MLIR-Based Code Generation for GPU Tensor Cores

Navdeep Katel
Department of CSA
Indian Institute of Science
Bengaluru, Karnataka, India.
PolyMage Labs
Entrepreneurship Centre, IISc
Bengaluru, Karnataka, India.
navdeep@polymagelabs.com

Vivek Khandelwal
Department of CSA
Indian Institute of Science
Bengaluru, Karnataka, India.
vivekkhandel@iisc.ac.in

Uday Bondhugula
Department of CSA
Indian Institute of Science
Bengaluru, Karnataka, India.
PolyMage Labs
Entrepreneurship Centre, IISc
Bengaluru, Karnataka, India.
udayb@iisc.ac.in

Abstract

The state-of-the-art in high-performance deep learning today is primarily driven by manually developed libraries optimized and highly tuned by expert programmers using low-level abstractions with significant effort. This effort is often repeated for similar hardware and future ones. In this work, we pursue and evaluate the more modular and reusable approach of using compiler IR infrastructure to generate libraries by encoding all the required optimizations as a sequence of transformations and customized passes on an IR. We believe that until the recent introduction of MLIR (Multi-level intermediate representation), it had been hard to represent and transform computation at various levels of abstraction within a single IR.

Using the MLIR infrastructure, we build a transformation and lowering pipeline to automatically generate near-peak performance code for matrix-matrix multiplication (matmul) as well as matmul fused with simple pointwise operators targeting tensor cores on NVIDIA GPUs. On a set of problem sizes ranging from 256 to 16384, our performance evaluation shows that we can obtain performance that is 0.95× to 1.19× and 0.80× to 1.60× of cuBLAS for `FP32` and `FP16` accumulate respectively on NVIDIA's Ampere based Geforce 3090 RTX. Furthermore, by allowing the fusion of common pointwise operations with matrix-matrix multiplication, we obtain performance ranging from 0.95× to 1.67× of a cuBLAS-based implementation. Additionally, we present matmul-like examples such as 3-d contraction and batched matmul, which the pipeline can efficiently handle while providing competitive performance. We believe that these results motivate further research and engineering on automatic domain-specific library generation using compiler IR infrastructure for similar specialized accelerators.

CCS Concepts: • **Software and its engineering → Compilers**.

Keywords: MLIR, GPU, tensor cores, matrix-matrix multiplication

ACM Reference Format:
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 (CC '22), April 02–03, 2022, Seoul, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3497776.3517770

1 Introduction

Recent advances in high-performance artificial intelligence and deep learning, heavily rely on high-performance computing. Highly tuned libraries provided by hardware vendors: for example, NVIDIA's cuDNN, cuBLAS, and Intel's MKL power a significant amount of high-performance deep learning is currently powered. Creating these libraries involves significant effort and expertise, and the development may have to be repeated with every major hardware update. There are also limits to what design points can be explored and optimized efficiently.

High-performance libraries for dense linear algebra have traditionally been, and continue to be, written in a combination of C/C++ and inline assembly [9]. However, work in the past decade on BLIS [19, 36] has made the art of developing dense linear algebra libraries more accessible and modular. But the process of developing such libraries still relies on combining pre-existing "inner" highly tuned kernels in assembly with outer nested loop implementations in C. Our goal in this work is of taking a completely different approach: one of using compiler intermediate representation (IR) infrastructure to auto-generate all code needed.

MLIR [4, 15, 18], standing for Multi-Level intermediate representation is a compiler infrastructure targeted at both domain-specific and general-purpose compilation. While LLVM [5, 14] itself has been used for code generation for numerous domain-specific languages targeted at high performance and potentially for generating libraries as well;

CC '22, April 02–03, 2022, Seoul, South Korea

ACM ISBN 978-1-4503-9183-2/22/04…$15.00
https://doi.org/10.1145/3497776.3517770

the use of LLVM in such projects has been typically for low-level compilation and optimization. MLIR, on the other hand, provides multiple levels of abstractions in a unified way to represent computation in various forms using a set of uniform and consistent concepts. One can use MLIR to naturally represent high-level structures like TensorFlow graphs, low-level machine instructions, and anything in between, such as loop nests.

Matrix-matrix multiplication (matmul) and matmul-like operations are at the heart of several compute-intensive high-performance computing and deep learning workloads. BERT [7] is one of the most popular transformers-based [37] natural language processing model that is used for tasks ranging from text summarization [17] and text mining [16] to improving search results [23]. Since BERT is based on transformers, a major chunk of computation comprising it is matmul. Given that the optimization of matmul has been extensively studied and published, it serves as an excellent starting point to evaluate the strengths of an approach itself, such as one based on new infrastructure. With GPUs being the dominant commodity accelerator for deep learning, the combination of matmul and deep learning is an appealing setup for our purposes.

There have been numerous recent works on optimizing matmul for GPUs, including both manual [8, 10, 13, 27, 34, 39, 40] and automatic [2, 35] ones. Although all of these works aim at achieving performance competitive with vendor libraries, they do not do so by using a compiler IR infrastructure like MLIR; we claim that the latter can make the process more systematic, modular, and reusable.

In this work, we design and evaluate an MLIR-based code generation pipeline targeting NVIDIA's tensor cores with the objective to obtain performance competitive with hand-optimized libraries while improving productivity. Our primary contributions are as follows:

1. we introduce warp matrix multiply-accumulate (WMMA) operations in an MLIR dialect along with their lowering to NVVM intrinsics [6];
2. we demonstrate how matmul and matmul-like operations can be systematically code generated as a sequence of MLIR transformations and dialect lowering passes while realizing an end-to-end code generation pipeline targeting tensor cores;
3. we leverage the same code generation pipeline to generate code for NVIDIA Turing and Ampere microarchitectures with two different accumulation precisions without writing or optimizing any code by hand;
4. we finally demonstrate the fusion of common pointwise operations such as ReLU, constant addition, and matrix addition with tensor-core matmul.

The rest of the paper is organized as follows. Section 2 presents the necessary background. Section 3 presents our design and implementation. Experimental evaluation is presented in Section 4 and 6. Section 5 presents insights into matmul-like examples. Related work and conclusions are presented in Section 8 and Section 9, respectively.

2 Background

This section presents the basics of MLIR and GPUs. This work targets tensor cores on NVIDIA GPUs, and hence we discuss in sufficient detail about the tensor cores and different ways to program them.

2.1 MLIR

Multi-Level Intermediate Representation (MLIR) [15, 22] is an SSA-based compiler intermediate representation that aims to provide reusable, extensible compiler infrastructure and reduce the cost of building domain-specific compilers. An operation is the basic entity in MLIR and is often referred to as an `Op`. Everything from instructions to functions to modules is modeled as an operation in MLIR. An operation may have regions and attributes attached to it. A region contains a list of blocks, and a block contains a list of operations that may further contain regions. The blocks inside the region make a Control Flow Graph (CFG). Each block ends with a `terminator` operation that may have `successor` blocks to which the control flow may be transferred. An attribute captures structured compile-time static information, e.g., integer constant values and string data.

The logical grouping of operations in MLIR is known as a dialect. Dialects provide multiple abstractions levels, which help in performing transformations and optimizations not possible or difficult at a single level. We mention about certain MLIR dialects [21] that we primiarily use in this work. The affine dialect uses techniques from polyhedral compilation to make dependence analysis and loop transformations efficient and reliable. The SCF dialect represents various control flow constructs in the IR like loops and conditionals without the restrictions of affine dialect. The GPU dialect models GPU-specific operations and programming constructs offered by CUDA or OpenCL. It is largely meant to be vendor-agnostic [11, 12, 18]. The NVVM dialect represents or is closely tied to NVIDIA-specific GPU intrinsics.

MLIR depends on the LLVM compiler infrastructure for the subsequent steps of code generation. After performing optimizations and transformations in MLIR, MLIR can be translated to LLVM IR. Code generation is then performed by the desired backend, for example, the `x86` backend for `x86`-based CPUs and `NVPTX` [6] backend for NVIDIA GPUs.

2.2 GPUs and Tensor Cores

GPUs are general-purpose massively parallel computing devices [31]. The processors on GPUs can be abstracted into a two-level hierarchy, the streaming multiprocessors (SMs) and computing cores inside the SM. The SM holds both general-purpose CUDA cores and specialized cores like tensor cores at the same level in the compute hierarchy.

Tensor Cores [1] are programmable matrix-multiply-and-accumulate (MMA) units present on NVIDIA GPUs. Significantly more throughput than CUDA cores makes them excellent for accelerating deep learning workloads. They perform small MMA operations represented as $\mathbf{D} = \mathbf{A} * \mathbf{B} + \mathbf{C}$. Tensor cores execute *warp synchronous* instructions like `HMMA` to perform the MMA operation. Tensor cores can be programmed explicitly using the `PTX` instructions or by using the NVIDIA-provided WMMA API. WMMA API provides larger matrix operations (16×16×16, 32×8×16) and utility functions to load and store operand matrices. The matrices loaded using WMMA API have an opaque layout once loaded into the registers, i.e., thread-data mapping is unknown. Programming tensor cores explicitly using assembly instructions is even more challenging because the programmer has to take care of complexities such as thread data mappings in registers and data movement between shared memory and registers. Still, the programmer could get rid of shared memory bank conflicts by choosing a swizzled layout in shared memory. This is impossible with the WMMA API because it supports only row-major and column-major layouts in shared memory. Good performance could be achieved using either the WMMA API or by programming the tensor cores explicitly.

The `NVPTX` backend in LLVM exposes WMMA API functionality through `intrinsics`. This makes it possible to program the tensor cores via MLIR. These intrinsics map one-to-one to WMMA API functions and exhibit identical behavior in terms of programming and usage.

3 Design and Implementation

As code generation using MLIR must happen as a series of transformations on the IR, we put together a lowering pipeline that encodes all the optimizations as IR transformations. This pipeline is shown in Figure 1. Using this pipeline we generate warp-centric, two-level tiled code specialized for tensor cores which has the structure shown in Algorithm 1.

Although there can be different lowering paths in MLIR to generate code for tensor cores, we argue that the one going through the affine dialect is particularly suitable as that helps creation and placement of fast memory buffers, tiling and convenient restructuring of loops while keeping IR compact, vectorization, and loop fusion.

Phase-ordering of passes in the pipeline is crucial when designing a code generation pipeline. The passes must be arranged in an order so that the pipeline is intuitive, progressive, and yields high performance. Arriving at such an order is non-trivial and requires solving challenging design issues that we describe in this section.

Starting Point: The starting point for our code generation approach can be a high-level operation on memref types like `lmhlo.dot` or `linalg.matmul` or simply an IR generated from a user-facing programming model targeting a linear algebra dialect. We can lower a named linear algebra operation to a canonical 3-d affine loop nest representing the matmul. Prior to this work, basic support to generate IR that could execute on tensor cores was not present in MLIR. We first introduce these operations along with their lowerings and then build upon them to put together a complete lowering pipeline while introducing passes and utilities where needed and reusing existing infrastructure for the rest.

Algorithm 1: Two-level tiled tensor core Matmul.

```
Global memory: A[M][K] B[K][N], C[M][N];
Shared memory: a_smem[tbm][tbk], b_smem[tbk][tbn];
Registers: c_reg[wm][wn], a_reg[wm], b_reg[wn];
for threadBlockK ← 0 to K step tbk do
    __syncthreads();
    All threads load tbm × tbk block from A to a_smem;
    All threads load tbk × tbn block from B to b_smem;
    Warp loads wm × wn block form C to c_reg[wm][wn];
    __syncthreads();
    // wmmaM, wmmaN, and wmmaK represent the WMMA
     intrinsic size;
    for warpK ← 0 to tbk step wmmaK do
        for warpM ← 0 to wm step wmmaM do
            Warp loads a fragment of A from a_smem into
             a_reg[warpM];
            for warpN ← 0 to wn step wmmaN do
                Warp loads a fragment of B from b_smem into
                 b_reg[warpN];
                c_reg[warpM][warpN] += a_reg[warpM] ×
                 b_reg[warpN];
            end
        end
    end
    Warp stores c_reg[wm][wn] to the respective block in C;
end
```

3.1 WMMA Operations and Types in MLIR

To program tensor cores using MLIR, we introduced the necessary types and operations based on the WMMA intrinsics exposed by LLVM. We first introduce an opaque `gpu.mma_matrix` type, representing a matrix held by a warp for matrix-matrix multiply-accumulate (MMA) operations. The shape and data type of the matrix is encoded in the IR by the operation that uses this type. `gpu.subgroup_mma` operations directly use these types for inputs and outputs. We introduce three `gpu.subgroup_mma` ops, `gpu.subgroup_mma_load_matrix`, `gpu.subgroup_mma_store_matrix`, and `gpu.subgroup_mma_compute`. Example usage of these operations is shown in Listing 1. The operations introduced in the gpu dialect are generic and can be used for different targets. We are specifically targeting tensor cores, so we introduce additional operations in the `NVVM` dialect, to which the operations in the gpu dialect are lowered.

In the following sections, we will talk about the design choices and implementation details of our pipeline and how

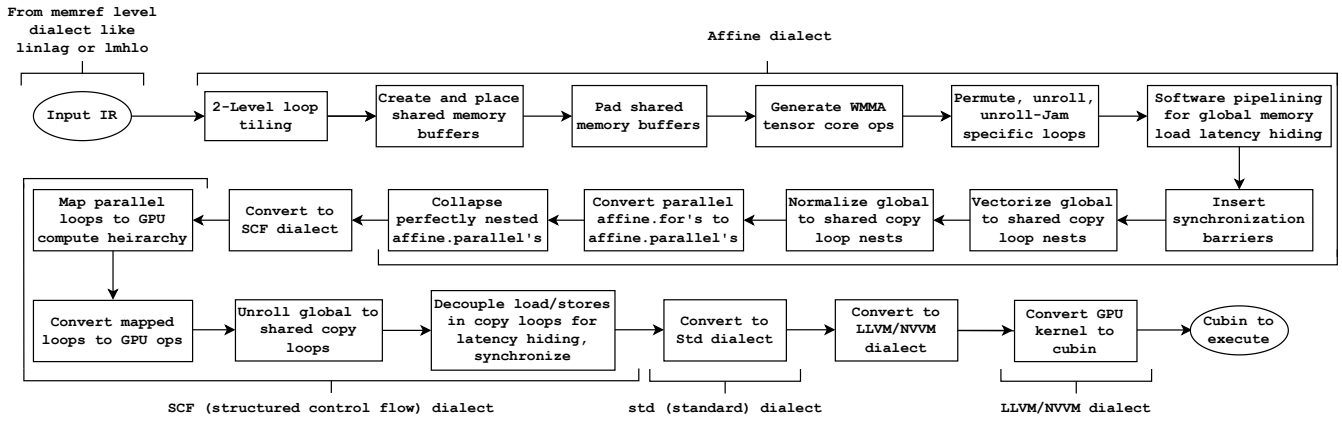

Figure 1. Lowering path in MLIR depicting successive optimizations and transformations for GPU tensor core code generation. Note: the *std* dialect has since been split primarily into the *memref*, *arith* and *func* dialects.

it progressively lowers a high-level matmul operation down to machine code. We will discuss major passes like tiling, shared memory buffer generation, copy loop vectorization, and global memory load latency hiding in the order as they are shown in Figure 1.

3.2 Tiling and Creating Shared Memory Buffers

To fully utilize the compute units of the GPU, two-level tiling is must. The first level of tiles is processed in parallel by thread blocks, while warps process the second level. The existing tiling utility in MLIR only supported one level of tiling. We extend the utility to tile loop nests multiple times. After tiling is performed, creating and placing shared memory buffers at the proper loop depth is next. We use `affineDataCopyGenerate` utility to generate global to shared copy loops. The utility is called on the thread block scoped k-loop or the main k-loop to generate copies for `A` and `B` operands right inside it. Shared memory buffers are modeled as `memref.global` as it best represents the semantics and scoping rules of shared memory in the CUDA programming model.

We pad the shared memory buffers to ensure shared memory access with reduced bank conflicts. Padding can change the mapping of elements in the shared memory banks and hence reduce bank conflicts. The padding factor must be a multiple of eight, i.e., 128-bits for eight `FP16` elements. This is because the WMMA load intrinsic of size `m16n16k16` requires the source address of each row of the matrix being loaded to be aligned to 128-bits. The official documentation says the minimum alignment requirement is 256-bits or 16 `FP16` elements. Our experiments found that 128-bit works without any issues and performs better than the 256-bit version. This detail is, however, undocumented.

The creation of shared memory buffers and the copy loops introduces all the memory accesses in the kernel. Parallel loops are now detected and marked with a user-defined (external), `parallel` unit attribute. We maintain this attribute for communicating information to the neighbouring passes in the pipeline. `copyLoopNest` is another such attribute which helps identify the loop nests performing global to shared memory copy. The marked `affine.for` operations will later be converted to `affine.parallel` operations. Perfectly nested parallel loops will be collapsed into a single operation. Collapsing helps capture the semantics of a perfect parallel nest in a single operation and helps in processing them conveniently. The IR in Listing 1 shows the padded shared memory buffers and their copy loops.

3.3 Mapping to WMMA Operations and More

The IR we have currently performs the computation on scalar elements, but tensor cores operate on small matrix operands, as discussed earlier in Section 2.2. Therefore, scalar operations inside the compute loop must be replaced by WMMA operations, and the loop steps must be adjusted according to the size of the WMMA operations used. We first generate the WMMA operations and then do the following IR transformations: 1. permute the outermost six loops from (`i`, `j`, `k`, `ii`, `jj`, `kk`) order to (`i`, `j`, `ii`, `jj`, `k`, `kk`) order. This helps in hoisting invariant load-store operations on `C` to the outermost position possible. 2. permute the innermost three loops form (`i`, `j`, `k`) to (`k`, `i`, `j`) and fully unroll them. This represents warp-level MMA operation as an outer product and enhances ILP, as pointed out by Bhaskaracharya et al. [2].

After performing the aforementioned steps, loads and stores on `C` in the innermost loop be hoisted. Unrolling the innermost loops completely reveals all loads on `A` and `B`. Application of `cse` on this IR will completely remove the redundant loads and achieve an unroll-jam kind of effect. The loop structure just after the said optimizations is shown in Listing 1. The main-k loop is now modified to take the loaded `C` operands as loop `iter_args`. After every iteration,

this loop `yields` accumulated products, which are passed as `iter_args` to the next iteration.

```
#map1 = affine_map<(d0) -> (d0 + 64)>
#map2 = affine_map<(d0) -> (d0 + 128)>
// Thread block `i` and `j` loop.
affine.for %i = 0 to 8192 step 128 {
  affine.for %j = 0 to 8192 step 128 {
    // References to shared memory buffers.
    %b_smem = memref.get_global @b_smem_global : ↩
          memref<64x136xf16, 3>
    %a_smem = memref.get_global @a_smem_global : ↩
          memref<128x72xf16, 3>
    // Warp `i` and `j` loop.
    affine.for %ii = 0 to 128 step 64 {
      affine.for %jj = 0 to 128 step 32 {
        // Hoisted loads on C.
        %c_reg_0 = gpu.subgroup_mma_load_matrix %C[%i + %ii, %j + ↩
              %jj] ... -> !gpu.mma_matrix<16x16xf32, "COp">
        ...
        // Main `k`-loop with loaded C operand as iter_args ↩
              yielding the results.
        %res:8 = affine.for %k = 0 to 8192 step 64 ↩
              iter_args(%c_in_0 = %c_reg_0 ...) -> ↩
              (!gpu.mma_matrix<16x16xf32, "COp">) {
          // Copy loops for B and A.
          affine.for %copykk = %k to #map1(%k) {
            ...
          } {copyLoopNest, parallel}
          affine.for %copyii = %i to #map2(%i) {
            ...
          } {copyLoopNest, parallel}
          %a0 = gpu.subgroup_mma_load_matrix %a_smem[%ii, %c48] ↩
                ... -> !gpu.mma_matrix<16x16xf16, "AOp">
          %b0 = gpu.subgroup_mma_load_matrix %b_smem[%c48, %kk] ↩
                ... -> !gpu.mma_matrix<16x16xf16, "BOp">
          %c_res_0 = gpu.subgroup_mma_compute %a0, %b0, c_in_0 ... ↩
                -> !gpu.mma_matrix<16x16xf32, "COp">
          ...
          affine.yield %c_res_0 ... : !gpu.mma_matrix<16x16xf32, ↩
                "COp"> ...
        } {isComputeLoopNest = true}
        // Hoisted stores on C.
        gpu.subgroup_mma_store_matrix %res#0, %C[%11, %12] ... : ↩
              !gpu.mma_matrix<16x16xf32, "COp">
        ...
      } {parallel}
    } {parallel}
  } {parallel}
} {parallel}
```

Listing 1. Affine matmul after tiling, creation of shared memory buffers, mapping to WMMA operations, loop permutation, and invariant load-store hoisting.

WMMA ops are generated early in the affine dialect as the access functions for the WMMA ops are affine. As the introduction of WMMA ops requires modification of the loop steps, generating them early also ensures the correctness of transformations like loop-unrolling.

3.4 Global Memory Load Latency Hiding and Synchronization Placement

Global to shared memory transfers have the highest latency and are slower than the compute in the main k-loop. To hide global load latency, we prefetch data for the next main k-loop iteration into the registers to be readily available. Software pipelining the main k-loop achieves this. Performance gains are obtained only when the stores are moved away from loads and compute finally overlaps with data copy. This technique has been previously used by Bhaskaracharya et al [2]. We postpone this optimization to a further point in the pipeline as the copy loops first need to be mapped appropriately to threads in a thread block, unrolled, and then moved. The complete optimization is described in Section 3.6.

Most of the parts in the IR which require synchronization have been generated. Therefore we insert synchronization barriers before reading from and writing into the shared memory buffers. More synchronizations will be placed when we complete the transformation on the copy loops inside the main-k loop for latency hiding in Section 3.6. IR with all the synchronization barriers is shown in Listing 2.

3.5 Global to Shared Copy Vectorization

It is well known that vector load and store instructions perform better than their scalar counterparts because it reduces the number of memory transactions and often results in better utilization of the available bandwidth [20]. We use the vectorization utility already present in MLIRX [33]. The utility first casts memrefs of scalar elements to memrefs of vectors using the `memref.vector_cast` operation, and then modifies the access functions. We experimentally observed that 128-bit wide vectors provide the best performance. All the copy loops generated have affine maps in their bounds. These bounds can be simplified by normalizing the loops. Normalization results in the elimination of redundant `floordiv` operations produced by vectorization utility, and reduces the number of PTX instructions generated fro address calculations. This results in gains up to 1.5 TFLOPS on some problem sizes. After these transformations, we lower the `affine` dialect to the `scf` dialect.

3.6 Mapping to the GPU Compute Hierarchy

After lowering down to `scf` dialect, we map the two-level tiled structure to thread blocks and warps, thereby mapping it to the GPU compute hierarchy and exploiting the available parallelism. MLIR already supported mapping to thread blocks but not to warps. We extend the existing mapper to provide this support. The outermost two loops in Listing 1 are mapped to thread blocks, and the following two loops are mapped to warps. Copy loops are mapped to thread blocks while the main k-loop is sequential and remains as is.

The outermost two loops being mapped to thread blocks are converted into `gpu.launch` operation. `gpu.launch` takes six parameters, three grid dimensions, and three thread block dimensions which are called the launch parameters. The launch parameters are set automatically using the tile sizes at both levels and the problem sizes.

3.7 Completing Latency Hiding

In Section 3.4, we described latency hiding and concluded that it is not complete until we decouple the shared memory stores from global memory loads. To achieve this, we first completely unroll the copy loops. Unrolling these loops exposes all the loads and stores and provides us the liberty to move them. We then move the stores after the main k-loop. This prevents the warps from stalling on the global memory loads and allows the compute to proceed in parallel with global memory loads in flight. Additional synchronization barriers are inserted as needed. The skeleton of the resultant IR is shown in Listing 2. This is our terminal point for us in terms of optimizations and was our last step while in the `scf` dialect.

3.8 Putting It All Together for Execution

GPU codes are typically divided into two components. A `host`-side component that runs on the CPU, and a `device`-side component, or the kernel, which runs on the GPU. The host-side code is converted to `std` dialect and then subsequently to `llvm` dialect. While conversion to `llvm` dialect, operations like `gpu.launch` are lowered to function calls to the CUDA driver API. The device-side code is also converted to `std` dialect and then to a mix of `llvm` and `NVVM` dialect. This is converted to LLVM IR and then to `PTX` by the `NVPTX` backend in LLVM. This `PTX` is converted to a CUDA binary using NVIDIA's compiler through the CUDA driver API.

4 Experimental Evaluation for Matmul

In this section, we evaluate the performance of our approach for matmul kernels on two different NVIDIA GPUs with two different accumulation precisions, `FP16` and `FP32`.

4.1 Experimental Setup

The evaluation was performed on two NVIDIA GPUs, Turing-based RTX 2080 Ti [24] and Ampere-based RTX 3090 [25]. The device driver version was 460.91.03 on Ampere and 460.32.03 on Turing. The SM clocks were set to the boost frequency mentioned in the whitepaper for all the experiments, which is 1635 MHz for the Turing GPU and 1695 MHz for the Ampere one. We limit ourselves to statically allocated shared memory, which is of 48 KB on both devices. The maximum number of registers per thread is set to 255. We use NSight Systems [29] for timing and consider only the kernel runtime for calculating the attained TFLOPS. This applies to our kernels as well as the baselines. We consider different combinations of thread block-level tiles and warp-level tiles and also enable and disable the unrolling of the innermost k-loop. Among all the combinations, we report the best-performing version. The reported performance is averaged over ten runs. In our experiments, we found that the space of suitable tile sizes is quite limited for matmul (combinations of [128,256]x[128,256]x[32,64] for shared memory/register tiles for large problem sizes). An auto-tuning-based approach or a simple analytical model would provide good performance here, which can be the subject of future work.

```
gpu.launch blocks(%blockIdX, %blockIdY, %blockIdX) in (%arg6 = %c64, %arg7 = %c64, %arg8 = %c1) threads(%threadIdX, %threadIdY, %threadIdZ) in (%arg9 = %c256, %arg10 = %c1, %arg11 = %c1) {
  ...
  %c_reg_0 = gpu.subgroup_mma_load_matrix %C[%26, %27] ... -> !gpu.mma_matrix<16x16xf32, "COp">
  ...
  // Peeled copy loops for iteration 0 of k-loop.
  scf.for %copy = %c0 to %c4 step %c1 { ... }
  scf.for %copy = %c0 to %c4 step %c1 { ... }
  gpu.barrier
  // Main k-loop
  %res:8 = scf.for %k = %c0 to %c8128 step %c64 iter_args(%c_in_0 = %c_reg_0 ...) -> (!gpu.mma_matrix<16x16xf32, "COp"> ...) {
    gpu.barrier
    // Global memory loads for iteration i + 1 of k-loop.
    %a_next_iter_0 = memref.load %a_cast[%74, %81] : memref<8192x1024xvector<8xf16>>
    %b_next_iter_0 = memref.load %b_cast[%94, %101] : memref<8192x1024xvector<8xf16>>
    ...
    scf.for %kk = %c0 to %c64 step %c32 iter_args(%arg16 = %c_in_0 ...) -> (!gpu.mma_matrix<16x16xf32, "COp">) { ... }
    gpu.barrier
    // Shared memory stores for iteration i + 1 of k-loop.
    memref.store %b_next_iter_0, %b_smem_cast[%51, %68] : memref<64x17xvector<8xf16>, 3>
    memref.store %a_next_iter_0, %a_smem_cast[%150, %167] : memref<128x9xvector<8xf16>, 3>
    ...
  }
  gpu.barrier
  // Peeled compute loop for iteration n-1 of k-loop.
  scf.for %kk = %c0 to %c64 step %c32 { ... }
  ...
  gpu.subgroup_mma_store_matrix %res#0, %C[%26, %27] ... : !gpu.mma_matrix<16x16xf32, "COp">
  ...
}
```

Listing 2. IR structure after performing global memory load latency hiding and converting loops to GPU operations.

We consider matmul of the form $C = AB + C$, where all three matrices are stored in a row-major layout in global memory. We use the `m16n16k16` version of the WMMA intrinsic and limit ourselves to square problem sizes ranging from 256 to 16384 with a step of 256. We assume the problem sizes are multiples of thread block tiles, which are again multiples of warp tiles, which are in turn multiples of the WMMA intrinsic. The standard baseline used for matmul kernels is cuBLAS-11.2 [26].

4.2 Mixed-Precision Matmul Performance

Matmul with `A`, `B` inputs in `FP16` and the output matrix `C` in `FP32` with multiplication and accumulation in `FP32` mode [30] is typically known as mixed-precision matmul.

Table 1. Performance counters for different mixed-precision MLIR-generated and cuBLAS kernels on Ampere for problem sizes 8192, 8448, and 8704.

Tool	Thread block tile size	Warp tile size	Active warps per SM	Pipeline stages	Stall barrier	Stall short scoreboard	Stall long scoreboard
MLIR	128x64x64	64x64x32	8	1	1.17, 1.07, 1.08	0.21, 0.13, 0.13	0.57, 0.46, 0.50
cuBLAS	256x128x32	—	8	3	0.10, 0.9, 0.08	0.01, 0.01, 0.01	0.50, 0.44, 0.40

Table 2. Performance counters for different half-precision MLIR-generated and cuBLAS kernels on Ampere for problem sizes 10240, 10496, 10752.

Tool	Thread block tile size	Warp tile size	Registers used	Static shared memory used	Dynamic shared memory used	Block limit registers	Block limit shared memory	Active warps per SM	Bank conflicts range
MLIR	128x256x32	64x128x16	254	27.14	0	2	3	8	$6.55 \cdot 10^7$—$7.06 \cdot 10^7$
cuBLAS	128x128x32	—	158	49.15	32.77	3	1	4	0

This version is of particular interest to us because of its importance in training deep learning models [28].

4.2.1 Performance on Ampere. We achieve performance that is consistently 0.95× to 1.19× of cuBLAS. Comparing our performance with the absolute peak of the device, we reach 95.4%, while cuBLAS reaches 97.7% of the device peak. Figure 2 shows the performance of our automatically generated kernels on Ampere RTX 3090. We observed that smaller thread block tile sizes like 64×64×64 performed better on smaller problem sizes. Table 1 shows performance for three problem sizes highlighting the differences between our kernels and cuBLAS. We have used slightly smaller tile sizes than cuBLAS at the thread block level and used single-stage pipelining for global memory load latency hiding. Smaller thread block sizes are advantageous when using single-stage pipelining as we have to wait on fewer data to be loaded. The tile sizes are still large enough to have a sufficient number of warps with enough work to keep the compute units busy, and hence we see performance comparable to cuBLAS. This is also observed by inspecting the warp stall states. Stall long and stall short scoreboard represent the number of cycles a warp spends waiting on global and shared memory loads. Stall barrier represents the number of cycles a warp spends waiting on synchronization barriers. We have comparable long scoreboard stalls but higher stalls on barriers, i.e., some warps spend time waiting at the barriers while others are busy in the compute. We also observe higher short-scoreboard stalls, which may be due to how WMMA intrinsics load data from shared memory and the fact that we did not use any latency hiding for the shared memory to register transfers to fit in the tight register budget. However, the impact of these stalls is minimal, as evidenced by the end performance that we obtained.

4.2.2 Performance on Turing. We obtain performance that is 0.86× to 1.11× of cuBLAS. We reach 90.2%, while cuBLAS reaches 99.5% of the device peak. Smaller problem sizes are benefited by smaller thread block tiles here as well. Moving to larger sizes, we see a 9-12% gap This is slightly greater than the gap we see on Ampere. A potential reason could be that the optimal tile sizes we found for most problem sizes have 32 as the leading dimension for the `A` operand, leading to bank conflicts while storing tiles in shared memory and slightly hurting the performance. Figure 3 shows the performance of our automatically generated kernels on Turing RTX 2080 Ti.

4.3 Half-Precision Matmul Performance

Here, we present the performance of our automatically generated half-precision kernels, where all three matrices `A`, `B`, and `C` are in `FP16`. The product of elements is performed at least in `FP32` mode, while the accumulation is performed in at least `FP16` mode [30]. On Ampere, we consistently achieve performance between 0.80× to 1.60× of cuBLAS. Figure 4 shows the performance of our automatically generated kernels on Ampere RTX 3090. We observe that cuBLAS has inconsistent performance on the problem sizes larger than `W = 8848`. On profiling cuBLAS kernels, we observe that thread block and warp tile sizes chosen by cuBLAS were slightly smaller than the ones for which we obtained optimal performance.

Moreover, cuBLAS kernels appear to use multiple pipelining stages as used by CUTLASS [27] to hide the latency of global memory loads. This results in increased shared memory usage to hold tiles for different stages and reduces the number of active warps per SM. This, combined with small tile sizes, results in the relatively lower performance of cuBLAS kernels. Details for three representative sizes are shown in Table 2. All three kernels have eight active warps per SM, while cuBLAS only has four active warps. The exact warp-tile sizes used by cuBLAS are unknown but are less than what we use, as hinted by the smaller thread block tile size, fewer registers used, and the same number of warps per SM.

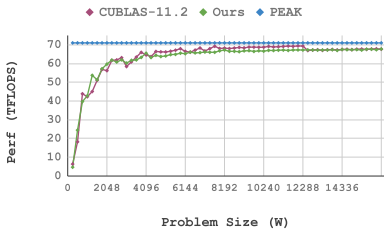

Figure 2. Mixed-precision matmul on Ampere.

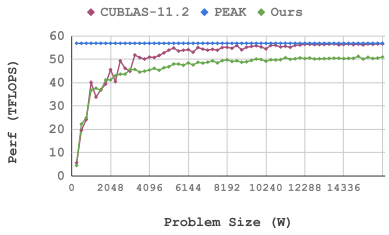

Figure 3. Mixed-precision matmul on Turing.

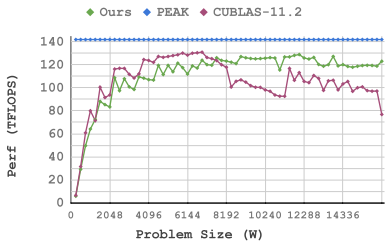

Figure 4. Half-precision matmul on Ampere.

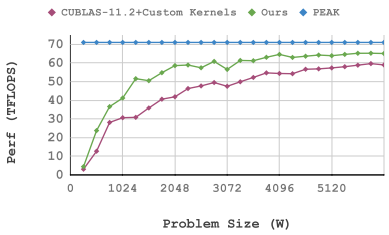

Figure 5. Mixed-precision fused matrix addition and ReLU on Ampere.

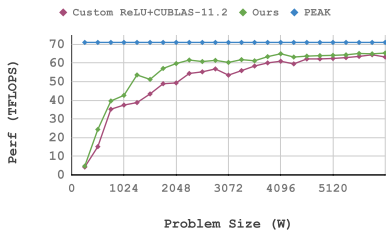

Figure 6. Mixed-precision fused ReLU on C input on Ampere.

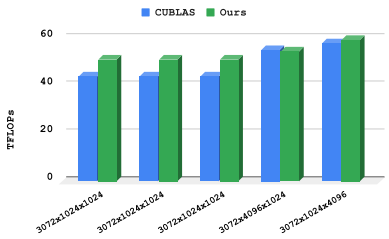

Figure 7. cuBLAS vs. MLIR on BERT SQUAD Inference like matmul.

On Turing, we achieve performance that is 0.72× to 0.89× of cuBLAS. This is lower than the performance achieved by us in other configurations. The primary reason for this is that the optimal register tile sizes like 128×64×8, pointed out in [39], lead to register spills in our kernels. The implementation of [39] is in SASS assembly; hence there is better control over register usage. As shown in Table 3, Turing exhausts all the available registers and spills to Local memory; Ampere runs the same kernel in 250 registers. Ampere has zero L1 hit rate for local loads and stores as there are no LDL and STL instructions. On Turing, the hit rate is meager, meaning that most of the LDL and STL instructions went to L2-cache or DRAM. The performance of the most optimal MLIR version gave was 79.48 and 123.25 TFLOPS on Turing and Ampere, respectively.

Additional performance here is left on the table on both the devices due to the bank conflicts we incur while storing tiles to shared memory as a result of non-swizzled layouts in shared memory (Section 2.2), and uncoalesced accesses to the global and shared memory by the WMMA load operations.

4.4 Impact of Individual Optimizations

Matrix-matrix multiplication gives optimal performance only when all the optimizations discussed in Section 3 are enabled. Figure 8 shows the impact of each individual optimization. Shared memory and register tiling give improvement over

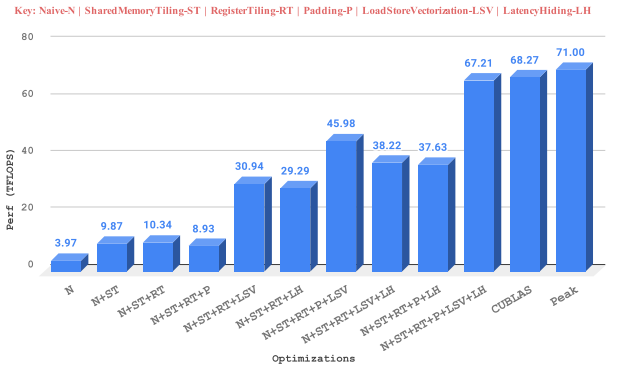

Figure 8. Effect of individual optimizations on mixed precision matmul (W=8192) on Ampere.

the naive version, as they reduce redundant memory accesses. Padding does not show any immediate benefits when enabled along with tiling as the bottleneck is global to shared memory copy. The vectorization of copy loops shows a 5× performance improvement over the tiled and padded version. Enabling global load latency hiding yields a 1.5× performance improvement over the vectorized version, taking us to 67.21 TFLOPS.

Table 3. Analysis of register spills for problem size 8192×8192×8192, shared memory tiles size 256×128×32, and register tile size 128×64×16.

	# of registers used	# of STL inst.	# of LDL inst.	L1 STL hit rate	L1 LDL hit rate	Perf TFLOPS
Turing	255+	$2.87 \cdot 10^5$	$2.87 \cdot 10^5$	1.25	4.28	64.18
Ampere	250	0	0	0	0	123.21

4.5 Performance on BERT-Like Matmul

Figure 7 presents the performance of our kernels on certain matmul's that are representative of those found in BERT [7]. We consider "BERT large" with a maximum sequence length of 384 and batch size of eight on the SQUAD inference task. BERT involves matmul's where the LHS operand is accessed as a transpose, and as we currently do not support accessing matrices in the transposed format, we compare the non-transpose version but with the exact sizes as found in the BERT setting. In three of the layers, *key*, *query*, and *value* (first three in Figure 7), we get an improvement of 16% over using cuBLAS for those matmul's. On the final two (*out* and *intermediate*), we perform nearly the same as cuBLAS.

5 Other Matmul-Like Examples

In this section, to demonstrate generality, we present and evaluate two more examples that have matmul computation at their core. We also compare these with vendor libraries providing these routines, all in mixed precision.

5.1 Batch Matmul

Batch matmul is a matmul-like computation, but with an extra data-parallel dimension: different input and output matrices are used for each batch. It can be expressed by: `C[b][i][j] +=` $\sum_{k=0}^{K-1}$`A[b][i][k] × B[b][k][j]`. We observe that the parallel data dimension representing the batch can be collapsed with an existing parallel dimension to canonicalize all 'memref's into a 2-d form. In general, any number of data-parallel batch dimensions can be collapsed with the parallel dimensions of matmul. Furthermore, collapsing the `batch` dimension with the `row` dimension of batch matmul puts the loop nest into a three-loop canonical form. As a result, the canonicalized form can be handled with no changes to our `affine` to `wmma` pass or the rest of pipeline. The performance of batch matmul generated kernels are compared with cuBLAS in Figure 10. We perform better than cuBLAS for all sizes reported, and the improvement from 1.03× to 1.28×. The reasons for the better performance is again due to smaller tile sizes 64×64×32 being used for all of the mentioned problem sizes and an implicit parallelization across the `batch` dimension after the collapsing.

5.2 3-d Tensor Contraction

A standard matmul can be viewed as a 2-d tensor contraction with the `k` dimension being contracted. We present another example with contraction happening in an additional dimension expressed by: `C[i][j] +=` $\sum_{k=0}^{K-1}\sum_{l=0}^{L-1}$`A[i][k][l] × B[l][k][j]`. Note that all the contracting dimensions of the matrix can be collapsed together to get to the three-loop canonical form. Figure 11 presents the performance comparison of a few representative sizes. We compared our implementation with cuTensor [32] version 1.3, which is an NVIDIA library for tensor contractions. We obtain performance that is 0.87× to 1.14× of cuTensor.

6 Operator Fusion

Feed-forward layers in deep learning models are generally followed by some pointwise operations such as ReLU and matrix addition. These operations can be termed as `consumers` of the output produced by matmul. They are much faster than matmul as they only have $O(n^2)$ computation. In an ordinary setting, the result of a matmul will be first stored to global memory and will be loaded again by the pointwise operation. This is avoided by applying the pointwise operation on the result of matmul in the registers, thereby fusing the operation with matmul.

As mentioned in Section 2.2, the elements of the registers used by the tensor core have an opaque layout. The fusion being performed here can broadly fall into two classes. The first one is of fusion with a *uniform* operation, i.e., the operation being fused is applied uniformly to all MMA register elements of a matmul input or output value and does not depend on the layout of elements in that value for correctness. Examples include ReLU and other point-wise binary operations with a uniform constant as their other operand. The second case is that of fusion with a "non-uniform" operation with an identically shaped operand: in this case, the operation is dependent on the layout of elements of the relevant matmul input or output. Since both operands are identical in shape, here, we load the value corresponding to the non-matmul operation into the registers using the `gpu.subgroup_mma_load` operation; this ensures that the layout of elements of the non-matmul input/output is the same as that of the matmul output/input in the registers. A simple point-wise addition with a matrix is an operation falling into this second class.

To further generalize, we detect loop nests representing operations to fuse using affine analysis available in MLIR. In a sequence of affine loop nests, after identifying the matmul loop nest of interest to fuse into, we inspect loop nests before and after the matmul to detect opportunities for fusion. Before inspecting operations and memory accesses for detecting the validity of fusion, we first ensure that affine maps used by the `affine.load` and `affine.store` operations in both loops nests are fully composed. The composition of affine maps is performed using utilities already available in MLIR. These utilities reduces all maps to a canonical form where all operands of the map are either loop IVs of

Table 4. Speedup achieved for mixed-precision fusion on Ampere and Turing by our MLIR based code-generator.

GPU Arch	Fused ReLU vs. cuBLASLt		Fused constant addition vs. cuBLAS + CUDA		Fused matrix addition vs. cuBLAS + CUDA		Fused matrix addition + ReLU vs. cuBLAS + CUDA	
	Min	Max	Min	Max	Min	Max	Min	Max
Ampere (RTX 3090)	0.95×	1.18×	0.99×	1.40×	1.00×	1.45×	1.03×	1.67×
Turing (RTX 2080)	0.86×	1.09×	0.91×	1.26×	0.84×	1.28×	0.86×	1.50×

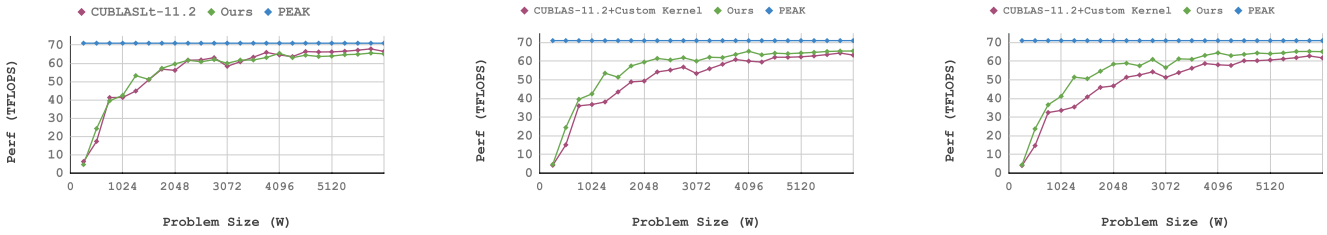

Figure 9. Performance of fused ReLU, constant addition, and matrix addition in that order on Ampere.

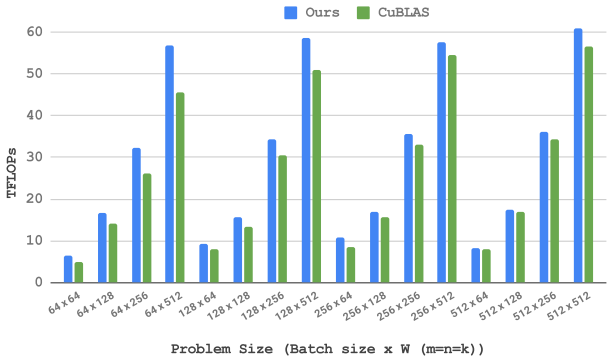

Figure 10. Batch matmul performance on Ampere.

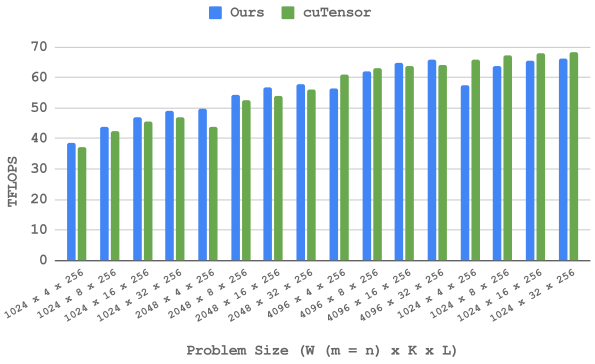

Figure 11. 3-d contraction performance on Ampere.

`affine.for` ops or terminal symbols. After canonicalizing to this form, we compare the load map in the consuming nest with the store map in the producing nest. It is sufficient to say that fusion is possible while ensuring a map to tensor cores if: 1. the load access in the consuming nest is to the same memref where the result of the matmul is stored in the source nest in case of a unary op or at least one of the load access is to the same memref in case of a binary op, and 2. the load access map in the target nest is a permutation of the store access map in the source such that there is always contiguity along the fastest varying dimension, e.g., if the source access is [i, j, k], then out of the six valid permutations, [k, j, i] and [j, k, i] are invalid as they do not provide contiguity in the fastest varying dimension of the memref involved. The restriction arises from the functionality of the WMMA load op, which provides a strided load where 16 contiguous elements are loaded every stride.

After detecting loops that are valid for fusion in this manner, we use the existing fusion utilities in MLIR to perform loop fusion on selected loops. We assume at this point that such fusion of pointwise ops is always profitable. In general case, evaluating the profitability of fusion is beyond the scope of this work.

7 Experimental Evaluation of Fusion

In this section, we evaluate our matmul kernels fused with different pointwise operations. The evaluation is performed on both Turing and Ampere with `FP32` accumulation. We use cuBLASLt [26] as the baseline for ReLU fusion. While for other examples, we use cuBLAS followed by a custom CUDA kernel performing the pointwise operation as the baseline for fused kernels that do not have a library implementation. We implement the custom CUDA kernels and ensure that performance-critical criteria such as vector load-stores and global memory coalescing are met. All MLIR fused kernels are tested with the same tile sizes, which had the optimal performance for the plain matmul kernels.

Consumer-side Fusion Performance: Table 4 summarizes the performance of the fused operations consuming the output of a matmul with the respective "unfused" baselines. For ReLU, both cuBLASLt and MLIR perform fusion while yielding the same execution time as that of the plain matmul kernels. For 'constant addition', a constant is added uniformly to the whole output matrix. 'Matrix addition' is a

typical pointwise addition of two matrices. A pattern similar to this is observed in BERT [7]. A batch matmul is followed by a pointwise division with a constant. The result produced is fed to a matrix addition. Performance gains are significant for matrix addition when compared to constant addition as the former involves loading a matrix from global memory. On larger sizes, the impact of fusion is not expected to be significant as the amount of computation and reuse in the matmul increases; thus, the producer-consumer reuse between the matmul and the point-wise operation becomes less important to exploit. We also evaluate our pipeline in the scenario where multiple operations are fused with matmul. In particular, we fuse matmul followed by matrix addition and ReLU. Results for Ampere are presented in detail in Figure 5 and Figure 9.

Producer-side Fusion Performance: Figure 6 presents results where pointwise operations are applied on the inputs of the matmul and are fused in with our approach. We present these results only as a proof of concept and consider a ReLU being applied on the `C` input. In general, the technique can be used to perform fusion for the `A` and `B` inputs as well. The speedup of ReLU fused with `C` input ranges from 0.98× to 1.38× on the Ampere and 0.91× to 1.33× on the Turing.

8 Related Work

cuBLAS and cuBLASLt are closed-source libraries provided by NVIDIA for linear algebra operations on their GPUs, delivering near-peak performance. However, obtaining the best possible performance for a composition of operations is a challenge. cuBLASLt tackles this problem by providing kernels with fused epilogues for common operations like ReLU and bias addition but is currently limited to just those. Our approach to using automatic code generation and compiler IR infrastructure is a fundamentally different way to create such libraries.

Triton [35] is a language and compiler aimed at rapidly getting peak performance with minimal programming effort from users. Triton uses tile-level semantics for analysis and optimizations at its core. The support for tensor cores relies on inline assembly. On the other hand, as our pipeline relies on lowering through WMMA intrinsics available via LLVM, it would work out of the box on any device with tensor cores where such support is available in LLVM.

Faingnaert et al. [8]'s work is centered around developing an API for easily writing flexible enough GEMM kernels in the Julia programming language. While we aim to generate all the code automatically using a set of IR transformations. More importantly, being based on MLIR, our approach provides a more gradual and small-step lowering pipeline as opposed to those that directly make use of LLVM.

CUTLASS [27] is a template-based CUDA C++ library providing abstractions for compute and data movement for a range of dense linear algebra operations. CUTLASS achieves high performance for matmul on tensor cores using a set of non-trivial optimizations and device-specific techniques such as the use of swizzled data layouts in shared memory. We currently do not use such device-specific implementations. We have however demonstrated that even without such techniques, performance competitive with that of cuBLAS and CUTLASS can be obtained. We currently do not implement all of the optimizations employed in CUTLASS – these include thread block to tile re-mapping to improve L2-cache locality and the parallelization of the reduction dimension. These are the subject of future work.

Bhaskaracharya et al. [2] exploit polyhedral code generation techniques to automatically generate code for matrix-matrix multiplication. Given a computation DAG, their approach constructs a schedule tree for matmul and any operations fused with it and passes it to ISL [38], generating the corresponding CUDA code. This approach is thus one of generating C/CUDA code as opposed to staying closed in a unified intermediate representation throughout.

Bondhugula's study [3] was the first to evaluate and demonstrate that performance competitive with highly tuned BLAS libraries on CPUs, in certain cases at least, could be achieved using MLIR and that the reliance on handwritten code could be potentially reduced. The experimentation focused on single-core matmul on CPUs and provided a comparison with state-of-the-art vendor libraries, including MKL and GotoBLAS. The work showed existing well-studied recipes for optimizing matmul could be implemented step-by-step using MLIR transformation utilities and passes while measuring the impact of each individual optimization to get close to machine peak performance. This work has incorporated such an analysis and evaluation for GPUs that required a far deeper and more complex pipeline.

9 Conclusions

We designed and evaluated an approach using MLIR infrastructure to generate close to peak performance implementations of matrix-matrix multiplication-like operations and the fusion of such operations with simple point-wise operations targetin NVIDIA's GPU tensor cores. The performance we obtained was competitive with that of CuBLAS, CuTensor, and CuBLAS-LT. We were able to enable the fusion of simple and common point-wise operations like ReLU, matrix addition, and constant addition with matmul while exploiting producer-consumer reuse in registers.

Acknowledgments

We express gratitude to all those who have contributed to the open-source MLIR project, in particular to its Affine, GPU, and LLVM dialect infrastructure.

References

[1] Jeremy Appleyard and Scott Yokim. 2017. Programming Tensor Cores in CUDA 9. https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/.

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

[3] Uday Bondhugula. 2020. High Performance Code Generation in MLIR: An Early Case Study with GEMM. *CoRR* (2020). https://arxiv.org/abs/2003.00532.

[4] C. Lattner and M. Amini and U. Bondhugula and A. Cohen and A. Davis and J. Pienaar and R. Riddle and T. Shpeisman and N. Vasilache and O. Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.

[5] LLVM Community. 2022. The LLVM Project. https://llvm.org/.

[6] LLVM community. 2022. User Guide for NVPTX Back-end. https://llvm.org/docs/NVPTXUsage.html.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* (2018). http://arxiv.org/abs/1810.04805

[8] Thomas Faingnaert, Tim Besard, and Bjorn De Sutter. 2020. Flexible Performant GEMM Kernels on GPUs. *CoRR* abs/2009.12263 (2020). https://arxiv.org/abs/2009.12263

[9] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of High-Performance Matrix Multiplication. *ACM Trans. Math. Softw.* 34, 3 (2008). https://doi.org/10.1145/1356052.1356053

[10] Scott Gray. 2017. A full walk through of the SGEMM implementation. https://github.com/NervanaSystems/maxas/wiki/SGEMM.

[11] Stephan Herhut. 2020. MLIR on GPUs. MLIR Open Design Meeting, Apr 16, 2020.

[12] Stephan Herhut and Oleksandr Zinenko. 2019. GPUs in MLIR. MLIR Open Design Meeting, Dec 12, 2019.

[13] Junjie Lai and Andre Seznec. 2013. Performance Upper Bound Analysis and Optimization of SGEMM on Fermi and Kepler GPUs. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (CGO '13)*. IEEE Computer Society. https://doi.org/10.1109/CGO.2013.6494986

[14] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. San Jose, CA, USA, 75–88.

[15] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. *CoRR* abs/2002.11054 (2020). http://arxiv.org/abs/2002.11054.

[16] Jinhyuk Lee, Wonjin Yoon, Sungdong Kim, Donghyeon Kim, Sunkyu Kim, Chan Ho So, and Jaewoo Kang. 2019. BioBERT: a pre-trained biomedical language representation model for biomedical text mining. *CoRR* abs/1901.08746 (2019). http://arxiv.org/abs/1901.08746

[17] Yang Liu and Mirella Lapata. 2019. Text Summarization with Pretrained Encoders. *CoRR* (2019). http://arxiv.org/abs/1908.08345

[18] LLVM/MLIR. 2020. MLIR: A multi-level intermediate representation. https://mlir.llvm.org.

[19] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Orti. 2016. Analytical Modeling Is Enough for High-Performance BLIS. *ACM Trans. Math. Softw.* 43, 2, Article 12 (Aug. 2016).

[20] Justin Luitjens. 2013. Increase Performance with Vectorized Memory Access. https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/.

[21] MLIR. 2020. MLIR dialects. https://mlir.llvm.org/docs/Dialects/.

[22] MLIR. 2020. MLIR language reference. https://mlir.llvm.org/docs/LangRef/.

[23] Pandu Nayak. 2019. Understanding searches better than ever before. https://www.blog.google/products/search/search-language-understanding-bert/.

[24] NVIDIA. 2018. NVIDIA Turing GPU Architechture. https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[25] NVIDIA. 2020. NVIDIA Ampere GA102 GPU Architechture. https://images.nvidia.com/aem-dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf.

[26] NVIDIA. 2021. cuBLAS. https://docs.nvidia.com/cuda/cublas/.

[27] NVIDIA. 2021. CUTLASS. https://github.com/NVIDIA/cutlass.

[28] NVIDIA. 2021. Mixed precision training in deep learning. https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html.

[29] NVIDIA. 2021. Nsight Systems. https://docs.nvidia.com/nsight-systems/UserGuide/index.html.

[30] NVIDIA. 2021. PTX programming guide. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[31] NVIDIA. 2022. CUDA Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[32] NVIDIA. 2022. cuTensor. https://docs.nvidia.com/cuda/cutensor/index.html.

[33] PolyMage Labs. 2020. MLIRX. https://github.com/polymage-labs/mlirx.

[34] Guangming Tan, Linchuan Li, Sean Triechle, Everett Phillips, Yungang Bao, and Ninghui Sun. 2011. Fast Implementation of DGEMM on Fermi GPU. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2063384.2063431

[35] 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*. Association for Computing Machinery. https://doi.org/10.1145/3315508.3329973

[36] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Softw.* 41, 3, Article 14 (June 2015).

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *CoRR* (2017).

[38] Sven Verdoolaege. 2010. ISL: An Integer Set Library for the Polyhedral Model. In *Mathematical Software – ICMS 2010*, Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama (Eds.).

[39] 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)*.

[40] Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. 2017. Understanding the GPU Microarchitecture to Achieve Bare-Metal Performance Tuning. *SIGPLAN Not.* (2017). https://doi.org/10.1145/3155284.3018755