_arrive`), the parity is inverted for the next consumer stage phase and hence invoked immediately following the `wgmma.wait_group.sync.aligned` operation.

3.5 Global to Shared Memory Data Transfer

This subsection describes our approach for generating copy operations that transfer data from global memory to shared memory.

```
DOMAIN: "[M, N, K, cwgIdx_y, cwgIdx_x, bid0, bid1, linearId] ->
  { ...; wgmma[i0, i1, i2] : wgmma_dom_constraints;
    aref_put[bid0, bid1, i2, 0, 0, i5, i6] :
      linearId = 0 and bid0 >= 0 and 128*bid0 < M and bid1 >= 0 and 128*bid1 < N
      and i2 >= 0 and 64*i2 < K and 0 <= i5 <= 127 and 0 <= i6 <= 63 }"
  SCHEDULE:
    [{ ...; wgmma[i0, i1, i2] -> [(bid0)]; aref_put[...] -> [(bid0)]
          : ... }]
    SEQUENCE:
    - FILTER: "{ init_0[...] }"
    - FILTER: "{ ...; wgmma[...]; aref_put[...] }"
      SCHEDULE:
        [{ wgmma[i0, i1, i2] -> [(floor((i2)/64))]; aref_put[...] ->
              [(i2)] }]
      SEQUENCE:
      - FILTER: "{ aref_put[...] }"
      - FILTER: "{ wgmma[...] }"
```

Figure 7: After copy insertion.

As identified through the reuse analysis described in Section 3.3, tiles of input tensors that exhibit reuse at the thread block level when computing a tile of the output tensor are promoted to shared memory. This requires defining iteration spaces and polyhedral schedules for all copy operations that move data tiles from global memory to shared memory at appropriate depths.

Consider the running example of a GEMM operation after tiling at the thread block level and warp-group level. At the thread block level, a tile of shape $(128, 64)$ from tensor A and a tile of shape $(64, 128)$ from tensor B are reused to compute a tile of shape $(128, 128)$ for output tensor C. The reuse analysis infers these tile shapes, which determine the iteration domains of the corresponding copy operations. These operations are scheduled at the thread block level, before the GEMM operation at the inferred reuse depth.

Copy iteration domains are first defined element-wise, but Hopper TMA operates at tile granularity. We therefore collapse all integer points in the domain into a single point, yielding a bulk copy that transfers an entire tile. Additionally, a TMA instruction must be initiated by a designated leader thread. Therefore, the copy domain is constrained by the thread-index parameter `linearId` (for example, `linearId = 0`), as shown in Figure 7.

Typically, separate copy operations are generated for each reused input data tile. However, the `AsyncRef` abstraction requires these operations to be combined, enabling a single copy operation to handle data movement for both tiles through the unified `put()` method, where the arguments specify the global memory offsets of the respective data tiles. A key scheduling feature is that wgmma operations are sequenced after the `aref.put()` operation, ensuring TMA loads occur before wgmma computation for each K tile. Therefore, the promoted shared-memory tiles are always produced before the corresponding `wgmma.mma_async` operations consume them.

3.6 Matrix Descriptor for WGMMA

This section describes the generation of GMMA descriptors and accumulator fragments for the WGMMA operation. We analyze the warp-group level access relations of the two input tensors to generate matrix descriptors. Similarly, we analyze the warp-group level access relations of the output tensor to generate accumulator fragments. These GMMA descriptors and accumulator fragments then enable the generation of the WGMMA operation.

```
extern "C" __global__ void __launch_bounds__(256)
kern_0(int M, int N, int K,
  const __grid_constant__ CUtensorMap tma_desc_A,
  const __grid_constant__ CUtensorMap tma_desc_B,
  half * __restrict__ M_3) {
  // ... init bid, cwgIdx_y from block/thread ...
  extern __shared__ half kern_sharedBuffer[];
  half * shared_A = (half *)&kern_sharedBuffer[64];
  half * shared_B = (half *)&kern_sharedBuffer[8256];
  GmmaDescriptor mma_A[1][4][1][1], mma_B[1][4][1][1];
  int64_t *mbar = (int64_t *)&kern_sharedBuffer[0];
  AsyncRef<half> aref(mbar, shared_A, shared_B,
    &tma_desc_A, &tma_desc_B, 1);
  __align__(16) float mma_C[1][1][1][64];
  int buffer0 = 0;
  if (linearId == 0) aref.allocate(1,256);
  __syncthreads();
  for (int c2 = 0; c2 <= (K - 1) / 64; c2 += 1) {
    if (linearId == 0)
      aref.put((128*bid0), (64*c2), 128, 64, 64,
        (128*bid1), (64*c2), 128, 64, 64);
    buffer0 = aref.get();
    asm volatile("wgmma.fence.sync.aligned;" ::);
    for (int c7 = 0; c7 <= 3; c7 += 1) {
      set_gmma_descriptor_a_row<1>(&mma_A[0][c7][0][0],
        0, c7, &shared_A[0], 64*cwgIdx_y, 16*c7, 128, 64);
      set_gmma_descriptor_b_col<1>(&mma_B[0][c7][0][0],
        0, c7, &shared_B[0], 0, 16*c7, 128, 64);}
    for (int c8 = 0; c8 <= 3; c8 += 1)
      mma_hopper_fp32_m64n128k16<0,0>(&mma_C[0][0][0][0],
        &mma_A[0][c8][0][0], &mma_B[0][c8][0][0],
              &mma_C[0][0][0][0]);
    asm volatile("wgmma.commit_group.sync.aligned;" ::);
    wgmma_wait_group<0>(aref);}
  wgmma_store_global_m64n_row<128>(&M_3[
    (64*cwgIdx_y+128*bid0)*N+128*bid1], &mma_C[0][0][0][0],
    M-(64*cwgIdx_y+128*bid0), N-(128*bid1), N,
    &kern_sharedBuffer[64], 0);}
```

Figure 8: Basic GEMM kernel with TMA and WGMMA.

Consider the running example above. The warpgroup-level access relations reveal that data tiles of shape 64×64 and 128×64 are needed to compute the warpgroup-level MMA operation of shape $64 \times 128 \times 64$. However, each WGMMA operates on tiles of shape $64 \times 128 \times 16$. Therefore, 4 WGMMA operations are required to cover a tile of shape $64 \times 128 \times 64$. We use two 1×4 arrays of GMMA descriptors, each for A and B input tensors, with each descriptor targeting one $64 \times 128 \times 16$ sub-tile consumed by a single WGMMA. Each GMMA descriptor requires the start address of its corresponding sub-tile in shared memory. We compute these shared memory offsets from the global memory tile offsets using the reuse analysis described in Section 3.3.

3.7 Synchronization

Since the memory and compute operations are asynchronous, we require additional synchronization operations within the kernel. We introduce these synchronization primitives after scheduling the copy operations. The generated kernel in Figure 8 shows that TMA operations from the `aref.put()` operation are followed by an `aref.get()` operation that waits for data transfer completion. Similarly, `wgmma.commit_group.sync.aligned` and `wgmma.wait_group()` follow `wgmma.mma_async` operations for synchronization. Together, these synchronization points, `aref.get()` and `wgmma.wait_group()`,

```
DOMAIN: "[M, N, K, cwgIdx_y, cwgIdx_x, bid0, bid1, linearId] ->
  { ...; wgmma[i0, i1, i2] : wgmma_dom_constraints;
    aref_put[bid0, bid1, i2, 0, 0, i5, i6] : aref_put_dom_constraints
        }"
  SCHEDULE:
    [{ ...; wgmma[i0, i1, i2] -> [(bid0)]; aref_put[...] -> [(bid0)]
        : ... }]
    SEQUENCE:
    - FILTER: "{ init_0[...] }"
    - FILTER: "{ ...; wgmma[...]; aref_put[...] }"
      SCHEDULE:
        [{ wgmma[i0, i1, i2] -> [(floor((i2)/64))]; aref_put[...] ->
            [(i2-1)] }]
      SEQUENCE:
      - FILTER: "{ aref_put[...] }"
      - FILTER: "{ wgmma[...] }"
```

Figure 9: After pipelining.

make shared-memory tiles visible before consumption by WGMMA and accumulator fragments visible before their subsequent use.

3.8 Accumulator Fragment to Global Memory

Accumulator fragments from `wgmma.mma_async` operations reside in the register file. Each accumulator fragment size depends on the specific `wgmma.mma_async` operation employed. For example, `wgmma.mma_sync` of shape $64 \times 128 \times 16$ produces an accumulator fragment of size 64. A final epilogue store operation moves the accumulator data to global memory. This is modeled as a copy operation that is scheduled after all `wgmma.mma_async` operations complete. In Figure 8, the `wgmma_store_global_m64n128_row()` operation is the corresponding copy operation. Since this final copy is scheduled after the asynchronous MMA operations complete, the epilogue store observes fully produced accumulator fragments.

3.9 Software Pipelining

Software pipelining of TMA and WGMMA operations is essential to mask global memory load latencies with compute operations. This is achieved by introducing a delay between the TMA load and the WGMMA operations via a loop shifting transformation. The outermost tiled k dimension corresponding to the WGMMA operation is shifted by one less than the number of pipeline stages (or the depth of the pipeline). This transformation requires larger shared memory buffers and mbarriers whose allocations are handled by the `AsyncRef` object. Furthermore, pipelining of WGMMA operations can also be performed by using the appropriate depth for the `wgmma_wait_group()` operation.

The key difference in the software pipelined schedule, shown in Figure 9, is that `aref.put()` operation is scheduled at $(i_2 - 1)$, pipelined ahead by one iteration. Therefore, the `aref.put()` operation for tile i_2 occurs during WGMMA operation of tile $i_2 - 1$. Hence the TMA loads for the next iteration overlap with the current WGMMA operation, effectively hiding memory latency without violating the data dependence.

3.10 Warp Specialization

Warp specialization dedicates separate warpgroups for distinct tasks. For example, one warpgroup handles loads (the load warpgroup) while one or more warpgroups perform math operations (math warpgroups). Consider a warp specialized kernel where each

```
// ... shared buffer, aref, accumulator fragment allocation
    (as in basic GEMM) ...
// Warp specialization: math warpgroup (WGMMA)
if (wgId >= 1) {
  for (int c2 = 0; c2 <= (K - 1) / 64; c2 += 1) {
    buffer0 = aref.get();
    ... // set descriptors, wgmma commit and wait (as in
        basic GEMM kernel)
  }
  ... // wgmma_store_global_m64n_row (as in basic GEMM kernel)
} else { // load warpgroup (TMA)
  if (linearId == 0)
    for (int c2 = 0; c2 <= (K - 1) / 64; c2 += 1)
      aref.put(...); // as in basic GEMM kernel
}
```

Figure 10: Warp specialized GEMM kernel.

thread block assigns one warpgroup to handle load operations and assigns the remaining warpgroups to perform WGMMA operations. To achieve this, we first apply all scheduling steps i.e., multi-level tiling, WGMMA instruction generation and memory promotions for asynchronous bulk copy operations, as described in previous sections. We then apply an additional constraint to the `aref.put()` operation's domain, restricting it to `wgId == 0` where `wgId` is a kernel parameter representing the warpgroup index. We also restrict the domains of all other operations by the constraint `wgId >= 1`, effectively dedicating them to math warpgroups. The resulting kernel resembles the basic WGMMA kernel from Figure 8 but includes additional warpgroup index predicates on both load and WGMMA operations as shown in Figure 10. Note that our approach for warp specialization is architecture and operation agnostic making it applicable to any operation on future generations of NVIDIA GPUs like Blackwell. These domain restrictions do not alter the logical schedule relative to the non-warp-specialized case; they only partition work across dedicated load and math warpgroups. Barrier synchronization then ensures that data produced by the load warpgroup is available in shared memory before the math warpgroups consume it.

3.11 Handling Partial Tiles

So far, we have assumed that problem sizes are integral multiples of block tile sizes. However, our kernel generator can handle arbitrary problem sizes through *domain expansion*. This step is performed before multi-level tiling. We inflate the iteration domains of all operations using their respective block level tile sizes. For the running example with tile sizes of 128, 128 and 64 along i, j, and k, the GEMM iteration domain becomes $0 \le i \le M{+}126$, $0 \le j \le N{+}126$, and $0 \le k \le K{+}62$. Domain expansion affects not only iteration domains but also access relations and dependences. Access indices corresponding to the expanded domain result in out-of-bounds accesses unless predicated. We predicate all global memory accesses using appropriate bounds from the original iteration domain to avoid out-of-bounds accesses and fill corresponding shared memory offsets with operation-specific identity values.

4 Convolutions

This section describes our approach to generate kernels for convolutions. We model convolutions as *implicit GEMM* operations [11], thereby reusing our GEMM kernel generation infrastructure from Section 3. We first provide a background for convolutions, then describe the implicit GEMM transformation and associated code generation enhancements.

4.1 Convolutions as Implicit GEMM

Given an input image $\mathcal{I}$ of shape $N \times C \times H \times W$ and a filter $\mathcal{F}$ of shape $K \times C \times R \times S$, the convolution operation produces an output $\mathcal{O}$ of size $N \times K \times P \times Q$ where spatial dimensions P and Q are determined by

$$P = 1 + \frac{H + 2 * p_H - (((R-1) * d_H) + 1)}{s_H},$$
$$Q = 1 + \frac{W + 2 * p_W - (((S-1) * d_W) + 1)}{s_W}, \tag{3}$$

where (p_H, p_W), (s_H, s_W) and (d_H, d_W) represent padding, stride and dilation parameters respectively. Each output element is computed as

$$\mathcal{O}_{n,k,p,q} = \sum_{c=0}^{C} \sum_{r=0}^{R} \sum_{s=0}^{S} \mathcal{I}_{n,c,p+r,q+s} \times \mathcal{F}_{k,c,r,s}. \tag{4}$$

For tensor-core execution, we assume NHWC tensors, although Hoppolyta supports both NHWC and NCHW layouts. The convolution in Equation 4 forms a 7-D loop nest with reductions over C, R, and S dimensions. Flattening these dimensions converts it to matrix multiplication by reshaping the image tensor from $N \times C \times H \times W$ to $(NPQ) \times (CRS)$ and the filter tensor from $K \times C \times R \times S$ to $K \times (CRS)$. Their transposed matrix product yields an $(NPQ) \times K$ result, which is reshaped to $N \times K \times P \times Q$. This transformation replicates input data by a factor of $R \times S$. Explicit GEMM [34] materializes this replication in global memory via `im2col` transformation, whereas implicit GEMM replicates data in shared memory thereby avoiding overhead of global memory transfers.

4.2 Data Layout Transformations

Converting convolution to implicit GEMM requires data layout transformations from 4D tensors to 2D matrices. The filter and output tensors can be directly flattened to 2D arrays whereas transforming the image tensor requires modeling of data replication. The replication of image tensor from 4D NCHW space to 2D $(NPQ) \times (RSC)$ proceeds through a two-stage mapping as illustrated in Figure 11. First, the 4D input access is mapped to a 6D (N, P, Q, R, S, C) intermediate space, then transformed to the final 2D representation. The 4D to 6D mapping is given by

$$\{\mathcal{I}[i_0, i_1, i_2, i_3] \to \mathcal{F}[o_0, o_1, o_2, o_3, o_4, o_5] \mid i_0 = o_0, i_1 = o_5, i_2 = (s_H \times o_1) + (o_3 \times d_H) - p_H, i_3 = (s_W \times o_2) + (d_W \times o_4) - p_W\}. \tag{5}$$

where o_4 and o_5 are bounded by the filter dimensions R and S respectively. The resulting 6D access function is then mapped to the final 2D $(NPQ) \times (RSC)$ space through

$$\{\mathcal{F}[i_0, i_1, i_2, i_3, i_4, i_5] \to \mathcal{M}[o_0, o_1] \mid o_0 = i_0 \times (P \times Q) + i_2 \times P + i_3, \; o_1 = i_1 \times (R \times S) + i_4 \times S + i_5\}. \tag{6}$$

Composing Equations 5 and 6 yields a complete transformation from 4D NCHW space to 2D $(NPQ) \times (RSC)$ space. This represents

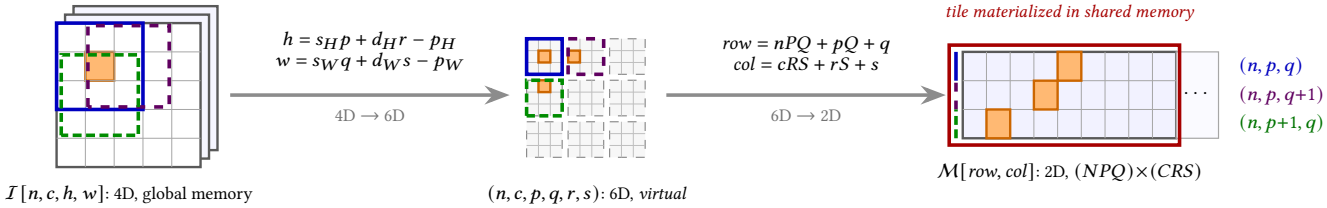


Figure 11: Two-stage implicit GEMM mapping of the image tensor. The 4D tensor in global memory is viewed as a virtual 6D space of $R\times S$ patches, then flattened to a 2D $(NPQ)\times(RSC)$ operand whose 2D tile is materialized in shared memory. The shaded element sits at a different (r, s) in every patch, and hence in a different column of each row.

the mapping from the 4D tensor in global memory and its 2D representation in shared memory, supporting the implicit GEMM computation while preserving spatial and channel relationships.

The data-layout transformer for the filter maps its 4D access function to a 2D access function. The filter tensor in NHWC layout is mapped to a matrix of shape $K \times (RSC)$ using the access function using the data layout transformation function

$$\{\mathcal{F}[i_0, i_1, i_2, i_3] \rightarrow \mathcal{F}[o_0, o_1] : o_0 = i_0 \wedge o_1 = i_2 \times (S \times C) + i_3 \times C + i_1\}, \quad (7)$$

and transpose the resulting matrix before multiplying it with the image matrix of shape $(NPQ) \times (RSC)$. The output of this GEMM operation is a matrix of size $(NPQ) \times K$. This matrix is further reshaped to a 4D tensor of shape $N \times K \times P \times Q$ using the data layout transformation function

$$\{O[i_0, i_1] \rightarrow O[o_0, o_1, o_2, o_3] : i_1 = o_1 \wedge i_0 = o_0 \times (P \times Q) + o_2 \times Q + o_3\}. \quad (8)$$

The polyhedral modeling of data layout transformations can be used to generate kernels for convolutions with constant shapes but not parametric ones. Moreover, data layout transformers, especially for the image tensor, generate complex access expressions leading to significant increase in compilation time, especially for convolutions with non-trivial padding, strides and dilations. Additionally the generated access expressions result in ternary operations that lead to significant loss of performance ($\approx$ 5×). We circumvent these challenges by computing the 4D access indices from 2D indices at runtime using helper functions, while preserving the computational benefits of the implicit GEMM approach.

As a result of the above modeling, the convolution operation becomes equivalent to matrix multiplication with a 3D iteration domain of shape $(NPQ) \times (RSC) \times K$. Our polyhedral scheduler mimics the approach for GEMM kernel generation described in Section 3. Note that the image tensor must be loaded to shared memory in im2col mode while the filter tensor must be loaded to shared memory in tiled mode. This is accomplished by suitably overloading the `put()` method in the AsyncRef data structure described in Section 3.4.

5 Fusion Scenarios

Matrix multiplication and convolutions form the computational backbone of deep learning workloads, but significant performance gains can be achieved by fusing these core operations with prologue and epilogue operations. Kernel fusion reduces memory traffic by exploiting temporal locality, leading to substantial performance improvements. Our framework broadly supports three fusion scenarios that we describe in the rest of this section.

```
// ... shared buffer, aref(depth 2), accumulator and register
      fragments allocation (as in basic GEMM) ...
for (int c2 = -1; c2 <= 11; c2 += 1) { // software pipelining
  if (linearId == 0 && 768 >= 64*c2+65)  // skip tma for c2=11
    aref.put(128*bid0, 64*c2+64, 128, 64, 64, 128*bid1,
          64*c2+64, 128, 64, 64, 2, 128, 15, 17, 256, 3, 2, 6,
     7, 1, 2, 2, 3, 3, 2); // conv 2D params
  if (c2 >= 0) { // consume only when pipeline has data
    buffer0 = aref.get();
    . . . // wgmma fence, set descriptors, mma, commit (as in
          basic GEMM)
    wgmma_wait_group<1>(aref); // pipelining: depth 2
  }}
// Epilogue fusion:
mma_epilog_convert_inplace_hopper<128,half>(
      &mma_C[0][0][0][0]);
mma_load128_fragment_from_gmem_matrix_hopper<128>(
\mma_D[0][0][0][0], &D[...], ...); // load residual
add((half *)&mma_C[0][0][0][0], &mma_D[0][0][0][0],
      &mma_out[0][0][0][0], 64); // pointwise
wgmma_store_global_m64n_row<128>( &out[...],
      &mma_out[0][0][0][0], ...);
```

Figure 12: Convolution kernel with epilogue fusion.

5.1 Epilogue Fusion

Epilogue fusion targets pointwise operations that consume the outputs of GEMM or convolution operations. The key insight is that the results of the WGMMA operation reside in accumulator fragments within the register file, enabling direct computation of the epilogue operation without intermediate memory transfers.

Our polyhedral kernel generator relies on ISL to obtain an initial fused schedule that is outer parallel. For a GEMM with pointwise epilogue operation, the initial fused schedule from ISL contains a single loop-nest with two parallel dimensions due to fusion of the pointwise operation's iteration domain with the i and j dimensions of the GEMM operation. After mapping the GEMM operation to WGMMA instructions, our fusion strategy leverages data reuse analysis to identify when accumulator fragments can be directly consumed by downstream operations. For unary pointwise operations, computations are performed entirely within registers. For n-ary operations requiring additional operands, we generate vectorized load operations (bypassing TMA) with vector width determined by problem size to fetch data from global memory.

```
// Initialize accumulators (registers)
acc_max = -inf; acc_sum = -1; O_tile = 0

// Loop over KV tiles (reduction dimension)
for each KV_tile in sequence:
  // BMM1: Compute attention scores for current tile
  S_tile = Q_tile @ K_tile^T

  // Compute tile statistics with updates
  tile_max = rowmax(S_tile)
  prev_max = acc_max
  acc_max = max(tile_max, prev_max)

  // Softmax numerator with updated max
  P_tile = exp(S_tile - acc_max)
  tile_sum = rowsum(P_tile)

  // Corrective updates (Flash Attention)
  correction = exp(prev_max - acc_max)
  prev_sum = acc_sum
  acc_sum = tile_sum + correction * prev_sum
  O_tile = O_tile * correction

  // BMM2: Accumulate output
  O_tile += P_tile @ V_tile

// Final normalization (deferred scaling)
O_final = O_tile / acc_sum
```

Figure 13: Fused MHA kernel pattern with online softmax and corrective updates.

In the case of implicit GEMM convolution, the convolution output is a 2D matrix that is reshaped as a 4D tensor. This reshape is then propagated across convolution consumers when valid, expressing the pointwise iteration domains and epilogue operands in the same flattened 2D space. This allows us to reuse the GEMM-with-pointwise fusion path. Figure 12 shows an example code snippet for this fusion scenario.

5.2 Fused Multi-Headed Attention

Multi-headed attention (MHA) takes three 4D tensors Q, K, V of shape $B \times H \times S \times D$ (batch, heads, sequence length, embedding dimension) and computes: (1) BMM1 — attention scores $S = Q \cdot K^T$ (shape $B \times H \times S \times S$), (2) Softmax — normalizing scores $X = \text{Softmax}(S)$, which decomposes into a row-wise max reduction (`max_reduction`), exponentiation of the difference (`exp_diff`), a row-wise sum reduction (`sum_reduction`) and a final division, and (3) BMM2 — computing the output $O = X \cdot V$ (shape $B \times H \times S \times D$).

The state-of-the-art approach to fuse an MHA graph is based on the Flash Attention algorithm [13, 14]. Flash Attention leverages properties of exponentiation ($\exp(x) \cdot \exp(y) = \exp(x + y)$ and $\exp(x)/\exp(y) = \exp(x - y)$) to enable tiled computation along the reduction dimension while maintaining numerical stability. These properties allow incremental computation of softmax statistics (row-wise max and sum) across tiles while maintaining numerical correctness through corrective updates. Algorithmic and scheduling extensions to handle the asynchronous nature of operations on the Hopper architecture have also been proposed in the literature [13, 28].

The MHA computation processes the key/value (KV) sequence in tiles. For each KV tile iteration, the algorithm maintains running accumulators: `acc_max` (running maximum for softmax stability), `acc_sum` (running sum for normalization), and `O_tile` (partial output accumulator). As each new KV tile is processed, the algorithm computes statistics for the current tile and updates the running accumulators using corrective factors that account for the change in the running max between iterations. This allows the softmax to be computed incrementally without materializing the full attention matrix. The fused kernel follows the pattern shown in Figure 13.

Figure 14 illustrates the schedule tree transformation pipeline for a representative problem (B=2, H=3, S=1024, D=128, Q/KV tile width 128). The remainder of this section describes the pipeline stages.

Initial Schedule from ISL: The initial schedule from ISL shown in Figure 14a consists of a sequence of three outer-parallel loop nests. The reduction dependences prevent fusion across the sequence boundaries.

Multi-level Tiling: The multi-level tiling pass strips-mines the query dimension (creating Q tiles) and the key/value sequence dimension (creating KV tiles), enabling block-level parallelism while maintaining the working set in shared memory. The query dimension is stripped-mined by 128 and the key/value sequence dimension is stripped-mined by 128. This creates 8 Q tiles (1024/128) per batch×head. Each Q tile processes 8 KV tiles sequentially. The division operation has a fixed tile coordinate since it operates on accumulated output. The resulting schedule is shown in Figure 14b.

Fusion with Dependence Relaxation: The fusion with dependence relaxation pass extracts a common outer band, hoisting parallel dimensions (batch, heads, Q tiles) while the KV tile dimension becomes sequential. All operations now share common outer dimensions, enabling fusion through the relaxed reduction dependence as shown in Figure 14b. The fifth dimension iterates over KV tiles for softmax operations.

Corrective Update Insertion: Once the KV dimension has been tiled, a plain softmax over a KV tile is not equal to the global softmax over the full sequence. A pattern-specific compiler pass turns this tiled softmax into the online softmax recurrence of Figure 13. The operations of the input MHA graph (Lines 7, 10, 15, 16, 25, and 28) namely BMM1, `max_reduction`, `exp_diff`, `sum_reduction`, BMM2, and `division`, respectively, and the KV tile loop (Line 5) are produced by the two preceding stages. The pass inserts every remaining statement, i.e., the accumulator initialization (Line 2), the running-max update `acc_max = max(tile_max, prev_max)` (Lines 11–12), the corrective factor `correction = exp(prev_max - acc_max)` together with the running-sum update `acc_sum = tile_sum + correction * prev_sum` (Lines 19–21), and the rescaling `O_tile = O_tile * correction` of the partial output (Line 22). It also rewires `exp_diff` to subtract the running `acc_max` (Line 15) and hoists `division` out of the KV tile loop as deferred normalization (Line 28). These updates are introduced in two forms: (1) source-level rewrites applied before the polyhedral IR is constructed, for instance the deferred normalization above, and (2) schedule-tree updates that connect a consumer to its upstream producer, for instance wiring `sum_reduction` to the `max_reduction` result so that the corrective factor can be computed.

```
SEQUENCE:
  - FILTER: {bmm1, cast_fp32, max_reduction}
    SCHEDULE: [(b), (h), (s1), (s2), (d)]
    coincident: [1, 1, 1, 1, 0]  // d is reduction dimension
  - FILTER: {exp_diff, sum_reduction}
    SCHEDULE: [(b), (h), (s1), (s2)]
    coincident: [1, 1, 1, 1]     // All parallel
  - FILTER: {bmm2, division}
    SCHEDULE: [(b), (h), (s1), (s2), (d)]
    coincident: [1, 1, 1, 1, 0]  // Different reduction dimension
```

(a) Initial schedule from ISL.

```
SCHEDULE: [
  all_operations -> [(floor(s1/128))],   // Q tile -> blockIdx.z
  all_operations -> [(b)],                 // Batch -> blockIdx.y
  all_operations -> [(h)],                 // Head -> blockIdx.x
  all_operations -> [(0)],
  bmm1,max_reduction,exp_diff,sum_reduction -> [(floor(s2/128))], // KV
  {bmm2 -> [(floor(d/128))]},
  division -> [(8)]                        // Fixed (after all tiles)
]
coincident: [1, 1, 1, 1, 0]  // 4 parallel, sequential (KV loop)
  SEQUENCE:
    - FILTER: {bmm1, cast_fp32, max_reduction}
    - FILTER: {exp_diff, sum_reduction}
    - FILTER: {bmm2, division}
```

(b) After fusion with dependence relaxation.

```
SEQUENCE:
  - FILTER: {bmm1, cast_fp32, max_reduction}
    SCHEDULE: [(floor(s1/128)), (b), (h), (0), (floor(s2/128))]
      SEQUENCE:
        - FILTER: {bmm1}
          child: [warpgroup/fragment tiling for BMM...]
        - FILTER: {cast_fp32, max_reduction}
          child: [warpgroup tiling for elementwise/reduction...]
  - FILTER: {exp_diff, sum_reduction}
    SCHEDULE: [(floor(s1/128)), (b), (h), (0), (floor(s2/128))]
    child: [warpgroup tiling for elementwise/reduction...]
  - FILTER: {bmm2, division}
    SCHEDULE: [(floor(s1/128)), (b), (h), (0), (floor(d/128))]
      SEQUENCE:
        - FILTER: {bmm2}
          child: [warpgroup/fragment tiling for BMM...]
        - FILTER: {division}
          child: [warpgroup tiling for elementwise...]
```

(c) After multi-level tiling.

```
DOMAIN: "[cwgIdx_y, bid0, bid1, bid2] -> {
  bmm1[bid1, bid2, i2, i3, i4]:
    cwgIdx_y ∈ {0,1} and // two warp groups per block
    0 <= bid1 <= 1 and 0 <= bid2 <= 2 and // batch and head indices
    64*cwgIdx_y + 128*bid0 <= i2 <= 63 + 64*cwgIdx_y + 128*bid0 and
    0 <= i3 <= 1023 and 0 <= i4 <= 127; ... }"
```

(d) After mapping to kernel parameters.

Figure 14: MHA schedule tree transformations (representative problem: B=2, H=3, S=1024, D=128, Q/KV tile width 128).

The pass is deliberately pattern-specific rather than a general reduction-fusion mechanism. It reconstructs the online softmax updates of Flash Attention 2 [13] and does not implement the GEMM-softmax pipelining of Shah et al. [28]. Only the single matmul-softmax reduction dependence is relaxed, and that relaxation, and hence the entire pass, is triggered solely by the MHA pattern hint emitted by the pattern matcher [12]. Correctness of the inserted updates is the property of the flash attention algorithm [13], and rests on the exponential identities described above. Since fusion is driven by dependences rather than by a fixed kernel template, an arbitrary composition of pointwise operations may also be sandwiched between BMM1 and the softmax, which the fixed matmul-softmax-matmul templates of the vendor libraries do not support.

Mapping to Kernel Parameters: Once the corrective updates are inserted, the iteration domains are constrained based on block and warp indices for warp specialization as shown in Figure 14d. The warpgroup `cwgIdx_y` handles 64 rows of the 128-row Q tile (2×64=128).

Memory Promotion: The reuse analysis identifies the Q tile that is reused across KV tile iterations, as well as the K and V tiles, for shared memory promotion. The accumulators `acc_max`, `acc_sum`, and `O_tile` remain in registers across iterations.

Additionally, we also perform software pipelining of TMA and WGMMA operations 3.9 and warp-specialization 3.10 where all loads are performed by a dedicated load warpgroup using TMA while the compute corresponding to the WGMMA instructions of the two GEMMs and softmax are performed by math warpgroups.

5.3 Layout Transformation Fusion

Our framework handles permute and reshape operations as affine and quasi-affine transformations respectively. These transformations are orthogonal to the above two fusion scenarios and are seamlessly integrated into the code generation process through relational composition applied to memory access relations. We treat layout transformations as mathematical relations on array access relations, avoiding explicit data movement operations and instead generating memory access patterns that directly implement the required transformations during load and store operations.

```python
import hoppolyta as hp
# Declare inputs and outputs
bmm1 = hp.matmul(
  Q, K.transpose([0, 1, 3, 2]), compute_type=hp.float32)
cast1 = R.astype(hp.float32)
smax = cast1.softmax([3]).astype(hp.float16)
X = hp.matmul(smax, V, compute_type=hp.float32)
hp.compile(..., (SM_90) (Pattern_FMHA))
# Allocate inputs and outputs in device memory
...
# Run the kernel by passing device memory pointers
hp.run(...)
```

Figure 15: Python DSL for FMHA

6 Implementation

The Hoppolyta compilation pipeline consists of frontend, runtime, and polyhedral kernel generation stages as shown in Figure 16.

6.1 Frontend and Runtime

Tensor operations are specified using a high-level Python DSL that is translated into Hoppolyta IR. Figure 15 shows an FMHA example using PyTorch-like syntax for composing multiple operators. Given a graph, the partitioning step decomposes the graph into subgraphs and provides a pattern hint for each subgraph using a pattern matcher [12]. The polyhedral backend generates CUDA kernels

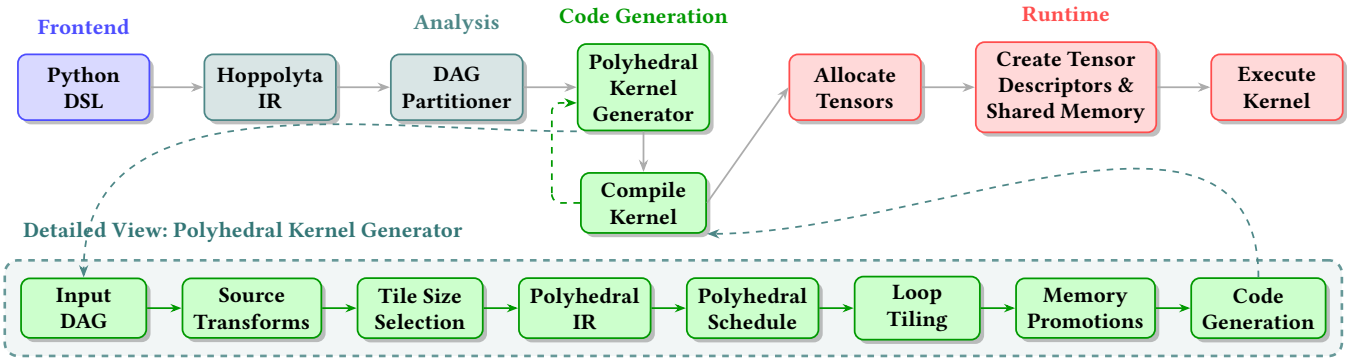


Figure 16: Hoppolyta compilation flow from Python DSL to kernel execution

for each partition and provides information necessary to facilitate tensor allocation and tensor map creation. The Hoppolyta runtime utilizes this information to create host-side tensor maps and allocate memory for temporary and output buffers required by the generated kernels. The generated kernels are then compiled using nvcc and launched on the GPU.

6.2 Polyhedral Backend

The polyhedral kernel generator transforms computation DAGs into optimized CUDA kernels through a multi-stage process. Each DAG partition may generate multiple kernels depending on the computational requirements. The backend first applies graph transformations including operator decomposition, padding insertion for vector alignment of convolutions and convolution-to-implicit-GEMM translations. These transformations produce a polyhedral IR consisting of iteration domains, access and dependence relations.

An initial schedule is computed using the ISL library whose scheduler aims to maximize parallelism and locality. The resulting schedule tree serves as the foundation for subsequent GPU-specific transformations. Multi-level tiling is applied to target the GPU compute and memory hierarchy, while reuse analysis identifies optimal locations for data movement operations that are injected into the schedule tree. To hide memory transfer latencies, TMA and WGMMA operations are software pipelined and appropriate synchronization primitives are inserted to coordinate asynchronous operations. Finally, the ISL AST generator produces CUDA code from the transformed schedule tree.

6.3 Tile Size Selection and Heuristics

The optimizations in Section 3 are architecture agnostic; only the emitted instructions change. Hoppolyta treats WGMMA shapes, tile sizes, pipeline depths, and warp specialization as compiler knobs. These are selected by heuristics over data type, problem size, graph type, and target architecture. Therefore these knobs also determine which transformations are applied to the initial ISL schedule.

For Hopper, we start with the following default configurations: (1) tile sizes of 128, 128, 64 at the block level and 64, 128, 64 at the warpgroup level, respectively, (2) Software pipelined TMA/WGMMA with a depth of 2. For small M, the N tile sizes are modified to fill available SMs. When both M and N are small, pipeline depth is increased to hide latencies. For $K < 64$, tile size of 32 along the K dimension is used to minimize redundant wgmma instructions. Warp specialization is applied to fused MHA kernels and GEMM/convolution operations with 8-bit floating point inputs.

Hoppolyta relies on TMA and WGMMA operations only when the fastest varying dimensions of the input tensors allow 128-bit vector loads and stores. Otherwise, the generated kernel incorporates 32 or 64 bit vector loads and stores that are pipelined with synchronous MMA instructions supported by the Ampere architecture. When vectorized loads are not possible, Hoppolyta generates CUDA code that does not use tensor cores by performing tiling at the levels of thread blocks, warps, threads and registers based on the techniques described in [4, 17].

Table 1: Summary of speedup of Hoppolyta over PyTorch (eager) for various patterns.

Network	Graph Type	Num Graphs	Speedup Over PyTorch (eager) <0.5	0.5-1	1-1.5	1.5-2	> 2	Max	Min	GM
TV	GEMM	22	–	20	1	1	–	1.63	0.69	0.83
	Conv	36	–	27	5	2	2	2.56	0.63	0.94
	GEMM+epi	636	81	304	83	35	133	6.23	0.37	1.01
	Conv+epi	3673	26	344	386	409	2508	14.85	0.37	2.73
	FMHA	8	–	–	–	–	8	4.10	2.24	3.34
HF	GEMM	16	4	10	2	–	–	1.03	0.47	0.71
	GEMM+epi	311	4	157	47	38	65	5.64	0.13	1.23
	FMHA	37	–	–	–	–	37	5.00	3.01	4.20

7 Experimental Evaluation

We evaluate Hoppolyta kernel generation on an NVIDIA H100 GPU with 80GB HBM3 memory running driver version 560.35.03. PyTorch (version 2.5.1) serves as our baseline, relying on cuBLAS (version 12.6.3) and cuDNN (version 9.5) libraries for matrix multiplication and convolution operations respectively. Our evaluation examines both standalone kernel performance and the effectiveness of operator fusion on real-world workloads.

Using the pattern matcher by Cutler et al. [12], we extracted matching graphs from 69 Torchvision (v0.20) and 57 Hugging Face Transformers (v4.20.1) models. The kernels generated by Hoppolyta for the graphs are compiled using nvcc 12.6. The evaluation covers five patterns: MatMul and convolution, their pointwise-epilogue fusion variants with broadcasting and optional reshape or permute

operations, and fused multi head attention (FMHA), a MatMul-Softmax-MatMul sequence with optional reshapes and permutes. All evaluations use 16-bit inputs and outputs and 32-bit accumulation, except FMHA softmax, which uses 32-bit throughout.

7.1 Pattern-wise Performance Evaluation

Table 1 summarizes graph counts and speedup distributions by network and graph type. Hoppolyta achieves performance comparable to hand optimized kernels through polyhedral transformations. Operator fusion provides significant performance improvements, 2.73× for convolution epilogues and 3.34× and 4.20× for FMHA on Torchvision (TV) and Hugging Face Transformers (HF), respectively. On TV and HF subgraphs, Hoppolyta also outperforms TorchInductor by 1.75× and 1.15× geomean, respectively. These results highlight the potential for automatic fusion strategies to enhance end-to-end model performance while maintaining competitive performance for core compute intensive operations.

We first provide details on the impact of optimizations on performance. Note that TMA and WGMMA instructions are tightly coupled. Disabling both will result in a kernel that can be targeted for the Ampere architecture. We observe that TMA and WGMMA instructions provide a geometric mean (geomean) speedup of 1.43× over kernels that do not use TMA and WGMMA instructions. With software pipelining, we observe an additional geomean speedup of 1.19× (max: 1.79×, min: 0.98×). We observed warp specialization to be sensitive to computation type, data type, and problem/tile sizes. It slowed fp16 GEMMs by 14% on average despite speedups of up to 1.3× in individual cases, but improved MHA partitions by 1.08× geomean (max 1.4×). Thus, Hoppolyta enables it only for MHA kernels, though the generator supports it for all graphs.

Hoppolyta achieves significant performance improvements (over 2×) primarily through operator fusion. In most of these cases, we observed that the underlying GEMM or convolution kernels (without fusion) also perform comparably to the baseline implementations. GEMM fusion scenarios demonstrate substantial performance gains that scale with the number of fused operations. We also observe that fusion improves performance only when the performance of the core operation reaches at least 70% of the baseline ninja implementation. For example, the minimum speedup that we observe for matmul-epilogue fusion in Torchvision is 0.37× (Table II, Row 3). In this instance, the matmul kernel performance is too low for epilogue fusion to have any significant impact on overall performance. Orthogonal to epilogue fusion, convolution kernels also benefit from our heuristics which choose small WGMMA instruction shapes for small output feature channels and suitable padding widths that enable efficient vector loads. Hoppolyta pads input feature channels to the nearest multiple of vector width (32, 64, or 128) when channel sizes misalign with vector operations.

MHA graphs benefit from the fused warp specialized flash attention kernel generated by Hoppolyta. Competitive speedups were observed despite not employing GEMM-softmax pipelining as described by Shah et al. [28]. Sequence lengths in all the MHA partitions we extracted were less than or equal to 512 with embedding dimension of 32 or 64. For a fair comparison, we manually modified the baseline implementation to use the fused cuDNN kernel for MHA using `scaled_dot_product_attention`, whenever possible. Even with the fused MHA baseline, we observe geomean speedup of 1.13× across all the MHA partitions due to fusion of permutes and reshapes at the inputs of the MHA graph.

Performance slowdowns (0.01–0.4×) in Hoppolyta stem from suboptimal heuristics or unsupported scenarios. GEMMs suffer slowdowns for very small M (e.g., 16) or sizes that preclude vectorized loads whereas cuBLAS handles these using transposed-output, column-major tensor-core kernels which Hoppolyta does not yet support. Some 2D convolutions suffer slowdowns when output channels are not multiples of 8. Hoppolyta pads to enable vectorized loads whereas cuDNN uses indexed-implicit GEMM without padding. Finally, our heuristics under-account for partial-tile over computation, for example, 16 valid rows can still compute 48 or 112 extra rows for tile sizes 64 or 128 respectively. More refined tile-size selection and heuristics can potentially improve performance.

7.2 End-to-end Results

Given a model in PyTorch, we intercept TorchScript to mine subgraphs using the pattern matcher by Cutler et al [12]. For each subgraph, we generate kernels with Hoppolyta and are compiled with nvcc. Unsupported operations in the input model fall back to use library implementations. Figure 17 provides detailed model-wise speedups for both Torchvision and HuggingFace Transformers networks. We compare against PyTorch (eager) and TorchInductor [1] baselines where the latter uses fused Triton kernels for certain operations. We used triton (v3.0) with the default autotuner for TorchInductor. Hoppolyta achieves **1.43**× and **1.41**× speedups over PyTorch (eager), and 1.24× over TorchInductor. It outperforms PyTorch (eager) and TorchInductor on 78% and 46% of models, respectively. On models from Hugging Face Transformers, TorchInductor achieves 1.46× speedup over Hoppolyta. However, Hoppolyta's generated kernels are 1.16× faster than TorchInductor on matched subgraphs, indicating that end-to-end slowdowns stem from graph-level optimizations outside of Hoppolyta's scope.

Our experiments demonstrate that systematic schedule transformations generate kernels competitive with hand-optimized implementations. Efficient GEMM and convolution kernels enable operator fusion benefits even without exactly matching baseline performance on core operations. The end-to-end evaluation reveals significant performance improvements and identifies optimization opportunities beyond kernel generation alone.

8 Related Work

Deep learning frameworks rely on optimized kernel libraries and compiler techniques to achieve high performance on GPU architectures. Tensor compilers such as TVM [9] and Halide [27] have demonstrated the effectiveness of separating algorithm description from optimization schedules. While these frameworks are powerful, they require the programmer to specify the schedules manually with detailed knowledge of the underlying hardware as well as the high level algorithm. Effectiveness of search based approaches [10, 35, 36] to automatically find schedules for tensor cores is yet to be demonstrated.

Polyhedral code-generators have widely adapted variants of the Pluto scheduling algorithm [6] for automatic optimization of loop

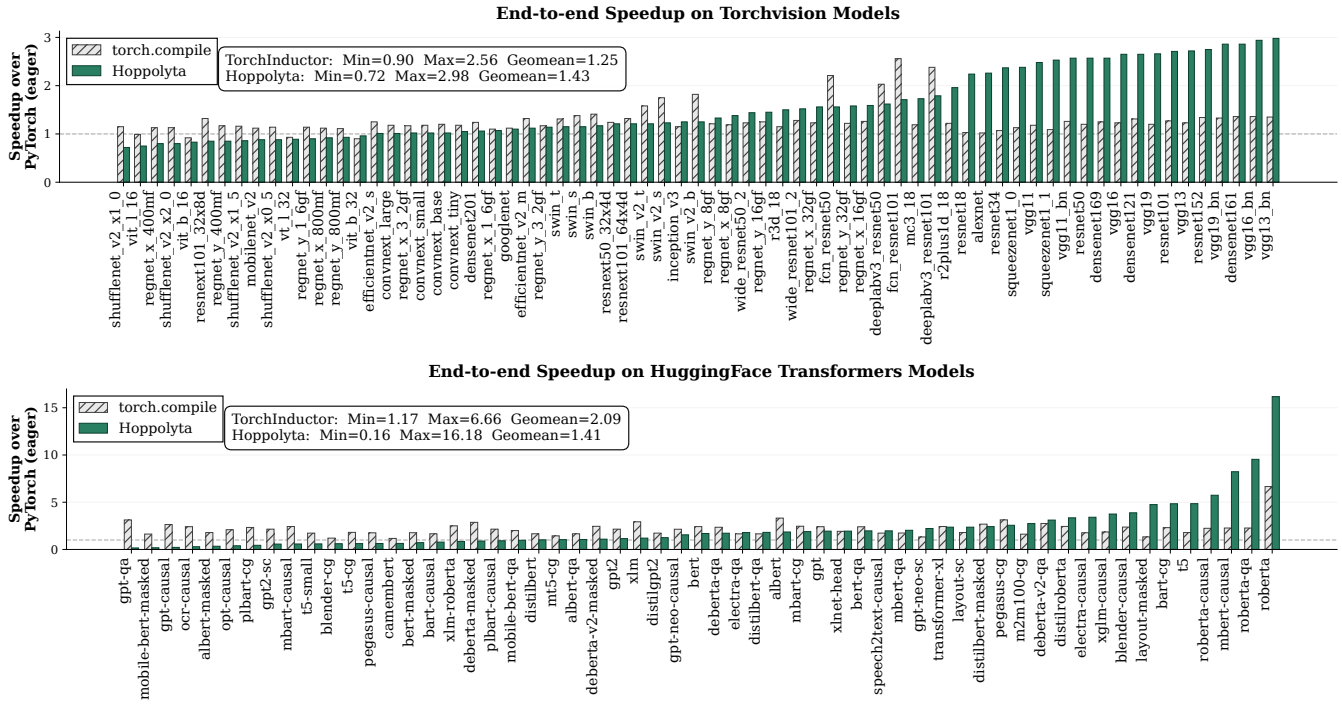


Figure 17: End-to-end speedup over PyTorch (eager) execution on 69 Torchvision and 57 HuggingFace Transformer models.

nests targeting GPUs [2, 4, 17, 32] both in the context of general-purpose and domain specific compilation. Traditionally PPCG [32] has been widely used for automatically generating CUDA kernels for general-purpose computation. However, in addition to not targeting tensor cores, it does not employ tile size selection models limiting its adaptability in a deep learning application.

Several prior works have explored polyhedral code generation for tensor cores. Zhao et al. [34] automatically generate kernels for Volta and Ampere tensor cores targeting GEMM and convolutions. However, they decompose convolutions into *im2col* followed by GEMM (explicit GEMM), which has been shown to underperform implicit GEMM approaches [37], and they cannot automatically generate fused kernels for MHA operations. The recently proposed Polyblocks infrastructure [5, 19] provides an MLIR-based framework to generate fused kernels for Ampere tensor cores. Bhaskaracharya et al. [4] also target earlier tensor core generations using polyhedral transformations. Hoppolyta extends these approaches to target Hopper tensor cores, which introduces significant new challenges due to the asynchronous nature of TMA memory operations and WGMMA compute operations. The performance impact of not generating TMA and WGMMA instructions has already been demonstrated in Section 7.

Though CUDA programming model [21] is powerful, it lacks a high level abstraction for describing GPU kernels and places the burden of memory management and synchronization on the programmer. Several higher-level programming models and intermediate representations that provide block-level abstractions whose primitives operate at tile granularity [15, 16, 18, 29, 30] have been proposed in literature. Task based programming models like Cypress [33] further relieve the programmer by generating asynchronous producer-consumer tasks for Hopper GPUs. However, these approaches still require the programmer to decompose computations into tasks or tiles, specify tile-level compute, and manually apply optimizations such as warp specialization, fusion, and tiling to achieve high performance.

9 Conclusions

This work presents Hoppolyta, the first automatic polyhedral compiler for generating high-performance fused kernels targeting Hopper tensor cores. Through systematic polyhedral loop transformations, Hoppolyta automatically generates efficient GEMM and implicit GEMM convolution kernels while supporting automatic fusion of pointwise epilogues, layout transformations (reshape, permute), and multi-headed attention. Our evaluation on subgraphs from Torchvision and Hugging Face Transformers shows geometric mean speedups of 2.33× and 1.35× respectively. End-to-end evaluation demonstrates performance improvements of 1.43× and 1.41× respectively over the PyTorch (eager) baseline.

Acknowledgments

We are thankful to the anonymous reviewers, whose careful reviews substantially improved the presentation of this paper. We thank Sean Lee and Hanfeng Chen at NVIDIA for building the CI/CD infrastructure and for hooking up the Torchvision and Hugging Face models that made our end-to-end experiments possible. Finally, we acknowledge the use of large language models in polishing the prose and refining the figures.

References

[1] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 929–947. doi:10.1145/3620665.3640366

[2] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: a polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, 193–205.

[3] C. Bastoul. 2004. Code Generation in the Polyhedral Model Is Easier Than You Think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*. Juan-les-Pins, 7–16.

[4] Somashekaracharya G. Bhaskaracharya, Julien Demouth, and Vinod Grover. 2020. Automatic Kernel Generation for Volta Tensor Cores. *CoRR* abs/2006.12645 (2020). arXiv:2006.12645 https://arxiv.org/abs/2006.12645

[5] Uday Bondhugula, Akshay Baviskar, Navdeep Katel, Vimal Patel, Anoop JS, and Arnab Dutta. 2026. PolyBlocks: A Compiler Infrastructure for AI Chips and Programming Frameworks. arXiv:2603.06731 [cs.PL] https://arxiv.org/abs/2603.06731

[6] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 101–113. doi:10.1145/1375581.1375595

[7] Chun Chen. 2012. Polyhedra scanning revisited. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 499–508. doi:10.1145/2254064.2254123

[8] Hongzheng Chen, Bin Fan, Alexander Collins, Bastian Hagedorn, Evghenii Gaburov, Masahiro Masuda, Matthew Brookhart, Chris Sullivan, Jason Knight, Zhiru Zhang, and Vinod Grover. 2026. Tawa: Automatic Warp Specialization for Modern GPUs with Asynchronous References . In *2026 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, Los Alamitos, CA, USA, 255–267. doi:10.1109/CGO68049.2026.11394849

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: an automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 579–594.

[10] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to optimize tensor programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 3393–3404.

[11] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. arXiv:1410.0759 [cs.NE]

[12] Joseph W. Cutler, Alex Collins, Bin Fan, Mahesh Ravishankar, and Vinod Grover. 2025. Pattern Matching in AI Compilers and Its Formalization. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization* (Las Vegas, NV, USA) *(CGO '25)*. Association for Computing Machinery, New York, NY, USA, 63–76. doi:10.1145/3696443.3708934

[13] Tri Dao. 2023. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. arXiv:2307.08691 [cs.LG] https://arxiv.org/abs/2307.08691

[14] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FLASHATTENTION: fast and memory-efficient exact attention with IO-awareness. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 1189, 16 pages.

[15] Yaoyao Ding, Bohan Hou, Xiao Zhang, Allan Lin, Tianqi Chen, Cody Yu Hao, Yida Wang, and Gennady Pekhimenko. 2025. Tilus: A Tile-Level GPGPU Programming Language for Low-Precision Computation. arXiv:2504.12984 [cs.LG] https://arxiv.org/abs/2504.12984

[16] 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. doi:10.1145/3575693.3575702

[17] Venmugil Elango, Norm Rubin, Mahesh Ravishankar, Hariharan Sandanagobalane, and Vinod Grover. 2018. Diesel: DSL for linear algebra and neural net computations on GPUs. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Philadelphia, PA, USA) *(MAPL 2018)*. Association for Computing Machinery, New York, NY, USA, 42–51. doi:10.1145/3211346.3211354

[18] 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. doi:10.1145/3582016.3582018

[19] Navdeep Katel, Vivek Khandelwal, and Uday Bondhugula. 2022. MLIR-based code generation for GPU tensor cores. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction* (Seoul, South Korea) *(CC 2022)*. Association for Computing Machinery, New York, NY, USA, 117–128. doi:10.1145/3497776.3517770

[20] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: scaling compiler infrastructure for domain specific computation. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '21)*. IEEE Press, 2–14. doi:10.1109/CGO51591.2021.9370308

[21] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable parallel programming with CUDA. In *ACM SIGGRAPH 2008 Classes* (Los Angeles, California) *(SIGGRAPH '08)*. Association for Computing Machinery, New York, NY, USA, Article 16, 14 pages. doi:10.1145/1401132.1401152

[22] NVIDIA Corporation 2025. Tensor Map Object Management. https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__TENSOR__MEMORY.html

[23] Nvidia Hopper Architecture 2023. Nvidia H100 Tensor Core Architecture. https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c

[24] NVIDIA PTX Documentation 2025. Asynchronous Warpgroup Level Matrix Multiply-Accumulate Instructions. https://docs.nvidia.com/cuda/parallel-thread-execution/#asynchronous-warpgroup-level-matrix-instructions

[25] NVIDIA PTX Documentation 2025. Data Movement and Conversion Instructions: cp.async.bulk.tensor. https://docs.nvidia.com/cuda/parallel-thread-execution/#data-movement-and-conversion-instructions-cp-async-bulk-tensor

[26] NVIDIA PTX Documentation 2025. Matrix Descriptor Format. https://docs.nvidia.com/cuda/parallel-thread-execution/#asynchronous-warpgroup-level-matrix-shared-memory-layout-matrix-descriptor

[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. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. doi:10.1145/2491956.2462176

[28] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision. arXiv:2407.08608 [cs.LG] https://arxiv.org/abs/2407.08608

[29] Benjamin F. Spector, Simran Arora, Aaryan Singhal, Daniel Y. Fu, and Christopher Ré. 2024. ThunderKittens: Simple, Fast, and Adorable AI Kernels. arXiv:2410.20399 [cs.LG] https://arxiv.org/abs/2410.20399

[30] 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. doi:10.1145/3315508.3329973

[31] Sven Verdoolaege. 2010. isl: An Integer Set Library for the Polyhedral Model. In *Mathematical Software - ICMS 2010*, Komei Fukuda, Joris Hoeven, Michael Joswig, and Nobuki Takayama (Eds.). Lecture Notes in Computer Science, Vol. 6327. Springer, 299–302.

[32] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization* 9, 4, Article 54 (Jan. 2013), 23 pages. doi:10.1145/2400682.2400713

[33] Rohan Yadav, Michael Garland, Alex Aiken, and Michael Bauer. 2025. Task-Based Tensor Computations on Modern GPUs. *Proc. ACM Program. Lang.* 9, PLDI, Article 163 (June 2025), 25 pages. doi:10.1145/3729262

[34] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, Peng Di, Kun Zhang, and Xuefeng Jin. 2021. AKG: automatic kernel generation for neural processing units using polyhedral transformations. In *Proceedings of the 42nd ACM SIGPLAN International Conference*

on Programming Language Design and Implementation (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1233–1248. doi:10.1145/3453483.3454106

[35] 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.

[36] 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. doi:10.1145/3373376.3378508

[37] Yangjie Zhou, Mengtian Yang, Cong Guo, Jingwen Leng, Yun Liang, Quan Chen, Minyi Guo, and Yuhao Zhu. 2021. Characterizing and Demystifying the Implicit Convolution Algorithm on Commercial Matrix-Multiplication Accelerators. arXiv:2110.03901 [cs.DC] https://arxiv.org/abs/2110.03901