Performance Analysis of Tiling for Automatic Code Generation for GPU Tensor Cores using MLIR

A PROJECT REPORT

SUBMITTED IN PARTIAL FULFILMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

Master of Technology

IN

Faculty of Engineering

BY

Vivek Khandelwal

Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

July, 2021

Declaration of Originality

I, **Vivek Khandelwal**, with SR No. **04-04-00-10-42-19-1-16943** hereby declare that the material presented in the thesis titled

Performance Analysis of Tiling for Automatic Code Generation for GPU Tensor Cores using MLIR

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2019-2021**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: Prof. Uday Kumar Reddy B. Advisor Signature

DEDICATED TO

My Family and Friends

&

Teachers and Students of IISc

&

To all those who lost their lives during the Covid-19 Pandemic.

Acknowledgements

First and foremost, I would like to extend my gratitude to my advisor, **Prof. Uday Kumar Reddy B.**, for his noble guidance and support with full encouragement and enthusiasm. I am grateful to him for his sincerity, encouragement, and continuous inspiration throughout my Master's degree.

I thank **Navdeep Kumar**, my partner in this project, for his continuous support and valuable insights, which played an important role in this project. The regular discussions we use to have were constructive in framing the right picture, formalizing, and solving the problem. The learning experience I have had would have been incomplete without him.

I want to thank my lab mates at the **Multicore Computing Lab (MCL)**. Also, I thank all my friends for making my IISc campus life fun, enjoyable and memorable. I would also like to thank all the office staff in the Department of Computer Science and Automation (CSA), including Mrs. Padmavathi, Mrs. Kushael, Mrs. Meenakshi, Mrs. Nishitha. Their effort to make administrative tasks smooth and easy for all the members of the department is invaluable. I am thankful to system administrator Akshay Nath for his technical support. Finally, I would like to thank my parents and sister for their continuous love, support, and guidance.

Abstract

With machine learning and deep learning operations becoming computationally expensive, the demand for accelerators such as GPUs is increasing rapidly. To meet these demands, GPU vendors are continuously introducing specialized functional units in GPUs. To exploit the full capability of these functional units, GPU vendors also provide efficient libraries which are manually fine-tuned. The problem with these manually tuned libraries is that they require people with specialized skills to write them and may involve redundant work. We resolve this problem by automating the process of generating efficient code for GPUs using the MLIR compiler framework.

In this thesis, we propose a solution for the automatic generation of kernels for matrix multiplication. We discuss optimization such as tiling, loop vectorization, memory padding, and loop unrolling performed by us for achieving the performance comparable to the state-of-the-art libraries such as cuBLAS. The main focus of this thesis is an analysis of the impact of tiling on the performance of the automatic code generation for the Tensor cores for the matrix multiplication. The experiments are performed on the NVIDIA Turing architecture and for the problem size 8192x8192x8192. I also want to mention that the best performance achieved by us is 48.9 TFLOPS which is nearly 90% of the performance of cuBLAS.

This thesis also discusses the multiple levels of tiling, its significance in the performance, and the optimal tile size selection problem. The analysis of the performance impact of two levels of tiling, namely the thread-block level and warp level, is done based on parameters like bank conflicts, compute to memory ratio, cache hit rate, and barrier stalls. We propose an analytical model for tile-size selection, which takes input data element size and shared memory capacity and gives the optimal tile size for thread-block level tiling. We propose some rules for the wap level tiling that needs to be followed for the warp level tile size selection. In the end, we also discuss the limitations of the tile-size selection model proposed by us, such as it doesn't work well for small problem sizes.

Publications based on this Thesis

In preparation for submission in Aug 2021.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

With the advances in machine learning and deep learning research, the demand for High-Performance Computing (particularly GPGPUs) has also increased. GPUs nowadays come with thousands of parallel processing cores capable of rapidly solving large problems with substantial parallelism. The latest GPU architectures such as Nvidia Turing [3], Ampere [4] consist of Tensor cores [5], [6], which can perform several mixed-precision matrix-multiply and accumulate computations in a single operation. Usually, matrix operations are among the commonly performed operations in the fields of HPC (High-Performance Computing) and ML (Machine Learning), which are highly parallel in nature. The matrix multiplication is the most common out of all the matrix operations performed, which in generalized form is known as GEMM (General Matrix Multiply) and is represented as $\boldsymbol{D} = \alpha\boldsymbol{A}\boldsymbol{B} + \beta\boldsymbol{C}$. The ordinary matrix multiplication $\boldsymbol{A}\boldsymbol{B}$ can be performed by setting α to one and $\boldsymbol{C}$ to an all-zeros matrix. GEMM in the field of HPC is used for dense linear algebra [7], [8], earthquake simulation [9], and climate prediction [10]. In ML, GEMM is used for training convolutional neural networks, long short-term memory (LSTM) cells, and natural language processing.

CUDA, a parallel computing API created by Nvidia, enables us to execute C, C++, etc., codes on Nvidia GPUs. Having GPUs and CUDA API is not enough; we also need a highly optimized kernel to run on these GPUs to maximize the usage of available resources. In general, GPU vendors provide manually tuned implementations such as cuBLAS [11], cuDNN [12], etc., which contain various deep learning primitives for several different matrix layouts. The problem with these manually tuned implementations is that they need to be tuned for every new architecture that comes into the market, and tuning them requires people with specialization in those domains.

The process of generating efficient fine-tuned implementations of deep learning primitives can be automated using polyhedral frameworks. The matrix operations, such as matrix multiplication, bias addition, etc., can be expressed as affine loop-nests, which can be easily analyzed and transformed using polyhedral frameworks.

The advantages of automating the process of generation of an efficient kernel are:

- It can be used for different architectures.
- It can be used for different matrix layouts and different matrix sizes.
- It also offers us to explore a non-trivial space of optimizations much more easily and automatically.
- We can get performance close to hand-tuned implementations of kernels without writing code manually.

The compiler infrastructure parts that we build for optimizations, such as tiling and unroll-and-jam, are common to both CPUs and GPUs. The compiler framework we are using to build this infrastructure is MLIR (Multi-Level Intermediate Representation) [13, 14, 15].

1.2 Our Contributions

The contributions made by us in this project are given below:

1. We provide a solution to automate the process of generating efficient kernels for Tensor cores for GPU using the MLIR compiler framework.
2. We show with experimental results that our solution achieves performance 85-95% of state-of-the-art libraries such as cuBLAS.
3. We analyze and discuss in detail the role of tiling over the performance of automatic code generation for GPUs.
4. We propose a tile-size selection model for thread-block level and warp level tiling, which yields us a tile size for optimal performance.
5. We also discuss the importance of several other optimizations such as loop vectorization, shared memory padding, unrolling, and delaying data copies.

1.3 Related Work

Several libraries already exist that include handwritten, highly optimized kernels for GPU. CUDA libraries such as cuDNN [12], and cuBLAS [11] provides highly tuned implementations of deep learning functions/algorithms and basic linear algebra subroutines. These libraries are optimized only for a limited set of matrix layouts, and thus, they don't provide much flexibility. CUB [16] an NVIDIA's library provides software components that are reusable for CUDA programming model,'s each layer such as Warp-wide, Block-wide, and Device-wide primitives for constructing high-performance, maintainable CUDA kernel code. The components provide by CUB give a state-of-the-art performance.

CUTLASS [17] is a CUDA C++ template library that contains components to instantiate performant kernels on GPUs. CUTLASS also contains support for mixed-precision computation for Tensor cores. It is not easy for end-users to extend the CUTLASS since its codebase is large and maintaining such a huge codebase is difficult. The performance achieved by CUTLASS [18] for WMMA GEMM is 95% of the performance of cuBLAS. cuTensor [19] is a CUDA library provided by NVIDIA for its GPUs, supports several Tensor operations, for example - pointwise operations with pointwise operator fusion support. cuBLAST a library that is a lighter version of CUBLAS, is made available by NVIDIA for basic linear-algebra subroutines dedicated to GEMM, which provides flexibility in supporting more data types for input and compute matrices and more matrix layouts.

Halide [20] is a DSL (embedded in the C++) compiler framework for image processing; it is also extended for supporting Tensor cores. The problem with Halide is that it does not support complex data types and the flexibility provided by it is limited; also, it supports only a single combination of memory layouts of the input matrices. Due to the Tensor comprehensions, [21] developed by Vasilache et al., and by using the polyhedral compilation techniques, the Halide compiler generates CUDA kernels for a given mathematical specification of a deep learning graph.

Some of the recent works in automatic kernel generation are being done by Somashekaracharya G. Bhaskaracharya et al. [22], and Thomas Faingnaert et al. [23]. Bhaskaracharya et al. [22] in their work used a polyhedral approach like ours to generate efficient CUDA kernels for matrix multiplication using inline assembly instructions. On the other hand, we have used MLIR for generating efficient CUDA kernels. They also proposed an extended approach for generating fused kernels such as matrix multiplication plus ReLU activation or bias addition. In contrast, we have generated kernels only for matmul.

Faingnaert et al. [23] framed the problem of manual code generation as a two-language

problem. Faingnaert et al. [23] stated that "efficient kernel generation require either low-level programming, which implies low programmer productivity, or using libraries that only offer a limited set of components." The author used the LLVM framework [24], and Julia programming language [25] for generating the efficient kernels. The Julia is a dynamic and flexible programming language suitable for numeric and scientific computations. The performance of Julia is comparable to traditional statically-typed languages like C and C++. The work of Faingnaert et al. [23] is different from our work in two ways: we have used the affine dialect based on the polyhedral techniques and the MLIR compiler framework [13, 14, 15]; instead, their work is based on the LLVM compiler framework and Julia programming language. The advantage of using MLIR over LLVM is that MLIR provides multiple levels of abstractions like loop abstractions, math abstractions, tensor, and vector abstractions. Also, one can define their own operations and abstractions in MLIR suitable for the problem they are trying to solve, which is the biggest difference from LLVM.

1.4 Outline

The rest of the report is organized as follows: We provide the necessary background on MLIR, GPU, and Tensor cores in Section 2. Then, in Section 3, we have explained the design and implementation of automatic kernel generation for Tensor cores. In Section 4, first, we have given the details on the experimental setup, then we have discussed in detail the impact of tiling on performance. Section 5 comprises of tile size selection model. The report ends with a conclusion in Section 6.

Chapter 2

Background

This section provides background on MLIR, GPU programming and Tensor cores on GPUs.

2.1 MLIR

Multi-Level Intermediate Representation(MLIR) [13, 14, 15] aims to build reusable, extensible compiler infrastructure and reduce the cost of building domain-specific compilers. MLIR is a hybrid IR that can be used for multiple different requirements, such as:

- It can represent dataflow graphs (such as in TensorFlow), including dynamic shapes, variables, etc.
- It can represent kernels in a form suitable for optimization for Machine Learning operations.
- It can host high-performance-computing-style loop optimizations across kernels (loop fusion, loop interchange, multi-level tiling, unroll-and-jam).
- The target-specific operations such as accelerator-specific operations can also be represented using MLIR.
- It can represent kernels at different levels of abstractions(dialects in MLIR), which can help perform transformations and optimizations not possible at a single level.

The MLIR structure is made up of the following components:

- **Operations:** This is the basic unit of semantics in MLIR and is commonly referred to as *Op*. In MLIR, everything is modeled as ops, whether it is instruction, module, or function. They take as input zero or more *values* called *operands* and produces zero or more *values* called *results* respectively.

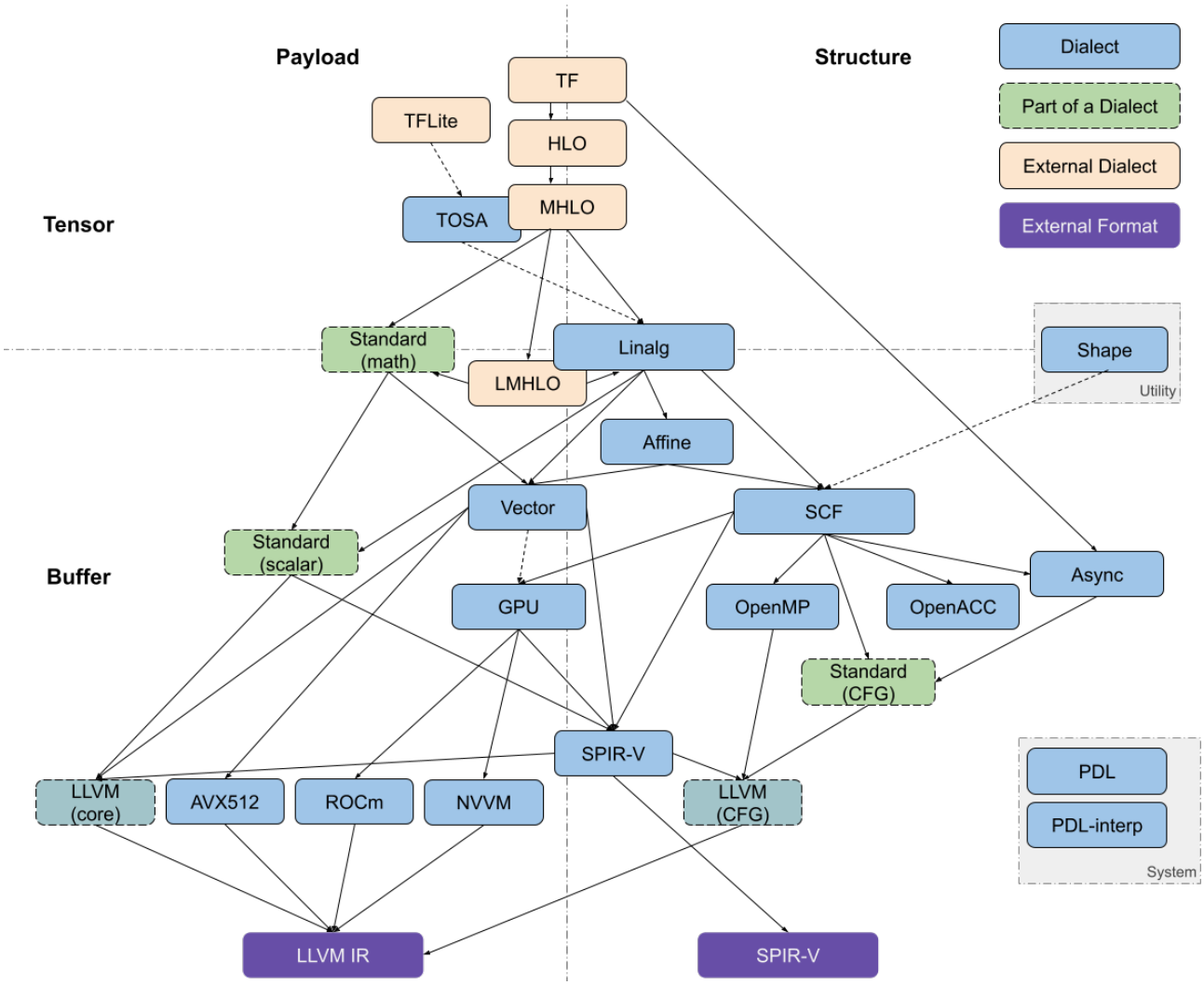

Figure 2.1: Different levels of abstraction and various dialects in MLIR, including Affine, Scf, Standard, and GPU dialects used in our solution. Source: Zinenko et al. [1].

```
func@main(%arg0: tensor<1x2xi32>, %arg1: tensor<2x1xi32>) {
  %matrixA = alloc() : memref<128x64xf16>
  %matrixB = alloc() : memref<64x128xf16>
  %matrixC = alloc() : memref<128x128xf16>
  // Initialize the matrices.
  linalg.matmul ins(%matrixA, %matrixB: memref<128x64xf16>, memref<64x128xf16>)
                outs(%matrixC: memref<128x128xf16>)
  %result = "mhlo.dot"(%arg0, %arg1) : (tensor<1x2xi32>, tensor<2x1xi32>)
                                       -> tensor<i32>
}
```

Figure 2.2: An example IR in MLIR comprising operation and types from different dialects such as Tensor, Standard (std), Linalg, MHLO.

```
func @dot_vector(%arg0: tensor<1x2xi32>, %arg1: tensor<2x1xi32>) -> tensor<i32> {
  %0 = "mhlo.dot"(%arg0, %arg1) : (tensor<1x2xi32>, tensor<2x1xi32>) -> tensor<i32>
  return %0: tensor<i32>
}
```

(a) MLIR-HLO abstraction level.

```
affine.for %i = 0 to %M {
  affine.for %j = 0 to %N {
    affine.for %l = 0 to %K {
      %a = affine.load %gpu_A[%i, %l] : memref<8192x8192xf16>
      %b = affine.load %gpu_B[%l, %j] : memref<8192x8192xf16>
      %c = affine.load %gpu_C[%i, %j] : memref<8192x8192xf32>
      %p = mulf %a, %b : f16
      %q = fpext %p : f16 to f32
      %co = addf %c, %q : f32
      affine.store %co, %gpu_C[%i, %j] : memref<8192x8192xf32>
    }
  }
}
```

(b) Affine dialect abstraction level.

Figure 2.3: Matrix multiplication representation in MLIR-HLO, a higher level dialect, and Affine dialect, a loop-optimization-friendly dialect in MLIR.

- **Attributes:** Attributes are structured compile-time static information, e.g., integer constant values, string data, etc. Each op instance has an open key-value dictionary from string names to attribute values.

- **Regions and blocks:** A *region* in MLIR is a list of blocks, and a block contains a list of operations that may further contain regions. The blocks inside the region form a Control Flow Graph (CFG). Each block may have *successor* blocks to which the control flow may be transferred, and it ends with a terminator op.

- **Dialects:** A dialect is a logical grouping of operations, attributes, and types. Operations from multiple dialects can coexist at any level of the IR at any point in time. Dialects allow extensibility and provide flexibility that helps in performing specific optimizations and transformations. There are many dialects in the MLIR, but the dialects we have worked with are Affine, GPU, LLVM, SCF, and Standard (std).

- **Functions and modules:** A module is an operation with a single region containing a single block and terminated by a dummy op that does not transfer the control flow. On the other hand, a function is an op with a single region, with arguments corresponding to function arguments.

Some of the MLIR dialects that we have used in our work are explained below:

- **Affine Dialect:** This dialect uses techniques from polyhedral compilation to make dependence analysis and loop transformations efficient and reliable. We have performed most of the optimizations and transformations at the level of affine dialect. The description of some ops and terminologies from the affine dialect that we have used in this document is as follows:

  - **Affine Expressions:** An affine expression is a linear expression plus a constant. MLIR extends this definition of affine expression to allow 'floordiv' (floor function), 'ceildiv' (ceiling function) and 'mod' (modulo operation) with respect to positive integer constants.

    Ex: (v1 + v2 + v3 mod 5) is an affine expression of literals v1, v2, and v3.

  - **Affine Maps:** They are mathematical functions that transform a list of inputs (dimensions and symbols) into a list of results, with affine expressions combining the dimensions and symbols. Affine maps help in doing powerful analysis and transformations due to the restrictions imposed on their form. Affine maps may be defined

inline at the point of use or maybe hoisted to the top of the file and given a name with an affine map definition and used by name.

Ex: affine_map< (d0, d1)[s0] → (d0, d0 + d1 + s0 floordiv 2) >, here d0, d1 are the input dimensions and s0 is the input symbol. The output of this affine map shown in rhs is a tuple of 3 values.

- **AffineForOp (affine.for):** This operation represents an affine loop nest. It has one region containing its body. It executes its body a number of times, iterating from a lower bound to an upper bound by a stride. The stride is represented by 'step', whose default value is one if it is not present. The lower and upper bounds are represented as the result of applying an affine mapping to a list of SSA values. Since the mapping may return multiple results, the minimum/maximum of values is used as lower/upper bound, respectively.
- **AffineParallelOp (affine.parallel):** It represents a hyper-rectangular affine parallel band, defining multiple SSA values for its induction variables. It has one region containing its body. This operation's lower/upper bound can have a list of values since this operation is multidimensional.

- **GPU Dialect:** GPU dialect models the general GPU programming paradigm similar to CUDA or OpenCL in MLIR. Its goal is to provide abstractions for kernel invocation not present at the lower level. GPU dialect is meant to be vendor agnostic.
- **NVVM Dialect:** Since we are focusing on Tensor core code generation, we use and extend another Nvidia specific dialect known as NVVM. This dialect provides operations that are directly mapped to the NVPTX back-end in LLVM.
- **LLVM Dialect:** The final stage of code generation involves lowering to LLVM IR, from where the LLVM back-end takes control and generates the target code. To model LLVM IR, we use this dialect called the LLVM dialect, the lowest level of abstraction present in MLIR.

2.2 GPU Programming

GPUs [26] are massively parallel processors, which means many threads execute the same function commonly referred to as kernel in parallel. Since our approach, analysis, and experimentation are only confined to NVIDIA GPUs, we will limit our discussion to NVIDIA GPUs, and the CUDA programming model [27]. The smallest unit of execution is a thread organized

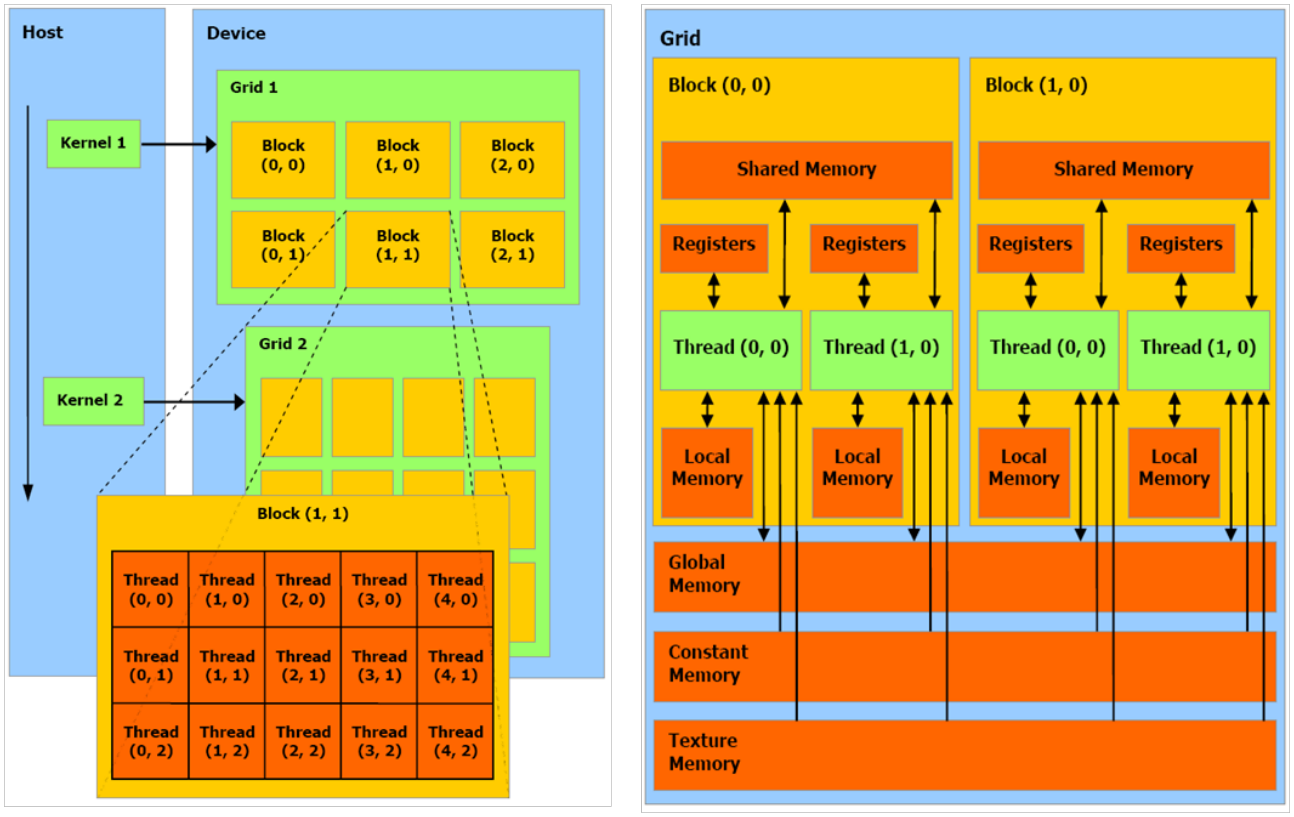

Figure 2.4: Memory hierarchy in GPUs, i.e., global memory, shared memory, and registers. Thread hierarchy in GPUs, i.e., thread grid, thread blocks, and threads. Source: NVIDIA Corporation [2].

in a thread hierarchy. The hardware groups threads into sets of 32 threads called warps. The 32 threads in the same warp execute in a SIMT (Single Instruction Multiple Thread) manners. In other words, these threads within a warp execute the same instruction simultaneously, possibly on different data. The threads are logically grouped into blocks known as thread-block. The set of all thread blocks on the GPU device is called the grid. Just like the threads, the GPU memory is also organized hierarchically. The smallest and fastest type of memory in GPUs is the register file or registers. Each thread in GPU typically has access to 255 registers. The threads within a thread block have their own shared memory set, which they use for communication. The largest capacity memory on the GPUs is global memory, but it also has the highest latency. All threads on the device (GPU) can access the global memory, irrespective of which thread block they belong to.

2.3 Tensor Cores

Tensor cores [5], [6] were first introduced in NVIDIA's Volta architecture [28] in 2017. Tensor cores are programmable matrix-multiply-and-accumulate units that provide a 4x4x4 matrix processing array which performs the operation $\mathbf{D} = \mathbf{A} * \mathbf{B} + \mathbf{C}$, where **A**, **B**, **C** and **D** are 4×4 matrices. Each Tensor core performs 64 floating-point fused-multiply-add mixed-precision operations per clock. Multiple Tensor cores are used concurrently by a full warp of execution. A warp comprising of 32 threads provide a larger 16x16x16 matrix operation to be processed by the Tensor cores. The Tensor cores operation are exposed by CUDA as warp-level matrix operations in the CUDA C++ WMMA (Warp Matrix Multiply Accumulate) API [29]. Only a limited set of data types are supported by tensor cores such as TF32, FP16, FP64, INT8 for input. We can use Tensor cores through libraries such as NVIDIA's cuDNN [12] library that contains Tensor cores kernels. NVIDIA's cuTENSOR [19] based on CUTLASS [17] contains Tensor-Core-accelerated kernels. LLVM also provides intrinsics which are mapped one-to-one with the functions provided by WMMA API.

Chapter 3

Design and Implementation

In this section, we have explained the approach and implementation details of automatic code generation (for matrix multiplication) for Tensor cores using MLIR.

3.1 Input IR

In Fig. 3.1, the input IR represents the basic matrix multiplication operation in some higher abstraction level such as MLIR-HLO [30] or TF. The IR is now lowered to affine dialect abstraction level to perform loop optimizations.

3.2 Loop Tiling

Loop Tiling, also known as blocking, important from parallelism and locality viewpoint, is performed for better data reuse and performance. The key objective of the tiling is to maximize the ratio of computation operations to memory operations. We have performed two-level tiling: thread-block level and warp level. The tile sizes are chosen based on the tile-size selection model discussed in Section 5.

3.3 Shared Memory Buffer Allocation

Since we are generating kernels targeting GPUs, we allocate buffers in GPU's shared memory. After the allocation, the data is copied from GPU's global memory into the shared memory buffers. We have allocated the shared memory buffers only for the input matrices (A and B), and this is a design choice that we have made. The thread-block level tile size determines the shared memory buffer size for both the matrices. The tile sizes are chosen such that the shared memory can be used maximally. The shared memory buffers are used for lower latency and higher bandwidth.

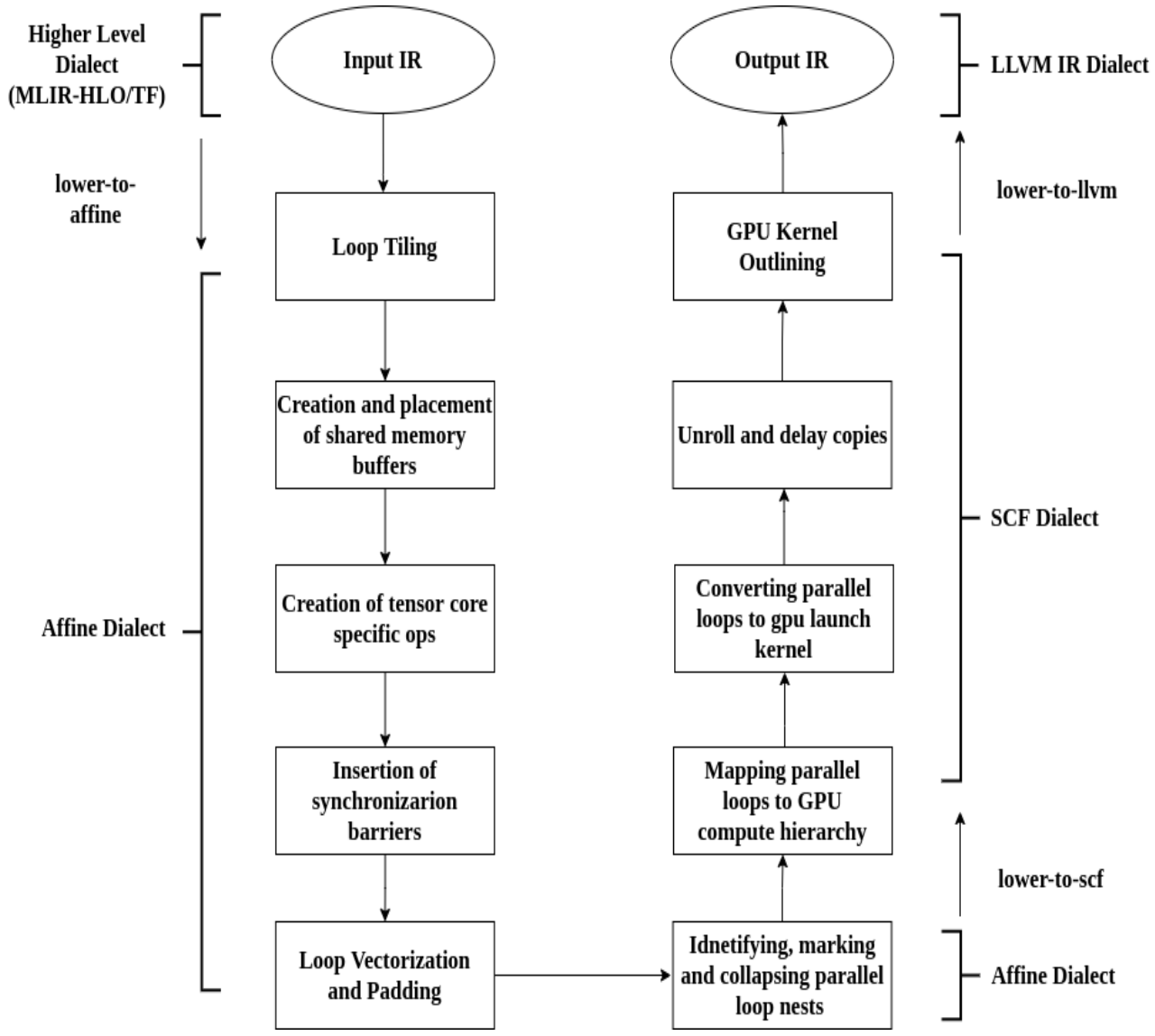

Figure 3.1: Design and Implementation details comprising the workflow of optimizations performed.

```
affine.for %arg0 = 0 to 8192 {
    affine.for %arg1 = 0 to 8192 {
        affine.for %arg2 = 0 to 8192 {
            %9 = affine.load %memref_1[%arg0, %arg2] : memref<8192x8192xf16>
            %10 = affine.load %memref_0[%arg2, %arg1] : memref<8192x8192xf16>
            %11 = affine.load %memref_2[%arg0, %arg1] : memref<8192x8192xf32>
            %12 = mulf %9, %10 : f16
            %13 = fpext %12 : f16 to f32
            %14 = addf %11, %13 : f32
            affine.store %14, %memref_2[%arg0, %arg1] : memref<8192x8192xf32>
        }
    }
}
```

(a) IR before loop tiling transformation.

```
affine.for %arg0 = 0 to 8192 step 128 {
    affine.for %arg1 = 0 to 8192 step 128 {
        affine.for %arg2 = 0 to 8192 step 64 {
            affine.for %arg3 = 0 to 128 step 64 {
                affine.for %arg4 = 0 to 128 step 64 {
                    affine.for %arg5 = 0 to 64 step 32 {
                        affine.for %arg6 = 0 to 64 {
                            affine.for %arg7 = 0 to 64 {
                                affine.for %arg8 = 0 to 32 {
                                    %9 = affine.load %memref_![%arg0 + %arg3 + %arg6,
                                    %arg2 + %arg5 + %arg8] : memref<8192x8192xf16>
                                    %10 = affine.load %memref_0[%arg2 + %arg5 + %arg8,
                                    %arg1 + %arg4 + %arg7] : memref<8192x8192xf16>
                                    %11 = affine.load %memref_2[%arg0 + %arg3 + %arg6,
                                    %arg1 + %arg4 + %arg7] : memref<8192x8192xf32>
                                    %12 = mulf %9, %10 : f16
                                    %13 = fpext %12 : f16 to f32
                                    %14 = addf %11, %13 : f32
                                    affine.store %14, %memref_2[%arg0 + %arg3 + %arg6,
                                    %arg1 + %arg4 + %arg7] : memref<8192x8192xf32>
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

(b) IR after loop tiling transformation.

Figure 3.2: Example IR before and after performing the loop tiling transformation.

3.4 Creating Tensor Core Specific Operations

Since the automatic code generation is done targeting Tensor cores available in the latest GPU architectures, we generate Tensor cores specific operations such as wmma.load, wmma.store, wmma.mma_sync, which replaces the scalar load, store, and compute operations. The additional details will be available in [31].

3.5 Insertion of Synchronization Barriers

The synchronization barriers are inserted in the code to ensure data is available before the computation begins. The synchronization barriers being used in our generated code are actually a thread-block level barrier to synchronize all threads in a thread block after copying the input matrix tiles into the shared memory. The number of synchronization barriers inserted is kept minimal because the barrier causes stall, ultimately degrading performance.

3.6 Loop Vectorization and Padding

The vectorization is performed to make use of the SIMD instruction set available in GPU architectures. For doing loop vectorization in MLIR, we have to convert memrefs of f16 (input matrices data type) into memrefs of a vector of f16 and transform the loop bounds and loop bodies accordingly. We have created the vectors, each comprising 8 elements. The vectorization is a crucial part of the solution because it provides a 2x speedup in the performance.

We know that memory is divided into banks and successive 32-bit words assigned to successive banks. Also, each bank can service one address per cycle. If multiple simultaneous accesses are made to a bank, then it results in a bank conflict. As a result, the conflicting accesses are serialized. Hence the memory access time increases, and the performance decreases. The solution to this is shared memory padding. The padding is done to ensure simultaneous bank accesses are reduced to the extent possible, thereby reducing the memory bank conflicts. For padding, we have tuned a parameter known as the padding factor, which specifies how much value the memory should be padded. Also, we have padded the k dimension of the matrices. The optimal padding factor turns out to be 8, which gives us the best performance.

3.7 Identification, Marking and Collapsing of Parallel Loop Nests

To map the parallel loop nests to GPU compute hierarchy, we first need to identify and mark the parallel loops in the IR. An affine.for is parallel if all its iterations can be executed parallelly,

```
func @coalesce_affine_parallel1() {
    %0 = alloc() : memref<1024x1024xf32>
    %c1 = constant 1 : index
    %1 = dim %0, %c1 : memref<1024x1024xf32>
    affine.for %i = 0 to %1 step 32 {
      affine.for %j = %i to 1024 {
        affine.for %k = 0 to 1024 {
          ...operations
        }
      }
    }
    return
  }
```

(a) Input IR.

```
func @coalesce_affine_parallel1() {
    %0 = alloc() : memref<1024x1024xf32>
    %c1 = constant 1 : index
    %1 = dim %0, %c1 : memref<1024x1024xf32>
    affine.for %i = 0 to %1 step 32 {
      affine.parallel (%j, %k) = (%i, 0) to (1024, 1024) {
        ...operations
      }
    }
    return
  }
```

(b) Output IR.

Figure 3.3: Example IR before and after the identification, marking and collapsing of parallel loop nests.

or we can say if all its iterations are independent of each other. We mark parallel loops as affine.parallel so that during the time of execution, there is a clear separation between the set of instructions that can execute in parallel and the instructions which need to be executed serially. The pass works as follows:

1. Walks over the entire input IR and collects all those affine.for which are parallel, using a utility that checks for all the dependencies inside the iteration space of affine.for.

2. One by one converts all affine.for obtained in step 1 into affine.parallel and keeps affine.parallel's bounds same as bounds of affine.for.

After marking the loops as parallel, we collapse perfectly nested affine.parallel ops into a single n-dimensional affine.parallel op where n is the sum of dimensions of all affine.parallel ops, which are coalesced together. This process happens as follows:

1. Collects all the affine.parallel ops, which are perfectly nested.

2. Collects all the affine expressions corresponding to the lower and upper bound map of affine.parallel ops obtained in step 1.

3. Collects all the lower and upper bound operands of affine.parallel ops.

4. Creates new lower and upper bound maps using affine expressions obtained in step 2.

5. Creates new coalesced affine.parallel op using maps and operands obtained in steps 4 and 3, respectively.

Collapsing the parallel loop nests helps in mapping the loops to GPU compute hierarchy since all the perfectly nested parallel loops are now grouped into a single op.

3.8 Mapping Parallel Loop Nests to GPU Compute Hierarchy

To execute the kernel on the GPU, the parallel loops need to be mapped to GPU entities such as thread blocks, warps, and threads. We greedily map loops starting from outermost to innermost to each GPU entity. Mapping the parallel loop means to specify which loop is distributed over a thread block grid, distributed over a thread block, or distributed over warps. For mapping the parallel loops, we move from the outermost loop to the innermost loop. Each parallel loop in between is mapped to some GPU compute hierarchy. The parallel loops at nesting level one are mapped to the thread block grid. The parallel loops at nesting level two are mapped to

```
func @main() {
  scf.for %arg0 = %c0 to %c8192 step %c1 {
    scf.for %arg1 = %c0 to %c8192 step %c1 {
        \\ Input matrix initialization.
    }
  }
  scf.parallel (%arg0, %arg1) = (%c0, %c0) to (%c8192, %c8192) step (%c128, %c128) {
    // operations...
    scf.parallel (%arg2, %arg3) = (%c0, %c0) to (%c128, %c128) step (%c64, %c64) {
      // operations...
      scf.parallel (%arg4) = (%c0) to (%c1024) step (%c1) {
        // operations...
      } {mapping = [{bound = #map, map = #map, processor = 0 : i64}]}
      scf.parallel (%arg4) = (%c0) to (%c1024) step (%c1) {
        // operations...
      } {mapping = [{bound = #map, map = #map, processor = 0 : i64}]}
      gpu.barrier
      %49:16 = scf.for %arg4 = %c0 to %c8128 step %c64 iter_args(/*arguments*/) {
        scf.parallel (%arg21) = (%c0) to (%c1024) step (%c1) {
          // operations...
        } {mapping = [{bound = #map, map = #map, processor = 0 : i64}]}
        scf.parallel (%arg21) = (%c0) to (%c1024) step (%c1) {
          // operations...
        } {mapping = [{bound = #map, map = #map, processor = 0 : i64}]}
        %51:16 = scf.for %arg21 = %c0 to %c64 step %c32 iter_args(/*arguments*/) {
          %52:16 = scf.for %arg38 = %c0 to %c32 step %c16 iter_args(/*arguments*/) {
            // operations...
          }
        }
      }
      gpu.barrier
      %50:16 = scf.for %arg4 = %c0 to %c64 step %c32 iter_args(/*arguments*/) {
        %51:16 = scf.for %arg21 = %c0 to %c32 step %c16 iter_args(/*arguments*/) {
          // operations...
        }
      }
    } {mapping = [{bound = #map, map = #map, processor = 7 : i64}, {bound = #map,
    map = #map, processor = 6 : i64}]}
  } {mapping = [{bound = #map, map = #map, processor = 1 : i64}, {bound = #map,
  map = #map, processor = 0 : i64}]}
  return
}
```

Figure 3.4: Example IR after the mapping of parallel loop nests to GPU compute hierarchy.

the thread block, and the ones at nesting level three are mapped to the warps. All parallel loops which are at nesting level four or more are marked as sequential. The mapping is done by attaching an attribute to each parallel loop.

3.9 Converting Parallel Loops to GPU Launch Kernel

In MLIR's GPU dialect, we have an operation known as LaunchOp, which launches a kernel on the specified grid of thread blocks. The body of the kernel is the single region that this operation contains. The parallel loops mapped to the thread block grid GPU compute hierarchy in the previous section are now converted into a LaunchOp. There are six operands of the LaunchOp, which are grid and block sizes. The grid and block sizes are determined using the thread-block level tile size and warp level tile size. The body region of launch op contains six arguments that are block identifiers and thread identifiers, along the x,y,z dimensions. The loop induction variable, lower and upper bounds of parallel loops mapped to the thread block, and warp in the previous section are also modified based on some computations performed using tile sizes of both the levels and linear thread id.

3.10 Unrolling and Delaying Copies

The loop unrolling provides better opportunities for instruction scheduling and register tiling. Loop unrolling also helps reduce control overhead, and reduced instruction count due to fewer number compare and branch instructions. The loops are unrolled based on a loop unroll factor, which specifies that by what extent the loop has to be unrolled. In our case, we have unrolled only the copy loops and have unrolled them completely. After unrolling the copy loops, we delay the data copy by moving the copy instructions after the computation instructions. The copies are delayed to hide the latency of loads from global memory. The additional details will be available in [31].

The GPU kernel outlining pass is run now, converting all the GPU dialect's LaunchOp into LaunchFuncOp which launches a function as a GPU kernel on the specified grid of thread blocks. Since we are using the LLVM backend for code generation, we now convert MLIR to LLVM IR. Now, the LLVM backend will generate the target code.

```
func @main() {
  scf.for %arg0 = %c0 to %c8192 step %c1 {
    scf.for %arg1 = %c0 to %c8192 step %c1 {
     // Input matrix initialization.
    }
  }
  gpu.launch blocks(%arg0, %arg1, %arg2) in (%arg6 = %c64, %arg7 = %c64, %arg8 = %c1)
             threads(%arg3, %arg4, %arg5) in (%arg9 = %c128, %arg10 = %c1, %arg11 = %c1) {
    // operations...
    scf.for %arg12 = %24 to %c128 step %c128 {
      scf.for %arg13 = %25 to %c128 step %c128 {
        // operations...
        scf.for %arg14 = %c0 to %c8 step %c1 {
          // operations...
        } {isCopyLoopNest = true}
        scf.for %arg14 = %c0 to %c8 step %c1 {
          // operations...
        } {isCopyLoopNest = true}
        gpu.barrier
        %50:16 = scf.for %arg14 = %c0 to %c8128 step %c64 iter_args(/*arguments*/) {
          scf.for %arg31 = %c0 to %c8 step %c1 {
            // operations...
          } {isCopyLoopNest = true}
          scf.for %arg31 = %c0 to %c8 step %c1 {
            // operations...
          } {isCopyLoopNest = true}
          %52:16 = scf.for %arg31 = %c0 to %c64 step %c32 iter_args(/*arguments*/) {
            %53:16 = scf.for %arg48 = %c0 to %c32 step %c16 iter_args(/*arguments*/) {
              // operations...
            }
          }
        }
        gpu.barrier
        %51:16 = scf.for %arg14 = %c0 to %c64 step %c32 iter_args(/*arguments*/) {
          %52:16 = scf.for %arg31 = %c0 to %c32 step %c16 iter_args(/*arguments*/) {
            // operations...
          }
        }
      // operations...
      }
    }
    gpu.terminator
  }
  return
}
```

Figure 3.5: Example IR after converting the parallel loops to GPU launch kernel.

Chapter 4

Performance Analysis of Tiling

Loop tiling, also known as blocking, is done for better data reuse and, all state-of-the-art implementations of GEMM perform two-level tiling. We have also performed two-level tiling, namely thread-block tiling and, warp-level tiling also known as register tiling. In this section, we first explain the experimental setup, and then we discuss the impact of different levels of tiling on the performance.

We have used the following symbols in the subsequent subsections:

- M, N, and K constitute the problem size for dimensions m,n, and k.
- T_m, T_n, and T_k is the thread-block level tile size for dimensions m,n, and k, respectively.
- W_m, W_n, and W_k is the warp level tile size for dimensions m,n, and k, respectively.

4.1 Experimental Setup

The experiments are done on a server with Intel Xeon Silver 4110 processor based on Intel Skylake architecture and NVIDIA GeForce RTX 2080 Ti GPU. The detailed information is given in Table 4.1. We have used NVIDIA Nsight Compute (Version: 2021.1.1.0), an interactive kernel profiler for CUDA applications, to profile the kernel executions and gather the details of performance metrics for analysis.

The problem size chosen for multiplication is 8192x8192x8192. The input data is of type FP16, and the accumulator/output is of type FP32.

4.2 Thread Block Level Tiling Analysis

For analyzing the impact of thread block tiling on the performance, we have divided the MxNxK matrix multiplication into multiples blocks/tiles of T_mxT_nxT_k , as shown in Figure 4.1. Out

Microarchitecture	Intel Skylake (64-bit)
Processors	2-socket Intel Xeon Silver 4110
Clock	2.10 GHz
Cores	16 (8 per socket)
Private caches	64 KB L1 cache, 1024 KB L2 cache
Shared cache	11,264 KB L3 cache
Memory	256 GB DDR4 (2.4 GHz)
Microarchitecture (GPU)	NVIDIA Turing
GPU	NVIDIA GeForce RTX 2080 Ti
Multiprocessors (SMs)	68
CUDA cores (SPs)	4352
GPU Base Clock	1350 Mhz
L1 cache/shared memory	96 KB
L2 cache size	5632 KB
Memory size	11.26 GB GDDR6
Memory bandwidth	616 GB/s
Tensor Cores	544
Register File Size/SM	256 KB
OS Linux kernel	4.18.0 (CentOS 8)
Compiler GNU C/C++ (gcc/g++)	8.3.1
CUDA version	10.2
NVCC version	10.2.89
Nvidia driver version	440.33.01
cuBLAS version	10.2
Peak performance (system)	56.9 TFLOPS
cuBLAS performance	55.2 TFLOPS

Table 4.1: System details of CPU and GPU architecture, compilers and library versions, peak performance of the system, and benchmark performance of cuBLAS library.

Parameter \Tile Size - Perf (in TFLOPS)	**32x32x64 - 14.9 (Worst perf)**	**64x256x32 - 34.3 (Average perf)**	**128x128x64 - 48.9 (Best perf)**
Utilization (%) - LSU functional unit	23.6	17.15	21.88
Utilization (%) - Tensor (FP) functional unit	7.84	18.14	21.7
Bank conflicts	343,989,534	58,158,461	7,506
L1 cache - hit rate	0.27	0.83	0.95
Stall barrier (in cycles)	10.7	0.76	0.5

Table 4.2: Thread-block level tiling performance analysis for problem size = 8192x8192x8192.

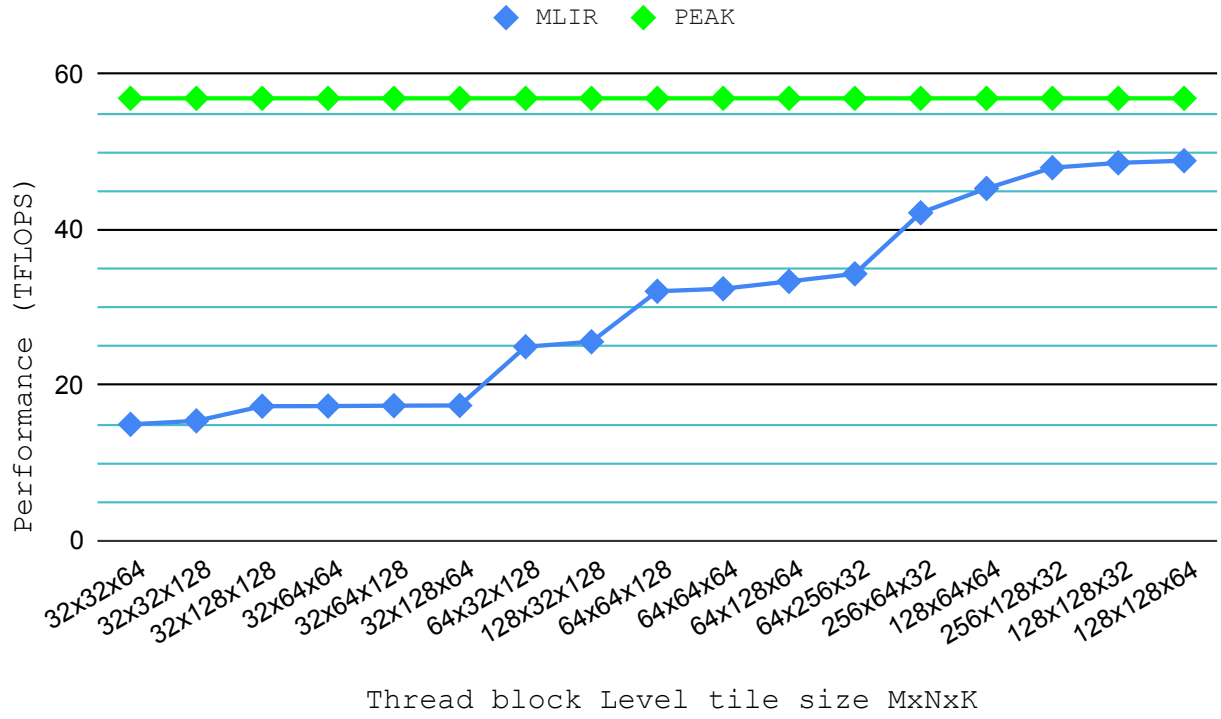

Figure 4.1: Performance of different thread-block level tile sizes. Problem size MxNXK = 8192x8192x8192.

of these many different configurations, we have chosen the following three sizes for analysis: T1 - 32x32x64, T2 - 64x256x32, and T3 - 128x128x64. As a result, T1 - gives the worst performance, i.e., 14.9 TFLOPS, T2 - gives an average performance, i.e., 34.3 TFLOPS, and T3 - gives a near-optimal performance, i.e., 48.9 TFLOPS. Furthermore, we have analyzed all three cases (tile sizes) based on some parameters: Load/Store functional unit utilization, Tensor (Floating Point) functional unit utilization. Bank conflicts in shared memory, L1 cache hit rate, and barrier stall cycles. The barriers are inserted in the code to ensure that all the threads finish copying data and computation for a given iteration before the next iteration begins. Since some threads take more time to complete than others; as a result, remaining threads have to wait at the barrier for completion, and the amount of time spent in waiting is known as stall barrier cycles. The comparison of T1, T2, T3 based on these parameters is given in Table 4.2.

The key objective of the tiling is to maximize the ratio of compute operations to memory operations, or, in other words, maximizing the ratio of utilization of the Tensor (FP) functional unit and Load/Store functional unit. For example, from Table 4.2, we can see that for T1, the ratio is close to 0.3, which is quite low, which means a large amount of time is spent on

data movement compared to the computations, which results in a bad performance, i.e., 14.9 TFLOPS. On the other hand, for T2 and T3, the ratio is close to 1, which means that both the data movement and computation operations take place in an equal amount during the kernel execution, which results in a better performance compared to T1. But the overall utilization of both the units is comparatively lesser for T2 than T3; thus, T2's performance is only 34.3 TFLOPS, and T3 gives 48.9 TFLOPS.

In general, for reducing the bank conflicts in shared memory, padding is performed for aligning the data so that the conflicts are reduced. But the padding alone can't reduce the bank conflicts. The combination of appropriate padding factor and tile size is required to minimize the shared memory bank conflicts. From the table, we can see that since the bank conflicts for T1 are much much high which results in terrible performance for T1. While the bank conflicts reduced by a 7 fold for T2, the performance also increases. For T3, the bank conflicts are almost negligible compared to T1 and T2, and hence for T3, the performance is maximum.

The L1 cache also plays a role in the performance because the output/accumulator matrices are not placed in the shared memory. For them, the data movement happens in this way: Global memory $\rightarrow$ L2 cache $\rightarrow$ L1 cache $\rightarrow$ registers. Since the tiling helps in data reuse, the better the data reuse, the better the performance. From Table 4.2, we can see that size of T1 is smaller. As a result, the data reuse is less. Hence the hit rate for T1 is the least, and also, it performs worst. While the size of T2 and T3 is large as compared to T1 and thus, the hit rate for T2 and T3 is also much higher than T1. As a result of this, their performance is better as compared to T1.

For T1, the stall at the barrier is quite high because the memory operations and computations are highly unbalanced for T1, and, as a result, some threads spend more time copying data. In comparison, others spend less time doing computation, due to which a large amount of time is spent waiting on the barrier for all threads to complete. On the other hand, for T2 and T3, the amount of time spent on data copying and computations is nearly equal; thus, very little time (< 1 cycle) is wasted stalling at the barrier.

From this analysis, we can conclude that the larger the tile size (factoring in the shared memory capacity), better the performance.

4.3 Warp-Level Tiling Analysis

The thread block-level tiling or level-1 tiling is performed to decide how much work is done by a thread block. The warp level tiling, also known as register tiling, is the second tiling level performed to distribute a thread block tile over the appropriate number of warps. We

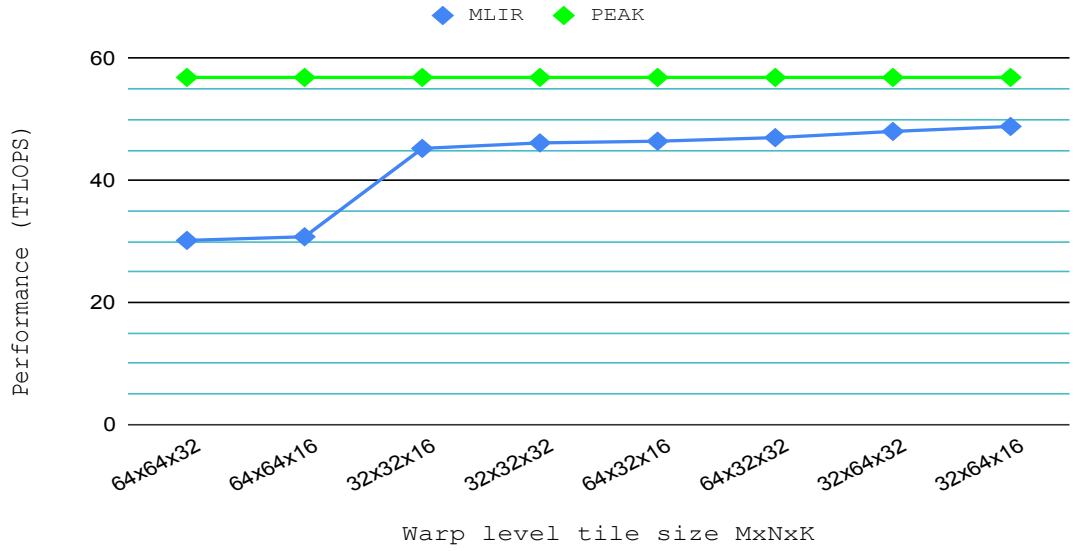

Figure 4.2: Warp level tile sizes performance for problem size = 8192x8192x8192 and thread-block level tile size = 128x128x64.

have chosen the thread block tile size 128x128x64 and divided it into eight different warp level tiles of sizes W_mxW_nxW_k for analyzing the warp level tiling shown in Figure 4.2. Out of these eight configurations, we have chosen the following two for analysis: W1 - 64x64x32 and W2 32x64x16. W1 gives the worst performance - 30.2 TFLOPS, W2 gives the best performance - 48.9 TFLOPS.

From Table 4.3, we can see that the ratio of compute operations to memory operations is close to 3/4 for W1 while it is close to 1 for W2. And, the overall utilization of the Tensor (FP) unit for W2 is high compared to W1. As a result, the performance achieved for W2 is 1.6 times more than the performance for W1. The observation that we made in thread-block level tiling was the larger the tile size (constrained by shared memory size), the better the performance; interestingly, that's not the case for warp-level tiling.

We can see from Table 4.3 that the number of bank conflicts for W1 is large as compared to W2, which reflects in the performance as well because while moving the data from shared memory into registers, the warp level tile size plays a key role; hence the padding along with appropriate warp level tile size is must for reducing the bank conflicts. We have discussed in Section 4.2 how the thread-block level tiling impacts the L1 cache hit rate. From Table 4.3, we can see that the warp level tiling does not have a big impact on the L1 cache hit rate. Although there's a significant difference in the performance of W1 and W2, the cache hit rate for W1 is slightly less than W2.

The number of registers available also limits the number of warps/threads that can be

Parameter \Tile Size - Perf (in TFLOPS)	**64x64x32 - 30.2 (Worst perf)**	**32x64x16 - 48.9 (Best perf)**
Utilization (%) - LSU functional unit	16.38	21.92
Utilization (%) - Tensor (FP) functional unit	13.49	21.74
Bank conflicts	2,052,360	8,597
L1 cache - hit rate	0.79	0.95
Register usage per thread	255	210
Number of threads	524,288	1,048,576

Table 4.3: Warp level tiling performance analysis for problem size = 8192x8192x8192. Thread-block level tile size chosen = 128x128x64.

issued/executed. We can see from Table 4.3 that the register usage per thread for W1 is high as compared to W2. As a result, fewer threads were executed for W1, and as a result, the W1's performance is less than W2.

Chapter 5

Tile-Size Selection Model

This section provides the tile-size selection model for thread-block level tiling and rules to be followed while choosing the warp level tile size.

5.1 Background

In Section 3, we have discussed various optimizations and transformations such as loop tiling, loop vectorization, loop unrolling, and shared memory padding. Each optimization has its own significance in improving the performance of automatic kernel generated for matrix multiplication. This thesis focused on the performance impact of tiling and the model for tile size selection. The performance impact of loop tiling on the automatic code generation is discussed in detail in Section 4.

Loop tiling, important from parallelism and locality viewpoint, is a widely used loop transformation. It is performed for better data reuse at a specific level of the memory hierarchy (such as L1/L2 cache or registers), enhanced data locality, and maximizing the ratio of computation operations to memory operations. Loop tiling is also known as blocking since the computation is executed block after block after performing the loop tiling. We can perform multiple levels of tiling by further tiling the tile or block obtained from the previous level of tiling. All state-of-the-art implementations of GEMM perform two-level tiling. We have also performed two-level tiling, namely thread-block level tiling (level 1) and warp-level tiling or register tiling (level 2). The thread-block level tiling is performed to distribute the amount of work done by a thread block, while the warp level tiling is performed to distribute the thread block tile over the appropriate number of warps.

For tiling the loops, we need to know the value of a parameter known as tile size, which specifies the size of the newly formed tile after performing the tiling. From Sections 4.2 and 4.3,

we can see how crucial is the selection of appropriate tile size for the performance. The selection of the best tile size for the problem we are solving and the given architecture is an NP-hard problem known as the Optimal Tile Size Selection Problem. Many works [32, 33, 34, 35, 36] had been done in this area for addressing this problem. Abid M. Malik [36] in their work proposed a tile size selection model using machine learning which predicts the tile sizes. The predicted tile sizes give the near-optimal performance (within 4% of the optimal performance). Khaled Abdelaal et al. [34] in their work proposed a solution for the tile size selection problem using polyhedral cross-compilation, which generates only two tile sizes achieving strong performance. We solve the optimal tile size selection problem by the model proposed in this section for thread-block level and warp level tiling.

5.2 Thread-Block Level Tile Size Selection Model

In Section 5.1, we have discussed the significance of loop tiling and the hardness of the optimal tile size selection problem. Also, we have mentioned in that section some of the models/solutions for the optimal tile size selection problem. In this section, we propose our analytical model for thread-block level tile size selection.

We can see from Section 4.2 that choosing a larger thread-block level tile size gives better data reuse and better performance. The rationale behind a larger thread-block tile performing better is that we want to utilize the shared memory maximally, and the larger tile size is suitable for that. But it is not the case that any large tile size will always perform better. Based on our analysis, we have found some constraints explained below. The main constraint is that the total amount of data residing in the blocks/tiles should not exceed the maximum amount of shared memory allowed to be used. Considering the points based on our analysis, we have derived a model, which is given below.

Let T_m, T_n, and T_k be the thread-block level tile size for dimensions m, n, and k, respectively. S_{Data} is the size of the input matrix element (in bytes), and S_{Mem} is the amount of shared memory capacity (in KB). Then, the tile-size selection model for thread-block level tiling is as follows:

$$T_m = T_n \tag{5.1}$$

$$T_m, T_n \geq 2 * T_k \tag{5.2}$$

$$\{(T_m * T_k) + (T_n * T_k)\} * S_{Data} \leq S_{Mem}. \tag{5.3}$$

By substituting the value of S_{Data} and S_{Mem} and solving these equations, we can find either the optimal tile size or a small set of configurations of tile sizes, out of which one will give the optimal performance. We can check the correctness of this model for the problem size we chose for the experimentation. When we substitute the $S_{Data} = 2$ bytes and $S_{Mem} = 48$ KB, it yields 128x128x64, which gives the best performance, i.e. 48.9 TFLOPS (as discussed in Section 4.2).

As we say that the model is derived based on the analysis that we have done, so the important points which we get from our analysis and on which this tile size selection model is based are as follows:

- The tile size for m, n dimension should be equal so that the data reuse across the dimensions is uniform.
- The tile size for m, n dimension should be greater than the k dimension and, since we are only choosing multiples of two as tile sizes, the value for the m and n dimension should be at least two times the value of the k dimension.
- The shared memory is limited, and during the tile size selection, we should consider this point and choose the tile sizes so that the tiles can fit inside the shared memory.

5.3 Warp Level Tile Size Selection Model

The warp level tiling, also known as register tiling, distributes the thread block tile over the appropriate number of warps. It is also performed for better register reuse. We have a model (discussed in Section 5.2) for thread-block level tiling in which values can be substituted, and we will get either a single tile size or a small set of tile sizes. For the warp-level tiling, based on our analysis in Section 4.3, we have derived some rules which, on being followed, give us a small set of tile sizes out of which one will give us the best performance.

Let W_m, W_n, and W_k be the warp level tile size for dimensions m,n, and k, respectively. S_{Warp} is the size of the warp (depends on the hardware), whose value is 32. Then, the rules that need to be followed for warp level tile size selection are as follows:

- We should choose the warp level tile size such that the thread-block level tile size should be at least twice of warp level tile size for each of the dimensions.
- The warp level tile size has to be greater than or equal to 16x16x16 because this is the smallest size supported by the WMMA API.

- The warp level tile should be such that the result of $(T_m/W_m) * (T_n/W_n) * S_{warp}$ should be less than or equal to 1024 because this is equivalent to the number of threads per block and the limit for this is 1024.

Following the above-mentioned rules, we get a small set of warp level tile sizes out of which one gives us the best performance.

5.4 Limitations

Some of the limitations of the tile size selection model proposed by us are as follows:

- The model proposed by us does not perform well for smaller problem sizes.
- The model is tested only for square size matrices multiplication, and it may not work well for non-square matrices multiplication.

This section discussed loop tiling, its significance, and the optimal tile size selection problem. Then in Sections 5.2 and 5.3, we have proposed the tile size selection model/rules for thread-block level tiling and warp level tiling, respectively. We have also mentioned the limitations of the model proposed by us in Section 5.4. The tile size selection model proposed yields us the tile size giving the best performance. The tile size model works well for the NVIDIA Turing and the Ampere (latest) architecture. The tile size selection model depends on the shared memory capacity, the size of the data type used, and the warp size. The tile size selection model does not depend on other hardware properties making it robust. We hope that the robustness of the tile size selection model will continue and make it work on the future architectures as it is or by slightly modifying it.

Chapter 6

Conclusions

We have tackled the problem of automatic and efficient code/kernel generation for matrix multiplication for GPU Tenosr cores using MLIR. We can conclude that our solution provides more flexibility to compiler developers, library developers, and hardware vendors. It uses the affine dialect of MLIR based on polyhedral transformation techniques for loop optimizations. The solution is front-end independent, i.e., it can be used along with any high-level programming language. We have shown with the experiments that our solution achieves performance comparable to hand-tuned libraries such as cuBLAS and CUTLASS for NVIDIA Turing and Ampere GPU architectures. In some sense, our work, based on MLIR, is modular and reusable to be used in several compilation frameworks that require a high-performance code generation path for GPUs.

As a part of this thesis, we have also proposed a tile-size selection model for thread-block level tiling and warp level tiling. The model proposed by us yields us either one tile size, which gives us the best performance, or it yields a small set of tile sizes, out of which one which gives us the best performance can be found out by the experimentation. The model proposed works well only for large problem sizes. As a part of future work, the model can be enhanced to become more robust and work well for all problem sizes.

In Section 3, we have discussed various optimization such as loop tiling, vectorization, memory padding, and loop unrolling. The padding factor required for performing padding is at present determined by experimentation. In the future, we can also work on deriving a model which can provide us the optimal padding factor just like the tile size selection model provides us the optimal tile size. Also as a part of future work, we can derive an analytical model for the loop unrolling factor.

The solution that we provide for the automatic kernel generation for matrix multiplication is restricted only to the GPUs comprising Tensor cores. In the future, this work can be ex-

tended to make this solution work for the GPUs not having tensor cores and also for the GPUs made available by hardware vendors other than NVIDIA. Our solution tackles the problem of automatic code generation only for matrix multiplication, and it can be further extended for problems such as automatic generation of fused kernels for matrix multiplication plus pointwise operations such as bias addition, ReLU activation, and convolution.

Bibliography

[1] Alex Zinenko, Nicolas Vasilache, Stephan Herhut, Mahesh Ravishankar, Geoffrey Martin-Noble. Codegen Dialect Overview.. https://llvm.discourse.group/t/codegen-dialect-overview/2723. vi, 6

[2] NVIDIA Corporation, CUDA C Programming Guide, 2011. vi, 10

[3] NVIDIA Corporation, Turing architecture, https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf. 1

[4] NVIDIA Corporation, Ampere architecture, https://www.nvidia.com/en-in/data-center/ampere-architecture/. 1

[5] Jeremy Appleyard and Scott Yokim. "Programming Tensor Cores in CUDA 9," 2017, https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/. 1, 11

[6] V. Mehta, "Getting started with Tensor Cores in HPC," 2019, NVIDIA GPU Technology Conference. 1, 11

[7] Ahmad Abdelfattah, Stanimire Tomov, and Jack Dongarra. Towards Half-Precision Computation for Complex Matrices: A Case Study for Mixed Precision Solvers on GPUs. In *2019 IEEE/ACM 10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, pages 17–24, 2019. 1

[8] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J. Higham. Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 603–613, 2018. 1

[9] Tsuyoshi Ichimura, Kohei Fujita, Takuma Yamaguchi, Akira Naruse, Jack C. Wells, Thomas C. Schulthess, Tjerk P. Straatsma, Christopher J. Zimmer, Maxime Martinasso,

Kengo Nakajima, Muneo Hori, and Lalith Maddegedara. A Fast Scalable Implicit Solver for Nonlinear Time-Evolution Earthquake City Problem on Low-Ordered Unstructured Finite Elements with Artificial Intelligence and Transprecision Computing. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 627–637, 2018. 1

[10] Vishal Mehta. "Getting started with Tensor Cores in HPC," 2019, NVIDIA GPU Technology Conference. 1

[11] NVIDIA Corporation, "cuBLAS" 2019, https://docs.nvidia.com/cuda/cublas/index.html. 1, 3

[12] NVIDIA Corporation, "cuDNN", https://docs.nvidia.com/deeplearning/cudnn/index.html. 1, 3, 11

[13] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques A. Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A Compiler Infrastructure for the End of Moore's Law. *CoRR*, abs/2002.11054, 2020. 2, 4, 5

[14] Chris Lattner and Jacques Pienaar. MLIR Primer: A Compiler Infrastructure for the End of Moore's Law, *Compilers for Machine Learning Workshop, CGO 2019*. 2, 4, 5

[15] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021. 2, 4, 5

[16] NVIDIA Corporation, CUB, https://docs.nvidia.com/cuda/cub/index.html. 3

[17] NVIDIA Corporation, "CUDA templates for linear algebra subroutines," 2019, vol. 21, no. 5, pp. 313–348, 1992. https://github.com/NVIDIA/cutlass. 3, 11

[18] NVIDIA Corporation, CUTLASS performance, https://github.com/NVIDIA/cutlass#performance. 3

[19] NVIDIA Corporation, "cuTENSOR: A high-performance CUDA library for Tensor primitives," 2019, https://docs.nvidia.com/cuda/cutensor/index.html. 3, 11

[20] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *SIGPLAN Not.*, 48(6):519–530, June 2013. 3

[21] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions, *Facebook AI Research Technical Report*. February 13, 2018. 3

[22] Somashekaracharya G. Bhaskaracharya, Julien Demouth, and Vinod Grover. Automatic Kernel Generation for Volta Tensor Cores. *CoRR*, abs/2006.12645, 2020. 3

[23] Thomas Faingnaert, Tim Besard, and B. D. Sutter. Flexible Performant GEMM Kernels on GPUs. *ArXiv*, abs/2009.12263, 2020. 3, 4

[24] LLVM Foundation, The LLVM Compiler Infrastructure. https://llvm.org/. 4

[25] The Julia language, 2021. https://julialang.org/. 4

[26] John Nickolls and William J. Dally. The GPU Computing Era. *IEEE Micro*, 30(2):56–69, 2010. 9

[27] NVIDIA Corporation, CUDA C++ programming guide. https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. 9

[28] NVIDIA Corporation, Volta architecture, https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. 11

[29] NVIDIA Corporation, "CUDA Toolkit Documentation," 2019, https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#warp-level-matrix-fragment-mma-884. 11

[30] Google Brain Team, Tensorflow/MLIR-HLO, https://github.com/tensorflow/mlir-hlo. 12

[31] Navdeep Kumar Katel, M.Tech. Research Thesis. Automatic Code Generation for GPU Tensor Cores using MLIR. In preparation, to be submitted in July 2021. 15, 19

[32] Jacqueline Chame and Sungdo Moon. A tile selection algorithm for data locality and cache interference. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, page 492–499, New York, NY, USA, 1999. Association for Computing Machinery. 28

[33] Jun Shirako, Kamal Sharma, Naznin Fauzia, Louis-Noël Pouchet, J. Ramanujam, P. Sadayappan, and Vivek Sarkar. Analytical bounds for optimal tile size selection. In Michael O'Boyle, editor, *Compiler Construction*, pages 101–121, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. 28

[34] Khaled Abdelaal and Martin Kong. Tile Size Selection of Affine Programs for GPGPUs Using Polyhedral Cross-Compilation. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '21, page 13–26, New York, NY, USA, 2021. Association for Computing Machinery. 28

[35] Nirmal Prajapati, Waruna Ranasinghe, Sanjay Rajopadhye, Rumen Andonov, Hristo Djidjev, and Tobias Grosser. Simple, Accurate, Analytical Time Modeling and Optimal Tile Size Selection for GPGPU Stencils. *SIGPLAN Not.*, 52(8):163–177, January 2017. 28

[36] Abid M. Malik. Optimal Tile Size Selection Problem Using Machine Learning. In *2012 11th International Conference on Machine Learning and Applications*, volume 2, pages 275–280, 2012. 28