FACULTY OF ENGINEERING AND ARCHITECTURE

Flexible matrix multiplication kernels on GPUs

Thomas Faingnaert
Student number: 01506418

Supervisor: Prof. dr. ir. Bjorn De Sutter
Counsellor: Dr. Tim Besard

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Academic year 2019-2020

GHENT UNIVERSITY

FACULTY OF ENGINEERING AND ARCHITECTURE

Flexible matrix multiplication kernels on GPUs

Thomas Faingnaert
Student number: 01506418

Supervisor: Prof. dr. ir. Bjorn De Sutter
Counsellor: Dr. Tim Besard

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Academic year 2019-2020

GHENT UNIVERSITY

Permission of use on loan

The author gives permission to make this master dissertation available for consultation and to copy parts of this master dissertation for personal use. In all cases of other use, the copyright terms have to be respected, in particular with regard to the obligation to state explicitly the source when quoting results from this master dissertation.

1st July 2020

Preface

Before you lies my master thesis "Flexible matrix multiplication kernels on GPUs", the culmination of my studies Computer Science Engineering at Ghent University. This thesis is the result of my 9 month long journey through the LLVM compiler framework, GPU programming, the Julia programming language, and Julia's GPU ecosystem. I have always been interested in compilers, and this thesis was an excellent opportunity to take this interest further. In particular, I enjoyed gaining experience with the widely used LLVM compiler framework. Additionally, I got the chance to explore GPGPU computing, a world which was new to me, using NVIDIA's CUDA platform.

First and foremost, I would like to thank my supervisors Prof. Bjorn De Sutter and Dr. Tim Besard for their advice and guidance. Secondly, I want to thank the Julia community, in particular Valentin Churavy and Jameson Nash, for reviewing and merging my pull requests. Next, I want to extend my sincere gratitude to my parents, for their willingness to proofread drafts of this thesis, and for their unwavering support during my studies. Finally, I would like to thank you, the reader, for taking the time to read my thesis. I hope you enjoy reading it as much as I enjoyed writing it.

Thomas Faingnaert

1st July 2020

Flexible matrix multiplication kernels on GPUs

Thomas Faingnaert
Student number: 01506418

Supervisor: Prof. dr. ir. Bjorn De Sutter
Counsellor: Dr. Tim Besard

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering
Faculty of Engineering and Architecture, Ghent University
Academic Year 2019 – 2020

Abstract

GEMM (General Matrix Multiplication) kernels are at the core of many computations in the fields of HPC (High Performance Computing) and ML (Machine Learning). GEMM is so prevalent that NVIDIA's latest GPUs (Graphics Processing Units) include Tensor Cores, a special type of processing cores that are designed to accelerate matrix multiplications. As the fields of HPC and ML evolve, we notice two trends. First, the low-level programming language C++ that is traditionally used for high-performance applications, is being replaced by higher level languages such as Python or Julia. Through the use of the Julia package CUDAnative, one can even program GPUs directly in Julia. The second trend is the increasing need for flexibility in GEMM kernels. State-of-the-art GEMM libraries typically contain a limited set of monolithic kernels, and thus lack flexibility.

In this thesis, we will design, implement, and evaluate a GEMM framework for Julia that allows users to create GEMM kernels that are tailored to their use case. Given that NVIDIA's Tensor Cores are already extensively used to accelerate ML and HPC workloads, we will mainly target GEMM using Tensor Cores. Our framework consists of three different APIs (Application Programming Interfaces). The first API provides an interface to access Tensor Cores from within Julia. The second API facilitates writing algorithms that use tiling techniques to improve performance, such as GEMM kernels. The final API consists of a set of customisable components that can be combined to implement a GEMM kernel for a specific purpose.

Keywords — GPU, Julia, Tensor Cores, Flexible GEMM

Flexible matrix multiplication kernels on GPUs

Thomas Faingnaert

Supervisor: Prof. dr. ir. Bjorn De Sutter
Counsellor: Dr. Tim Besard

***Abstract*—GEMM (General Matrix Multiplication) kernels are at the core of many computations in the fields of HPC (High Performance Computing) and ML (Machine Learning). GEMM is so prevalent that NVIDIA's latest GPUs (Graphics Processing Units) include Tensor Cores, a special type of processing cores that are designed to accelerate matrix multiplications. As the fields of HPC and ML evolve, we notice two trends. First, the low-level programming language C++ that is traditionally used for high-performance applications, is being replaced by higher level languages such as Python or Julia. Through the use of the Julia package CUDANATIVE, one can even program GPUs directly in Julia. The second trend is the increasing need for flexibility in GEMM kernels. State-of-the-art GEMM libraries typically contain a limited set of monolithic kernels, and thus lack flexibility.**

In this thesis, we will design, implement, and evaluate a GEMM framework for Julia that allows users to create GEMM kernels that are tailored to their use case. Given that NVIDIA's Tensor Cores are already extensively used to accelerate ML and HPC workloads, we will mainly target GEMM using Tensor Cores. Our framework consists of three different APIs. The first API provides an interface to access Tensor Cores from within Julia. The second API facilitates writing algorithms that use tiling techniques to improve performance, such as GEMM kernels. The final API consists of a set of customisable components that can be combined to implement a GEMM kernel for a specific purpose.

***Index Terms*—GPU, Julia, Tensor Cores, Flexible GEMM**

I. Introduction

Scientific computing applications typically rely on GPUs instead of CPUs, as the computational capabilities of the former are significantly larger than those of the latter. Traditionally, GPUs are programmed using low-level languages such as C++. As the field of scientific computing evolves, these low-level languages are being replaced with alternative languages with a higher-level syntax, but that are still able to match the performance of C++. The Julia programming language is an important example of a language with a high-level syntax, but a performance comparable to that of C++ [5]. Using the package CUDANATIVE, it is possible to program NVIDIA GPUs directly in Julia [4].

Matrix multiplication, commonly called GEMM (General Matrix Multiplication), is at the core of many computations in scientific computing. GEMM is used for neural networks in the field of ML (Machine Learning), and in several HPC (High Performance Computing) applications. Matrix multiplication is so prevalent that the latest NVIDIA GPUs now include Tensor Cores, a type of processing cores that accelerate matrix multiplications.

GPU vendors provide highly optimised implementations of GEMM in libraries, such as NVIDIA's CUBLAS. These libraries contain a set of GEMM kernels designed for a specific purpose, and hence lack flexibility. This lack of flexibility is problematic if the needs of the application are not addressed by one of the kernels contained in the library. For example, inference in neural networks can be computed using a matrix multiplication, followed by the elementwise application of the activation function of the artificial neuron. Deep learning libraries typically only support a limited set of the most commonly used activation functions. This means that ML researchers cannot easily experiment with new activation functions without writing a performant GEMM kernel from scratch.

Another example is tensor contraction, the generalisation of matrix multiplication to multiple dimensions. One approach for performant tensor contractions, GETT (GEMM-like Tensor-Tensor contraction), builds on top of a performant GEMM kernel [16]. GETT reshuffles the input tensors, so that the tensor contraction is equivalent to a matrix multiplication. This reshuffling can be performed before launching a prebuilt GEMM kernel, but this introduces unnecessary overhead. This overhead can be avoided if the reshuffling is fused in the GEMM kernel instead. Of course, this is only possible if the underlying GEMM kernel is flexible.

Over the course of this thesis, we will design and implement a framework to instantiate customisable GEMM kernels on NVIDIA GPUs. We will use the high-level Julia programming language, and its package CUDANATIVE. Given their use in various ML and HPC workloads, we will mainly focus on GEMMs using NVIDIA's Tensor Cores.

We will start with an introduction to the necessary background information in Section II. This section discusses GPU programming, the Julia programming language, CUDANATIVE, and Tensor Cores. Our framework consists of three different APIs that interact. We will discuss each of these in a separate section. Section III describes the design of an API that allows us to use Tensor Cores from Julia. Next, Section IV discusses an API that facilitates writing algorithms that use tiling techniques to improve performance, such as matrix multiplications. The final API consists of a set of customisable components that together implement a performant GEMM kernel, and will be the subject of Section V. We will demonstrate and evaluate this framework using three different examples in Section VI. Finally, Section VII concludes the paper.

II. Background

This section discusses the relevant aspects of GPU programming, the Julia programming language, the Julia package CUDAnative, and Tensor Cores.

A. GPU programming

The main difference between programming GPUs versus CPUs is that the underlying programming model is different. GPUs are massively parallel processors, meaning that a large number of threads execute the same function in parallel. In GPU parlance, this function is commonly referred to as a *kernel*.

GPU threads are organised in a thread hierarchy [12]. Since our main interest is in NVIDIA GPUs, we will limit our discussion to NVIDIA's CUDA programming model. In CUDA, the thread hierarchy consists of:

- *Threads*: Threads are the smallest unit of execution in the hierarchy.
- *Warps*: Threads are grouped by the hardware into a set of 32 threads called a warp. Threads in the same warp execute in a SIMT (Single Instruction Multiple Thread) fashion. This means that these threads must execute the same instruction at the same time, possibly on different data.
- *Blocks*: Threads are grouped by the programmer into blocks. Threads in the same block can communicate efficiently, so that they can cooperate on a common task.
- *Grid*: The set of all blocks on the GPU device is called the grid.

Similarly to threads, GPU memory is also ordered hierarchically. We are mainly interested in three parts of this hierarchy, corresponding to the levels in the thread hierarchy:

- *Registers*: The register file is the fastest type of memory. Each thread typically has access to 255 registers.
- *Shared memory*: Each block has its own set of shared memory, that may be used by threads in the same block to communicate.
- *Global memory*: Global memory can be accessed by all threads on the device, regardless of which block they belong to. Global memory has the largest capacity of the memory hierarchy, but also has much higher latency and lower throughput.

B. The Julia programming language

Julia is an open source programming language featuring a high-level syntax. A central paradigm in the design of the language is the way it handles dispatch, the process by which the compiler determines which implementation of a function to use for a given function call. Julia uses a *multiple dispatch* scheme, which means that this choice depends on the types of *all* of a function's arguments [6].

Like most high-level languages, Julia's type system is *dynamic*, meaning that the types of expressions are not necessarily known at compile time. However, Julia also inherits some of the advantages of static type systems through several features of its compiler. The Julia compiler applies type inference to deduce the types of values used by the program. If the compiler can deduce the types of all of the arguments in a function call, then this function can be *specialised* for these types. This specialised function is then compiled *just-in-time* to an efficient implementation, devoid of any dynamic type checks.

Julia's compiler is built on top of LLVM, a compiler infrastructure project commonly used in research and industry [9]. Julia's compilation process consists of a couple steps. First, Julia code is converted to an IR (Intermediate Representation) that is used for type inference, called Julia IR. Next, Julia IR is lowered to LLVM IR, the representation that LLVM uses. From this point onwards, the LLVM framework takes control of the compilation process. LLVM contains a set of backends, one for each target architecture that LLVM supports. The backend corresponding to the current architecture will then convert this LLVM IR to native instructions.

C. Programming GPUs in Julia using CUDAnative

CUDAnative is a Julia package that allows executing kernels written in Julia on NVIDIA GPUs. It reuses part of the Julia compilation process that we explained in the previous section. In particular, the compilation pipeline is run until the point where Julia IR is lowered to LLVM IR. The generated LLVM IR is intercepted, and sent to the LLVM NVPTX backend instead of the backend corresponding to the host architecture. This NVPTX backend converts LLVM IR to PTX instructions, the virtual instruction set of NVIDIA GPUs. NVIDIA GPUs are not capable of executing this PTX directly. PTX is first compiled by the GPU driver to SASS, the generation-specific instruction set that the GPU understands.

D. Tensor Cores

Each Tensor Core performs a matrix multiply-accumulate, i.e. an expression of the form $D = A \cdot B + C$. Tensor Cores only support a limited set of possible data types for these matrices. One peculiarity of these Tensor Cores is that the multiply-accumulate is performed in mixed precision. For example, if the A and B matrices are stored as 16-bit floating point values, the C and D matrices are 32-bit floating point.

NVIDIA exposes Tensor Cores in C++ in an API that they call WMMA (Warp Matrix Multiply Accumulate). As the name suggests, WMMA instructions must be used by all threads in a warp, in a SIMT fashion. Each thread that cooperates in a warp-wide WMMA operation holds a part of each matrix in its registers. This part is referred to as a *fragment* in WMMA terminology.

In WMMA parlance, we say that A is an $M \times K$ matrix, B is a $K \times N$ matrix, and C and D are $M \times N$ matrices. The tuple (M, N, K) is called the *shape* of the WMMA operation. Note that not all possible values of M, N, and K are allowed, as WMMA restricts the set of possible shapes.

Conceptually, WMMA consists of three separate steps:

1) Load the input matrices A, B, and C from memory into WMMA fragments using a WMMA load operation.

2) Perform the matrix multiply-accumulate using a WMMA mma operation, resulting in a fragment of the D matrix.
3) Store the resultant D fragment to memory using a WMMA store operation.

In CUDA C++, each of these steps corresponds to an overloaded C++ function. Calls to these functions are converted to the correct WMMA PTX instructions by the CUDA C++ compiler.

III. Abstractions for programming Tensor Cores in Julia

To support Tensor Cores in Julia, our WMMA API thus needs to make sure that CUDANATIVE generates the correct PTX instructions. We can reuse some existing functionality in the LLVM NVPTX backend for this purpose. In the context of compilers, an *intrinsic* or *intrinsic function* is a function that is handled in a special way by the compiler. Backends in the LLVM framework can define intrinsics that are specific to that backend. The NVPTX backend already includes intrinsics for WMMA, which are converted to the corresponding PTX instructions.

Our Julia API for Tensor Cores consists of two different layers. The first layer is a set of Julia functions that wrap the pre-existing intrinsics in the NVPTX backend. These wrapper functions are one-to-one, meaning that each intrinsic corresponds to one Julia function.

The Julia compiler already includes an `llvmcall` function, that allows programmers to call LLVM intrinsics directly from Julia. The compiler first determines the Julia type of each of the arguments of `llvmcall`, converts these to the corresponding LLVM type, and inserts a call to the correct intrinsic. This mapping of Julia types to LLVM types is hardcoded in the Julia compiler. To support WMMA in Julia, we had to adapt this mapping, as the WMMA intrinsics use some LLVM types that did not have a corresponding Julia equivalent. We bundled the necessary changes to Julia's code generation in one pull request, that has since been merged in the upstream Julia repository.

One possibility is to write these wrapper functions manually for each intrinsic, but that would be a very tedious process. Instead, we use Julia's powerful metaprogramming capabilities to generate these wrappers automatically. We define a limited set of variables that contain the information necessary to generate these wrappers, such as the possible shapes of WMMA operations. We then simply iterate over the possible configurations, and dynamically generate the corresponding wrapper functions.

The second layer is a high-level interface, similar to CUDA C++'s version of WMMA. Each of the steps of WMMA corresponds to a different Julia function. For example, the A matrix is loaded with a call to `load_a`, and the resultant D matrix is stored to memory using `store_d`. Note that the correct PTX instruction depends on the shape of the WMMA operation, and the datatype of the accumulator matrices C and D. This information is passed to the high-level interface using a `conf` argument, which is a type that contains both the WMMA shape and accumulator element type. We then use Julia's multiple dispatch mechanism to redirect calls to our high-level API to the correct intrinsic wrapper function.

Our WMMA API will serve as a building block to implement flexible GEMMs later on. One important operation that we must support for flexible GEMM is the application of elementwise transformations, such as linear scaling or activation functions in neural networks. To support elementwise operations, we integrate our WMMA API in Julia's broadcasting framework. This way, we may apply an elementwise operation `f` to a WMMA fragment `frag` using Julia's dot syntax: `f.(frag)`. The resulting high-level WMMA API and intrinsic wrappers were bundled in one pull request, that has since been merged in CUDANATIVE.

IV. Abstractions for recursive blocking

Matrix multiplication is an application that is rich in data reuse. A matrix multiplication of square matrices of size N requires $\mathcal{O}(N^3)$ floating point operations, but only $\mathcal{O}(N^2)$ storage. This results in each element being reused roughly $\mathcal{O}(N)$ times. This data reuse can be used to improve the performance of GEMM kernels. The general idea is to copy tiles of the input matrices from a slower type of memory to a faster type. This process is then repeated for every level in the memory hierarchy. The size of the tiles in each step is chosen such that they fit in the relevant part of the memory hierarchy.

On the GPU, we first copy a tile from global memory to shared memory. We can then reuse the data in shared memory, which is faster than global memory loads. In a next step, we can load smaller tiles from shared memory to the register file. The massively parallel nature of GPUs allows us to improve performance of GEMM even further. Note that the computations of different tiles of the resultant matrix are independent, and can thus be performed completely in parallel.

Recall that their is a one-to-one mapping between the levels of the thread and memory hierarchy. For example, shared memory is inherently linked to blocks, since only threads in the same block can communicate via shared memory. Each of the tiled copy operations is thus performed cooperatively, by all threads in the relevant part of the thread hierarchy. Consider the case of a GEMM $D = A \cdot B + C$, where A is an $M \times K$ matrix, B is a $K \times N$ matrix, and C and D are $M \times N$ matrices. More concretely, this GEMM will consist of the following steps:

1) Copy a tile of C from global memory to shared memory, cooperatively by all threads in a block.
2) Copy a tile of C from shared memory to registers, cooperatively by all threads in a warp.
3) Iterate over the K dimension, according to the tiling size of a block.
    a) Copy a tile of A from global memory to shared memory, cooperatively by all threads in a block.
    b) Copy a tile of B from global memory to shared memory, cooperatively by all threads in a block.

c) Iterate over the K dimension, according to the tiling size of a warp.
   i) Copy a tile of A from shared memory to registers, cooperatively by all threads in a warp.
   ii) Copy a tile of B from shared memory to registers, cooperatively by all threads in a warp.
   iii) Compute a tile of D, given the A, B, and C tiles, cooperatively by all threads in a warp.
4) Copy a tile of D from registers to shared memory, cooperatively by all threads in a warp.
5) Copy a tile of D from shared memory to global memory, cooperatively by all threads in a block.

Most GEMM implementations on NVIDIA GPUs use explicit tiling to improve performance [11, 14, 10, 2, 8, 3]. Apart from GEMM, tiling is also used for batched GEMMs or tensor contractions [1, 3, 13, 7]. Given the multitude of different applications, a tiling API could prove very useful.

To that end, we have developed a tiling API in Julia that facilitates writing algorithms that use tiling techniques to improve performance. The most important operation in our tiling API is *parallelisation*. The parallelisation operation first divides a tile in subtiles of a given size. The resulting set of subtiles is then parallelised across all blocks on the device. Given a set of N subtiles and M blocks, each block will handle $\frac{N}{M}$ subtiles. This parallelisation operation can be applied recursively. For example, the subtile of each block can be recursively parallelised across all the warps in the same block.

This parallelisation operation can be used to implement the general structure of a GPU GEMM that we described before. Our tiling API contains another operation that we call *linearisation*, that converts a tile to a linear offset in memory. Linearisation is used to calculate the memory address corresponding to each element in a tile. Our tiling API thus replaces the manual calculation of memory addresses, which is less maintainable, and more prone to errors.

The tiling API that we developed will be used as a building block to implement flexible matrix multiplication kernels in our GEMM API. Nevertheless, we have designed our tiling API to be as generally applicable as possible. For example, tiles in our API can have any number of dimensions, so that they can be used for tensor contractions as well.

V. Abstractions for flexible matrix multiplication kernels

The final API in our flexible GEMM framework is the GEMM API itself. It uses the tiling API we described in the previous section to implement the general structure of a performant GEMM. Our API splits this GEMM kernel in a set of building blocks having a predetermined interface. Each of these building blocks corresponds to a way in which GEMM kernels need to be adapted, and can have different implementation depending on the use case. For example, one building block could determine how the A matrix is stored in global memory. Specific implementations of this building block would include a column major, and a row major storage format.

Each of these building blocks corresponds to one or more calls to a set of functions with a predetermined name. For example, the building block that determines how A is stored can have two functions `load` and `store` that load and store a tile of A, respectively. The first argument to these functions is a type, such as `ColumnMajor` or `RowMajor`, that determines the memory layout of A. Using Julia's multiple dispatch mechanism, we can customise the behaviour of these functions depending on this type, thereby adding flexibility to our GEMM kernel.

Note that the introduction of flexibility using this approach has no performance impact at runtime. Through type inference, the Julia compiler is able to infer the types passed to the `load` or `store` functions. The same applies to all different building blocks, so that it is known at compile time which implementations of each function will be called. These implementations are then combined, and compiled just-in-time to a kernel tailored to a specific use case.

Of course, the most important question we should answer is which building blocks we need to cover a wide range of different use cases. To answer this question, we drew inspiration from two sources. First, we looked into NVIDIA's open source CUTLASS library, which contains a set of components for performant GEMM or GEMM-like kernels. Second, we conducted a literature search to get an idea of the most common ways in which GEMM kernels need to be adapted. From these two sources, we propose 5 different building blocks: params, layouts, transforms, operators, and epilogues.

The *params* building block contains a set of fields that determine the tiling sizes that should be used for each step of the GEMM kernel. For example, the `BLOCK_SHAPE` field determines the size of the tiles loaded and stored by each block. The params building block also contains information regarding the launch configuration of the kernel, such as the number of warps per block.

The tiling sizes in the params component are specified in logical coordinates, i.e. with a meaning specified by the user. To load and store tiles, we need to convert this logical index to a physical offset in memory. This conversion is handled by the *layout* building block. Possible implementations of this building block are `RowMajor` and `ColMajor`. To provide maximal flexibility, users can specify the memory layout separately for each matrix, and for each level of the memory hierarchy. For example, the A matrix may be stored differently in global and shared memory. This is especially useful for shared memory, as accessing shared memory efficiently typically requires more complicated memory layouts.

Transforms are arbitrary Julia functions that are automatically called after every load, and before every store. Their main use case is to apply elementwise transforms to the input or output matrices of a GEMM kernel. This can range from the case of a simple scaling for a normal GEMM, to the application of activation functions in the case of neural

networks.

Operators determine how the computation of the matrix product is performed in the inner loop. Possible implementations of this building block include a WMMA operator that performs the computation using Tensor Cores, or an implementation that uses the traditional floating point hardware instead.

The final component in our GEMM API is the *epilogue*. As the name suggests, the epilogue building block is situated at the very last stage of GEMM. It offers more control than the layouts over how tiles of the resultant matrix are copied from shared memory to global memory. For example, a custom epilogue could apply a reduction operation over the tiles stored in shared memory across all blocks.

VI. Evaluation

To evaluate the performance and flexibility of our framework, we will implement three different GEMM variants using our GEMM API. The first example is a normal mixed-precision GEMM that uses Tensor Cores via our WMMA API. Next, we introduce the necessary components to extend this GEMM to matrices of complex numbers. Our last example will change this complex GEMM so it also supports dual numbers.

We will compare the performance of our GEMM kernels with the state-of-the-art GEMMs in NVIDIA's cuBLAS and CUTLASS libraries. CUDAnative dispatches matrix multiplications to cuBLAS, but only for datatypes that are compatible with cuBLAS. For other datatypes, CUDAnative includes a generic kernel as a fallback. We will also compare our performance with CUDAnative's generic implementation.

A. Mixed-precision GEMM

To support a mixed-precision GEMM in our GEMM API, we will create an operator that builds on top of our WMMA API. This operator simply calls the correct WMMA function for each step in the GEMM's inner loop.

Since Julia is column major, we also implemented a `ColMajor` layout component. This layout is suitable for global memory, but leads to inefficient memory accesses for shared memory. On NVIDIA GPUs, shared memory is split into *memory banks*. Different memory banks can be accessed in parallel, but memory accesses to addresses that map to the same bank are serialised. This serialisation process is often referred to as a *bank conflict*.

The most common way to reduce these bank conflicts for a column major layout, is to add padding to every column. This changes the mapping of matrix elements to banks, thereby reducing the number of bank conflicts. To introduce padding in our GEMM API, we added another layout `PaddedLayout`. The way this layout works is slightly different than `ColMajor`. It does not specify the mapping of logical indices to physical offsets directly, but is a wrapper for another layout. For example, `PaddedLayout{ColMajor, 8}` is a column major layout, where each column is padded by 8 elements.

The epilogue we use for our mixed-precision GEMM simply copies the corresponding tile from shared memory to global memory. Finally, we can implement a custom transform to apply elementwise operations such as scaling to the input matrices or the result of the matrix multiplication.

Table I compares the peak performance of our GEMM kernel with the generic implementation in CUDAnative, and the state-of-the-art implementations in cuBLAS and CUTLASS. CUDAnative's generic kernel uses no tiling techniques, and is only able to achieve a peak performance of $0.37\,\mathrm{TFLOPS}$. The CUTLASS (WMMA) implementation uses the same parameters as our implementation: the tiling sizes are the same, and the matrix product in the inner loop is also computed using WMMA. We notice a perfect performance parity between this CUTLASS kernel and our implementation.

Both our implementation and CUTLASS (WMMA) achieve a performance of about 75% that of cuBLAS. To understand this gap, we have also benchmarked two other kernels in the CUTLASS library. These kernels also use Tensor Cores, but do so using `mma` instructions instead of the WMMA API. These `mma` instructions are specific to the GPU generation, and allow more direct control over Tensor Cores compared to the portable WMMA abstraction layer. The performance results of Table I were obtained using an NVIDIA RTX 2080 Ti, which is a GPU of the Turing-generation. The `mma.m16n8k8` instruction is optimised for Turing GPUs, and gets very close to cuBLAS's performance. We also benchmarked a mixed-precision GEMM using the `mma.m8n8k4` instruction, which is aimed at Volta GPUs, Turing's predecessor. As we can see, the performance of this Volta-style `mma` is even worse than WMMA, indicating that these `mma` instructions are highly generation specific.

Table I
A comparison of the peak performance of our mixed-precision GEMM kernels with the state-of-the-art (higher is better).

Implementation	Performance [TFLOPS]
Ours	35
CUDAnative generic	0.37
cuBLAS	46
CUTLASS (WMMA)	35
CUTLASS (mma.m8n8k4)	22
CUTLASS (mma.m16n8k8)	40

B. Mixed-precision complex GEMM

To implement a mixed-precision GEMM of complex numbers, we can reuse most of the components of our previous example. There are only two main differences between a normal mixed-precision GEMM, and a mixed-precision complex GEMM [2].

First, the WMMA multiply-accumulate operation in the inner loop is replaced by four WMMA operations: `A.real * B.real`, `A.real * B.imag`, `A.imag * B.real`, and `A.imag * B.imag`. In our GEMM API, this can be implemented using another operator that is based on the WMMA operator of the previous section, but performs four multiply-accumulates in the inner loop instead of one.

The second difference between normal GEMMs and complex GEMMs, is the memory layouts that are used. In global memory, complex matrices are stored in an interleaved layout, where the real and imaginary parts of the same element are stored contiguously in memory. To load the real and imaginary parts from shared memory into WMMA fragments, we need to use a split-complex layout. This layout stores the real and imaginary parts of the matrix separately. This is needed because WMMA implicitly assumes that elements in the same column of a column major matrix are stored at adjacent memory addresses. In our GEMM API, this can be done using two new layout components: `InterleavedComplex` and `SplitComplex`.

Table II shows the peak performance of our complex GEMM, CUDANATIVE's generic implementation, and CUTLASS. At the time of writing, CUTLASS only includes an `mma` variant of complex GEMM, so we cannot compare our implementation with CUTLASS's WMMA. Still, we were able to achieve about 50% of the performance of CUTLASS's `mma` kernel, without implementing any optimisations specific to complex GEMM. Once again, we see that our flexible kernel significantly outperforms the generic implementation in CUDANATIVE.

Note that the gap between our WMMA kernel and CUTLASS's `mma` is larger for complex GEMM than for normal mixed-precision GEMM. This is likely because complex GEMMs have higher arithmetic intensity: the multiplication of two complex numbers requires four real multiplications. This means that the importance of the compute stage of GEMM increases, so that replacing WMMA with `mma` is more important for complex GEMM than for normal GEMM.

Table II
A COMPARISON OF THE PEAK PERFORMANCE OF OUR COMPLEX MIXED-PRECISION GEMM WITH THE STATE-OF-THE-ART (HIGHER IS BETTER).

Implementation	Performance [TFLOPS]
Ours	22
CUDANATIVE generic	1.2
CUTLASS (mma.m16n8k8)	42.7

C. Mixed-precision dual GEMM

In our final example, we study the case of the multiplication of matrices containing dual numbers. Dual numbers extend the set of real numbers with a new element ε, similar to the imaginary unit i in the case of complex numbers. Addition and multiplication of dual numbers is similar to the complex case, with the only difference that $\varepsilon^2 = 0$, whereas $i^2 = -1$. An example application of dual numbers in scientific computing is automatic differentiation of functions [15].

Because of the similarity between complex numbers and dual numbers, most of the discussion of the previous example applies here as well. For example, we can simply reuse the split and interleaved layouts we developed in the previous subsection. The only difference with the complex case is the operator component. Instead of four WMMA multiplications, we only need to perform three: `A.real * B.real`, `A.real * B.dual`, and `A.dual * B.real`. The use cases of complex and dual numbers are thus an excellent illustration of the reusability of the components in our GEMM API.

Performance-wise, we observe similar behaviour as the complex GEMM in the previous subsection. This is not surprising, as the only difference between complex and dual GEMMs is the operator used in the inner loop. Note that neither CUBLAS nor CUTLASS include kernels that multiply matrices of dual numbers. Thus, multiplying dual matrices stored on the GPU in Julia will dispatch to CUDANATIVE's generic implementation, which is many orders of magnitude slower than peak device performance. This means that using our GEMM API already results in a significant speedup for dual matrices, even though we can still improve the performance of our complex and dual GEMMs.

VII. CONCLUSION

We designed, implemented, and evaluated a framework to instantiate flexible GEMM kernels on GPUs using the Julia programming language. We build on top of CUDANATIVE, a Julia package that compiles Julia code to the GPU's PTX instruction set using the LLVM NVPTX backend. Our framework consists of three separate APIs.

The first API allows Julia kernels to program Tensor Cores, a type of processing core in the latest NVIDIA GPUs that accelerate matrix computations. This API consists of two layers: a low-level layer that directly wraps the correct intrinsics in the LLVM NVPTX backend, and a high-level WMMA API that is similar to the way Tensor Cores are exposed in CUDA C++.

The second API is a tiling API, and aims to facilitate writing algorithms that make use of tiling techniques. These tiling techniques are used in GEMM kernels, batched GEMM kernels, and tensor contractions to improve performance.

Finally, our third API consists of a set of components that together implement a GEMM. These components can be customised by the user, thereby introducing the necessary flexibility in the GEMM kernel.

We have evaluated the performance and flexibility of our framework using three different examples: a normal mixed-precision GEMM, a complex GEMM, and a dual GEMM. Our kernels perform similarly to the state-of-the-art implementations in the CUTLASS library, provided that the same parameters are used. We also indicated how we could bridge the remaining performance gap with CUBLAS by using the generation-specific `mma` instructions instead of the portable WMMA interface.

REFERENCES

[1] A. Abdelfattah, S. Tomov and J. Dongarra. 'Fast Batched Matrix Multiplication for Small Sizes Using Half-Precision Arithmetic on GPUs'. In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2019, pp. 111–122.

[2] A. Abdelfattah, S. Tomov and J. 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)*. 2019, pp. 17–24.

[3] Ahmad Abdelfattah et al. 'Performance, Design, and Autotuning of Batched GEMM for GPUs'. In: *High Performance Computing*. Ed. by Julian M Kunkel, Pavan Balaji and Jack Dongarra. Cham: Springer International Publishing, 2016, pp. 21–38. ISBN: 978-3-319-41321-1.

[4] T. Besard, C. Foket and B. De Sutter. 'Effective Extensible Programming: Unleashing Julia on GPUs'. In: *IEEE Transactions on Parallel and Distributed Systems* 30.4 (2019), pp. 827–841.

[5] JuliaLang.org. *Julia Micro-Benchmarks*. 2020. URL: https://julialang.org/benchmarks.

[6] JuliaLang.org. *The Julia Language Official Documentation*. 2020. URL: https://docs.julialang.org/en/v1.

[7] Jinsung Kim et al. 'A Code Generator for High-Performance Tensor Contractions on GPUs'. In: *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*. CGO 2019. Washington, DC, USA: IEEE Press, 2019, pp. 85–95. ISBN: 9781728114361.

[8] J. Lai and A. Seznec. '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)*. 2013, pp. 1–10.

[9] LLVM contributors. *The LLVM Compiler Infrastructure Project*. 2020. URL: https://llvm.org.

[10] Rajib Nath, Stanimire Tomov and Jack Dongarra. 'An Improved MAGMA GEMM For Fermi Graphics Processing Units'. In: *International Journal of High Performance Computing Applications* 24.4 (Nov. 2010), pp. 511–515. ISSN: 1094-3420. DOI: 10.1177/1094342010385729. URL: http://dx.doi.org/10.1177/1094342010385729.

[11] NVIDIA. *cuBLAS: CUDA Toolkit Documentation*. 2020. URL: https://docs.nvidia.com/cuda/cublas/index.html.

[12] NVIDIA. *CUDA C++ Programming Guide*. 2020. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide.

[13] NVIDIA. *cuTENSOR: A High-Performance CUDA Library for Tensor Primitives*. 2020. URL: https://docs.nvidia.com/cuda/cutensor/index.html.

[14] NVIDIA. *CUTLASS: CUDA Templates for Linear Algebra Subroutines*. 2020. URL: https://github.com/NVIDIA/cutlass.

[15] J. Revels, M. Lubin and T. Papamarkou. 'Forward-Mode Automatic Differentiation in Julia'. In: *arXiv:1607.07892 [cs.MS]* (2016). URL: https://arxiv.org/abs/1607.07892.

[16] Paul Springer and Paolo Bientinesi. *Design of a high-performance GEMM-like Tensor-Tensor Multiplication*. 2016. arXiv: 1607.00145 [cs.MS].

Contents

List of Figures

List of Abbreviations

API Application Programming Interface.

AST Abstract Syntax Tree.

ATLAS Automatically Tuned Linear Algebra Software.

BF16 Bfloat16.

BLAS Basic Linear Algebra Subprograms.

BLIS BLAS-like Library Instantiation Software.

CPU Central Processing Unit.

CTA Cooperative Thread Array.

CUDA Compute Unified Device Architecture.

CUTLASS CUDA Templates for Linear Algebra Subroutines.

DL Deep Learning.

DLA Dense Linear Algebra.

FP16 Half Precision Floating Point.

FP32 Single Precision Floating Point.

FP64 Double Precision Floating Point.

FPU Floating Point Unit.

GEMM General Matrix Multiplication.

GETT GEMM-like Tensor-Tensor contraction.

GPGPU General-purpose computing on graphics processing units.

GPU Graphics Processing Unit.

GTC GPU Technology Conference.

HPC High Performance Computing.

IR Intermediate Representation.

ISA Instruction Set Architecture.

JIT Just-In-Time.

LoG Loop-over-GEMMs.

MAC Multiply Accumulate.

MAGMA Matrix Algebra on GPU and Multicore Architectures.

MKL Math Kernel Library.

ML Machine Learning.

OpenCL Open Computing Language.

OpenGL Open Graphics Library.

PTX Parallel Thread Execution.

SASS Streaming Assembler.

SIMT Single Instruction Multiple Thread.

SM Streaming Multiprocessor.

SSA Static Single Assignment.

TBLIS Tensor BLIS.

TF32 TensorFloat-32.

TTGT Transpose-Transpose-GEMM-Transpose.

WMMA Warp Matrix Multiply Accumulate.

1 Introduction

The days where GPUs (Graphics Processing Units) were only used as a coprocessor to perform graphics-related tasks have long since passed. Due to their massively parallel nature, GPUs can be used to significantly accelerate scientific computations as well. The use of GPUs for these purposes is commonly referred to as GPGPU (General-purpose computing on graphics processing units) programming. Several fields use GPUs to increase performance, including ML (Machine Learning) and HPC (High Performance Computing) [1, 61, 22, 18, 43].

GPGPU programming is typically performed in low-level languages such as C++, whereas researchers would prefer to use a high-level language such as Python or Matlab. Traditionally, the choice between high-level or low-level languages comes down to a trade-off between performance and programmer productivity. Through the use of modern compiler techniques, it is possible to eliminate this trade-off and use a high-level language without sacrificing performance. Julia is one example of a programming language with a high-level syntax, but performance comparable to that of C++ [25]. Using the CUDAnative project, researchers are even able to program GPUs directly in Julia.

Matrix multiplication, commonly called GEMM (General Matrix Multiplication), is at the core of many computations in ML and HPC, ranging from neural networks to earthquake simulation and weather prediction [1, 33, 22, 43]. The multiplication of matrices is so prevalent that NVIDIA has introduced Tensor Cores in their latest GPUs. Tensor Cores are hardware units specifically designed to compute matrix multiplications quickly. One peculiarity of these Tensor Cores is that the matrix multiplication is performed in mixed precision, meaning that both 16-bit and 32-bit floating point is used during

the computation. The use of mixed precision has resulted in significant speedups in the training of neural networks and in several HPC applications [50, 18].

Efficient implementations of GEMM are bundled in vendor-specific libraries, such as cuBLAS in the case of NVIDIA GPUs. These libraries contain a set of pre-optimised GPU functions, called kernels, that are built for a specific purpose, and hence lack flexibility. This is problematic if the variant of GEMM needed for a given algorithm is not supported. In those cases, programmers typically have to spend a significant amount of time and effort implementing a performant kernel from scratch.

The focus of this thesis is the design and implementation of a high-level framework to instantiate performant GEMM kernels on GPUs. To increase programmer productivity, we will use the high-level programming language Julia and the package CUDAnative instead of C++. Given the prevalence of mixed precision in both ML and HPC, the end goal is a GEMM targetting NVIDIA's Tensor Cores that is both performant and flexible. We will work towards this goal in three main steps.

Tensor Cores can be programmed in C++ using NVIDIA's WMMA API (Application Programming Interface), which CUDAnative did not originally support. The first goal is thus the design and implementation of an API to interact with Tensor Cores from within Julia.

Once we can program Tensor Cores directly in Julia, we can use them to implement a matrix multiplication kernel. A possible approach is to divide the output matrix in small tiles, and to calculate each of these tiles using Tensor Cores. As we shall see, this implementation does not use the computational capabilities of the GPU optimally, and is hence extremely inefficient. The next goal is thus to research the design principles behind performant matrix multiplication kernels on GPUs, and to apply these principles to implement a performant mixed-precision GEMM.

Finally, we need to make this performant GEMM kernel flexible. To achieve this, we first conduct a literature search to understand which types of flexibility are important, and how other frameworks approach this requirement. From this search, we will distil a

list of the main use cases for flexible matrix multiplication kernels, and propose a set of abstractions needed to cover them. As a final step, we will introduce this flexibility in the mixed-precision GEMM kernel that we developed.

We will start with an introduction to the necessary background information regarding GPU computing, the Julia programming language, and mixed-precision arithmetic in Chapter 2. Chapter 3 will go into more detail about the main motivation for this thesis by describing why flexible matrix multiplication kernels are needed. The flexible matrix multiplication interface developed in this thesis consists of three main APIs, each of which will be the subject of a separate chapter. In Chapter 4, we will design and implement the API to interface with NVIDIA's Tensor Cores. Chapter 5 focuses on tiling, one of the core principles behind performant GEMMs. Chapter 6 then builds on top of this tiling API to implement a flexible mixed-precision GEMM. Finally, Chapter 7 concludes the thesis, and lists several possible directions for future research.

2 Background

This chapter gives an overview of the background knowledge required for the rest of this thesis. We will start with an introduction to the core concepts of CUDA (Compute Unified Device Architecture), NVIDIA's platform for GPGPU programming. Next, we will explore the relevant features of the Julia programming language, and how the package CUDAnative.jl allows writing CUDA kernels in Julia. To conclude this chapter, we discuss the concept of mixed precision arithmetic, why it is gaining traction, and how NVIDIA has incorporated this recent trend in their latest GPUs.

2.1 The CUDA programming model

Traditionally, GPUs were used as a coprocessor to offload compute-intensive graphics computations from the CPU (Central Processing Unit). Several APIs were designed to faciliate this, chief among which were Microsoft's DirectX [46] and Silicon Graphics's OpenGL (Open Graphics Library) [32], now maintained by the non-profit Khronos Group. It became clear, however, that GPUs were not only useful in the domain of computer graphics, but were also able to accelerate scientific computations. These early efforts of using GPUs for general-purpose computing were promising, but still required expressing these computations in terms of the underlying graphics primitives used by the hardware.

In November 2006, NVIDIA revolutionised the GPGPU landscape with the announcement of CUDA: their vision for a framework for general-purpose programming on NVIDIA

GPUs [58]. Some time later, Microsoft extended their DirectX graphics API with GPGPU capabilities with the launch of DirectCompute [45]. Similarly, through a joint effort of Apple and the Khronos Group, OpenCL (Open Computing Language) was released in 2009 [31]. While CUDA is only supported on NVIDIA hardware, both OpenCL and DirectCompute have the advantage that they are vendor independent. Nevertheless, for the purposes of this thesis, we will mainly focus our attention on CUDA, since it is more mature in its tooling and software support.

With CUDA's introduction also came a plethora of new terminology specific to CUDA and NVIDIA GPUs. The first chapters of the official CUDA programming guide give an excellent overview of these newly introduced terms, and how they interact [52]. In contrast to traditional CPU programming, the CUDA programming model is one of massive parallelism. Whereas a modern CPU typically has $8 - 16$ cores, an NVIDIA GPU contains a large set of processors that NVIDIA refers to as SMs (Streaming Multiprocessors), each consisting of a number of CUDA cores.

In CUDA, data is processed by a large set of threads executing in parallel. These threads are organised in a thread hierarchy, consisting of:

- *Threads*: Threads are the smallest unit of execution in the hierarchy.

- *Threadblocks*: Threads can be further grouped into threadblocks (or simply blocks). All threads in a threadblock are guaranteed to be executed on the same SM, and can thus cooperate on a common task. For this reason, a block is also referred to as a CTA (Cooperative Thread Array).

- *Grids*: Threadblocks themselves are organised in a grid, representing the set of all blocks on the device.

Threads are not scheduled individually on an SM, but in groups of 32 threads called *warps*. All threads in the same warp execute in a SIMT (Single Instruction Multiple Thread) fashion, meaning that they should execute the same instruction. If threads in a warp disagree on the execution path, e.g. because of a data-dependent conditional branch,

the SM will execute each branch taken, and disable all threads not on the currently executing branch. This serialises the execution of the threads in a warp, and is commonly referred to as *divergence*. Even though warps are primarily related to the hardware implementation, CUDA programmers should be aware of their existence. As we shall see later, some instructions operate at the warp level, and must be executed in lockstep by all threads in a warp.

Similarly to threads, GPU memory is also ordered in a hierarchy. In this thesis, we mainly care about three types, corresponding to the levels in the thread hierarchy:

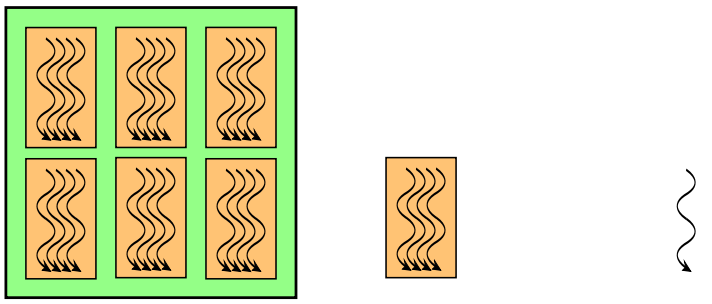
- *Registers*: The register file is the fastest type of memory. In contrast to a CPU core with only a few registers, each CUDA thread can use up to 255 registers on modern GPUs.

- *Shared memory*: Each block has its own set of shared memory, that may be used by threads in the same block to communicate.

- *Global memory*: In contrast to the register file or shared memory, global memory is off-chip, and thus has much higher latency and lower throughput. However, it has the largest capacity of the memory hierarchy, and can be accessed by all threads on the device, regardless of which block they belong to.

A summary of each component in the CUDA thread hierarchy and the corresponding level in the memory and hardware hierarchy is shown in Figure 2.1. Take note of the one-to-one correspondence between the components in the thread, memory, and hardware hierarchy.

CUDA uses the C++ programming language as a base, but extends it with concepts specific to GPUs. A simple kernel that adds two arrays `A` and `B` elementwise, and stores the result in a third array `C` is given in Listing 1. Note that the function `VecAdd` is annotated with the keyword `__global__`. These functions are executed on the GPU (also called device) rather than the CPU (also called host), and are commonly referred to as *kernels*. When calling a regular C++ function, it is only executed once. In contrast,



Thread hierarchy	grid	threadblock	thread
Memory hierarchy	global memory	shared memory	register file
Hardware hierarchy	GPU device	SM	CUDA core

Figure 2.1: An overview of each component in the CUDA thread hierarchy, along with the corresponding level in the memory and hardware hierarchy.

calling kernels will result in the creation of a set of N threads, each executing the same kernel in parallel. The example kernel is very simple: it accesses the identifier of the executing thread using the built-in `threadIdx` variable, and adds the elements in `A` and `B` at that position. Finally, on Line 10, this kernel is launched using the triple bracket syntax `<<<M, N>>>`, which starts the kernel with `M` blocks of `N` threads each.

2.2 The Julia programming language

Julia is an open source programming language featuring a high-level syntax, and performance on par with lower level languages such as C [25, 26]. In Julia, *functions* may have different behaviour depending on the exact types of the arguments [27]. In Julia parlance, each definition of such behaviour is referred to as a *method*. A central paradigm in the design of the language is the way it handles *dispatch*, the process by which the compiler determines which method to use for a given function call. Julia uses a *multiple dispatch* scheme, which means that this choice depends on the number of arguments, and types of *all* of the function's arguments.

```cpp
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // ...
    VecAdd<<<1, N>>>(A, B, C);
    // ...
}
```

Listing 1: A simple CUDA C++ kernel that adds two arrays elementwise. Adapted from the CUDA programming guide [52].

One feature of Julia's type system is that it is *parametric*. This means that generic types may optionally be annotated with information through the use of type parameters. For example, we can parametrise the generic `Array` type to a parametric form `Array{T}` that also includes the type of each element. Like most high-level programming languages, Julia's type system is also *dynamic*, meaning that the types of expressions are not necessarily known at compile time. However, Julia also has some advantages of static type systems through several features of its compiler, which contrast it with the interpreters used in other high-level programming languages such as Python or R.

It achieves this through the combination of JIT (Just-In-Time) compilation, function specialisation, and type inference. Consider for example a simple Julia function that adds its two arguments: `add(x, y) = x + y`. In traditional dynamically typed languages, executing this function will involve a dynamic lookup of the types of `x` and `y`, and a dynamic call to the `+` operator corresponding to these types. The Julia compiler takes a different approach. Suppose we call this function using two 64-bit integers: `add(1, 2)`. Through *type inference*, the compiler knows that both `x` and `y` are of type `Int64`. The

`add` function is then *specialised* for these argument types, and compiled *just-in-time* to an efficient implementation consisting of a simple `leaq (%rdi, %rsi), %rax` on x86-64.

Julia's compiler is built on top of LLVM, a compiler infrastructure project commonly used in research and industry [37]. The compilation process is illustrated in Figure 2.2. First, Julia code is parsed into an AST (Abstract Syntax Tree), a tree representation of the syntactical structure of the source code. This AST is then converted to a Julia-specific IR (Intermediate Representation), that is used for type inference. Next, Julia IR is lowered into the SSA (Static Single Assignment) form that LLVM uses, called LLVM IR. In the last step, this LLVM IR is then converted to native instructions by the LLVM backend corresponding to the current architecture, such as x86-64.

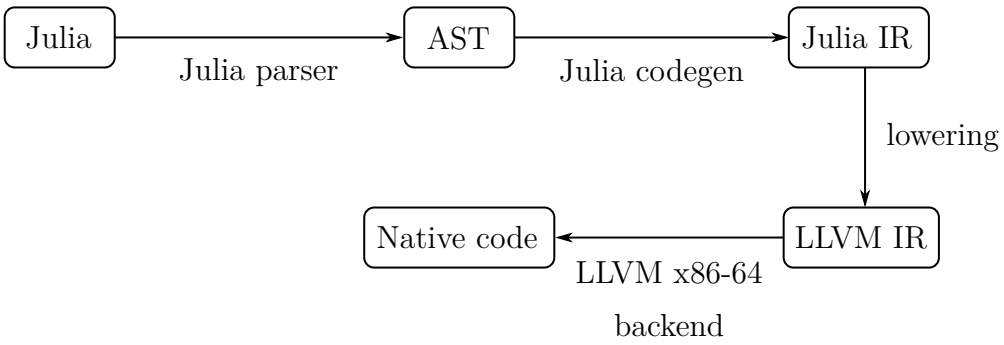


Figure 2.2: An overview of the compilation pipeline of the Julia compiler.

Julia provides access to these intermediate representations from within the language itself. For example, Julia's introspection capabilities include several macros that print the result of each step in the compilation pipeline. `@code_lowered` is used to display the Julia IR, `@code_typed` and `@code_warntype` to display the lowered form after type inference. At a lower level, `@code_llvm` can be used to print the LLVM IR, and `@code_native` for the final machine code.

Apart from querying the intermediate representations, we can also change some of them within Julia. Similar to LISP, Julia represents the AST of parsed code as a data structure in the language itself, called an `Expr`. This is for example used to implement macros, which return an `Expr` that is then compiled at parse time. More powerful is the concept of *generated functions*, which, unlike macros, are expanded at a time when the types of the arguments are known. Like macros, generated functions return an `Expr` that is

subsequently compiled. However, they can generate custom code for each combination of argument types.

Another way in which we can influence the compilation pipeline, is through the use of `llvmcall`. This function allows embedding LLVM IR directly into Julia source code. First, Julia determines the LLVM types corresponding to the Julia types of the arguments and return value, e.g. Julia's `Int64` will be mapped to LLVM's `i64`. Next, the compiler will generate an LLVM function with those types, containing the specified LLVM IR. As a last step, the `llvmcall` is replaced by a call to this function, and, if necessary, instructions to convert the arguments and return value to the correct LLVM type.

2.3 CUDAnative.jl: Executing Julia kernels on CUDA hardware

CUDANATIVE.JL is a Julia package that allows executing kernels written in Julia on CUDA-enabled hardware [10]. To accomplish this, it uses the LLVM backend for NVIDIA GPUs, NVPTX, along with several features of the Julia programming language we have seen in Section 2.2. For example, `llvmcall` is used to call CUDA-specific LLVM instrinsics directly, as is the case for e.g. synchronising all threads in a threadblock.

Similarly, the CUDA memory hierarchy is exposed through the use of parametric types: CUDANATIVE's type for device pointers, `DevicePtr`, has a parameter that determines the address space of the pointer, e.g. global or shared. When loading from or storing to this pointer, the multiple dispatch mechanism is used to dispatch to a custom implementation for device pointers. This implementation makes use of `@generated` functions to generate custom LLVM IR for each pointer type, and thus for each possible address space.

Through the use of CUDANATIVE, one can replace C++ with Julia in the CUDA software stack. For example, Listing 2 shows the equivalent of the simple vector addition kernel of Listing 1. Kernels, such as `vecadd`, are regular Julia functions, that may

```
1 function vecadd(a, b, c)
2     i = threadIdx().x
3     c[i] = a[i] + b[i]
4     return
5 end
6
7 # ...
8 @cuda threads=N vecadd(a, b, c)
9 # ...
```

Listing 2: The CUDAnative.jl equivalent of the simple vector addition kernel in Listing 1.

additionally contain CUDA-specific code, such as the call to `threadIdx()` on Line 2. Note that kernels are only allowed to use a subset of the Julia language. Features that do not map well to GPUs, such as garbage collection and exceptions, are not supported. This kernel is then run on the device using the `@cuda` macro, which serves as a replacement of the triple bracket syntax of CUDA C++.

CUDANATIVE reuses part of the Julia compiler infrastructure we have seen in Figure 2.2. In particular, this compilation pipeline is run up until the lowering stage. The generated code is then intercepted at the LLVM IR level, and sent to the LLVM NVPTX backend, instead of the backend corresponding to the host architecture. The adapted compilation pipeline is given in Figure 2.3. Note that, compared to the original pipeline, CUDANATIVE slightly changes the lowering process from Julia IR to LLVM IR.

Instead of x86-64 assembly instructions, the NVPTX backend emits PTX (Parallel Thread Execution) instructions. PTX is a low-level virtual ISA (Instruction Set Architecture), suitable for execution on CUDA-enabled NVIDIA GPUs. The main goal of PTX is to provide a stable ISA that is portable to different GPU architectures [60]. Note that this ISA is only virtual: for example, PTX uses an infinitely large set of virtual registers, that do not necessarily correspond to physical GPU registers. The GPU itself is not able to execute PTX directly. For this, the NVIDIA graphics driver contains a compiler that

translates this PTX into SASS (Streaming Assembler), the generation-specific ISA that the GPU uses.

Mimicking standard Julia's introspection capabilities, CUDANative also includes macros that print the IRs of GPU kernels. These are named similarly to their Julia counterparts: it suffices to add a prefix `device` to their name, so that `@code_llvm` becomes `@device_code_llvm`. CUDANative also introduces `@device_code_ptx` and `@device_code_sass` to inspect the generated PTX and SASS, respectively. These features are extremely useful to find out which instructions the LLVM backend or the CUDA driver generate for our kernels.

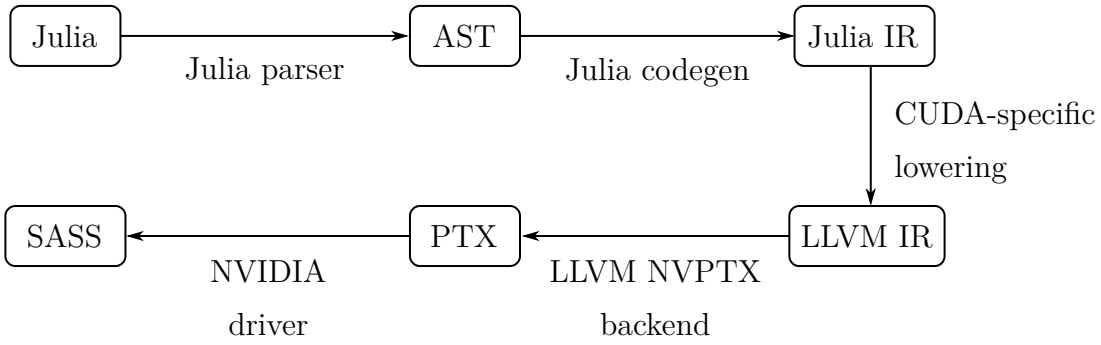


Figure 2.3: An overview of Julia's compilation pipeline adapted by CUDAnative.jl.

Finally, we should briefly mention CuArrays.jl, a package built on top of CUDAnative.jl that defines a new array type, `CuArray` [12]. This package complements CUDANative.jl by introducing several higher-level abstraction on top of the kernels generated by CUDANative.jl. For example, the `CuArray` type implements Julia's broadcast operation, which applies an operation elementwise. At the Julia language level, this is exposed using the special dot-syntax. As such, the vector addition from Listing 2 may be more succinctly written as `c .= a .+ b`, if all the given variables are of the type `CuArray`.

2.4 Mixed precision arithmetic

While FP64 (Double Precision Floating Point) arithmetic has long been the de facto standard in the domain of scientific computing, mixed precision arithmetic is quickly gaining traction [16]. This new concept originated in the fields of ML and DL (Deep Learning), where numerical accuracy is of lesser importance than performance. Clearly, as long as the neural network converges to a model of similar accuracy, it does not matter that intermediate computations have reduced accuracy. Performance, on the other hand, is pivotal: the training and inference process is repeated for each batch of input samples, and for each choice of the hyperparameters of the model.

Mixed precision is, as its name suggests, the process of combining two precisions: FP32 (Single Precision Floating Point) and FP16 (Half Precision Floating Point). The general idea is to perform the main computations in FP16, but store a minimal amount of information in FP32, which is important to guarantee convergence of neural networks. There are several advantages to using mixed precision, compared to the traditional FP64 or FP32 approach [50]:

1. *Power consumption:* Hardware implementing mixed precision arithmetic tends to consume less power than traditional FPUs (Floating Point Units) for FP32 or FP64.

2. *Computation time:* For workloads that are mainly compute-bound, mixed precision will increase performance, since they are typically faster than full FP32.

3. *Memory bandwidth:* Since FP16 are only 2 bytes long compared to FP32's 4 bytes, the required memory bandwidth to load/store them is twice as small, which is beneficial for memory-bound tasks.

4. *Memory usage:* Similarly, an array of FP16 elements will only require half as much storage in global memory. In the context of neural networks, this means that we can double the batch size used during training.

It has been demonstrated that mixed precision models are able to achieve the same accuracy as their full precision counterparts [44]. What makes this even more remarkable is the fact that this can be accomplished without any changes to the model architecture or hyperparameters. Through the use of techniques such as gradient scaling and maintaining a single-precision copy of the weights, this holds true for a variety of different ML models.

Compared to ML, HPC applications are much more sensitive to the precision loss induced by the use of mixed precision. However, it is possible to limit this loss in precision using precision refinement techniques [41]. Consider the case of the multiplication of two matrices A_{FP32} and B_{FP32}, initially stored in FP32. We convert each of these to FP16, and calculate the residual matrices $A_{\text{res}} = A_{\text{FP32}} - A_{\text{FP16}}$ and $B_{\text{res}} = B_{\text{FP32}} - B_{\text{FP16}}$. We can then obtain a higher precision result using four mixed-precision multiplications as $A_{\text{FP32}}B_{\text{FP32}} = A_{\text{FP16}}B_{\text{FP16}} + A_{\text{FP16}}B_{\text{res}} + A_{\text{res}}B_{\text{FP16}} + A_{\text{res}}B_{\text{res}}$. This scheme can be improved by applying the refinement iteratively. Such iterative refinement techniques have already proven their use in mixed-precision solvers for linear systems [17].

Since ML and DL typically use GPUs, it was only a matter of time before NVIDIA saw the importance of supporting mixed precision. In 2017, NVIDIA first incorporated mixed precision in their Volta-generation GPUs. This GPU generation introduced Tensor Cores, a new type of processing core specifically designed for mixed precision arithmetic [5]. In terms of the traditional GPU hardware hierarchy, we may think of them as being "inside of the CUDA cores", complementing the traditional FPU that performs FP32 operations. Tensor Cores of the Volta generation perform a $4 \times 4 \times 4$ matrix MAC (Multiply Accumulate) operation, i.e. an operation of the form $D = A \cdot B + C$, where A, B, C and D are 4×4 matrices. This MAC operation is performed in mixed precision, meaning that the input matrices A and B are stored in FP16, whereas the accumulation matrices C and D may be either FP16 or FP32.

The second generation Tensor Cores were introduced in 2018 with the launch of NVIDIA's Turing architecture [57]. Turing Tensor Cores extend the first generation Tensor Cores with support for 8-bit, 4-bit, and 1-bit integer data types. These are useful for ML workloads that are more tolerant to precision loss, and do not require the full 16-bit precision of FP16.

On 14 May 2020, NVIDIA presented the Ampere GPU architecture, and the new NVIDIA A100 GPU [34]. The A100 includes the third generation Tensor Cores, which support all the operations of Volta's and Turing's Tensor Cores, and additionally add new capabilities. Tensor Cores of the Ampere generation support FP64, which is useful for HPC applications, and BF16 (Bfloat16), a truncated version of the IEEE FP32 format. Ampere Tensor Cores also implement operations in a newly introduced floating point format, TF32 (TensorFloat-32), a hybrid of the standardised FP16 and FP32 formats. Like FP16, TF32 uses a 10-bit mantissa, since that is typically sufficient for the precision requirements of ML applications. The main limitation of FP16 is that the numeric range is quite limited, since it only has 5 bits allocated to the exponent. For that reason, TF32 uses the same 8-bit exponent that is used for FP32, so that the same range is supported.

3 Motivation and related work

In this chapter, we will focus on matrix multiplication. To start off, we explain why we are interested in matrix multiplication kernels in the first place. Next, we illustrate why there is a need for flexiblity in matrix multiplication kernels, and argue that the current state-of-the-art is lacking in this respect. Finally, we conclude by summarising the main goals of this thesis, and how each of these is mapped to the remaining chapters.

3.1 The case for matrix multiplication

Matrix multiplication kernels are interesting for several reasons. First, matrix multiplication is one of the most common operations in linear algebra. This is evidenced by the fact that matrix multiplication is standardised in the BLAS (Basic Linear Algebra Subprograms) specification, a collection of low-level computational routines that has evolved into a de facto standard [14]. BLAS functionality is separated in three separate categories, called *levels*: vector-vector operations (BLAS level 1), matrix-vector operations (BLAS level 2), and matrix-matrix operations (BLAS level 3). In this work, we are mainly interested in BLAS level 3, and more specifically, in its GEMM routine, which calculates a scaled matrix MAC, i.e. an expression of the form $D = \alpha A \cdot B + \beta C$.

Because GEMM is a matrix-matrix operation, there are many opportunities to improve performance by exploiting memory locality. Consider for example a GEMM where all matrices have size $N \times N$. In this case, one matrix contains N^2 elements, leading to a total memory consumption of $\mathcal{O}(N^2)$. Each of the N^2 elements of the output requires N

additions and N multiplications, totalling a computational cost of $\mathcal{O}(N^3)$. As a result, each input element is reused $\mathcal{O}(N)$ times, and we can improve performance by exploiting this reuse. However, writing an efficient GEMM is not trivial, and requires knowledge of the underlying hardware. A framework to automatically instantiate GEMMs could thus prove very useful.

Apart from a standalone matrix multiplication, GEMM is also used as a building block in other computational kernels. For example, the area of DLA (Dense Linear Algebra) deals with linear algebra computations involving dense matrices, i.e. matrices that do not have a special structure that can be exploited. Two important routines in DLA are the LU factorisation of matrices, and solving linear systems of equations. In order to exploit code reuse, these are typically implemented on top of a performant GEMM kernel [3, 17]. By expressing the computationally intensive steps of LU factorisation and linear solvers in terms of matrix multiplication, one can achieve high performance without having to implement these steps from scratch.

Matrix multiplication has also proven its use in several applications in the field of HPC. Of course, of significant importance in HPC is the traditional use case of the linear solvers, as mentioned before [17]. In computational physics and chemistry, tensor contractions, the generalisation of GEMM to multiple dimensions, also play an important role. With the introduction of hardware specialised for matrix multiplication, however, we see several more complex HPC applications being rephrased in terms of GEMMs. For example, using NVIDIA Tensor Cores instead of the traditional FP32 FPUs can result in a massive boost in performance. For the NVIDIA V100, a GPU typically found in datacentres, the FP32 peak performance is 16.4 TFLOPS, whereas Tensor Cores are able to achieve 130 TFLOPS at peak capacity [59]. This has proven useful to accelerate the computations in several HPC applications, such as earthquake simulations [22], plasma visualisation [18], and weather and climate prediction [43].

Finally, we focus on the area of ML and DL. Training neural networks typically involves the calculation of a large number of dot products $y_i = f(w_i^T x_i + b_i)$. In this expression, $x_i \in \mathbb{R}^{n\times 1}$ represents the input features, $w_i \in \mathbb{R}^{n\times 1}$ is the weight vector, $b_i \in \mathbb{R}$ is the bias, f the activation function of the artificial neuron, and $y_i \in \mathbb{R}$ the output. The computation

of these dot products can be parallelised, and rephrased in terms of matrix multiplication. This allows the use of highly optimised GEMM kernels, thus achieving high performance while reusing existing functionality. This technique is used for, amongst others, fully connected layers in traditional neural networks, convolutional neural networks, long short term memory cells, and natural language processing [74, 56].

3.2 The need for flexible matrix multiplication kernels

Now that we have established the importance of GEMM, we will argue why flexibility is an important criterion for matrix multiplication kernels. Google Brain, a DL research team at Google, has recently published a paper that provides an excellent overview of why this flexibility is needed [9]. The main focus is on Capsule networks, a novel ML idea that is similar to the traditional neural networks, but where the neurons are matrix-valued rather than scalars [19]. Compared to the traditional convolutional neural networks, Capsule networks only require a fourth of the number of floating point operations for a network of similar complexity. However, when implementing these Capsule networks in the existing ML frameworks TensorFlow [1] and PyTorch [61], the authors noticed that both performance and memory usage was much worse.

The underlying reason of this inefficiency turned out to be the inflexibility of both frameworks. Because it is not straightforward to introduce custom behaviour, they had to rephrase the computations in terms of components already supported by these frameworks. The most efficient implementation they found consisted of several steps:

1. Split up the input image in partially overlapping patches, and materialise them to main memory.

2. Reshuffle this data so that the dimensions are ordered correctly.

3. Call the pre-optimised, inflexible matrix multiplication kernel.

4. Shuffle the data back to the original layout, essentially undoing step 2.

5. Interpret the resulting data as a multidimensional tensor, and sum this tensor over three dimensions.

This materialisation and reshuffling, necessary to adapt the input to the format that the kernel expects, has a massive impact on performance and memory usage. Even for small input sizes, the authors indicate that they needed to copy, rearrange, and materialise two orders of magnitude more data than necessary.

Unfortunately, the issues do not stop there. Capsule networks have a layered structure, and between each layer a custom computation based on the expectation maximisation algorithm is run. Ideally, this operation should be fused in the matrix multiplication. Due to the inflexibility of the underlying matrix multiplication kernel, however, neither framework is able to do so. This need for operation fusion also occurs in traditional (convolutional) neural networks. Existing ML frameworks typically only implement a limited set of commonly used activation functions, although researchers may want to experiment with new ones. Of course, both of these use cases can be implemented with a separate kernel that is run after the GEMM. That introduces extra kernel launch overhead, however, which can be avoided when operations are fused instead. Additionally, this fusion avoids the need for intermediate storage, and loading the same data multiple times.

Another argument for the flexibility of GEMM is the need to support a multitude of different memory layouts. A basic GEMM only has a few possible layouts: each of the involved matrices may be stored either in row-major or column-major storage format. In the case of convolutions, the design space is much larger because there are more than two dimensions involved. For images, ML frameworks typically use four dimensions: a batch of N images with C channels, each consisting of $W \times H$ features. While any permutation of these is possible, a common choice is to order these as either NCHW or NHWC [56].

Nevertheless, some computational kernels are pre-optimised for a single data layout that is optimal for that specific kernel. As such, they have no support for inputs that are

stored in a different layout. This is especially an issue for ML models that involve a sequential chain of convolutions, as is the case for ResNet [9]. In case the used layouts in each step differ, a conversion must be performed through the use of transpose kernels, which are typically expensive. If the used kernels were flexible with respect to memory layout, we would no longer be constrained by the requirements of the kernel, and could assign the layouts differently. For example, we might assign layouts to each step to optimise the performance globally. Such an assignment may not be optimal for each step individually, but it allows the entire sequence to execute faster because we can avoid the expensive transposition between each step.

Finally, let us focus our attention on tensors, the generalisation of matrices to multiple dimensions. In particular, we are interested in the *contraction* of two tensors, the analog of matrix multiplication for tensors. This operation is common in several scientific fields, such as fluid dynamics [66], electromechanics [62], and computational chemistry [7].

Whereas a matrix-matrix multiplication $C_{mn} = \sum_k A_{mk} \cdot B_{kn}$ only has three different indices $\{m, n, k\}$, tensor contractions involve an arbitrarily large set of indices. The transpositions of A and B in the case of GEMM are extended to arbitrary permutations of the indices of the tensors $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$. As an example, consider this contraction of two 4D tensors $\mathcal{A}$ and $\mathcal{B}$, resulting in another 4D tensor $\mathcal{C}$: $\mathcal{C}_{m_1 n_1 n_2 m_2} = \sum_{k_1,k_2} \mathcal{A}_{m_1 k_1 m_2 k_2} \cdot \mathcal{B}_{n_2 k_2 n_1 k_1}$. In this case, the set of indices is $\{m_1, m_2, n_1, n_2, k_1, k_2\}$. Typically, this index set is partitioned in three disjoint sets. The set $\{k_1, k_2\}$ of indices that summed over is referred to as the set of *contracted* indices. The indices m_1 and m_2, which occur in both $\mathcal{A}$ and $\mathcal{C}$, are denoted as the *free* indices in $\mathcal{A}$. Similarly, n_1 and n_2 are called the *free* indices in $\mathcal{B}$.

First, note that the number of possible data layouts for tensor contractions is significantly higher than was the case for matrix multiplication. For matrices, each input matrix A and B may be transposed or not, leading to 4 different possibilities. This corresponds to the position of the summation index k in both matrices: A_{mk} vs. A_{km}, and B_{kn} vs. B_{nk}. On the other hand, the previous tensor contraction has a total of $4! \times 4! \times 4! = 13824$ different memory layouts. For tensor contractions of even higher dimension, this number grows exponentially fast.

The main challenge in implementing a framework for generic tensor contractions is thus the large number of possible cases that must be considered. Given the importance of tensor contractions, a lot of research has been done to implement this efficiently. Springer and Bientinesi classify the traditional approaches to tensor contraction in three main categories: loop nesting, LoG, and TTGT [69].

The first approach, loop nesting, is closest to the mathematical description of the tensor contraction. The contraction is implemented as a set of nested loops, where each loop corresponds to one index [6, 40, 49]. Additionally, loop transformations such as loop reordering and loop fusion may be applied to improve performance. These approaches exhibit suboptimal memory access patterns due to the high stride accesses [68]. As such, they are not sufficient for larger tensor contractions where the input tensors do not fit in the cache.

The second approach, LoG (Loop-over-GEMMs), is built on the idea of reusing efficient GEMM kernels to implement tensor contractions [47, 36]. Conceptually, the tensor contraction is split up in two dimensional slices, which are then interpreted as a matrix. These 2D subtensors are then multiplied using a regular GEMM kernel. While this approach exploits the high performance of pre-implemented GEMM kernels, performance may suffer if the resulting 2D slices are small [68].

The third approach, TTGT (Transpose-Transpose-GEMM-Transpose), has been implemented by several tensor contraction frameworks in use today, including the Cyclops Tensor Framework [67] and Tensor Toolbox [8]. It operates according to the same principle as LoG, i.e. to reuse efficient GEMM implementations. However, whereas LoG involves multiple calls to GEMM, possibly of very small matrices, TTGT only calls GEMM once. Because the data layout of general tensors is not suitable for use by GEMM, TTGT involves reshuffling the tensors $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ so they have the same memory layout as expected by GEMM.

Figure 3.1 illustrates TTGT on the example tensor contraction of before: $\mathcal{C}_{m_1 n_1 n_2 m_2} = \sum_{k_1,k_2} \mathcal{A}_{m_1 k_1 m_2 k_2} \cdot \mathcal{B}_{n_2 k_2 n_1 k_1}$. First, the input tensors $\mathcal{A}$ and $\mathcal{B}$ are reshuffled such that the free indices are on the outside, and the contraction indices are on the inside. This

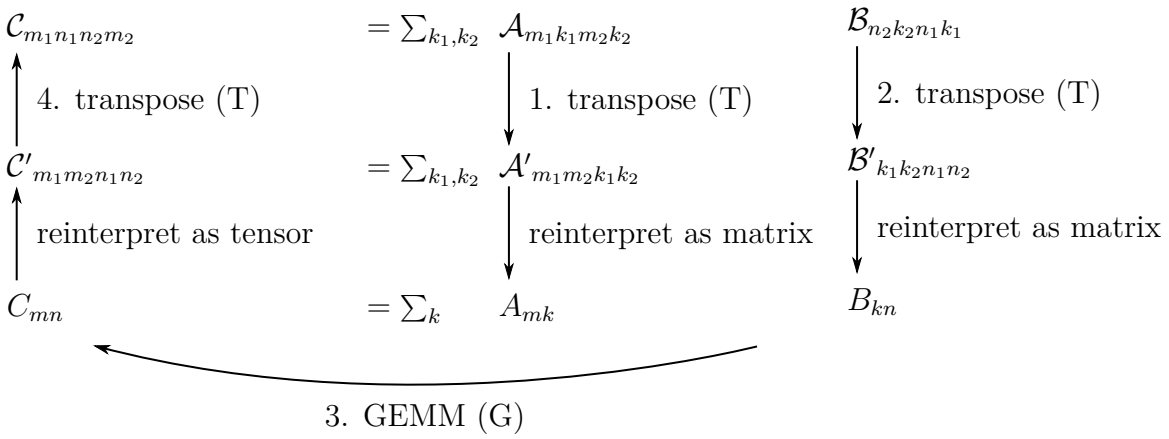


Figure 3.1: An illustration of the TTGT algorithm for tensor contractions.

step involves two transposes, leading to the first two T's in the acronym TTGT. Next, note that consecutive indices in tensors can be interpreted as a single virtual index, so that we may reinterpret these transposed tensors as matrices. A performant GEMM kernel is then executed, leading to the G in the acronym. This results in a new matrix C, which may then subsequently be reinterpreted as a tensor. Finally, the last T in TTGT signifies the transposition of this intermediate tensor $\mathcal{C}'$ to obtain the result $\mathcal{C}$.

It has been shown that the TTGT approach works especially well for compute-bound tensor contractions [68]. However, it also has some significant disadvantages [33]. The transpose steps require extra kernel launches that constitute pure overhead, and additionally need intermediate storage to store the transposed tensors. Moreover, the resulting matrices A and B may be highly rectangular. Typically, GEMM routines have much higher performance for square or nearly-square matrices, resulting in lower performance for TTGT.

In 2016, Springer and Bientinesi proposed another method for tensor contractions, GETT (GEMM-like Tensor-Tensor contraction) [68], that has since been adopted by other tensor contraction implementations [33, 42]. GETT is based on the principes of TTGT, but implicitly reorganises tensors while loading them. As such, the overhead associated with the explicit transpositions is avoided. We may thus think of GETT as a variant of TTGT, where the transposes are fused in the GEMM kernel itself. Clearly, this fusion requires that the underlying GEMM kernel is flexible.

3.3 Related work

Having looked into the need for flexibility in matrix multiplication kernels, we will now discuss the most prominent GEMM implementations, and evaluate their flexibility. Since GEMM is standardised in the BLAS specification, plenty of implementations exist, including both commercial and open-source variants. On the CPU, the most notable are Intel's MKL (Math Kernel Library) [23], and the open source projects OpenBLAS [73], and ATLAS (Automatically Tuned Linear Algebra Software) [72]. However, due to their strict adherence to the BLAS specification, their flexibility remains limited. For example, the only supported memory layouts for real matrices are row-major and column-major. As such, these standard BLAS implementations are unsuitable for the GETT approach to tensor contractions.

In 2015, Van Zee and van de Geijn introduced their framework BLIS (BLAS-like Library Instantiation Software) [71]. BLIS relaxes the monolithic nature of classical BLAS implementations by exposing its kernels as separate components. These components may then be reused to implement computational kernels not present in BLAS. BLIS also extends the row-major and column-major layouts to generalised matrix layouts. The generalised layouts may have any arbitrary stride, which is important for GETT. The flexibility of BLIS is used by TBLIS (Tensor BLIS), an implementation of tensor contraction built on the BLIS framework [42]. While TBLIS was independently developed from Springer's GETT, the underlying idea is the same: fusing the transpositions in the GEMM kernel.

Similar observations hold for the GPU landscape of GEMM. The open source MAGMA (Matrix Algebra on GPU and Multicore Architectures) project focuses on DLA on GPUs and heterogeneous architectures [48]. Proprietary solutions include NVIDIA's cuBLAS, the de facto standard for BLAS on CUDA-enabled hardware [51]. cuBLAS contains a set of GEMM kernels written directly in SASS for optimal performance. Starting with CUDA 10, the cuBLAS APIs also include cuBLASLt, a lightweight library dedicated to GEMM. cuBLASLt generalises the BLAS interface by adding more flexibility to the types used in GEMM. For example, cuBLASLt includes support for mixed-precision

through the use of Tensor Cores. Nevertheless, both MAGMA and cuBLAS suffer from the same inflexibility imposed by the fixed BLAS interface.

Another application of GEMM, especially on GPUs, is the efficient implementation of convolutions in convolutional neural networks. NVIDIA has collected a set of DL primitives for this purpose in the cuDNN library [53]. cuDNN is flexible in the sense that the most common data layouts, NHWC and NCHW, are supported. However, it only includes support for a limited set of activation functions. Since cuDNN is a proprietary library, this cannot be easily changed by end users without sending a feature request to NVIDIA.

The Julia package CUDAnative includes high-level wrappers for both cuBLAS and cuDNN. For example, when two matrices of the correct types are multiplied, this call is dispatched to cuBLAS. Note that this only holds for types supported by cuBLAS, and will not work for custom types defined by the user. In those cases, a generic matrix multiplication routine is used as a fallback. However, because the underlying implementation does not make use of any memory reuse techniques such as tiling, its performance is many orders of magnitude worse than that of cuBLAS.

Perhaps the most interesting project is NVIDIA's open source CUTLASS (CUDA Templates for Linear Algebra Subroutines) [55]. CUTLASS contains a set of C++ templates that together implement a performant GEMM on CUDA-enabled GPUs. It is thus similar in spirit to BLIS, where the GEMM kernel is decomposed in reusable parts. Due to its open source nature, CUTLASS serves as an excellent starting point for the purposes of this thesis. CUTLASS is not only useful as a source of inspiration for possible abstractions, but is also a benchmark to assess the performance of our GEMM kernels. CUTLASS will be discussed in more depth when we implement the GEMM API in Chapter 6. We will then highlight the relevant implementation details of CUTLASS, and explain the similarities or differences between it and our API. While CUTLASS is definitely a step in the right direction, it still requires low-level programming with CUDA C++. Additionally, it consists of a large amount of components that interact in ways that may not be obvious on first sight. Both these observations make it unsuitable for rapid prototyping in the context of ML and DL.

Finally, we should discuss cuTENSOR, a tensor contraction library by NVIDIA [70]. Similarly to TBLIS, cuTENSOR follows the GETT approach to tensor contraction, whereby transpositions are fused in GEMM. It is built on top of the flexible matrix multiplication kernels of CUTLASS, and offers support for a wide variety of elementwise operations [54]. However, these elementwise transformations are still specified as a finite set of pre-implemented operations. True operator fusion can only be obtained when users can pass any arbitrary functor to the computational kernels.

3.4 Goals of this thesis

In this thesis, we will design, implement, and evaluate abstractions to instantiate flexible and performant GEMM kernels. We will mainly target researchers in the ML and DL fields, that frequently iterate over new ideas, and want to test them out quickly without writing a complete kernel by hand. As such, we will make the abstractions as intuitive as possible, and use the high-level programming language Julia instead of CUDA C++.

Given the prevalence of mixed precision in ML, we will evaluate the result using a mixed precision GEMM using NVIDIA's Tensor Cores. To accomplish this, we identify 3 goals, each of which will be covered in a separate chapter. Chapter 4 focuses on the implementation of abstractions to program Tensor Cores from within Julia. In Chapter 5, we will lay the foundations for the GEMM by developing an API to easily implement blocking. Such blocking techniques are necessary to improve the performance of GEMM by exploiting its memory reuse property. Finally, Chapter 6 discusses the main matter at hand: the matrix multiplication API itself.

Most of the work in this thesis was performed using CUDAnative version 3.0, and Julia version 1.5. We had two setups at our disposal. The first setup contains a Volta-generation NVIDIA V100, and runs on CUDA 10.1. The second setup contains a Turing-generation RTX 2080 Ti, and runs on CUDA 10.2.

4 Abstractions for programming Tensor Cores in Julia

In this chapter we will focus on the first concrete goal of this thesis: the implementation of an API for programming NVIDIA's Tensor Cores from within Julia. Section 4.1 serves as an introduction to this chapter, and explains the way that Tensor Cores are exposed in CUDA C++. In Section 4.2, we briefly list the main requirements we impose on the API in advance. The main content is Section 4.3, which describes the implementation of the API itself. Finally, Section 4.4 evaluates the developed API according to the criteria specified in Section 4.2.

4.1 WMMA: CUDA C's interface to Tensor Cores

Recall that each Tensor Core calculates an expression of the form $D = A \cdot B + C$, where A, B, C, and D are 4×4 matrices. This operation is performed in mixed precision: the input matrices A and B are stored in FP16, and the accumulator matrices C and D may be either FP16 or FP32. CUDA does not expose Tensor Cores directly to programmers [52]. Instead, multiple Tensor Cores must be used cooperatively by all threads in a warp, in a SIMT fashion. Failure to do so will result in undefined behaviour. For that reason, CUDA's API for Tensor Cores is referred to as WMMA (Warp Matrix Multiply Accumulate).

Like the underlying Tensor Cores, WMMA calculates a matrix multiply accumulate $D = A \cdot B + C$. However, whereas Tensor Cores operate on 4×4 matrices, WMMA uses matrices of a larger size. We say that A is an $M \times K$ matrix, B is a $K \times N$ matrix, and C and D are $M \times N$ matrices. WMMA restricts the possible values of M, N, and K. In WMMA parlance, the tuple (M, N, K) is referred to as the *shape* of the multiply accumulate.

Conceptually, WMMA consists of three separate steps:

1. Load the input matrices A, B, and C from memory to registers using a WMMA load operation.

2. Perform the matrix multiply accumulate using a WMMA MMA operation. This results in the matrix D, stored in registers.

3. Store the resultant D matrix from registers to memory using a WMMA store operation.

Each thread that cooperates in a warp-wide WMMA holds a part of each matrix in its registers. In WMMA terminology, this part is referred to as a WMMA *fragment*. Note that the exact mapping between matrix elements and fragments is unspecified, and subject to change in later CUDA versions.

To illustrate WMMA in CUDA C++, a mixed-precision matrix multiply accumulate of 16×16 matrices is implemented in Listing 3. The given kernel `wmma_example` has four arguments, containing pointers to the first element of the matrices A, B, C, and D. The kernel should be executed by one warp, i.e. one block of 32 threads. In Lines 3 – 6, the WMMA fragments for each matrix are defined. In CUDA C++, the type of these fragments is used during overload resolution, so they must be defined upfront. These fragments are C++ templates, where the parameters contain information such as the element type (`half`, `float`), data layout (`wmma::col_major`), matrix use (`wmma::matrix_a`), and overall shape of the WMMA operation (`16, 16, 16`).

```
1  __global__ void wmma_example(half *a, half *b, float *c, float *d)
2  {
3      wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::col_major> a_frag;
4      wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::col_major> b_frag;
5      wmma::fragment<wmma::accumulator, 16, 16, 16, float> c_frag;
6      wmma::fragment<wmma::accumulator, 16, 16, 16, float> d_frag;
7
8      wmma::load_matrix_sync(a_frag, a, 16);
9      wmma::load_matrix_sync(b_frag, b, 16);
10     wmma::load_matrix_sync(c_frag, c, 16, wmma::mem_col_major);
11
12     wmma::mma_sync(d_frag, a_frag, b_frag, c_frag);
13
14     wmma::store_matrix_sync(d, d_frag, 16, wmma::mem_col_major);
15 }
```

Listing 3: A mixed-precision $16 \times 16 \times 16$ matrix multiply accumulate, implemented using WMMA in CUDA C++.

Lines 8 – 14 contain the main WMMA steps, as outlined before. Step 1, the loading of the matrices, is shown in Lines 8 – 10. Note that the parameter 16 refers to the leading dimension (stride) of the matrix being loaded. This leading dimension may differ from the WMMA shape in the case of submatrices that are embedded in a matrix of larger dimensions. Line 12 implements step 2, the multiply accumulate operation itself. Finally, step 3 is shown on Line 14, that stores the resultant D matrix back to main memory. The parameter 16 once again refers to the leading dimension of D, not to the WMMA shape itself.

The CUDA compiler then converts the given CUDA C++ to PTX instructions. The initial definition of the fragments in Lines 3 – 6 is only used for overload resolution, and is not lowered to PTX. Listing 3 is converted to five WMMA PTX instructions: three `wmma.load`s (one for each of A, B, C), one `wmma.mma`, and one `wmma.store` (for D). The names of these PTX instructions all follow a pattern. For example, the loading of

A is lowered to PTX's `wmma.load.a.sync.aligned.col.m16n16k16.global.f16`. The `aligned` infix specifies that the instruction must be called by all threads in a warp. Similarly, `sync` indicates that the executing thread will block until all other threads in the same warp have reached the same instruction. Both of these infixes are mandatory, but for our purposes, they bear little relevance. The name of this instruction contains some information that was specified explicitly in CUDA C++, such as the data layout (`col`), shape (`m16n16k16`), and element type (`f16`). Additionally, the instruction also contains some information that we did not specify at all, but that was derived from the context. In this case, the compiler inferred that the pointer arguments must refer to global memory, and adds an infix `global` to the instruction name.

Finally, the CUDA driver compiles this intermediate PTX to device-specific SASS. Since the loads and stores are inferred to refer to global memory, they are lowered to instructions specific to global memory, `LDG.E.SYS` and `STG.E.SYS`. The way that `mma` is handled depends on the target GPU architecture. On GPUs of the Volta generation, such as the NVIDIA Tesla V100, the `mma` is converted to 16 `HMMA.884.F32.F32` instructions, each calculating an $8 \times 8 \times 4$ slice of the result [24]. On the NVIDIA RTX 2080 Ti, which is a Turing-generation GPU, the `mma` corresponds to 4 `HMMA.1688.F32` instructions, each performing a $16 \times 8 \times 8$ matrix multiplication [74].

4.2 Requirements

Before jumping straight into the implementation of the WMMA API for Julia, we should impose some requirements on the result. We identify four important criteria:

1. *Zero-cost*: Using the WMMA API in Julia should not result in extra runtime overhead compared to a manual implementation, such as calling the PTX instructions directly. For example, we need to avoid overheads in the form of superfluous instructions, extra memory footprint, and increased execution time.

2. *Building block for subsequent chapters*: In the context of a complete matrix multiplication kernel, WMMA is an abstraction at a lower level. Later chapters will build on top of the WMMA API to implement a flexible GEMM kernel. As such, we need to make sure that the WMMA API supports the operations needed by the higher abstraction layers that make use of it. In particular, the Julia WMMA API must have support for elementwise operations, such as scaling. Elementwise operations are by far the most common, ranging from a simple scaling in the case of simple matrix multiplication, to the activation functions in neural networks. Julia's syntax even includes first-class support for elementwise operations, through its dot syntax [27]. For example, applying a function `f` elementwise to an array `a` may be succinctly written as `f.(a)`.

3. *Future-proof*: The future-proof requirement imposes that the implementation should be adaptable to WMMA changes in the future, preferably without breaking changes to the API exposed to the end users. When WMMA was first introduced, it only supported a $16 \times 16 \times 16$ multiply accumulate of FP16 matrices [52]. With the launch of Turing's second-generation Tensor Cores, WMMA was extended with new data types (8-bit, 4-bit, and 1-bit integers) and new shapes ($8 \times 32 \times 16$ and $32 \times 8 \times 16$). On 14 May 2020, NVIDIA published details on the changes to WMMA for the third-generation Tensor Cores in Ampere [63]. Once again, WMMA was extended with support for new data types (FP64, TF32, BF16) and new shapes ($8 \times 8 \times 4$ and $16 \times 16 \times 8$). In general, we must ensure that the WMMA implementation in Julia can easily adapt to the introduction of new data types and shapes.

4. *Close to CUDA*: The API should be similar to CUDA C++'s version of WMMA. This ensures that the API is familiar to users coming from CUDA C++, so that programmers wishing to program at a lower level of abstraction should have little trouble using WMMA in Julia. One thing to keep in mind is that Julia and C++ are quite different languages. We should thus not try to exactly replicate CUDA C++. Instead, we may deviate from the C++ version to make the resulting API fit in the Julia programming language better.

4.3 A WMMA API for Julia

Recall that when WMMA is used from CUDA C++, the source code is first compiled to generation-independent PTX instructions. The CUDA driver then converts these intermediate PTX instructions to device-specific SASS. The compilation process of the Julia package CUDAnative is more complicated. We have already illustrated the compilation pipeline of CUDAnative in Figure 2.3, but for clarity, we repeat it here in Figure 4.1. The last two steps, PTX and SASS, are the same as CUDA C++, since these are fixed by NVIDIA. Rather than directly converting Julia to PTX, Julia is first lowered to LLVM IR. CUDAnative then uses the LLVM NVPTX backend to convert the LLVM IR to the necessary PTX instructions.

The LLVM framework supports a wide range of platforms, each having a specific backend that converts LLVM IR to the machine instructions of that platform. In order to avoid cluttering the language specification, platform-specific features are not added to LLVM IR. Instead, each backend can define a set of functions that have a special meaning for that backend. These functions are commonly referred to as *intrinsic functions* or *intrinsics*. Since WMMA is specific to the NVPTX backend, it is exposed in LLVM IR through a set of these intrinsics.

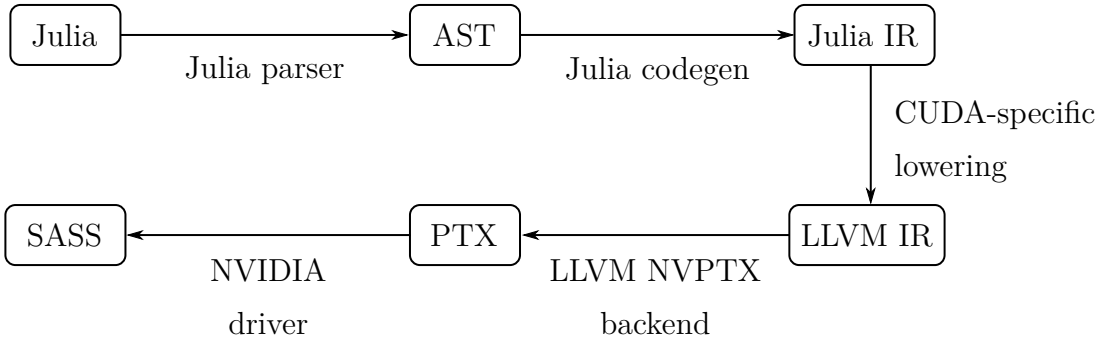


Figure 4.1: An overview of Julia's compilation pipeline adapted by CUDAnative.jl.

To support WMMA in Julia, we will build on top of the existing WMMA intrinsics of the NVPTX backend. A schematic overview of the WMMA API for Julia that we will develop is given in Figure 4.2. The bottom part of the figure displays the pre-existing components in the LLVM NVIDIA GPU stack that we will reuse: the WMMA intrinsics,

the PTX generated by the NVPTX backend, and the device-specific SASS that the CUDA driver emits. Our Julia WMMA API is shown at the top of Figure 4.2, and consists of two parts. As a first step, we will focus on exposing the NVPTX WMMA intrinsics in Julia. To accomplish this, we will implement a set of Julia wrapper functions, where each wrapper function corresponds to one WMMA intrinsic. We can then use these intrinsics in Julia by simply calling the corresponding Julia wrapper. The design and implementation of these wrapper functions will be the subject of Section 4.3.1. Of course, we cannot expect users to directly call these wrappers every time they want to use WMMA. They will want to use a higher level API similar to the way WMMA is used from within CUDA C++. Section 4.3.2 will thus focus on abstracting these intrinsics to an API that is similar to CUDA's WMMA.

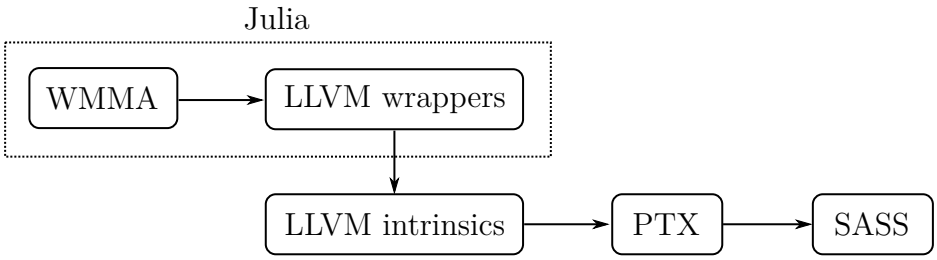


Figure 4.2: A schematic overview of the WMMA API that we will develop for Julia (top) and the pre-existing components it relies on (bottom).

Note that, to use a given WMMA operation, all components in Figure 4.2 need to support it. In particular, if the LLVM NVPTX backend does not have intrinsics for a specific WMMA variant, we cannot use it from Julia either. At the time when we implemented Julia's version of WMMA, the LLVM version that Julia used only supported the Volta-generation WMMA operations. As such, the implementation described in the next sections will only offer support for the Volta-generation WMMA types and shapes. Nevertheless, during our implementation of WMMA, we kept the need for future extensibility to Turing and Ampere in mind. While we only have support for Volta-style WMMA at the NVPTX level, we have both Volta and Turing GPUs at our disposal. Because WMMA is forward compatible, we will test our developed API on both Volta and Turing hardware.

4.3.1 Wrappers for the LLVM intrinsics

To call the WMMA NVPTX intrinsics from Julia, we can use Julia's `llvmcall`, as discussed in Section 2.2. The Julia compiler will first convert the Julia types passed to `llvmcall` to their LLVM IR equivalent. This mapping of Julia types to LLVM IR types is hardcoded in the Julia compiler. To support WMMA in Julia, we need to adapt this mapping, since the LLVM WMMA intrinsics use some LLVM types that have no corresponding Julia equivalent:

1. `half`: Julia's `Float16` type is lowered to an LLVM 16-bit integer, `i16`, instead of the half-precision floating point type `half`.

2. `i8 addrspace(x)*`: LLVM pointer types such as `i8*` can be optionally annotated with an address space. NVPTX uses this to map each pointer to a level of the GPU memory hierarchy. For example, `i8 addrspace(1)*` represents a pointer to global memory, whereas `i8 addrspace(3)*` refers to a location in shared memory [39]. While the Julia type `Ref{T}` is lowered to `i8*`, it does not support address spaces.

3. `{float, ..., float}`: WMMA fragments are represented in LLVM as a structure aggregate type, indicated by curly braces. However, since Julia `structs` are lowered to LLVM arrays, LLVM structures have no equivalent Julia type.

We thus extended Julia's code generation to support these LLVM types. Because these changes should only apply to `llvmcall`, a boolean parameter was added to the Julia-to-LLVM type conversion functions. This boolean indicates whether the type mapping is meant for use with `llvmcall`, or will be used in regular Julia code. Supporting `half` and LLVM structures is fairly straightforward: it suffices to override the default type mapping of `Float16` and `structs` in case `llvmcall` is used. The case of pointers with address spaces is somewhat trickier. For compatibility reasons, we cannot just add an address space parameter to `Ref{T}`. Instead, a new built-in type `Core.AddrSpacePtr{T, AS}` was added to the Julia language, that gets lowered to an LLVM pointer type to the

correct address space `AS`. The required adaptations to the Julia code generation were sent to the upstream developers, and have since been merged into Julia[1].

This type mapping is not the only problem we encountered while adding support for WMMA in Julia. It turned out that NVIDIA had changed the names of the WMMA PTX instructions in PTX version 6.3. The LLVM NVPTX backend was later updated to accommodate this change, but Julia used an LLVM version that did not include this patch yet. The next step in supporting WMMA for Julia was thus to port this patch to Julia's LLVM version. During this porting, we noticed that this patch depended on two other patches, one of which refactored the WMMA implementation in NVPTX. The second patch added a new feature to TableGen, LLVM's domain-specific language that is used in the backends to lower LLVM IR to machine-dependent instructions [38]. This new TableGen feature was used by the NVPTX backend to emit different instructions depending on the PTX version that was targeted. The resulting three patches were reapplied on top of Julia's LLVM version, and bundled in one pull request that has since been merged in upstream Julia[2].

Now that the Julia compiler supports the needed LLVM types, and NVPTX generates the correct PTX instructions, we can begin wrapping the WMMA intrinsics. Like the PTX instructions, the LLVM intrinsics are named according to a pattern. For example, the loading of the A matrix is represented as a call to the intrinsic function `@llvm.nvvm.wmma.m16n16k16.load.a.col.stride.f16.p1i8`. Note that, like PTX, the LLVM intrinsic includes information such as the memory layout (`col`), address space (`p1`, referring to global memory), and WMMA shape (`m16n16k16`) in its name.

To satisfy our future-proof requirement, we will not be defining each individual wrapper manually. Instead, we first define a set of configuration variables that determine, amongst others, the possible WMMA shapes, and the WMMA fragment sizes for each matrix and element type. We can then iterate over all possible configurations, and generate the

[1]Pull requests available at: `https://github.com/JuliaLang/julia/pull/33970` (Float16), `https://github.com/JuliaLang/julia/pull/34760` (AddrSpacePtr), and `https://github.com/JuliaLang/julia/pull/34996` (LLVM structures).

[2]Pull request available at `https://github.com/JuliaLang/julia/pull/34043`.

```
1 for N in unique(values(map_frag_sizes))
2     struct_ty = Symbol("LLVMStruct$N")
3
4     @eval struct $struct_ty{T}
5         Base.Cartesian.@nexprs $N i -> x_i::T
6     end
7
8     @eval Base.convert(::Type{NTuple{$N, T}}, x::$struct_ty{T}) where {T} =
      ↪  ntuple(i -> getfield(x, i), $N)
9 end
```

Listing 4: Dynamically generating Julia structs with the correct number of fields, using Julia's metaprogramming functionality.

name of the wrapper and LLVM intrinsic depending on the values of the configuration variables. A function with that name that calls the correct LLVM intrinsic is then created dynamically.

This approach allows for future extensibility, but it complicates the generation of the types needed for the `llvmcall`. In particular, because the fragment sizes are determined by the configuration variables, we need a way to dynamically generate a Julia `struct` that contains the correct amount of fields. To solve this, one can make use of Julia's powerful metaprogramming abilities, as shown in Listing 4. The variable `map_frag_sizes` maps each matrix and element type to its corresponding fragment size. Line 1 iterates over the unique values in this `map`, i.e. the fragment sizes, and dynamically generates a `struct` with the corresponding number of fields. Line 2 defines a `Symbol` representing the name of the generated `struct`. Lines 4–6 then generate a parametric `struct` with this name, and one type parameter `T`. The generated `struct` contains N fields `x_1`, `x_2`, ..., `x_N`, each of type `T`. Finally, we may want to convert this `struct` to a Julia `NTuple`, which is a better fit for statically-sized data such as WMMA fragments. For example, contrary to `struct`s, `NTuple`s support indexing through the subscript operator `some_ntuple[i]`. To allow this conversion, Line 8 defines how this newly generated `struct` can be converted to Julia's `NTuple` type.

4.3.2 A WMMA interface for Julia

Since we cannot expect users to directly use the wrappers of the LLVM intrinsics, we need to implement a CUDA C++-like abstraction layer on top of them. The main difference between the CUDA C++-like API that we will develop, and the lower level wrappers, is that the former enforces several constraints when working with WMMA. For example, it ensures that the A fragment argument to a `wmma.mma` instruction was obtained by a call to `wmma.load.a`, and not `wmma.load.b` or `wmma.load.c`. Additionally, it makes sure that the data type and the storage layout of the `wmma.load`/`wmma.store` operations and the `wmma.mma` operation match.

We can achieve this by adding extra information to the fragment types, similar to CUDA C++. To represent WMMA fragments, we will use a parametrised type `WMMA.Fragment{M, N, K, FS, T, L, U}`, as shown in Listing 5. The type parameters determine the WMMA shape (`M`, `N`, `K`), the size `FS` and element type `T` of the fragment, the storage layout `L` (row- or column-major), and the use `U` of the corresponding matrix (A, B, or one of the accumulator matrices). Each of these parameters is specified in the type domain. For example, the use of a fragment is given as a `struct` type that is a concrete version of the abstract `FragmentUse` type. This way, we can use Julia's multiple dispatch mechanism to select the correct variant of WMMA instructions to call.

The `WMMA.Fragment` type is a `struct` with one field `x`. This field `x` is an `NTuple` of the correct size and element type, and contains the fragment's data. Users can access a fragment's elements through the `x` field, like in CUDA C++. However, the Julia language allows us to extend this further. In Julia, indexing can be implemented for custom types by specialising a limited set of functions: `getindex`, `setindex!`, `firstindex`, and `lastindex`. For example, `X[i]` is converted a call to `getindex(X, i)` [27]. We can thus implement the required methods for parameters of type `WMMA.Fragment`, and redirect the function to the `x` field instead. This way, we may either write `frag.x[i]`, or `frag[i]` to access the `i`th element of a fragment `frag`. Once again, Julia's metaprogramming capabilities allow us to do this easily, as shown in Listing 6. The given snippet iterates over the functions to specialise in the loop of Line 1. Line 2 then specialises each

```
abstract type FragmentLayout end
struct RowMajor <: FragmentLayout end
struct ColMajor <: FragmentLayout end
struct Unspecified <: FragmentLayout end

abstract type FragmentUse end
struct MatrixA <: FragmentUse end
struct MatrixB <: FragmentUse end
struct Accumulator <: FragmentUse end

struct Fragment{M, N, K, FS, T, L <: FragmentLayout, U <: FragmentUse}
    x::NTuple{FS, T}
end
```

Listing 5: The definition of a fragment in the Julia WMMA API.

```
for f in (:getindex, :setindex!, :firstindex, :lastindex)
    @eval Base.$f(frag::Fragment, args...) = $f(frag.x, args...)
end
```

Listing 6: Defining indexing for WMMA fragments using Julia's metaprogramming.

function for arguments that are instances of the `WMMA.Fragment` type. The underlying implementation forwards this call to the same function, where the first argument is changed from `frag` to `frag.x`.

One peculiarity of the LLVM intrinsics is that half-precision types are stored as an LLVM vector type of two elements. That is to say, a WMMA fragment of FP16 values is represented as `{<2 x half>, ..., <2 x half>}`, which corresponds to Julia's type `NTuple{N, NTuple{2, VecElement{Float16}}}`. Using this Julia type to store a fragment's data in `WMMA.Fragments` complicates elementwise operations, so we take a different approach. We store the values in a fragment using the flattened type `NTuple{2 * N, Float16}` instead. Of course, we need to convert to/from this flattened type when calling the LLVM intrinsics. This flattening and unflattening process is implemented

using `@generated` functions, which recursively generate an `Expr` that flattens or unflattens the input. Due to the `@generated` nature of these functions, this recursion only happens during type inference, and does not result in an overhead at run time.

Despite the similarities between the Julia API and CUDA C++, we depart from CUDA C++ in the way we handle fragments. In CUDA C++, the fragments must be declared upfront, because their type is used during overload resolution to select the correct variant of WMMA instructions to call. This means that the fragments must be passed as a by-reference argument, which obscures the data flow. In essence, in CUDA C++, the fragments have two separate purposes: the storage of intermediate results, and the configuration of the parameters of WMMA.

In contrast, the Julia API separates these in two types: `WMMA.Fragment`, as seen before, and a new type `WMMA.Config`. Instead of taking the resultant fragment by-reference, the Julia API just returns it. This makes the dataflow clearer, but it also means that the fragment's type cannot be used during dispatch. As such, there is a limited amount of information that cannot be inferred from the argument types, but that is nonetheless needed to select the correct WMMA instruction. This is accomplished using a separate parametrised type `WMMA.Config{M, N, K, T}`, where `(M, N, K)` represents the WMMA shape, and `T` represents the type used for accumulation.

Through the use of `WMMA.Config`, we can avoid having to declare the fragments upfront. We only need to create an instance of this parametrised type once, and give it as an argument to all WMMA functions. These WMMA functions are `@generated`, and dynamically determine the correct intrinsic to call dynamically, based on the types of the arguments. Because most information is baked into the argument types, we only need a limited number of WMMA functions in the high-level API. For example, rather than having a myriad of functions for loading WMMA matrices, the high-level API only has three: `WMMA.load_a`, `WMMA.load_b`, and `WMMA.load_c`.

As an illustration of the Julia WMMA API, consider the multiply accumulate of 16×16 matrices given in Listing 7. This code fragment is the Julia equivalent of the CUDA C++ version shown in Listing 3. Note that, contrary to CUDA C++, there is no need to define

```
function wmma_example(a, b, c, d)
    conf = WMMA.Config{16, 16, 16, Float32}

    a_frag = WMMA.load_a(pointer(a), 16, WMMA.ColMajor, conf)
    b_frag = WMMA.load_b(pointer(b), 16, WMMA.ColMajor, conf)
    c_frag = WMMA.load_c(pointer(c), 16, WMMA.ColMajor, conf)

    d_frag = WMMA.mma(a_frag, b_frag, c_frag, conf)

    WMMA.store_d(pointer(d), d_frag, 16, WMMA.ColMajor, conf)

    return
end
```

Listing 7: A mixed-precision $16 \times 16 \times 16$ matrix multiply accumulate using the WMMA API in Julia.

the fragments upfront. The only definition that is needed before calling the WMMA API, is a single `WMMA.Config` instantiation. Compared to CUDA C++, the dataflow is also a lot clearer. For example, we can immediately see that `a_frag` is the result of the call to `WMMA.load_a`.

Finally, we have one remaining requirement: implementing elementwise operations through Julia's dot syntax. In Julia, dot expressions are lowered to calls to the built-in `broadcast` function. This `broadcast` function is extensible through the multiple dispatch mechanism [27]. For example, the first step in `broadcast` is to convert the values involved in the broadcast to types that support indexing. This is implemented using a call to `broadcastable`, which can be specialised. In our case, we do not require any specific conversion, so we can define `broadcastable` as the identity function: `Base.broadcastable(frag::WMMA.Fragment) = frag`.

The next two steps are selecting an appropriate output array, and applying the broadcast in an efficient manner. In Julia's broadcast framework, these are defined simul-

taneously by a broadcast *style*. We thus define a new `WMMA.FragmentBroadcastStyle` specifically for WMMA fragments. Through the same multiple dispatch mechanism used for `broadcastable`, we are able to have this style take precedence over the style used for scalars, so that a scalar multiplication of a WMMA fragment is handled by our custom style. Similarly, by specialising `Base.BroadcastStyle(...)`, we can link `WMMA.Fragment`s to this new broadcast style.

Finally, `Base.copy` is called with the arguments of the broadcast operation, and the broadcast style. This method is responsible for implementing the broadcast operation itself. In our case, we specialise `Base.copy` for our broadcast style to provide a custom implementation of broadcast. First, we check if all arguments are either 1-dimensional (arrays) or 0-dimensional (scalars), since WMMA broadcasting only makes sense in those cases. Next, we apply the given operation elementwise, artificially extending all arguments to a 1-dimensional array. This extension is important in case one of the arguments was a scalar. Finally, we find the type of the first `WMMA.Fragment` used in the broadcast call, and return a new fragment of the same type, containing the broadcasted data.

With these changes in place, we can now use dot expressions on WMMA fragments. For example, to scale the D matrix in Listing 7 by 2, it suffices to add the following after Line 8: `d_frag = 2 .* d_frag`.

The resulting high-level WMMA API and LLVM wrappers were bundled in one pull request to CUDAnative, that has since been merged upstream[3].

4.4 Evaluation

In this section, we will evaluate the developed WMMA API based on the criteria given in Section 4.2. The support for elementwise operations is mostly a functional requirement that we have already discussed in Section 4.3.2. We do not repeat the functional

[3]Pull request available at `https://github.com/JuliaGPU/CUDAnative.jl/pull/494`.

aspects here, but we will look into the impact of elementwise operations on the zero-cost criterion.

4.4.1 Zero-cost

To evaluate the zero-cost requirement, we will look at the generated LLVM IR and PTX of the $16 \times 16 \times 16$ matrix multiply accumulate given in Listing 7. A shortened version of the LLVM IR corresponding to Listing 7 is given in Listing 8. The `extractvalue`s on Line 2 – 3 are generated by CUDAnative to extract the pointer to the arguments `a`, `b`, `c`, and `d`. Similarly, the `inttoptr` and `addrspacecast` on Lines 5 – 6 and Lines 23 – 24 are due to the way CUDAnative represents pointers: as integers in the generic address space.

The WMMA specific code is converted to five `@llvm.nvvm.wmma` instructions: 3 loads for A, B, and C, one `mma`, and one store for D. Additionally, a number of `extractvalue`s is generated after each `load` and the `mma`. These are required because the `load` and `mma` return an LLVM structure, whereas `mma` and `store.d` have the individual elements as arguments. Note though that no unneeded LLVM IR instructions are generated. In particular, the flattening and unflattening process is completely optimised away at the LLVM IR level.

The NVPTX backend generates the PTX shown in Listing 9. First, the base address of each matrix is loaded into registers on Lines 3 – 6. These addresses are converted from the generic address space to global addresses on Lines 8, 12, 15, and 25. These `cvta.to.global.u64` instructions correspond to LLVM's `addrspacecast`. The only other instructions are related to WMMA itself, i.e. the extra `extractvalue`s from the LLVM IR have disappeared, because they were only necessary to pass the correct types to the WMMA intrinsics.

Similar observations hold for the elementwise operations using Julia's dot syntax. For example, let us consider the case of the scaling of the D matrix using `d_frag = 5 .*`

```
define void @kernel({ [2 x i64], i64 }, ...) {
  %a.addr = extractvalue { [2 x i64], i64 } %0, 1
  ; similarly for B, C, D

  %a.ptr = inttoptr i64 %a.addr to i8*
  %a.asc = addrspacecast i8* %a.ptr to i8 addrspace(1)*
  %a.frag = call {<2 x half>, ...} @llvm...m16n16k16.load.a.col...(
    i8 addrspace(1)* %a.asc, i32 16)
  %a.frag.0 = extractvalue {<2 x half>, ...} %a.frag, 0
  ; ...
  %a.frag.7 = extractvalue {<2 x half>, ...} %a.frag, 7

  ; similarly for B and C

  %d.frag = call {float, ...} @llvm...m16n16k16.mma.col.col...(
    <2 x half> %a.frag.0, ..., <2 x half> %a.frag.7,
    <2 x half> %b.frag.0, ..., <2 x half> %b.frag.7,
    float %c.frag.0, ... float %c.frag.7)
  %d.frag.0 = extractvalue {float, ...} %d.frag, 0
  ; ...
  %d.frag.7 = extractvalue {float, ...} %d.frag, 7

  %d.ptr = inttoptr i64 %d.addr to i8*
  %d.asc = addrspacecast i8* %d.ptr to i8 addrspace(1)*
  call void @llvm...m16n16k16.store.d.col...(
    i8 addrspace(1)* %d.asc, float %d.frag.0, ..., float %d.frag.7,
    i32 16)

  ret void
}
```

Listing 8: The LLVM IR corresponding to the program given in Listing 7, as generated by Julia.

```
.visible .entry wmma_example(.param .align 8 .b8 a_array[24], ...)
{
    ld.param.u64  %rd1, [a_array+16];
    ld.param.u64  %rd2, [b_array+16];
    ld.param.u64  %rd3, [c_array+16];
    ld.param.u64  %rd4, [d_array+16];

    cvta.to.global.u64  %rd5, %rd1;
    mov.u32   %r1, 16;
    wmma.load.a.sync.aligned.col.m16n16k16.global.f16
        {%hh1, ..., %hh8}, [%rd5], %r1;
    cvta.to.global.u64  %rd6, %rd2;
    wmma.load.b.sync.aligned.col.m16n16k16.global.f16
        {%hh9, ..., %hh16}, [%rd6], %r1;
    cvta.to.global.u64  %rd7, %rd3;
    wmma.load.c.sync.aligned.col.m16n16k16.global.f32
        {%f1, ..., %f8}, [%rd7], %r1;

    wmma.mma.sync.aligned.col.col.m16n16k16.f32.f32
        {%f9, ..., %f16},
        {%hh1, .., %hh8},
        {%hh9, ..., %hh16},
        {%f1, ..., %f8};

    cvta.to.global.u64  %rd8, %rd4;
    wmma.store.d.sync.aligned.col.m16n16k16.global.f32 [%rd8],
        {%f9, ..., %f16}, %r1;
    ret;
}
```

Listing 9: A cleaned up version of the PTX generated by the LLVM NVPTX backend for the program given in Listing 7.

`d_frag`. For such a FP32 scaling, we only have a few extra `fmul float %..., 5.0e+0` instructions at the LLVM IR level, which get lowered to PTX's `mul.f32`.

The case of FP16 operations is a little more complicated. For FP16, Julia does not implement arithmetic operations such as multiplication and addition directly. Instead, the FP16 values are first converted to FP32. The operation is then run on these FP32 intermediate values, and the result is converted back to FP16[4]. While this is fine for CPUs, where FP32 and FP64 are the norm, this is suboptimal for GPUs, which have hardware support for FP16. Scaling an FP16 WMMA fragment still works, but extra instructions will be generated to implement the conversion to and from FP32.

Obviously, we cannot change this behaviour in Julia directly, as this will break code running on the CPU. The cleanest way to solve this would be with a Julia package called Cassette.jl [65]. Cassette.jl implements contextual dispatch, where each function call has an extra implicit argument, the context, that may be used during dispatch. Contextual dispatch allows override existing Julia functions with context-specific behaviour. We can use this mechanism to contextually override the implementation of FP16 multiplication on the GPU.

Cassette support for CUDAnative is already planned, since it could be used to provide custom implementations of built-in functions such as `sin`, but it is still a work-in-progress[5]. In the meantime, the same behaviour can be achieved by using inline PTX assembly that multiplies two FP16 values. Note that the NVPTX backend also uses this approach for CUDA C++ code that applies an elementwise scaling to the A or B fragments.

As an illustration, Listing 10 shows an example implementation using LLVM.jl, a Julia package that wraps the LLVM C APIs [11]. While overriding the built-in $*$ operator is possible, doing so is inadvisable because this will break CPU code. Instead, we define a new function `multiply_fp16`, which means we need to adapt our kernel accordingly.

[4]See the implementation of FP16 arithmetic at `https://github.com/JuliaLang/julia/blob/97e3fe88d7d3e4f748195a4250b34ed593419b56/base/float.jl#L397-L399`.

[5]See `https://github.com/JuliaGPU/CUDAnative.jl/pull/334`.

```
1  using LLVM
2
3  multiply_fp16(a::Float16, b::Float16) =
4      Base.bitcast(Float16,
5                   LLVM.Interop.@asmcall(
6                                         "{mul.f16 \$0,\$1,\$2;}",
7                                         "=h,h,h",
8                                         false,
9                                         Int16,
10                                        Tuple{Int16, Int16},
11                                        Base.bitcast(Int16, a),
12                                        Base.bitcast(Int16, b)
13                                       )
14                  )
```

Listing 10: Multiplying two half-precision floating point values using inline PTX.

That is, instead of `a_frag = Float16(5) .* a_frag`, we should now write `a_frag = multiply_fp16.(Float16(5), a_frag)`. LLVM.jl expects the input and output to be of type `Int16`, so Line 4, Line 11, and Line 12 `bitcast` the `Float16` values to or from `Int16`. Line 9 and Line 10 specify the return type, and argument types, respectively. The boolean `false` on Line 8 indicates that the inline assembly has no side-effects. Finally, the most important part of the `@asmcall` is the PTX template on Line 6, containing the call to the `mul.f16` instruction. Line 7 determines the constraints corresponding to the placeholders `$0`, `$1`, and `$2`: all of the placeholders are half-precision (`h`), and the first placeholder is an output rather than an input (`=`).

In general, we can state that the broadcasting abstractions themselves do not introduce superfluous instructions. However, the implementation of the underlying elementwise operation can result in extra instructions, at both the LLVM IR level, and the PTX level. In the case of FP16 operations, this can be remedied by using Cassette.jl's contextual dispatch mechanism, or by using inline PTX using LLVM.jl.

4.4.2 Future proof

The next criterion for the WMMA API is that it must be future proof. We have already discussed this during the implementation, but in this section we will evaluate the future-proofness in more detail. To accomplish this, we identify three different ways in which the Julia WMMA API may need to change in the future:

1. The names of the WMMA instrinsics of the NVPTX backend may change. For example, LLVM 6 and LLVM 8 use slightly different naming schemes for the WMMA intrinsics.

2. To support the increasing capabilities of Tensor Cores, WMMA can be extended with new datatypes. For example, Volta-style Tensor Cores only support FP16, but Tensor Cores of the Turing and Ampere generation support way more datatypes.

3. NVIDIA can add new shapes to WMMA. For example, the initial WMMA API only supported $16 \times 16 \times 16$ WMMA operations. With the launch of Turing, the list of supported shapes was extended to also include $8 \times 32 \times 16$ and $32 \times 8 \times 16$.

Ideally, the WMMA API in Julia should be able to handle these changes with minimal adaptations to the underlying implementation, or to user code that makes use of WMMA.

Recall from Section 4.3.1 that the LLVM intrinsics are wrapped in Julia functions. Because the latter do not change names, end-users do not need to change their code if the underlying LLVM intrinsics change. It suffices to adapt the logic that is used in CUDAnative to dynamically generate the names of the LLVM intrinsics. As such, the first option for changes to WMMA is covered.

The other two options are similar, in that they only add functionality. As a result, these do not introduce breaking changes to user code that relies on WMMA. Nevertheless, we still want to support these new shapes or types with minimal changes to CUDAnative.

This is solved through the use of the configuration variables. To add these new types and shapes, one should only need to extend these variables with the new functionality. This way, new wrappers will be generated for the corresponding LLVM intrinsics. Because the high-level WMMA functions are `@generated`, changing the configuration variables will extend the support for the new types and shapes to the higher level interface. Of course, adding support for new types may need additional changes to Julia's compiler as well. If the new datatype does not have an equivalent in Julia, it needs to be added to the Julia language before CUDAnative can support it.

4.4.3 Similar to CUDA C++

The last aspect of the evaluation is the similarity of the Julia WMMA API to that of CUDA C++. While the main idea of Julia's WMMA is the same as CUDA C++, there are a few notable deviations. These differences were introduced either to make the resulting API more readable, or to make it fit better in the Julia language. As an illustration, compare the implementation of the $16 \times 16 \times 16$ matrix multiply accumulate in Julia (Listing 7) and in CUDA C++ (Listing 3). In both versions, the basic steps of WMMA are the same. First, a fragment of the A, B, and C matrices is loaded. Next, a fragment of the D matrix is obtained by a `wmma.mma` operation. The resultant D fragment is then stored to memory. Despite the similarities, we observe a few cases where the Julia API deviates from CUDA C++:

- *Specification of the WMMA shape*: In CUDA C++, the WMMA shape must be defined for each fragment separately. This means that if the shape is changed later, this change needs to be propagated to all four WMMA fragments. The Julia API only specifies the WMMA shape once, through the `WMMA.Config` variable. This means that the Julia version is less cluttered, and also makes changing the WMMA shape less prone to errors, since the shape is defined in one location only.

- *Passing the resultant WMMA fragments*: The CUDA API takes all WMMA fragments as by-reference arguments, which obscures the data flow. Consider for

example the case of the multiply accumulate operation: `wmma::mma_sync(frag_1, frag_2, frag_3, frag_4)`. It is not immediately obvious which of the arguments will be changed after the call. In the Julia API, fragments that are changed are not taken as arguments, but are instead returned by the function. The call above would thus be represented as `frag_1 = WMMA.mma(frag_2, frag_3, frag_4)`. The syntax of the assignment statement makes it immediately clear that `frag_1` contains the resultant WMMA fragment.

- *Specification of the data layout*: WMMA fragments may be stored as row major or column major. This memory layout must be specified when issuing a WMMA load or store. In CUDA C++, this is done in an inconsistent manner. For A and B, the layout must be specified in the type of the fragment. The accumulator matrices C and D have a different approach. In these cases, the memory layout is specified as an extra argument of the call to the load or store. Additionally, when specifying the data layout using the fragment type, `wmma::col_major` and `wmma::row_major` must be used, whereas the argument method requires `wmma::mem_col_major` or `wmma::mem_row_major`. These two forms of inconsistency make using WMMA in CUDA C++ more difficult without referring to the documentation. In Julia, both these inconsistencies are eliminated. All memory layouts are specified as an extra argument `WMMA.ColMajor` or `WMMA.RowMajor` to the load or store functions.

- *Verbosity*: CUDA C++ requires fragments to be defined upfront, which makes the resulting code more verbose than necessary. In the sample $16 \times 16 \times 16$ matrix multiply accumulate of Listing 3, these definitions constitute almost half of the WMMA-specific lines of source code. The verbosity is increased even further because the types used in these declarations have long names themselves. In contrast, the Julia API does not require declaring the fragments upfront. There is also no need to fully specify the WMMA fragment type.

- *Purpose of WMMA fragments*: In CUDA C++, the purpose of WMMA fragments is twofold. On the one hand, they are temporary arrays to store the intermediate per-thread results when using WMMA. On the other hand, their type contains information, such as the WMMA shape, that is used during overload resolution to

select the correct WMMA variant. The Julia API separates this in two different concepts. A `WMMA.Fragment` stores the intermediate results, and the `WMMA.Config` type is used to configure the parameters to WMMA.

- *Indexing WMMA fragments*: Finally, CUDA's and Julia's version of WMMA differ in the way fragments can be indexed. Both allow elementwise access to a fragment's data using the `x` member. The Julia API extends this by implementing the indexing interface for `WMMA.Fragment`s, so that one may write `frag[i]` instead of `frag.x[i]`. In the Julia standard library, functions that operate on arrays are written to make use of these generic indexing interfaces. As long as a type supports the necessary interfaces, it can be passed directly to these functions. By supporting the indexing interfaces for `WMMA.Fragment`s, we can thus reuse functionality in Julia's standard library or its packages.

For the aforementioned reasons, we argue that the underlying differences improve readability, and make the API fit better into a high-level language such as Julia. Despite these differences, the resulting API is still close enough to the original version of WMMA, so that programmers coming from CUDA C++ should have no trouble familiarising themselves with the Julia API.

4.5 Conclusion

In this chapter, we have developed a Julia WMMA API to interface with NVIDIA's Tensor Cores. The resulting abstractions are inspired by CUDA's version of WMMA, but depart from it to increase readability, or to better fit into the Julia programming language. The WMMA abstractions also support the most common operation, elementwise transformations, through Julia's special dot syntax. The developed WMMA API implementation has been merged into CUDAnative, and will allow us to wite a mixed precision GEMM in a later chapter.

5 Abstractions for recursive blocking

In the previous chapter, we implemented a WMMA interface to access Tensor Cores from within Julia. To calculate a matrix product $C = A \cdot B$ using Tensor Cores, we can subdivide the C matrix in submatrices of size 16×16 so that we can use WMMA. We can then calculate these submatrices in parallel by calculating each of them in one warp. Unfortunately, this simple implementation does not achieve high performance.

The subject of this chapter is the design of abstractions for recursive blocking. These techniques are used to increase performance in GEMM kernels, by improving the temporal locality of the implementation. Section 5.1 makes the case for these tiling abstractions by discussing in which contexts they are used. In Section 5.2, we will describe the non-functional requirements of the API, and list which operations must be supported. Section 5.3 describes the tiling API for Julia that we developed. Finally, we will evaluate the resulting API in Section 5.4.

5.1 The case for recursive blocking

Consider the in-place matrix multiply accumulate operation $C := AB + C$ of $N \times N$ square matrices. The most basic implementation of this operation is the triple loop nest shown in Listing 11. The outer two loops iterate over the rows i and columns j of the output matrix C. Each element `C[i, j]` is then calculated as the inner product of the ith row of A and the jth column of B using another loop over the k dimension.

```
for i = 1 : N
    for j = 1 : N
        for k = 1 : N
            C[i, j] += A[i, k] * B[k, j]
        end
    end
end
```

Listing 11: A naïve implementation of the in-place matrix multiply-accumulate operation $C := AB + C$.

This approach is inefficient due to the large working sets required to store slices of the A and B matrices. For example, calculating the first row of C requires the first row of A and the entire B matrix, as illustrated in Figure 5.1. If this row of A does not fit in the cache, it needs to be fetched for every element in the first row of C. Similarly, once we reach the second row of C, we need the entire B matrix once again. If N is large, chances are that a large part of B has already been evicted from the cache. As such, this naïve algorithm will only run as fast as global memory, which is much slower than the computational capabilities of the GPU.

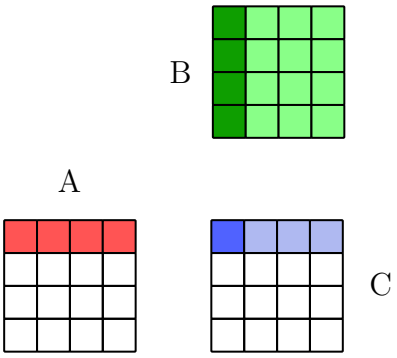


Figure 5.1: An illustration of the triple loop nest approach to GEMM.

In order to improve performance, we can exploit the data reuse property of GEMM. A matrix multiplication of square matrices of dimension N requires $\mathcal{O}(N^3)$ floating point operations, and $\mathcal{O}(N^2)$ storage. This results in each element being reused roughly $\mathcal{O}(N)$ times. This data reuse can be exploited by improving the temporal locality of the

implementation. That is, we want memory accesses to the same location to be as close as possible in time.

If N is small, the algorithm has perfect temporal locality, since each of the matrices fit in the cache. For large N, we have to rewrite the GEMM as a set of matrix products of smaller size. We can then choose the size of these matrices to be small enough such that they fit in a part of the memory hierarchy. On the CPU, the L1, L2, and L3 caches are used, whereas on the GPU, we can reuse data in shared memory. There are two main ways this can be accomplished: explicit blocking techniques, and cache-oblivious algorithms. Both are designed to make use of the memory hierarchy, but do so in different ways.

Cache-oblivious algorithms, like Strassen, do not rely on the cache size being known at compile time [21]. As such, they can be expected to perform well on machines with memory hierarchies of different size. Typically, they operate on the principle of divide-and-conquer, where the problem is recursively subdivided in subproblems of smaller size. For example, Strassen recursively subdivides an $N \times N$ matrix in four submatrices of size $\frac{N}{2} \times \frac{N}{2}$. Eventually, these submatrices will become small enough to fit in a level in the memory hierarchy. Typically, the runtime of these cache-oblivious algorithms has large constant factors, so they are only efficient for large matrices.

In contrast to cache-oblivious algorithms, explicit blocking techniques have the cache size as a parameter to the GEMM. The tiling size is chosen in such a way that tiles of the A, B, and C matrices fit in the relevant parts of the memory hierarchy. Explicit blocking applied to a GEMM on the GPU is illustrated in Figure 5.2. First, tiles of A and B are loaded from global memory to shared memory. These tiles are then multiplied, resulting in an intermediate tile of C in shared memory. This process is repeated for all the tiles of A and B that contribute to a given C tile. The contributions of all these A and B tiles are accumulated together, and the resultant C tile is stored back to global memory.

Note that this only calculates a single tile of the C matrix. We need to repeat this process for every tile to obtain the entire C matrix. The computations needed for each of these tiles do not depend on any of the other tiles of C. As such, we can perform

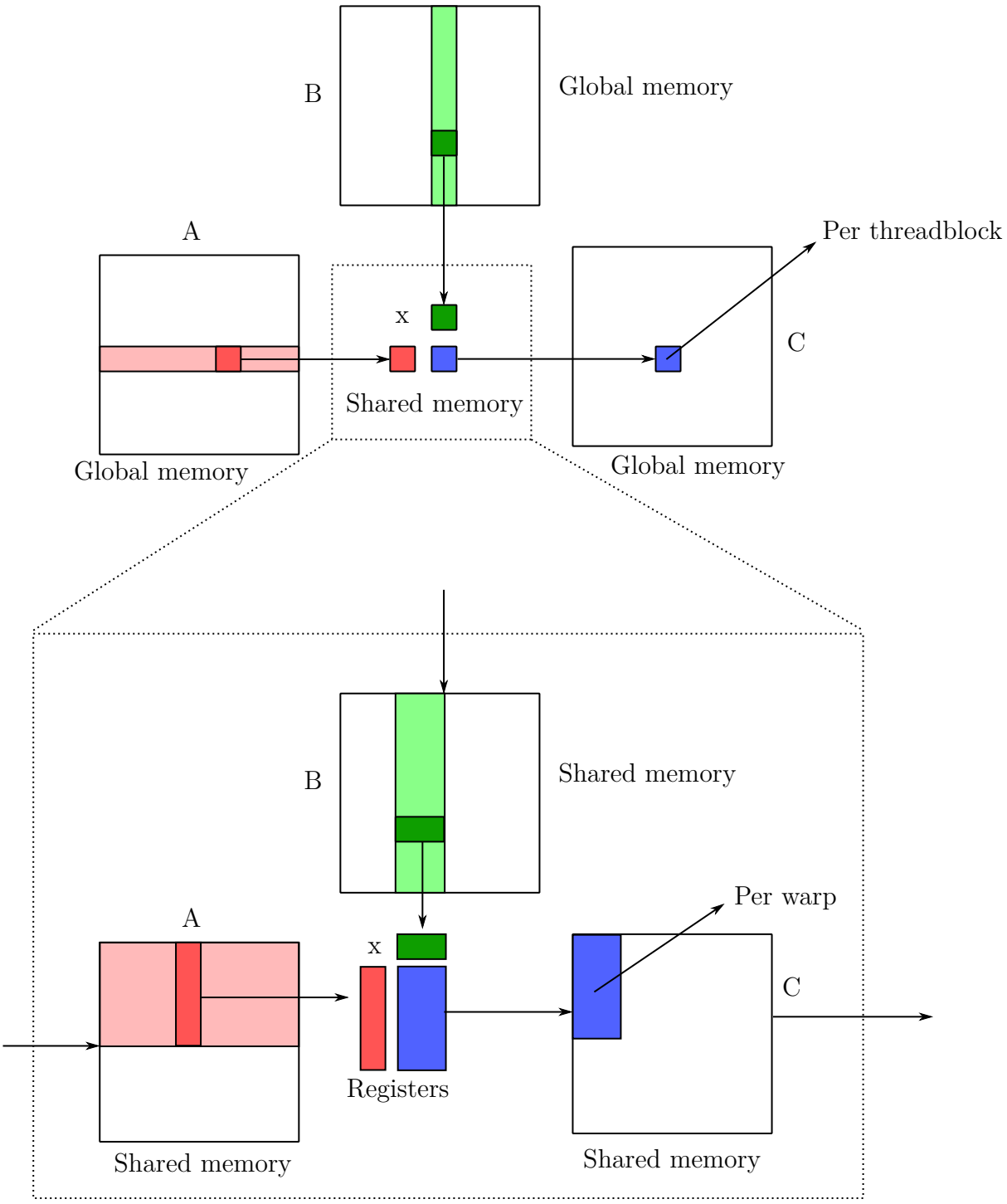


Figure 5.2: An illustration of the blocking approach applied to the GEMM problem on GPUs.

the computations for different tiles in parallel. Shared memory is inherently linked to threadblocks, since only threads in the same threadblock may communicate via shared memory. Consequently, we assign each of these tiles to one threadblock, and launch a kernel with a sufficient number of threadblocks to cover the entire C matrix.

This subdivision process is then repeated for the next level of the thread and memory hierarchy, as illustrated on the bottom of Figure 5.2. Tiles of A and B are loaded from shared memory into registers, multiplied, and stored back to shared memory. These tiles are computed in parallel by a set of warps in one threadblock.

Explicit blocking techniques typically perform well for a variety of different matrix sizes, but performance heavily depends on the values of these parameters. Because the optimal parameter values are linked to the properties of the underlying hardware, explicit blocking is less portable than cache-oblivious algorithms. In the context of GPU GEMMs, explicit blocking is sometimes also referred to as shared memory or register blocking, depending on the relevant part of the memory hierarchy. Another term that is used frequently in literature is double-sided recursive blocking. Recursive refers to the fact that this tiling is performed for each level of the memory hierarchy. The term double-sided indicates that the optimal tiling parameters differ for the memory operations and the computation stage of a GEMM. Both of NVIDIA's implementations of performant GEMMs, cuBLAS and CUTLASS, make use of this technique [51, 55]. Similarly, most implementations of GEMM for GPUs in literature use explicit blocking [48, 3, 35, 75].

Batched GEMM, which calculates many small matrix multiplications in parallel, also uses tiling techniques in each batch [2, 4]. Furthermore, tensor contractions also rely on blocking techniques for performance [54, 33]. This can be either explicitly in kernels built for a specific tensor contraction, or implicitly in approaches built on top of GEMM, such as GETT. In general, any algorithm that can be made more cache efficient by processing the input in blocks, benefits from tiling. Given the multitude of different applications, a tiling API could thus prove useful. Such a tiling API can improve the readability of kernels making use of it. Instead of performing the necessary address manipulations manually each time, this can be handled transparently by the code underneath the API.

5.2 Requirements and design of abstractions

To make the resulting tiling API as useful as possible, we identify three important design criteria:

1. *Readability*: The main goal of the tiling API is to make writing kernels that use blocking techniques easier. At a high level, these kernels are typically expressed in terms of high-level operations on tiles, such as subdividing a tile in tiles of a smaller size. In the implementation phase, programmers typically have to convert these high-level operations to manual calculations of address offsets, which is hard to read and prone to errors. Consider the example of Listing 12, which corresponds to the first step in a GEMM kernel: copying a tile of the C matrix from global memory to shared memory. Note that the memory addresses in this example are calculated manually. Without reverse engineering these address calculations, it is not obvious what this code does. Our tiling API will replace these manual calculations with high-level operations on tiles, and perform the necessary address computations behind the scenes.

2. *Zero-cost*: Using the abstractions should incur no performance cost, compared to manually calculating the addresses.

3. *Support for multiple dimensions*: Since our focus is GEMM, we will mainly be using two dimensional tiles, as each matrix has two dimensions. Note though that the underlying iteration space of GEMM has three dimensions. The first and second dimension correspond to the rows and columns of the resultant C matrix, and together specify an element of C. Each element of the C matrix is calculated as a dot product, which involves iterating over the elements in a row of A and a column of B. This iteration can be regarded as a third dimension in the iteration space of GEMM. As such, support for multiple dimensions is necessary to implement a GEMM.

There is also a second reason to support multiple dimensions. The tiling API can be used for problems related to GEMM as well. For example, batched GEMM calculates a batch of small matrix products in parallel. Thus, batched GEMM needs an extra dimension compared to regular GEMM, corresponding to individual matrices in the batch. Tensor contractions extend GEMM even further, to any arbitrary amount of dimensions. In order for the tiling API to be applicable to many different applications, it should work for any arbitrary amount of dimensions.

In this thesis, we are mainly interested in matrix multiplication kernels. To that end, the tiling API will need to support the operations needed to implement a performant GEMM kernel. In particular, we propose 4 different operations on tiles: projection, parallelisation, translation, and linearisation. During our extensive literature study, we found no comparable approaches.

Projection Recall that a GEMM calculates an expression of the form $C := A \cdot B + C$, where A is an $M \times K$ matrix, B is a $K \times N$ matrix, and C is an $M \times N$ matrix. The compute stages of GEMM will use tiles that refer to the three dimensional iteration space (M, N, K). In the memory stages of GEMM, we typically only need two of these dimensions. For example, to load a slice of the A matrix, we are only interested in the M and K dimension. Projecting a tile reduces its dimensionality by dropping one or more of its dimensions, as shown in Figure 5.3. The projection abstraction thus allows us to easily reduce the original three dimensional tile to a tile containing only the relevant dimensions.

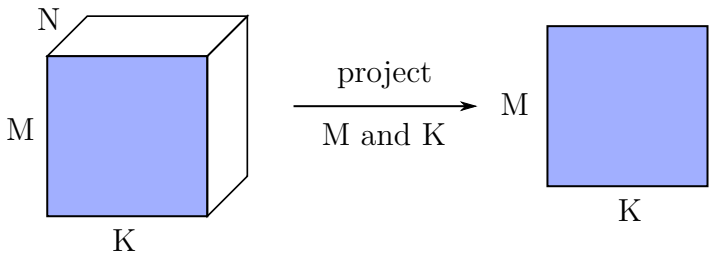


Figure 5.3: An illustration of the projection of a three dimensional tile to two dimensions M and K.

```
@unroll for warp_offset = 0 : WARPS_PER_BLOCK : (BLOCK_M * BLOCK_N) ÷ (MEM_CD_WARP.M * MEM_CD_WARP.N) - 1
  NUM_WARP_ROWS = BLOCK_M ÷ MEM_CD_WARP.M

  base_warp_i = (warpId % NUM_WARP_ROWS) * MEM_CD_WARP.M
  base_warp_j = (warpId ÷ NUM_WARP_ROWS) * MEM_CD_WARP.N
  warp_i = (warp_offset % NUM_WARP_ROWS) * MEM_CD_WARP.M
  warp_j = (warp_offset ÷ NUM_WARP_ROWS) * MEM_CD_WARP.N

  @unroll for thread_offset = 0 : 32 : (MEM_CD_WARP.M * MEM_CD_WARP.N) ÷ (MEM_CD_THREAD.M * MEM_CD_THREAD.N) - 1
    NUM_THREAD_ROWS = MEM_CD_WARP.M ÷ MEM_CD_THREAD.M

    base_thread_i = (laneId % NUM_THREAD_ROWS) * MEM_CD_THREAD.M
    base_thread_j = (laneId ÷ NUM_THREAD_ROWS) * MEM_CD_THREAD.N
    thread_i = (thread_offset % NUM_THREAD_ROWS) * MEM_CD_THREAD.M
    thread_j = (thread_offset ÷ NUM_THREAD_ROWS) * MEM_CD_THREAD.N

    global_linear_base   = (block_i + base_warp_j + base_thread_j) * global_M + (block_j + base_warp_i + base_thread_i)
    global_linear_offset = (warp_j + thread_j) * global_M + (warp_i + thread_i)
    shared_linear_base   = (base_warp_j + base_thread_j) * shared_M + (base_warp_i + base_thread_i)
    shared_linear_offset = (warp_j + thread_j) * shared_M + (warp_i + thread_i)

    global_ptr = pointer(global_c, global_linear_base)
    shared_ptr = pointer(shared_c, shared_linear_base)

    # Load at address global_ptr, with offset global_linear_offset
    # Store at address shared_ptr, with offset shared_linear_offset
  end
end
```

Listing 12: Implementing the first step in GEMM using manual calculation of addresses.

Parallelisation Parallelisation is the most important operation of the tiling API. It corresponds to the recursive subdivision of tiles in smaller tiles, and the subsequent parallelisation of the resulting subtiles over a set of collaborating entities, such as threadblocks or warps. Consider the example parallelisation operation shown in Figure 5.4. A tile of size $4M \times 4N$ is divided in subtiles, each of size $M \times N$. These subtiles are handled in parallel by a set of 8 cooperating warps, indicated by the numbers $0 - 7$ in each subtile. Note that the set of all cooperating warps do not need to cover the entire tile. Indeed, in this example, there are 16 subtiles but only 8 warps. This means that each warp will handle 2 of these 16 subtiles. This parallelisation can be applied recursively, by dividing each of these subtiles into sub-subtiles, where each sub-subtile is handled by one thread.

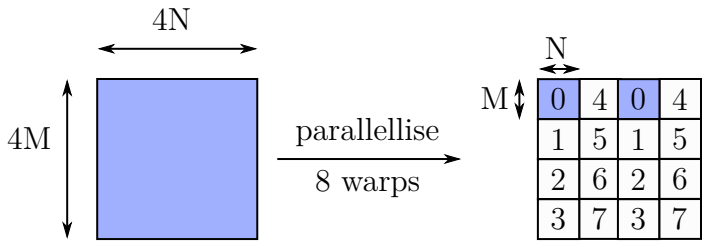


Figure 5.4: An illustration of the parallelisation of a tile over a set of 8 cooperating warps.

Translation The third operation is translation. As the name suggests, translation moves a tile over a specified distance in each dimension. In the example of Figure 5.5, a two dimensional tile is moved over a distance m in the M dimension, and a distance n in the N dimension. The translation operation is useful in cases where the reference point of a tile needs to be changed. For example, consider a tile referring to a submatrix stored in global memory. The coordinates of this tile are specified relative to the first element in the first row of the parent matrix in global memory. To copy this submatrix to shared memory, we need to express the tile relative to the first element stored in shared memory, which may be different. To accomplish this, we can simply translate the tile over the correct distance.

Linearisation The last supported operation is linearisation, which is used to convert a tile's location from a cartesian index to a linear index. This conversion is needed to

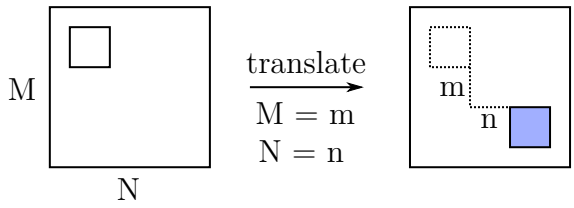


Figure 5.5: An illustration of the translation of a tile.

calculate the offset of a tile in memory, relative to the base pointer of the parent tile. In the example of Figure 5.6, we consider a subtile at a cartesian offset of (m, n) from its parent tile which has size (M, N). Linearisation results in the linear offset of this tile, relative to the top-left corner of the parent tile. Note that the linearisation process assumes that the matrix is stored in column major ordering, as this is the convention that Julia uses. In this case, we need to span n columns of M elements each, and an additional m elements to reach the subtile. This corresponds to a linear index of $nM + m$.

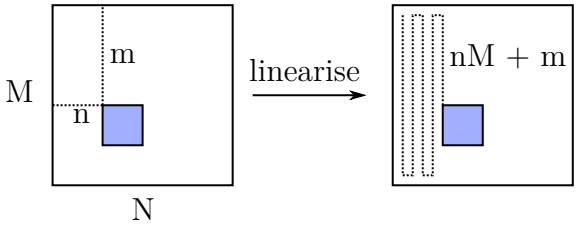


Figure 5.6: An illustration of the conversion of a tile to a linear index.

5.3 A tiling API for Julia

A tile is fully determined by its position and its size. Our tiling API contains a `Tile` struct that stores this information. One could store the size as a field in this struct, but this poses a problem. In the implementation of GEMM, we use these tiles to copy slices of a matrix from shared memory to registers. In order for the compiler to know how many registers are needed, the size of the tile needs to be available at compile time. In Julia, this can be done using the package StaticArrays, which contains a set of statically sized arrays [15]. The term statically sized refers to the fact that the size of

each of these arrays can be determined from their type, which means it is available to the compiler after type inference.

Now suppose we define a function `load_tile(mat, tile::Tile)` that, given a matrix `mat` and a `Tile`, returns a `StaticArray` that contains the data stored in `mat` at the positions corresponding to that tile. The size of this returned `StaticArray`, and hence its type, will depend on the size of the tile that is passed as an argument. If we store the size of a tile as a field, the return type of this `load_tile` function will not be inferable from the types of the arguments `mat` and `tile`. Instead, this return type depends on the *value* of the arguments passed to `load_tile`. This will make type inference in the Julia compiler fail, which will result in dynamic type checks at runtime. In Julia parlance, this is referred to as *type instability* [27]. In order to avoid type instability in this `load_tile` function, we have to ensure that the return type is inferable from the types of the arguments `mat` and `tile`. We can accomplish this by mimicking `StaticArrays`, and storing the size in the `Tile` type. `Tile` will thus become a parametrised type, where one of the type parameters is the size of the tile.

The position of an N-dimensional tile is a tuple of N elements, and can be represented in Julia as an instance of the built-in type `NTuple`. One disadvantage of using `NTuple`s is that they are indexed by a number. If `pos` is an `NTuple` that represents the position of a tile, then we can obtain the position along the first dimension as `pos[1]`. This inherently imposes an ordering of the dimensions of the tile.

If we want to implement GEMM using tiling, we typically do not think in terms of the first or second dimension of a tile. Instead, a tile that represents a slice of the A matrix of size $M \times K$ has an M-dimension and a K-dimension. Rather than writing the position as `pos[1]`, we can increase readability by naming the dimensions, so that we may write `pos.M`. Julia includes a special type for this purpose, called `NamedTuple`. As their name suggests, `NamedTuple`s are tuple-like collections of values, where each value is not associated with a numerical index, but with a unique name. To improve the readability of the tiling API, we will use `NamedTuple`s to store both the position and size of a tile.

```
struct Tile{size, names, T}
    base::NamedTuple{names, T}
    offset::NamedTuple{names, T}
end
```

Listing 13: The definition of a Tile in the Julia tiling API.

Before we started working on the tiling API, we had already implemented a proof-of-concept performant GEMM implementation. In that implementation, we calculated memory offsets manually, but we noticed that a lot of superfluous instructions were being generated. Upon closer inspection, it turned out that these were the result of the address calculations. These calculations involve expressions that are quite complex, and hence introduce overhead in the form of extra arithmetic instructions.

In order to eliminate these extra instructions, we had to split up each expression as a sum of two parts. The first part contained the base of the memory address, and depended on values only known at runtime, such as the identifier of the currently executing thread. The second part contained offsets from this base address, which were constants known at compile time. Due to this splitting, the compiler was able to eliminate the superfluous instructions. The base part of the addresses could be calculated once, at the start of the kernel, and stored in a register. To perform a load or store at a specified offset from this base address, the compiler used the register + constant addressing mode, eliminating the extraneous arithmetic operations.

To avoid these superfluous instructions when using the tiling API, we will not store the position of the tile as one field. Instead, the position is split up in a base index and offset from that base. The final definition of the parametrised `Tile` type is shown in Listing 13. Note that the base and offset are separated, and that both are stored as a `NamedTuple`. The `Tile` type is statically sized by including the size in the type parameters. `NamedTuple`s are themselves parametrised types with two type parameters `names`, which contains the name associated with each entry in the tuple, and `T`, which determines the type of each entry. These two type parameters are included in the definition of `Tile` as well.

Now that we have defined a `Tile` datastructure, we need to implement the four operations that we need for the GEMM kernel. The implementation for the translate operation is fairly simple. We define the function `translate(tile, dist)` that returns a new tile with the same size and offset, but where the base is the elementwise sum of the original tile's base and the argument `dist`. This essentially moves the multi-dimensional tile over the distance specified by the argument.

The `linearise(coord, dimensions)` function converts a cartesian index to a linear index. The first argument `coord` represents the coordinate of the tile. Note that we do not take the tile itself as an argument, so that `linearise` can be used for both the base and offset of a tile. Instead of having a separate `linearise` function for base and offset, we may simply write `linearise(tile.base, ...)` and `linearise(tile.offset, ...)`. The second argument `dimensions` represents the size of the parent tile. To convert the cartesian index to a linear index, we use the `LinearIndices` type from the Julia standard library. This way, we can both reuse functionality, and ensure the linearise operation works for any arbitrary amount of dimensions.

To project tiles, we could define another function `project(tile, dims)`, where `dims` contains a list of the dimensions we want to keep. A projection of a tile to the M and K dimension could then be written as `project(tile, (:M, :N))`. We will take a slightly different approach by making use of Julia's extensibility. In Julia, the syntactic construct `a.b` is converted to a call to `Base.getproperty(a, :b)` [27]. Through the multiple dispatch mechanism, we can thus override this with custom behaviour. We will use this to write our projection operation as `tile.MN` instead of `project(tile, (:M, :N))`.

Listing 14 shows a part of the implementation of the projection operation. As mentioned previously, the construct `tile.MN` is first converted to the call `Base.getproperty(tile, :MN)`. The type of the second argument, `:MN`, is a `Symbol`, indicated by the colon prefix. `Symbol`s are similar to strings, except that they are immutable and only one copy of each distinct value is stored [27]. The `Base.getproperty` function is specialised for arguments of type `Tile` on Line 1. The value of the `sym` argument of this function determines the name of the field that was accessed. To generate custom projection implementations for

each sets of dimensions, we want to dispatch on the *value* `:MN` of this argument, rather than its *type* `Symbol`. To do this, we can use Julia's `Val` type, a parametric type with one type parameter. When we call the constructor of `Val` as `Val(sym)`, a new instance of `Val` is created where the type parameter is set to `sym`. This essentially moves the value of `sym` to the type domain, so that we may use the multiple dispatch mechanism. After creating a `Val` type, we dispatch to another function `getproperty_impl` that implements the projection itself.

To make the abstraction zero-cost, we use `@generated` functions that generate custom code depending on the argument types, as shown on Line 3 of Listing 14. Since we moved the name of the field to the type domain, we can generate a different implementation for each possible projection operation. First note that accesses to the base or offset of a tile using `tile.base` or `tile.offset` also get converted to calls to `Base.getproperty`. We handle this case in Lines 4 – 7 by checking whether the symbol passed is either `base` or `offset`. If so, we just return the value of the field by calling `getfield` on Line 7. Recall that `@generated` functions must return an `Expr`, that is subsequently compiled. To easily create an `Expr` corresponding to a block of Julia code, we can surround the block with the `quote ... end` construct, as shown in Lines 6 – 8.

The projection itself is implemented in Lines 10 – 22. First, we convert the symbol representing the field name to a `String` on Line 11. Line 12 then converts this symbol to a tuple containing the individual dimensions. For example, if `sym` is `:MN`, then `sym_str` and `new_names` are `"MN"` and `('M', 'N')`, respectively. In Lines 16 – 18, an `Expr` is generated to create new `NamedTuple`s that only contain the relevant dimensions for the base, offset, and size. Finally, Line 21 wraps these newly generated `NamedTuple`s in the `Tile` struct that represents the project tile, and returns that tile.

The parallelise operation is exposed to the user as a function call with four arguments: `parallelise(tile, tiling_size, index, count)`. The `tile` argument has type `Tile` and is the parent tile that will be subdivided and parallelised over a set of entities. These entities can be a set of blocks, warps, or threads that cooperate. The second argument, `tiling_size`, determines the tile size that each entity will handle, and the last argument `count` refers to the number of cooperating entities. Finally, the argument `index` is an

```
@inline Base.getproperty(tile::Tile{size, names, T}, sym::Symbol) where {size, names, T} = getproperty_impl(tile, Val(sym))

@generated function getproperty_impl(tile::Tile{size, names, T}, ::Val{sym}) where {size, names, T, sym}
    if sym == :base || sym == :offset
        # standard fields
        return quote
            getfield(tile, sym)
        end
    else
        # tile projection
        sym_str = String(sym)
        new_names = ntuple(i -> Symbol(sym_str[i]), length(sym_str))

        return quote
            # create new NamedTuples with the correct dimensions
            new_base   = ...
            new_offset = ...
            new_size   = ...

            # return projected tile
            return Tile{new_size, new_names, ...}(new_base, new_offset)
        end
    end
end
```

Listing 14: An overview of the implementation of tile projection in the Julia tiling API.

integer from 0 to `count - 1`, and determines the identifier of the currently executing entity.

Consider the example parallelisation given in Figure 5.7. This operation starts with a parent tile of size $4m \times 2n$, divides it in subtiles of size $m \times n$, and parallelises the subtiles across 2 warps. The number 0 or 1 in each subtile indicates which warp is responsible for it. We may write this parallelisation operation as `parallelise(Tile(M = 4 * m, N = n), Tile(M = m, N = n), warpId, 2)`, where `warpId` is either 0 or 1, corresponding to the identifier of the currently executing warp.

To make the parallelisation operation generalisable to multiple dimensions, we again reuse the indexing functionality from Julia's standard library. The information needed for iteration is then stored in a new struct, a `TileIterator`, that is returned by the `paralellise` function. Julia allows us to write customised implementations for iterating over user-defined types. For loops are converted to calls to the `Base.iterate` function, which may be specialised for our own types. In order to iterate over `TileIterator`s using a for loop, we must thus specialise the `Base.iterate` method for `TileIterator`s. `Base.iterate` is called for each iteration of the for loop, and must return the value associated with each iteration. In the case of `TileIterator`s, each call to `Base.iterate` will return a `Tile` corresponding to the tile of that iteration.

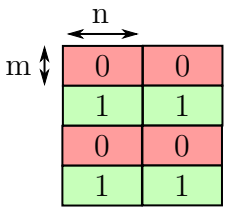


Figure 5.7: A parallelisation operation over 2 warps handling a 4×2 set of subtiles in parallel.

5.4 Evaluation

This section evaluates the Julia tiling API according to the criteria we gave in Section 5.2: the support for multiple dimensions, readability, and zero-cost. For our discussion, we will use three different examples. Given that our main interest is matrix multiplication kernels, each example will correspond to a different step in a performant implementation of GEMM. From our discussion in Section 5.1, we identify the following steps for a GEMM $D = A \cdot B + C$, where A is an $M \times K$ matrix, B is a $K \times N$ matrix, and C and D are $M \times N$ matrices:

1. Copy a tile of C from global memory to shared memory, cooperatively by all threads in a block.

2. Copy a tile of C from shared memory to registers, cooperatively by all threads in a warp.

3. Iterate over the K dimension, according to the tiling size of a block.

    3.1. Copy a tile of A from global memory to shared memory, cooperatively by all threads in a block.

    3.2. Copy a tile of B from global memory to shared memory, cooperatively by all threads in a block.

    3.3. Iterate over the K dimension, according to the tiling size of a warp.

        3.3.1. Copy a tile of A from shared memory to registers, cooperatively by all threads in a warp.

        3.3.2. Copy a tile of B from shared memory to registers, cooperatively by all threads in a warp.

        3.3.3. Compute a tile of D, given the A, B, and C tiles, cooperatively by all threads in a warp.

4. Copy a tile of D from registers to shared memory, cooperatively by all threads in a warp.

5. Copy a tile of D from shared memory to global memory, cooperatively by all threads in a block.

We will pick three steps from this process to illustrate and evaluate the tiling API: copying a tile of C from global memory to shared memory (step 1), the computation of the matrix product in the inner loop (step 3.3), and copying a tile of D from registers to shared memory (step 4).

Note that these three examples cover the entire GEMM, as the other steps are just variants of the steps we selected. For example, the only difference between step 1 and step 5 is the direction in which memory is copied: either global-to-shared, or shared-to-global. Similarly, step 2 and step 4 are the same, if we ignore the direction of the copy operation. Finally, step 3.1 and step 3.2 are the same as step 2, except for the matrix that is being copied, and possibly the data type.

5.4.1 Multiple dimensions

In order to support multiple dimensions, our tiling API reuses functionality from Julia's standard library. All operations on `Tile`s are built on top of interfaces from Julia that work for any arbitrary number of dimensions. For example, the position and size of each `Tile` is stored using Julia's `NamedTuple`s, which support any amount of dimensions. Similarly, the parallelisation and linearisation operations, which involve computations using multidimensional indices, are written using Julia's generic indexing interfaces.

Our set of three examples cover the case of tiles with two and three dimensions. Step 1 (copying C from global to shared memory) and step 4 (copying D from registers to shared memory) are memory operations, and only use the two dimensions of the matrix that is being copied. Step 3.3 (the matrix product itself) is a computation step, and

uses all three dimensions M, N, and K. Given that our main focus is GEMM, most of the discussions of the other criteria will be limited to the two or three dimensional case. Nevertheless, we argue that similar observations should hold for the case of more than three dimensions. Julia's indexing interfaces are extensively used in the standard library itself, and in its package ecosystem. One of these packages is TiledIteration.jl, which aims to facilitate writing cache-efficient algorithms in Julia [20]. It includes functionality to iterate over disjoint tiles of a larger array. Like our tiling API, TiledIteration.jl supports multiple dimensions by building on top of Julia's indexing interfaces. To date, no issues regarding dimensionality have been reported. As such, we can reasonably assume that the conclusions we draw for our tiling API in the 2D and 3D case are generalisable to tiles of higher dimensionality as well.

One may wonder why we are not using TiledIteration.jl instead of designing a new tiling API. The reason is that TiledIteration.jl is a simple package, mainly targeted towards code running on the CPU. TiledIteration.jl is not sufficient for our use case, as it does not support several GPU-specific operations, such as parallelisation across threadblocks and warps.

5.4.2 Readability and zero-cost

We will evaluate the readability and zero-cost together, using the three different examples we have given before. Given our end goal of a mixed-precision matrix multiplication, we will mainly focus on a GEMM using WMMA. For each of these examples, we first explain at a high level which operations must be performed, and implement them using our tiling API in Julia. Finally, we will study the generated PTX to ensure that our zero-cost criterion is met.

Example 1: Copying a tile of C from global to shared memory

To copy a tile of C from global to shared memory, we will follow the approach that is illustrated in Figure 5.8. The corresponding implementation using our tiling API is given in Listing 15. Each block will copy a separate tile, and we will launch the GEMM kernel with enough blocks to fully cover the C matrix. The size of each of these tiles is determined by the `block_tile` variable. Note that `block_tile` initially has three dimensions, so we first project it to the M and N dimension using `block_tile.MN` on Line 1 in Listing 15.

Next, we divide `block_tile` in subtiles, and parallelise the resulting `warp_tile`s over a set of cooperating warps in the block. This parallelisation operation is shown in Line 1 on Listing 15. Note that we are using the `@unroll` macro from the Julia package GPUifyLoops.jl [13], which informs LLVM to fully unroll the loop. Each of these `warp_tile`s has size `(M = MEM_CD_WARP.M, N = MEM_CD_WARP.N)`.

Similarly, we parallelise this `warp_tile` over the set of 32 threads in a warp. The corresponding parallelisation operation is shown on Line 2. The `laneId` variable is an integer 0 – 31 that identifies the threads within a warp. Each thread handles a tile of size `(M = MEM_CD_THREAD.M, N = MEM_CD_THREAD.N)` in each iteration. In the case of an FP32 C matrix, the best choice is `MEM_CD_THREAD.M = 4`, and `MEM_CD_THREAD.N = 1`. This way, each thread loads/stores 4 adjacent FP32 elements, so that we can issue one 128-bit load/store, the largest memory transaction size supported by the GPU. This optimisation reduces the total number of instructions compared to issuing four separate 32-bit loads/stores, and is referred to as *vectorisation of memory accesses*.

Note that the positions of all these tiles are specified relative to the top-left corner of the current block's tile. This means that `thread_tile.index == (M = 0, N = 0)` corresponds to a linear index of 0. Because shared memory only stores the tile of the current block, this is the correct index for shared memory. For global memory, we need to offset this tile depending on the currently executing block. To accomplish this, we translate this `thread_tile` over the correct distance on Line 3. Finally, Lines 5–9 convert the base and

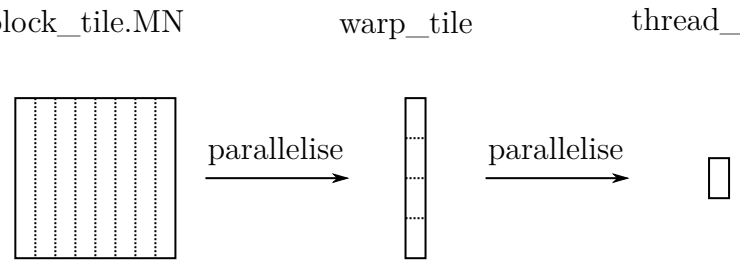


Figure 5.8: An illustration of copying a tile of the C matrix from global memory to shared memory.

```
@unroll for warp_tile = parallelise(block_tile.MN, Tile(MEM_CD_WARP), warpId,
↪  WARPS_PER_BLOCK)
  @unroll for thread_tile = parallelise(warp_tile, Tile(MEM_CD_THREAD),
  ↪  laneId, 32)
    global_thread_tile = translate(thread_tile, (M = block_i, N = block_j))

    global_linear_base   = linearise(global_thread_tile.base, (M = global_M,
    ↪  N = global_N))
    global_linear_offset = linearise(global_thread_tile.offset, (M = global_M,
    ↪  N = global_N))

    shared_linear_base   = linearise(thread_tile.base, (M = shared_M, N =
    ↪  shared_N))
    shared_linear_offset = linearise(thread_tile.offset, (M = shared_M, N =
    ↪  shared_N))

    global_ptr = pointer(global_c, global_linear_base)
    shared_ptr = pointer(shared_c, shared_linear_base)

    # Load at address global_ptr, with offset global_linear_offset
    # Store at address shared_ptr, with offset shared_linear_offset
  end
end
```

Listing 15: Copying a tile of the C matrix from global to shared memory using our tiling API.

offset of each of these `thread_tile`s to a linear index. We can then create a pointer to the correct memory location on Lines 11–12, and perform the load or store. To separate the constant parts of the memory addresses, we create a pointer using the linearised base, and only add the linearised offset afterwards.

This example corresponds to Listing 12, which we used to illustrate the readability criterion in Section 5.2. For clarity, we have repeated Listing 12 as Listing 16. The outer for loop on Lines 1 – 7 corresponds to the first parallelisation operation, and the inner for loop on Lines 9 – 15 is the equivalent of the second parallelisation operation. In the body of both loops, the bases and offsets of tiles are calculated manually. Lines 17 – 20 convert the bases and offsets to linear indices, and are thus the equivalent of the linearise operations in our tiling API. Note that the translation operation is handled by the addition of the translation offsets `block_i` and `block_j` on Line 17. Compared to Listing 16, the implementation using our tiling API in Listing 15 is significantly less verbose. Additionally, our tiling API increases the readability and maintainability by replacing the manual address calculations with high-level operations on tiles, such as parallelisation and linearisation.

The corresponding PTX of Listing 15 is shown in Listing 17. First, the base addresses for each thread are calculated in the registers `%rd20` and `%rd13` for shared and global memory, respectively. The loads and stores are performed using vectorised memory instructions, indicated by the suffix `v4.f32`. The stores to shared memory are shown on Lines 7, 13, and 21. Because the size of shared memory is known at compile time, the offset from this base address can be resolved to a constant. This allows the compiler to use the register plus constant addressing mode for the shared memory stores. In contrast, the size of the matrix in global memory is not known by the compiler. This means that the linearised offset is not known at compile time, even though the offsets in the M and N dimensions are constants. To calculate the address in global memory, LLVM emits a multiplication (using a bitshift `shl.b64`), and an addition. In general, we can see that no superfluous instructions are generated, for both the loads from global memory and the stores to shared memory.

```
@unroll for warp_offset = 0 : WARPS_PER_BLOCK : (BLOCK_M * BLOCK_N) ÷ (MEM_CD_WARP.M * MEM_CD_WARP.N) - 1
  NUM_WARP_ROWS = BLOCK_M ÷ MEM_CD_WARP.M

  base_warp_i = (warpId % NUM_WARP_ROWS) * MEM_CD_WARP.M
  base_warp_j = (warpId ÷ NUM_WARP_ROWS) * MEM_CD_WARP.N
  warp_i = (warp_offset % NUM_WARP_ROWS) * MEM_CD_WARP.M
  warp_j = (warp_offset ÷ NUM_WARP_ROWS) * MEM_CD_WARP.N

  @unroll for thread_offset = 0 : 32 : (MEM_CD_WARP.M * MEM_CD_WARP.N) ÷ (MEM_CD_THREAD.M * MEM_CD_THREAD.N) - 1
    NUM_THREAD_ROWS = MEM_CD_WARP.M ÷ MEM_CD_THREAD.M

    base_thread_i = (laneId % NUM_THREAD_ROWS) * MEM_CD_THREAD.M
    base_thread_j = (laneId ÷ NUM_THREAD_ROWS) * MEM_CD_THREAD.N
    thread_i = (thread_offset % NUM_THREAD_ROWS) * MEM_CD_THREAD.M
    thread_j = (thread_offset ÷ NUM_THREAD_ROWS) * MEM_CD_THREAD.N

    global_linear_base   = (block_i + base_warp_j + base_thread_j) * global_M + (block_j + base_warp_i + base_thread_i)
    global_linear_offset = (warp_j + thread_j) * global_M + (warp_i + thread_i)
    shared_linear_base   = (base_warp_j + base_thread_j) * shared_M + (base_warp_i + base_thread_i)
    shared_linear_offset = (warp_j + thread_j) * shared_M + (warp_i + thread_i)

    global_ptr = pointer(global_c, global_linear_base)
    shared_ptr = pointer(shared_c, shared_linear_base)

    # Load at address global_ptr, with offset global_linear_offset
    # Store at address shared_ptr, with offset shared_linear_offset
  end
end
```

Listing 16: Implementing the first step in GEMM using manual calculation of addresses (repeated from Listing 12).

```
// Calculate the base addresses in %rd13 and %rd20...

shl.b64                 %rd22, %rd13, 5;
add.s64                 %rd23, %rd17, %rd22;
cvta.to.global.u64      %rd24, %rd23;
ld.global.v4.f32        {%f5, %f6, %f7, %f8}, [%rd24];
st.shared.v4.f32        [%rd20+4096], {%f5, %f6, %f7, %f8};

shl.b64                 %rd25, %rd13, 6;
add.s64                 %rd26, %rd17, %rd25;
cvta.to.global.u64      %rd27, %rd26;
ld.global.v4.f32        {%f9, %f10, %f11, %f12}, [%rd27];
st.shared.v4.f32        [%rd20+8192], {%f9, %f10, %f11, %f12};

// ...

mul.lo.s64              %rd64, %rd13, 480;
add.s64                 %rd65, %rd17, %rd64;
cvta.to.global.u64      %rd66, %rd65;
ld.global.v4.f32        {%f61, %f62, %f63, %f64}, [%rd66];
st.shared.v4.f32        [%rd20+61440], {%f61, %f62, %f63, %f64};
```

Listing 17: The PTX generated by the NVPTX backend for the Julia code given in Listing 15.

Example 2: Computation of the matrix product

To implement the computation of the matrix product in the inner loop using the tiling API, we will follow the approach illustrated in Figure 5.9. A `block_tile` represents the three dimensional iteration space (M, N, K) used to calculate the tile of the D matrix corresponding to one block. The M and N dimensions are shown vertically and horizontally, respectively. The K dimension is represented as perpendicular to the page. Let us consider the case where a `block_tile` has size $(M, N, K) = (128, 128, 16)$. This means that each block calculates an $M \times N = 128 \times 128$ part of D, by multiplying all $M \times K = 128 \times 16$ tiles of A in the same row, and all $K \times N = 16 \times 128$ tiles of B in the same column. This results in a set of D-tiles, which are subsequently accumulated by summing over the K dimension.

We want to parallelise this computation over all warps in a block. In the example of Figure 5.10, each block contains 8 warps, in a 2×4 arrangement. Each warp calculates a 64×32 tile of D in each iteration, by multiplying a 64×16 tile of A, and a 16×32 tile of B. Of course, we want tiles in this three dimensional space with the same M and N indices to be mapped to the same warp, so that we can accumulate across the K dimension. Recall that our tiling API is column major. In our case, this means that the warps are assigned to tiles in the order of the M, N, and K dimension. We can thus simply use a parallelisation operation of size $(M, N, K) = (64, 32, 16)$ across 8 warps, as shown in Figure 5.10. The 8 warps fully cover the M and N dimensions, as indicated by the 0 – 7 in each tile. In the next iteration, we have advanced along the K dimension, but the division along the M and N dimension is the same. This means that with this choice of tiling size, the parallelisation operation implicitly iterates over the K dimension.

This parallelisation operation is shown on Line 1 of Listing 18, and returns a three dimensional `warp_tile`. To compute the matrix product using WMMA, we first need to load the A and B tiles into WMMA fragments. To load A, we are only interested in the M and K dimension, so we first project `warp_tile` on Line 3. This results in a tile of size $(M, K) = (64, 16)$, which thus consists of four 16×16 WMMA fragments. To load the WMMA fragments, we first translate the tile in the M dimension over 0, 16, 32, and

48 elements on Line 3. Lines 5–6 then convert this translated base and offset to a linear index, which can then be used to create the pointer argument to `WMMA.load_a` on Line 8. Lines 11–18 do the same thing for the B matrix: the `warp_tile` is projected to the K and N dimensions, translated, and converted to a linear index. Finally, Lines 20 – 24 calculate the 64×32 product of D using the `WMMA.mma` function.

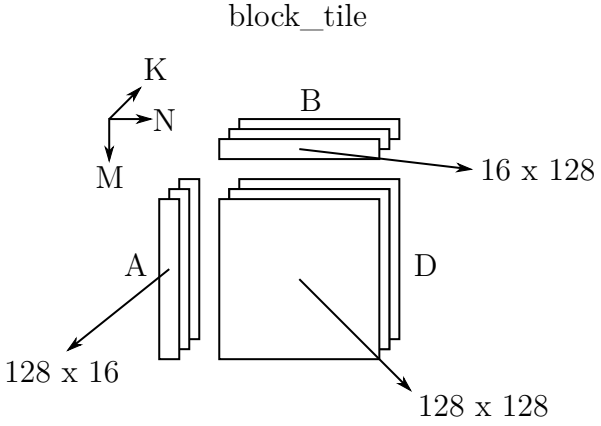


Figure 5.9: An illustration of the three dimensional iteration space in the inner loop of the matrix product.

At the PTX level, we again observe the same behaviour as in the previous example: the base addresses of A and B for each warp are calculated once, and stored in registers. The code in Listing 18 is converted to a set of `wmma.load.a`, `wmma.load.b`, and `wmma.mma` instructions. The addresses of the load operations are expressed as a constant offset from the base addresses stored in registers. This once again indicates that the tiling abstractions do not introduce any superfluous instructions.

Example 3: Copying a tile of D from registers to shared memory

In the previous example, we studied the calculation of the matrix product in the inner loop of GEMM. After this step, each warp has a part of the D matrix stored in WMMA fragments. To store these WMMA fragments to shared memory, we follow the approach illustrated in Figure 5.11. `block_tile` represents the same tile as in example 2, i.e. the three dimensional iteration space used to calculate a tile of the D matrix corresponding

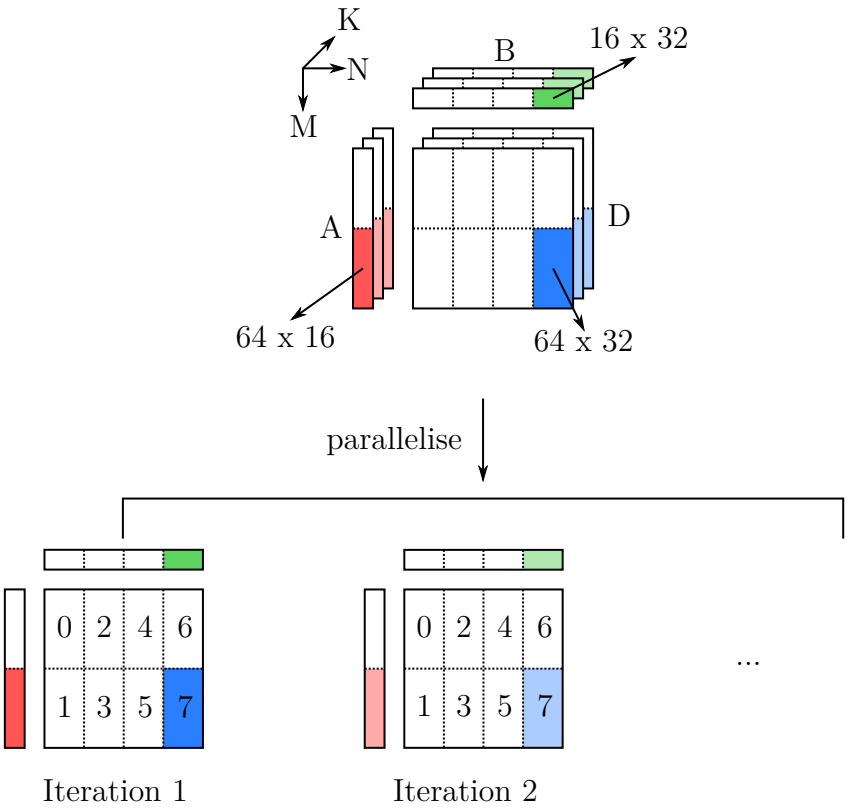


Figure 5.10: An illustration of the computation of the matrix product in the innermost loop.

```
@unroll for warp_tile = parallelise(block_tile, Tile(M = 64, N = 32, K = 16),
↪  warpId, 8)
  @unroll for i = 1 : 4
    a_tile = translate(warp_tile.MK, (M = (i-1)*16, K = 0))

    linear_base   = linearise(a_tile.base, ...)
    linear_offset = linearise(a_tile.offset, ...)

    a_frags[i] = WMMA.load_a(...)
  end

  @unroll for j = 1 : 2
    b_tile = translate(warp_tile.KN, (K = 0, N = (j-1)*16))

    linear_base   = linearise(b_tile.base, ...)
    linear_offset = linearise(b_tile.offset, ...)

    b_frags[j] = WMMA.load_b(...)
  end

  @unroll for i = 1 : 4
    @unroll for j = 1 : 2
      acc_frags[i, j] = WMMA.mma(a_frags[i], b_frags[j], acc_frags[i, j],
      ↪  ...)
    end
  end
end
```

Listing 18: The computation of the matrix product, implemented using our tiling API.

to one block. To copy D, we are only interested in the M and N dimension, so we project this tile to these dimensions first.

Next, we parallelise this tile over a set of warps. This parallelisation should have the same parameters as the matrix computation in the previous example. Obviously, the tiling size is only specified in the M and N dimension, instead of in the in the three dimensions M, N, and K. Figure 5.11 uses the same tiling sizes as our previous example: `block_tile` is a 128×128 matrix, and is parallelised across 8 warps, each handling a 64×32 subtile. The corresponding parallelisation operation returns a `warp_tile`, and is shown on Line 1 of Listing 19. Note that the for loop of Line 1 only has 1 iteration in this case, since 8 warps fully cover the entire `block_tile`.

Finally, this `warp_tile` is divided in a 4×2 arrangement of WMMA fragments, like in example 2. The for loops on Line 2 and Line 3 iterate over these 8 WMMA fragments. Line 4 then translates the tile in the M and N dimension over 0, 16, 32, or 48 elements to obtain the final tile corresponding to each WMMA fragment. Line 6 and Line 7 then convert this cartesian index to a linear index, so that it may be used to create pointers for the WMMA `store.d` on Line 9.

To conclude this example, we will study the PTX that is generated for Listing 19. The PTX is very similar to the previous two examples: first, the base address of D for each warp is calculated and stored in a register. After this computation, 8 `wmma.store.d` instructions are emitted, which use this register as a base address, and constant offsets. Once again, we conclude that the use of the tiling API does not introduce any extra overhead.

Summary

We have proposed four operations on tiles in our tiling API: parallelisation, projection, translation, and linearisation. In this section, we have demonstrated how these operations can be combined to implement more complex behaviour. To that end, we selected three different steps in a mixed-precision GEMM using WMMA. Using the tiling API for each

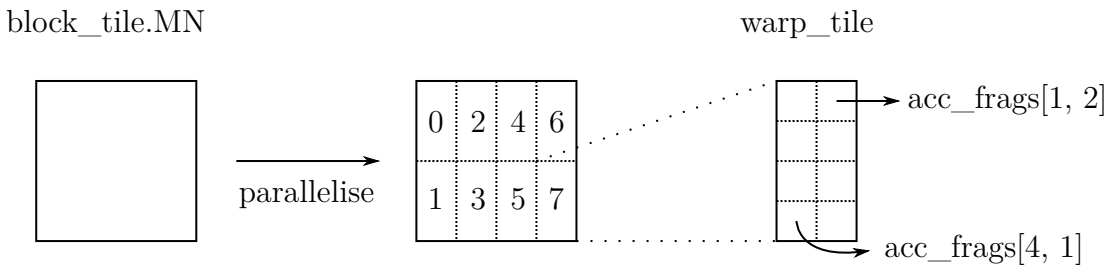


Figure 5.11: An illustration of copying a tile of the D matrix from registers to shared memory.

```
@unroll for warp_tile = parallelise(block_tile.MN, Tile(COMPUTE_WARP).MN,
  ↪  warpId, WARPS_PER_BLOCK)
  @unroll for i = 1 : 4
    @unroll for j = 1 : 2
      tile = translate(warp_tile, (M = (i-1)*16, N = (j-1)*16))

      linear_base   = linearise(tile.base, ...)
      linear_offset = linearise(tile.offset, ...)

      WMMA.store_d(..., acc_frags[i, j], ...)
    end
  end
end
```

Listing 19: Copying a tile of the D matrix from registers to shared memory using our tiling API.

of these significantly improves readability, compared to performing the necessary address calculations manually. For each of these examples, we also demonstrated that the tiling API does not introduce any superfluous instructions. Because each tile stores the base and offset separately, the compiler was able to calculate the base address once, store it in a register, and then use the register + offset addressing mode.

To determine which part of an address is a constant, and thus part of the offset, we make a few assumptions. For example, we assume that the parallelisation operation is fully unrolled through the use of `@unroll`. This way, values that are dependent on the iteration index are constants. A possible improvement in our tiling API would be to let the compiler handle this separation of base and offset instead of doing it manually. There is an optimisation in LLVM, the so-called `SeparateConstOffsetFromGEP` pass, that does this. Unfortunately, `SeparateConstOffsetFromGEP` did not work when we tested it, because of the way CUDAnative stores pointers. These pointers are stored in the generic address space, and converted to the global/shared address space just before the load/store. This conversion is performed using LLVM's `addrspacecast` instruction, which is treated as a black box in most LLVM optimisations. CUDAnative has recently changed the way pointers are stored, thus eliminating the need of the extra `addrspacecast`. More future work is needed to ascertain that this eliminates the need for the manual separation of base and offset, but our initial testing showed promising results.

5.5 Conclusion

In this chapter, we have developed an API for recursive blocking. This technique is important to improve performance in several kernels, such as normal GEMMs, batched GEMMs, and tensor contractions. While our main goal was applying the blocking techniques for the specific case of GEMM, the tiling API is designed with these other applications in mind. We illustrated how the tiling API can be used to implement a performant mixed-precision GEMM kernel using WMMA. In the next chapter, we will discuss GEMM in more depth, and introduce the necessary flexibility in the kernel.

The tiling API and the GEMM API of the next chapter were bundled in one pull request to CUDAnative, that is currently under review[1]. The tiling API part of this pull request consists of 300 lines of Julia source code.

[1]Pull request available at `https://github.com/JuliaGPU/CUDAnative.jl/pull/629`.

6 Abstractions for flexible matrix multiplication kernels

In Chapter 5, we designed tiling abstractions that can be used to implement a performant GEMM kernel. In this chapter, we will introduce the necessary flexibility in this GEMM, so that users can instantiate a wide variety of GEMM variants. In Chapter 3, we already indicated that the open source CUTLASS project was a source of inspiration for this thesis. Section 6.1 will discuss the relevant aspects of CUTLASS in more detail. In Section 6.2, we list the criteria we will use to evaluate our GEMM API. The main content is Section 6.3, which discusses the design and implementation of the GEMM API we developed. Section 6.4 illustrates this GEMM API using an example. Finally, Section 6.5 evaluates the resulting API using the criteria given in Section 6.2.

6.1 CUTLASS

The de facto standard for GEMM on GPUs is NVIDIA's cuBLAS library, containing a set of GEMM kernels written in SASS assembly. cuBLAS works well if the needs of the application are covered by the kernels implemented in the library. If a certain operation is not supported by cuBLAS, application developers typically have to implement the required kernels from scratch. In the case of custom elementwise operations, one could run another kernel after cuBLAS, but this results in the same data being loaded twice: once for the GEMM in cuBLAS, and once for the kernel that performs the elementwise operation.

NVIDIA's open-source CUTLASS project takes a different approach. Rather than having monolithic GEMM kernels, CUTLASS contains a set of CUDA C++ template classes that together implement a GEMM. These classes are organised according to CUDA's execution hierarchy: CUTLASS contains templates at the thread, warp, block, and device level. Each of these templates can be specialised for custom computations, tiling sizes, element data types, and memory layouts. Thanks to this modular design, these flexible components can be reused as building blocks to implement custom GEMM or GEMM-like kernels.

The use of templates in CUTLASS not only increases flexibility, but also allows the compiler to optimise better. For example, instead of passing tiling sizes as arguments to functions, CUTLASS stores the tiling size as template arguments. This allows the compiler to store thread-local arrays in the register file, since their size is known at compile-tile. Similarly, most loops can be unrolled, since the number of iterations is known by the compiler.

The aim of the CUTLASS project is to provide a set of reusable templates to implement GEMM or GEMM-like algorithms. For that reason, CUTLASS contains a lot of different interacting components. NVIDIA annually organises a technical conference called GTC (GPU Technology Conference). In GTC 2018 and GTC 2019, one of CUTLASS's developers presented two talks that provide an excellent overview of the structure of the project [29, 30]. Since we cannot cover all the different components of CUTLASS in this thesis, we will limit our discussion to the components that are most relevant for our use case.

CUTLASS includes generic algorithms that iterate over tiles of matrices with constant size. The *tile iterator* abstraction determines how exactly this iteration is performed. Not all of the algorithms in CUTLASS require the same operations on tile iterators. For that reason, CUTLASS's tile iterators can implement one or more *concepts*. Each concept specifies a set of operations that tile iterators satisfying this concept must implement. For example, *readable tile iterators* must define a `load` operation that loads a part of the matrix from memory. *Contiguous memory tile iterators* implement a `add_pointer_offset` function, that adds an offset to the memory address that the iterator represents.

The tile iterator templates additionally have arguments that can be used to customise the behaviour of the iterator. The *thread map* argument determines which thread is responsible for each matrix element. Each concrete thread map must define a function `initial_offset` that, given the identifier of the current thread, returns the initial position of the iterator for that thread. The thread map also specifies the number of iterations for each thread, and the offset from the initial position that should be used in each iteration. We can thus think of the thread map as a generalisation of the parallelisation operation in our tiling API.

The initial position and offsets returned by thread maps are expressed as a multidimensional index in a logical coordinate system. Tile iterators have another argument, called the *layout*, that converts this logical index to a memory address. The purpose of the layout components in CUTLASS is twofold. First, they contain functions that map logical indices of tiles to physical offsets in memory. For example, a logical index (i, j) in an $M \times N$ matrix will be converted to a physical offset $jM + i$ for a column-major layout, and to $iN + j$ for a row-major layout. The second purpose of layouts is to define a type system that can be used to specialise other CUTLASS components. Consider the case of a component that calculates a matrix product using WMMA. Recall from Chapter 4 that the WMMA instructions differ depending on whether the matrices are stored in row-major or column-major format. CUTLASS's WMMA templates thus have layouts as template arguments to determine the memory layout of the matrices A, B, and C.

These tile iterators, thread maps, and layouts are used to efficiently move tiles of the input matrices through the GPU memory hierarchy. Another abstraction that is relevant for our purposes, is the *transform* component. Transforms are functions that are used for the global-to-shared memory stream of the A and B matrices. Transforms are applied after a tile is loaded from global memory, but before it is stored to shared memory. This component was initially introduced to implement an 8-bit GEMM, but having a functor built into the global-to-shared memory stream is useful for other cases as well. For example, one could store the matrix in global memory using a reduced precision to save capacity. Using a transform, this lower precision datatype can then be converted to a higher precision to increase the accuracy of the matrix product.

The purpose of the abstractions discussed before is to copy tiles of the input matrices from global to shared memory, and from shared memory into registers. These abstractions are used in the components that implement the GEMM computation itself. CUTLASS's GEMM components are ordered according to the CUDA execution hierarchy, as discussed in Section 2.1. The matrix multiplication for the thread-level GEMM components is performed per thread, and the inputs are stored in each thread's registers. Components at the warp-level first load tiles from shared memory to registers using tile iterators, and then compute the matrix product. Similarly, threadblock-level components use tile iterators to load tiles from global memory into shared memory.

All of these GEMM components are heavily specialised C++ templates. Some of the template arguments determine the class and shape of the matrix multiplication operation. For example, in the case of a warp-level WMMA operation, the class would be "WMMA Tensor Operation", and the shape would be $16 \times 16 \times 16$. These templates are also specialised on the element type, memory layout, and tile iterator corresponding to each of the A, B, and C matrices. The CUTLASS source code contains the necessary specialisations for a range of GEMM variants, including FP32 and FP64 GEMMs using traditional FPUs, and mixed-precision GEMMs using Tensor Cores.

After the computation step, each threadblock has calculated a tile of the resultant matrix. This tile is distributed over the registers of all threads in a block, so it still needs to be written back to global memory. During the computation phase, the mapping of threads to elements was chosen to maximise the performance of the matrix computation. This mapping does not necessarily lead to optimal access patterns if we were to update global memory directly. The *epilogue* component in CUTLASS is the last phase of the matrix multiplication kernel. Its purpose is to update global memory efficiently. In the default implementation of CUTLASS's epilogue, the threads in one block first store their part of the result matrix to shared memory. This tile in shared memory is then copied to global memory, cooperatively by all threads in the block. This indirection through shared memory allows CUTLASS to use a different mapping of threads to matrix elements, so that global memory can be updated more efficiently.

Custom epilogues can override this with custom logic to update global memory. For example, one could implement a custom epilogue that adds a bias to the matrix multiplication result. This epilogue would load the bias vector from global memory, and add it to the matrix multiplication result before writing it back to global memory. Epilogues are also the phase in CUTLASS where custom elementwise operations on the resultant matrix can be fused in the GEMM kernel. CUTLASS includes a set of epilogues for common elementwise operations such as scaling and converting the matrix elements from one type to another.

The set of C++ templates we have described only constitute a part of the CUTLASS project, called the CUTLASS Template Library. This part of CUTLASS serves as an inspiration for abstractions in our GEMM API. It also gives us an idea of possible optimisations that we can apply to improve the performance of our GEMM kernels. The CUTLASS project also contains two other parts that are interesting for our purposes: the CUTLASS Instance Library, and the CUTLASS Profiler.

The CUTLASS Instance Library contains a set of scripts that instantiate and compile GEMM kernels using the CUTLASS Template Library. These kernels are instantiated with a wide range of different tiling sizes, data types, memory layouts, and operations. The CUTLASS Profiler is a command-line application that can load the kernels in the Instance Library, execute them, and evaluate their performance. Some of the command-line arguments of the profiler specify which kernel types should be run, so that we may apply a filter to the set of all kernels in the Instance Library. For example, the `--op_class=wmma_tensorop` argument indicates that we are only interested in GEMM kernels targeting Tensor Cores using the WMMA API. Other arguments can be used to change the tiling sizes for each step in the GEMM, such as the threadblock shape.

The CUTLASS profiler is interesting for our use case for two reasons. First, the CUTLASS profiler allows us to easily quantify the influence of different parameters of the GEMM on the performance of the kernel. Secondly, by running the profiler with the same parameters as our GEMM kernel, we have an excellent benchmark to compare our implementation with.

6.2 Requirements

Before designing and implementing our GEMM API in Julia, we should impose a set of requirements on the end result. We identify three important requirements for our GEMM API:

1. *Flexibility*: Since the main goal of this thesis is to build flexible GEMMs, flexibility is the most important criterion for our API. The cuBLAS library contains a set of monolithic kernels, whose behaviour cannot be adapted by the programmer. To increase flexibility, our GEMM API should split the GEMM kernel in separate components that can be customised by the user.

2. *Performance*: Of course, flexibility is of no benefit if the resulting kernel performs poorly. The performance of GEMM kernels that are built using our API should be on-par with the state-of-the-art implementations, such as cuBLAS or CUTLASS.

3. *Portability*: To ensure that the GEMM kernels built using our API are able to perform well on a wide range of devices, we should keep portability in mind. This means that we should make as few assumptions about the underlying hardware as possible. For example, the size of shared memory depends on the GPU, so our API needs to be able to handle these differing sizes. Similarly, only GPUs of the Volta, Turing, and Ampere generation include Tensor Cores to accelerate the computation of matrix multiplications. While our main focus is on GEMM using Tensor Cores, our API should also allow the matrix product to be calculated using traditional FPUs.

6.3 A flexible GEMM API for Julia

CUTLASS's main aim is to provide a set of pre-built components that can be combined to build custom kernels. While these components are used in CUTLASS to implement a

variety of different GEMMs using the explicit blocking technique, they are reusable for other applications as well. For example, Huang et al. have implemented a performant family of Strassen-like matrix multiplication kernels that are built on top of the abstractions provided by CUTLASS [21]. The large set of components in CUTLASS is useful if one wants to write custom kernels that operate on matrices. However, for use cases where tweaking a pre-built GEMM kernel is sufficient, it introduces unnecessary complexity.

This thesis has a different goal. Our main interest is GEMM kernels that can be customised by the user, instead of a set of components that can be used to build GEMM or GEMM-like kernels. To accomplish this, we implement the general structure of a performant GEMM kernel beforehand. To introduce the necessary flexibility, we split this GEMM in a small set of different building blocks having a predetermined interface. Each of these building blocks corresponds to a way in which GEMM kernels need to be adapted, and can have different implementations depending on the use case. For example, one building block could determine how the A matrix is stored in global memory. Specific implementations of this building block would include a column major and a row major storage format.

In our Julia GEMM API, each of these building blocks corresponds to one or more calls to a set of functions with a predetermined name. Using Julia's multiple dispatch mechanism, we can customise the behaviour of these functions depending on the types of their arguments. For example, consider the building block that determines the memory layout of the A matrix. We can define two types `RowMajor` and `ColumnMajor` that represent possible implementations. This building block can have functions `load(layout_type, tile, ...)` that loads a tile of A, and `store(layout_type, tile, ...)` that stores a tile. The first argument `layout_type` is a type, and can be either `RowMajor` or `ColumnMajor` in our example. Using the multiple dispatch mechanism, we can have different implementations of this building block, depending on which type is passed for `layout_type`.

The major strength of this scheme is that this flexibility has no impact at run time. Through type inference, the Julia compiler is able to infer the types passed to each of these functions, so that it is known at compile time which implementation of a building block will be called. This same process is repeated for all the different building blocks,

which are subsequently combined. The resulting code is then compiled just-in-time to a kernel tailored to the needs of the user.

The most important question we should answer is which building blocks we should introduce in the GEMM, so that they can cover a range of different use cases. CUTLASS already gives us some inspiration for possible abstractions, as described in Section 6.1. To get a better idea of the required level of flexibility, we also conducted a literature search to list the most common ways in which GEMM kernels need to be adapted. From these two sources, we propose five different building blocks: params, layouts, transforms, operators, and epilogues. In what follows, we will explain each of these building blocks, and list some use cases for each of them. Where relevant, we will also compare our design with that of CUTLASS.

Recall that CUTLASS includes tile iterators that can implement one or more concepts depending on which operations they support. The elements in each tile are then manually mapped to individual threads using the thread map template argument. A thread map specifies the initial offset of each thread, the number of iterations for each dimension of the tile, and the offset for each iteration. Our focus is on pre-built GEMM kernels that are flexible, so we will take a different approach. We will implement the general structure of a GEMM kernel using the tiling API we designed in Chapter 5. Users of our GEMM API do not need to specify a thread map manually. Instead, our GEMM kernel will call the parallelise operation on tiles, which implicitly maps matrix elements to threads.

Of course, we still want the user to be able to customise the tiling size of each step of the GEMM kernel. This is the purpose of the *params* abstraction. This abstraction is essentially a structure that is passed to the kernel, and contains a set of configuration fields. Some of these fields determine the tiling sizes, and others specify the launch configuration of the kernel, such as the number of warps per block. Note that the user does not need to specify all fields manually. We have implemented a set of heuristics that choose reasonable defaults for fields that are not set explicitly. For example, if the tiling size per threadblock is not set, we choose the largest square ($N \times N$) or nearly-square ($2N \times N$) tile that still fits in shared memory. For the time being, these heuristics

are mainly aimed at GEMMs using Tensor Cores, but future work could expand these heuristics to other cases as well.

Like CUTLASS, the positions of tiles in our tiling API are expressed in logical coordinates. To convert these logical coordinates to offsets in physical memory, we introduce another abstraction, called *layouts*. This abstraction corresponds to three functions that can be customised using Julia's multiple dispatch. The `size(layout_type, logical_size)` function determines the size in physical memory of the layout for a given size in logical coordinates. This physical size is not necessarily the same as the size in logical coordinates. For example, to access shared memory efficiently, it is sometimes necessary to add p padding elements to every column of a column major matrix. In this case, for a logical size of $M \times K$, the corresponding physical size would be $(M + p) \times K$. The `size(...)` function is used so that our GEMM API known how many bytes it has to reserve in shared memory. This function is also used by the heuristics in the params abstraction to select the optimal tiling size in shared memory, as this depends on how much memory a given memory layout requires.

The other two functions are `load(layout_type, tile, ...)` and `store(layout_type, tile, ...)`. As their name suggests, these functions are responsible to load or store the tile at the logical coordinates represented by the `tile` argument. This is different from CUTLASS, where the layouts only convert logical indices to physical offsets. By having separate load and store functions, users of our GEMM API can implement arbitrary logic to load or store the matrix elements corresponding to a given tile. For example, recall that NVIDIA GPUs can load and store vectors of 16 bytes (128 bits) in a single instruction. This vectorisation of memory accesses is only possible if the base address of the load or store is aligned, i.e. divisible by 16. An `AlignedColumnMajor` layout can indicate that the necessary alignment requirements are met, so that the corresponding `load` and `store` functions can issue vectorised loads and stores.

In the case of a classic GEMM kernel, the most obvious examples of implementations of the layout building block are `RowMajor` and `ColumnMajor`. As mentioned before, each of these layout can be adapted to aligned or padded layouts. To add padding, one could have `PaddedRowMajor` and `PaddedColumnMajor` layouts, but Julia's type system allows

us to do this more cleanly. We can make a parametrised type `PaddedLayout{Layout, Padding}`, where `Padding` represents the padding in number of elements, and `Layout` is the base layout we wish to modify, such as `RowMajor` or `ColumnMajor`. The `load` and `store` functions for padded layouts would then dispatch to the implementations for the underlying `Layout`. The `size` method simply calls `size` for the underlying layout, and adds the padding specified in the type parameters of the `PaddedLayout`.

The layout building block can also be used to create a GEMM with a more complicated mapping between logical indices and physical offsets. In Section 3.2, we discussed the GETT approach to tensor contraction, where multidimensional tensors are reinterpreted as matrices. In our framework, this reinterpretation can be performed using a custom implementation of the layout building block. Note that a layout does not even need to correspond to a matrix that is materialized in memory. Consider a matrix multiplication where the elements of one of the matrices can be calculated from the position, i.e. $a_{ij} = f(i, j)$ for some function f. In this case, we implement a layout where the `store` function is a no-op, and the `load` function generates the necessary elements on the fly. Similar strategies can be used for other matrices with a special structure, such as sparse matrices. We can only store the non-zero elements in memory, and create a custom layout that implements the necessary logic to load or store the correct elements.

The next building block in our GEMM API are *transforms*. Transforms are arbitrary Julia functors, i.e. functions or structures implementing the function call operator `()`. They are called after every load and before every store operation. This is different from CUTLASS's transforms, which only cover the global-to-shared memory stream of the input matrices A and B. Our transforms can thus also be used for elementwise operations of the result matrix, whereas in CUTLASS, this is the task of the epilogue component. By having a transform after every load and before every store, elementwise operations to the input and result matrices can be applied consistently in our API.

Transforms can be used for elementwise operations, such as a simple scaling in the case of GEMM, and the application of activation functions for artificial neurons in neural networks. Another use case of transforms is to implement type conversions immediately after loading data from global memory. This is useful if one wants to use a higher

precision data type to compute the GEMM, but store the matrices in lower precision in global memory to save capacity.

So far, we have discussed three parts of our GEMM API: params, layouts, and transforms. Figure 6.1 illustrates how these three components interact to copy a tile of the A matrix from global to shared memory. A similar structure is used to copy tiles of the B, C, or resultant D matrix. This copy operation is performed cooperatively by all threads in a threadblock, using the parallelisation operation of our tiling API. First, the params component determines the tiling size that should be used for the tile iterator corresponding to the parallelise operation. The GEMM kernel then iterates over this tile iterator, which returns a tile in each iteration. The base and offset of this tile are specified in logical coordinates. To load the correct matrix elements from global memory, the `load` function is called using this tile and the layout of A in global memory. This `load` function returns a tuple that contains the correct matrix elements. This tuple is then sent to the transform for the global-to-shared stream of the A matrix, resulting in a transformed tuple. Finally, the `store` function corresponding to the layout of A in shared memory is called with this transformed tuple and the logical index of the current tile.

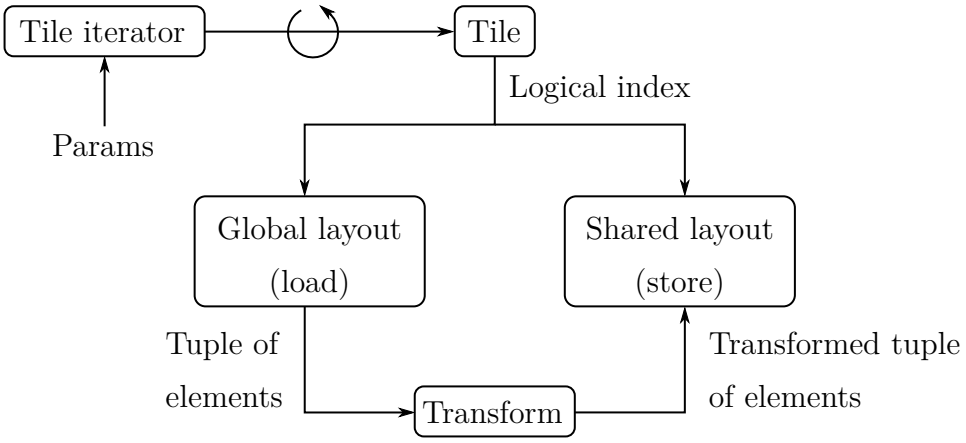


Figure 6.1: Copying a tile of A from global to shared memory using the params, layouts, and transforms components in our GEMM API.

The previous building blocks together copy tiles from global to shared memory. The next step in GEMM is to compute the matrix product itself. In our GEMM API, this is the task of the *operator* building block. CUTLASS contains GEMM components for each level in the CUDA execution hierarchy. The components at the threadblock level

have tile iterators as template arguments that are used to copy tiles from global memory to shared memory. In our GEMM API, this copying is implicit in the structure of the pre-built GEMM kernel, and can be customised using the layout abstraction that we have discussed before. The operator building block in our API approximately corresponds to the warp-level GEMM component in CUTLASS. The purpose of operators is to load tiles from shared memory, perform the matrix multiplication, and store the resulting tile back to shared memory.

In CUTLASS, the loads and stores are implemented by tile iterators, which typically are specific for the GEMM component. For example, the warp-level WMMA GEMM component has specific tile iterators that call the `nvcuda::wmma::load_matrix_sync` function of the CUDA C++ WMMA API. Because the warp-level GEMM components and its tile iterators are so closely tied together, they are merged in one building block in our GEMM API, the operator. Instead of having separate tile iterator arguments, the operator building block has five functions associated with it. The `load_a`, `load_b`, and `load_c` functions load tiles of the A, B, and C matrix from shared memory to registers. The matrix computation itself is performed by the `mma` function, and the result is stored back to shared memory using the `store_d` function. Like the layout building block, the `load_a`, `load_b`, `load_c`, and `store_d` functions have a `tile` argument that represents the logical coordinate of the tile that should be loaded or stored. The load and store functions also have an argument that determines the shared memory layout of the corresponding matrix, so that we can dispatch to the different implementations depending on the memory layout that is used. Finally, the `mma` function has three arguments `a_frag`, `b_frag`, and `c_frag` that represent parts of the A, B, and C matrices stored in registers. The function should perform the multiply-accumulate operation `res_frag = a_frag * b_frag + c_frag`, and return the resulting fragment `res_frag`.

Note the similarity of the operator building block with the WMMA API we developed in Chapter 4. There is a one-to-one correspondence between the basic steps of WMMA, and the functions we listed in the previous paragraph. This is no coincidence, as both the operator building block and the WMMA API are warp-level matrix multiply-accumulate operations. Because of this correspondence, it is fairly easy to define an implementation of the operator building blocks that uses Tensor Cores using our WMMA API. It suffices

to convert the `tile` argument to the load and store functions to a memory address, and call the `load`, `store`, and `mma` functions of the WMMA API.

The operator building block has several use cases. First, they can be used to provide a custom implementation for the computation in the inner loop of GEMM. This is useful if the data type of our matrices has a custom multiplication operator, such as complex numbers. The operator building block also improves the portability of the GEMM kernel. For example, the WMMA operator may be parametrised with the WMMA shape, so that we can select the WMMA shape that is optimal for our GPU. Additionally, we cannot always use WMMA, since not all GPUs have Tensor Cores. In order for our kernel to be portable to devices with different capabilities, we can define an alternative operator that calculates the matrix product using the traditional FPUs instead of Tensor Cores.

Recall that CUTLASS's epilogue component performs three different tasks. It applies elementwise operations to the result, stores the transformed matrix to shared memory, and updates global memory efficiently. In our API, the elementwise operations are handled by the transform component, so that elementwise operations to the A, B, C, and D matrices can be applied in the same way. While we can use transforms to apply elementwise operations to the resultant matrix, we still need an epilogue building block. Epilogues are needed so that users of our API can customise the way global memory is updated at the last stage of GEMM. For example, without an epilogue, it would not be possible to apply a reduction operation across all threadblocks.

The purpose of epilogues in our API slightly differs from CUTLASS's approach to epilogues. In CUTLASS, the epilogue also stores the resultant matrix to shared memory. Because this step heavily depends on the way the GEMM was computed in the inner loop, CUTLASS includes quite a few default epilogue components. For example, CUTLASS has separate epilogues for GEMMs using WMMA, and GEMMs using traditional FPUs.

Our epilogue building block only has one purpose: to copy tiles of the resultant matrix from shared memory to global memory. The storing to shared memory is moved to the operator building block that we mentioned before. This means that we need fewer epilogues overall, since our epilogue is decoupled from the way GEMM is computed in

the inner loop. As long as two different operators store the resultant matrix to shared memory in the same layout, we can use the same epilogue for each. By default, we only include one epilogue that simply copies the current threadblock's tile in shared memory to the correct position in global memory. This default epilogue uses the previously mentioned layout building block to determine the memory layout of the resultant D matrix.

6.4 Example

We can imagine that the discussion in the previous section was somewhat abstract, so we would like to clarify it with an example. Consider the first step in a GEMM kernel: copying a tile of the C matrix from global to shared memory. Listing 15 on page 70 shows an implementation of this step using our tiling API. For clarity, we repeat it here as Listing 20.

In our GEMM API, this first step is implemented as shown in Listing 21. The code has a similar structure to Listing 20, but the linearisation, loads, and stores are replaced by generic calls to `Layout.load` and `Layout.store`. The first arguments of these functions, `GLOBAL_LAYOUT` and `SHARED_LAYOUT`, are types that determine the memory layout of C for global and shared memory, respectively. The `transform_global_to_shared_c` is a Julia function that represents the transform that should be applied during the global-to-shared memory stream of the C matrix.

Suppose that we have defined the necessary components (such as layouts, operators, ...) for a given use case. To instantiate and execute GEMM kernels that use these components, we use the user-facing interface of our GEMM API, which is illustrated in Listing 22. This code fragment calculates a mixed-precision matrix product of the form $D_{ij} = \max(\sum_k A_{ik}B_{kj} + C_{ij}, 0)$. These types of matrix products are common in neural networks, where the activation function $\max(\cdot, 0)$ is commonly referred to as a rectified linear unit (ReLU). Lines 1 – 4 declare the two-dimensional arrays that represent the A, B, C, and D matrices. In Lines 6 – 11, we configure the parameters of our GEMM

```
@unroll for warp_tile = parallelise(block_tile.MN, Tile(MEM_CD_WARP), warpId,
↪  WARPS_PER_BLOCK)
  @unroll for thread_tile = parallelise(warp_tile, Tile(MEM_CD_THREAD),
  ↪  laneId, 32)
    global_thread_tile = translate(thread_tile, (M = block_i, N = block_j))

    global_linear_base   = linearise(global_thread_tile.base, (M = global_M,
    ↪  N = global_N))
    global_linear_offset = linearise(global_thread_tile.offset, (M = global_M,
    ↪  N = global_N))

    shared_linear_base   = linearise(thread_tile.base, (M = shared_M, N =
    ↪  shared_N))
    shared_linear_offset = linearise(thread_tile.offset, (M = shared_M, N =
    ↪  shared_N))

    global_ptr = pointer(global_c, global_linear_base)
    shared_ptr = pointer(shared_c, shared_linear_base)

    # Load at address global_ptr, with offset global_linear_offset
    # Store at address shared_ptr, with offset shared_linear_offset
  end
end
```

Listing 20: Copying a tile of the C matrix from global to shared memory using our tiling API (repeated from Listing 15).

```
@unroll for warp_tile = parallelise(block_tile.MN, Tile(MEM_CD_WARP), warpId,
    WARPS_PER_BLOCK)
  @unroll for thread_tile = parallelise(warp_tile, Tile(MEM_CD_THREAD),
      laneId, 32)
    global_thread_tile = translate(thread_tile, (M = block_i, N = block_j))

    x = Layout.load(GLOBAL_C_LAYOUT, c, global_thread_tile)
    y = transform_global_to_shared_c(x, thread_tile)
    Layout.store(SHARED_C_LAYOUT, shmem_c, y, thread_tile)
  end
end
```

Listing 21: Copying a tile of the C matrix from global to shared memory in our GEMM API.

kernel, such as the overall shape of the GEMM, the operator to be used in the inner loop, and the memory layouts of the A and C matrices. The missing fields are automatically set to reasonable default values. For example, if the memory layout of the B matrix is not specified, it is automatically set to the memory layout of the A matrix.

A GEMM kernel that uses this configuration is executed in Lines 13 – 16. The argument `transform_regs_to_shared_d` determines the transform that should be applied when copying tiles of the resultant D matrix from the register file to shared memory. The call to `MatMul.matmul` will execute each step in the GEMM kernel, using the components given by the user. For example, Listing 21 will be executed with `GLOBAL_C_LAYOUT = Layout.AlignedColMajor{Float32}`.

The equivalent of Listing 22 using CUTLASS is shown in Listing 23 and Listing 24. These listings were adapted from one of the examples available in the CUTLASS repository[1]. Listing 23 contains the configuration step of the GEMM, and thus corresponds to the call to `MatMul.get_config` in our API. Recall that elementwise operations in CUTLASS

[1]Original example available at the permalink `https://github.com/NVIDIA/cutlass/blob/1ab1027954bafc513cef2d3ca673d0e2c1eebb24/examples/12_gemm_bias_relu/gemm_bias_relu.cu`.

```
a = CuArray(rand(Float16, (M, K)))
b = CuArray(rand(Float16, (K, N)))
c = CuArray(rand(Float32, (M, N)))
d = similar(c)

conf = MatMul.get_config(
    gemm_shape = (M = M, N = N, K = K),
    operator = Operator.WMMAOp{16, 16, 16},
    global_a_layout = Layout.AlignedColMajor{Float16},
    global_c_layout = Layout.AlignedColMajor{Float32}
                          )

MatMul.matmul(
    a, b, c, d, conf;
    transform_regs_to_shared_d = Transform.Elementwise(x -> max(x, 0))
              )
```

Listing 22: Calculating the matrix product $D_{ij} = \max(\sum_k A_{ik} \cdot B_{kj} + C_{ij}, 0)$ using our GEMM API.

need to be implemented using a custom epilogue. Lines 2 – 6 instantiate the pre-defined `LinearCombinationRelu` epilogue for FP32 values. Lines 9 – 25 create a type `Gemm` that contains the configuration for the GEMM kernel, such as the datatype and layout of each matrix. The type parameter on Line 22 determines the epilogue, and is set to the epilogue we defined on Lines 2–6.

Note that each elementwise operation corresponds to a different CUTLASS epilogue. The CUTLASS codebase contains epilogues for two activation functions: ReLU and the sigmoid function $\sigma(x) = (1 + \exp(-x))^{-1}$. If CUTLASS does not contain an epilogue for a certain elementwise transform, users need to implement one themselves. Because these epilogues are typically 150 – 200 lines long[2], this process requires significant effort. In our API, we can use another activation function by replacing `Transform.Elementwise(x -> max(x, 0))` with any arbitrary function. Of course, we can also apply custom elementwise transforms in CUTLASS using a separate kernel, but this introduces extra kernel launch overhead, and results in the same data being loaded multiple times.

Listing 24 launches a GEMM kernel using this configuration, and thus corresponds to `MatMul.matmul` in our API. Lines 2 – 7 create the arguments of the GEMM kernel, such as the overall shape of the GEMM, and the addresses of the A, B, C, and D matrices. Some CUTLASS components need extra memory to store intermediate results, called a workspace. A workspace of the correct size is allocated on Lines 10–11. Finally, Lines 14–16 initialize and run the GEMM kernel.

Launching a GEMM kernel in CUTLASS uses a lot more components than our GEMM API. These components have quite a few template arguments, all of which need to be specified by the user. In contrast, our API has a set of heuristics for default values, so that end users do not need to specify every configuration parameter explicitly.

[2]For examples of pre-defined epilogues in CUTLASS, see `https://github.com/NVIDIA/cutlass/tree/1ab1027954bafc513cef2d3ca673d0e2c1eebb24/include/cutlass/epilogue/thread`.

```
// Instantiate the epilogue
using EpilogueOp = cutlass::epilogue::thread::LinearCombinationRelu<
  float,  // datatype of the output
  4,      // number of elements per memory access
  float,  // datatype of the accumulator
  float>; // datatype used to calculate the ReLU

// Configure the GEMM kernel
using Gemm = cutlass::gemm::device::Gemm<
  cutlass::half_t,                        // datatype of A
  cutlass::layout::ColumnMajor,           // layout of A
  cutlass::half_t,                        // datatype of B
  cutlass::layout::ColumnMajor,           // layout of B
  float,                                  // datatype of C and D
  cutlass::layout::ColumnMajor,           // layout of C and D
  float,                                  // datatype used for accumulation
  cutlass::arch::OpClassTensorOp,         // operator in inner loop
  cutlass::arch::Sm75,                    // Turing-generation Tensor Cores
  cutlass::gemm::GemmShape<128, 128, 32>, // threadblock shape
  cutlass::gemm::GemmShape<64, 64, 32>,   // warp shape
  cutlass::gemm::GemmShape<16, 8, 8>,     // operator shape
  EpilogueOp,                             // epilogue
  cutlass::gemm::threadblock::GemmIdentityThreadblockSwizzle<>,
      // determines how threadblocks are scheduled on the GPU
  1>;                                     // number of pipeline stages
```

Listing 23: Configuring a GEMM kernel to calculate the matrix product $D_{ij} = \max(\sum_k A_{ik} \cdot B_{kj} + C_{ij}, 0)$ in CUTLASS.

```
// Create the arguments of the kernel
typename Gemm::Arguments arguments{
  cutlass::gemm::GemmCoord(M, N, K), // overall shape of GEMM
  A, B, C, D,                        // references to A, B, C, D
  {1, 1},                            // scaling factors used for A * B and D
  1                                  // split factor along the K-dimension
};

// Allocate workspace
size_t workspace_size = Gemm::get_workspace_size(arguments);
cutlass::device_memory::allocation<uint8_t> workspace(workspace_size);

// Run the kernel
Gemm gemm;
gemm.initialize(arguments, workspace.get());
gemm();
```

Listing 24: Launching a CUTLASS GEMM kernel using the configuration of Listing 23.

6.5 Evaluation

In this section, we will evaluate our GEMM API according to the three criteria we described in Section 6.2. In Section 6.5.1, we will discuss the flexibility and performance of our GEMM API using three different examples. For each of the three examples, we will first indicate which building blocks need to be instantiated. Next, we will compare the performance of our kernels with the state-of-the-art GEMMs in cuBLAS and CUTLASS. Recall that CUDAnative also includes a generic matrix multiplication kernel as a fallback for datatypes that are incompatible with cuBLAS. We will also compare the performance of our kernels with CUDAnative's generic implementation. Finally, Section 6.5.2 will explain how our proposed building blocks increase the portability of the GEMM kernels that are built using our API.

6.5.1 Flexibility and performance

In this section, we will evaluate both the flexibility and performance using three example use cases. First, we will implement a mixed-precision GEMM using the WMMA API we developed in Chapter 4. Next, we introduce the necessary components to extend this GEMM to matrices containing complex numbers. Our last example will change this complex GEMM so it also supports dual numbers. Dual numbers extend the set of real numbers with a new element ε, similar to the imaginary unit i in the case of complex numbers. Addition and multiplication of dual numbers is similar to the complex case, with the only difference that $\varepsilon^2 = 0$, whereas $i^2 = -1$.

Example 1: Mixed-precision GEMM

To implement a mixed-precision GEMM, we will create an operator that calls our WMMA API. We have already mentioned that the interface of operators and WMMA is similar: both have functions to load A, B, and C, calculate the matrix product, and store the

resultant D matrix. To use WMMA in our GEMM API, it is thus sufficient to convert the logical index of the tile to a memory address, and call the correct WMMA function.

Julia uses column major storage for matrices. The next step for a mixed-precision GEMM using our API is thus to implement a `ColMajor` layout. We used this memory layout for both global and shared memory, but we noticed that performance was not on par with CUTLASS's WMMA kernels. Upon closer inspection, we found that this performance gap was due to two reasons.

First, tiles of the FP16 A and B matrices were loaded using 16-bit memory instructions. This resulted in a large amount of memory transactions, which resulted in stalls because the load-store queue was full. To reduce the number of instructions, we can use vectorised accesses instead. The largest transaction size on NVIDIA GPUs is 128 bits, so that we can replace eight 16-bit loads with one 128-bit load, reducing the total number of memory instructions by a factor 8×. We chose to do this vectorisation manually, by casting the pointer from a `Float16` pointer to a vector type of 128-bits. Note that this is the same approach that CUDA C++ uses for explicit vectorisation.

The second reason for the performance gap with CUTLASS is due to the structure of shared memory. On NVIDIA GPUs, shared memory is split into *memory banks*. Different memory banks can be accessed in parallel, but memory accesses to addresses that map to the same bank are serialised. This serialisation process is often referred to as a *bank conflict*. Consider the example of Figure 6.2, where shared memory consists of 4 banks. Suppose we store a 4×4 matrix in a row major format. If multiple threads access the elements 0, 4, 8, and 12 in the same column, a bank conflict occurs. CUTLASS's WMMA kernels eliminate these bank conflicts by using padded layouts. In our example, we can add one element of padding after every row, as shown in Figure 6.3. This changes the mapping of elements to banks, so that the elements 0, 4, 8, and 12 now map to different memory banks. To use padding in our GEMM API, we use the parametrised `PaddedLayout{Layout, Padding}` that we mentioned in Section 6.3. `PaddedLayout` adds the specified number of padding elements, and reuses the functionality of the wrapped `Layout`.

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

Figure 6.2: Storing a 4×4 matrix in shared memory in a non-padded layout.

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

Figure 6.3: Storing a 4×4 matrix in shared memory using padding.

To evaluate the performance of our mixed-precision GEMM, we will compare our implementation with cuBLAS, CUTLASS, and the pre-existing generic matrix multiplication in CUDAnative. We will measure this performance as the number of floating point operations per second, expressed in TFLOPS. We have two experimental setups at our disposal for this evaluation. The first setup contains an NVIDIA V100, a GPU of the Volta generation. The resulting performance is graphed in Figure 6.4. First note that CUDAnative's generic matrix multiplication performs poorly, and levels off to a maximum quite quickly. This is because this generic kernel does not use tiling techniques, so that it is limited by the bandwidth of global memory. This is many orders of magnitude slower than the computational capabilities of the GPU. The purple curve represents CUTLASS's GEMM kernel that uses WMMA. To have a fair comparison, we used the CUTLASS Profiler to select the same tiling sizes for CUTLASS and our implementation.

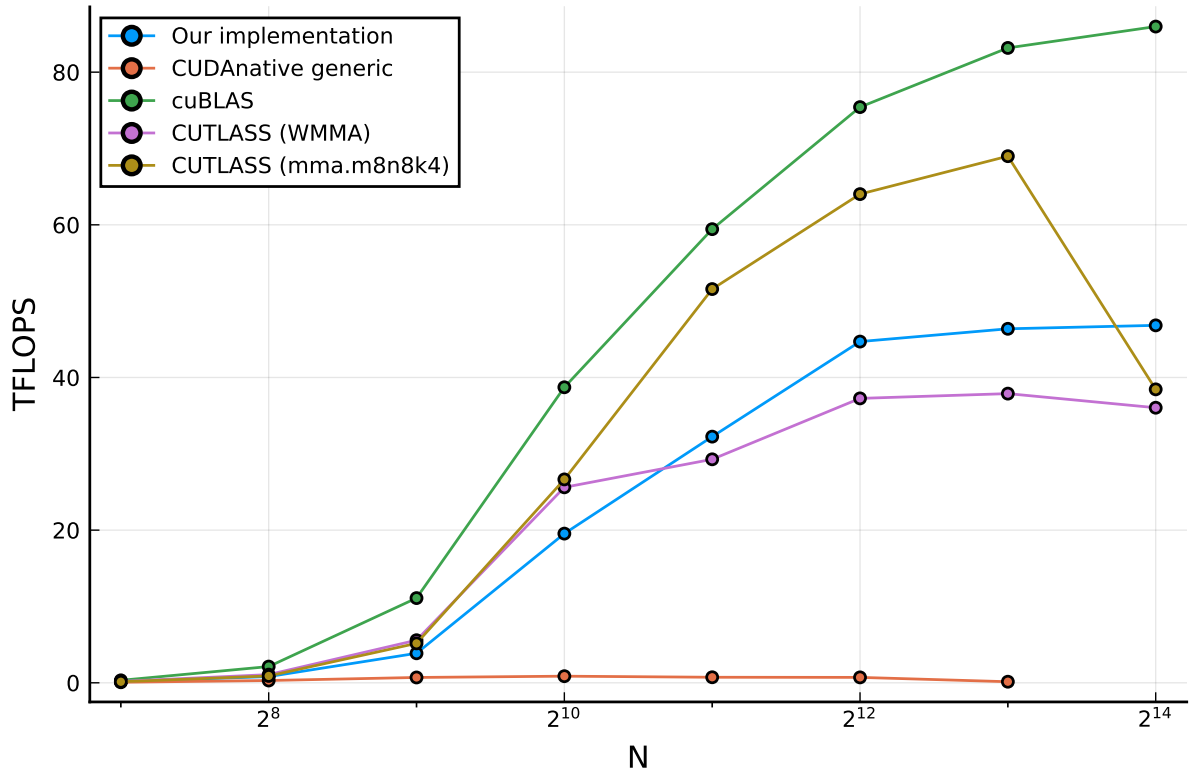


Figure 6.4: A comparison of the performance of our mixed-precision GEMM with state-of-the-art implementations on the NVIDIA V100 GPU.

As we can see, our kernel performs slightly better than CUTLASS's GEMM. This seems to be the result of slight differences in the generated SASS assembly, such as the order of memory instructions. Unfortunately, we have no control over the process that generates the SASS, since this is performed by the closed-source NVIDIA CUDA driver. We have observed that when we tweak our GEMM, the CUDA driver may generate slightly different SASS instructions, resulting in different performance results. In general, the performance stays close to 40 TFLOPS, so that the graph of Figure 6.4 still gives a good idea of the performance of our kernels. This same remark holds for the other performance graphs that we will discuss: if two implementations have similar performance, we cannot really say with absolute confidence that one is more performant than the other.

Our GEMM kernels achieve about 55% of the performance of cuBLAS on Volta. To get an idea on how to bridge this gap, we also plotted the performance of a CUTLASS GEMM kernel that uses `mma` instead of WMMA. `mma` is a set of PTX instructions that use Tensor Cores, similar to WMMA. These `mma` instructions are generation specific, and thus less portable than the WMMA abstraction layer. The `mma.m8n8k4` of Figure 6.4 calculates an $8 \times 8 \times 4$ matrix multiply-accumulate, and is optimised for GPUs of the Volta generation. The main difference between WMMA and these `mma` instructions, is that the latter gives more control over how the matrix multiply-accumulate is performed. For example, WMMA calculates a $16 \times 16 \times 16$ matrix multiply-accumulate per warp. Volta's `mma.m8n8k4` calculates one $8 \times 8 \times 4$ matrix multiply-accumulate per quarter warp, i.e. four $8 \times 8 \times 4$ multiply-accumulates per warp.

Another difference between WMMA and `mma` is that `mma` does not have explicit load and store instructions. This means that we have more control over how the matrix elements are loaded, stored, and distributed over the different threads. This is used by CUTLASS to replace the padded shared memory layout, which results in some overhead due to the unused padding elements, with an alternative layout. This alternative layout permutes the elements in shared memory in such a way that simultaneous memory accesses map to different memory banks. Since this technique heavily depends on the GPU architecture and type of GEMM kernel, we did not implement this. Nevertheless, to use this optimisation in our GEMM API, one could write a custom layout for the

permuted layout in shared memory, and a custom operator to replace WMMA with `mma`.

Our second experimental setup contains a Turing-generation RTX 2080 Ti. The performance results of this setup are shown in Figure 6.5. We observe similar performance behaviour for this graph as was the case for Volta: our implementation and WMMA CUTLASS have similar performance, whereas CUDAnative's generic kernel only achieves a fraction of peak device performance. On Turing, our mixed-precision GEMM using WMMA is able to achieve 75% of the performance of cuBLAS, compared to Volta's 55%. Note that the Volta-style `mma.m8n8k4` performs poorly on Turing, only achieving about 20 TFLOPS. Figure 6.5 also shows the performance for `mma.m16n8k8`. This `mma` operation is aimed at Turing GPUs, and has a performance of about 40 TFLOPS, very close to cuBLAS. This again illustrates that the `mma` operations are highly generation specific.

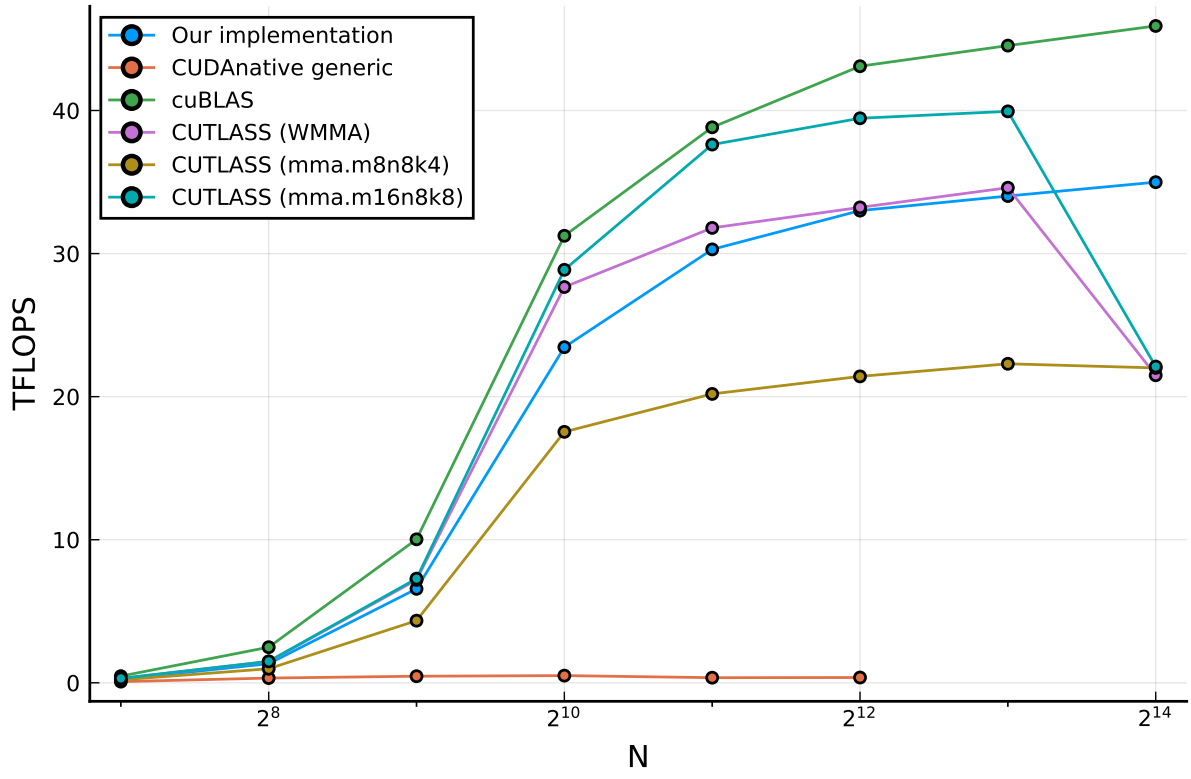


Figure 6.5: A comparison of the performance of our mixed-precision GEMM with state-of-the-art implementations on the NVIDIA RTX 2080 Ti GPU.

Example 2: Mixed-precision complex GEMM

There are two main differences between a normal mixed-precision GEMM, and a mixed-precision GEMM of complex numbers. First, the WMMA multiply-accumulate operation in the inner loop is replaced by four WMMA operations: `A.real * B.real`, `A.real * B.imag`, `A.imag * B.real`, and `A.imag * B.imag`. In our GEMM API, this can be done using a custom operator `WMMAComplexOp`. This operator is based on the `WMMAOp` operator, but performs four $16 \times 16 \times 16$ matrix multiplications instead of one. `WMMAComplexOp` also loads or stores two WMMA fragments, one for the real part, and one for the imaginary part. Note that the real part of the result matrix is `A.real * B.real - A.imag * B.imag`, so that the fragment corresponding to `A.imag * B.imag` needs to be negated. This corresponds to an FP16 multiplication with a constant -1. Recall from Section 4.4.1 that Julia first casts FP16 values to FP32, and then multiplies the resulting FP32 values. This significantly reduces the performance of our GEMM kernel. In Listing 10, we showed a possible solution to this problem by using inline PTX that multiplies FP16 values directly. To improve performance, we apply this same approach to our mixed-precision complex GEMM.

The second difference between normal GEMMs and complex GEMMs, is in the memory layouts that are used. In global memory, complex matrices are stored in an interleaved layout, as illustrated at the left in Figure 6.6. In the interleaved layout, the real and imaginary parts of a single element are stored contiguously. To load the real and imaginary parts using WMMA, we need to use a split-complex layout, as shown at the right of Figure 6.6. This layout stores the real and imaginary parts separately, so that both the matrix consisting of the real part, and the matrix consisting of the imaginary part are stored contiguously. This is needed because WMMA implicitly assumes that elements in the same column of a column major matrix are stored at adjacent memory addresses. In our GEMM API, we hence defined two memory layouts `InterleavedComplex` and `SplitComplex`. These layouts have custom `load` and `store` functions that are used to convert between the interleaved and split layouts.

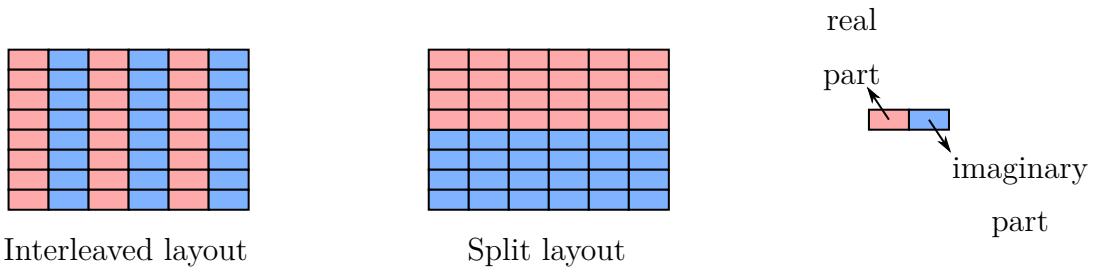


Figure 6.6: The difference between an interleaved and split memory layout to store matrices of complex numbers. Memory addresses increase from left-to-right, and then from top-to-bottom, i.e. in a row-major fashion.

CUTLASS 2.1 includes an example that performs a mixed-precision complex GEMM. We compare the performance of our complex GEMM kernel with CUDANATIVE's generic implementation and this CUTLASS example in Figure 6.7 (Volta) and Figure 6.8 (Turing). Once again, we see that CUDANATIVE's generic implementation only achieves a fraction of the peak device performance, because it does not make use of any tiling techniques. For Volta, our kernel achieves a performance of 37% of CUTLASS, whereas on Turing, we get a performance slightly over 50%. The fact that we were able to achieve decent performance by just implementing two components is promising, even though there is still a performance gap. A possible explanation for this gap is that the CUTLASS example uses `mma` instead of WMMA. Unfortunately, complex GEMMs are not added to the CUTLASS profiler yet, so we cannot easily check the performance of complex GEMM in CUTLASS with WMMA. A possible direction for future work is to research the case of complex GEMMs more thoroughly, in order to bridge this remaining performance gap.

Example 3: Mixed-precision dual GEMM

In our final example, we will study the case of the multiplication of matrices containing dual numbers. We have already explained that dual numbers are similar to complex numbers, in that they are a two-dimensional extension of the real numbers. A dual number is written as $a + \varepsilon b$, where ε has the same role as the imaginary unit i for complex numbers. Addition and multiplication of dual numbers happens similarly to complex numbers, with the only difference that instead of $i^2 = -1$, we have $\varepsilon^2 = 0$. Addition of

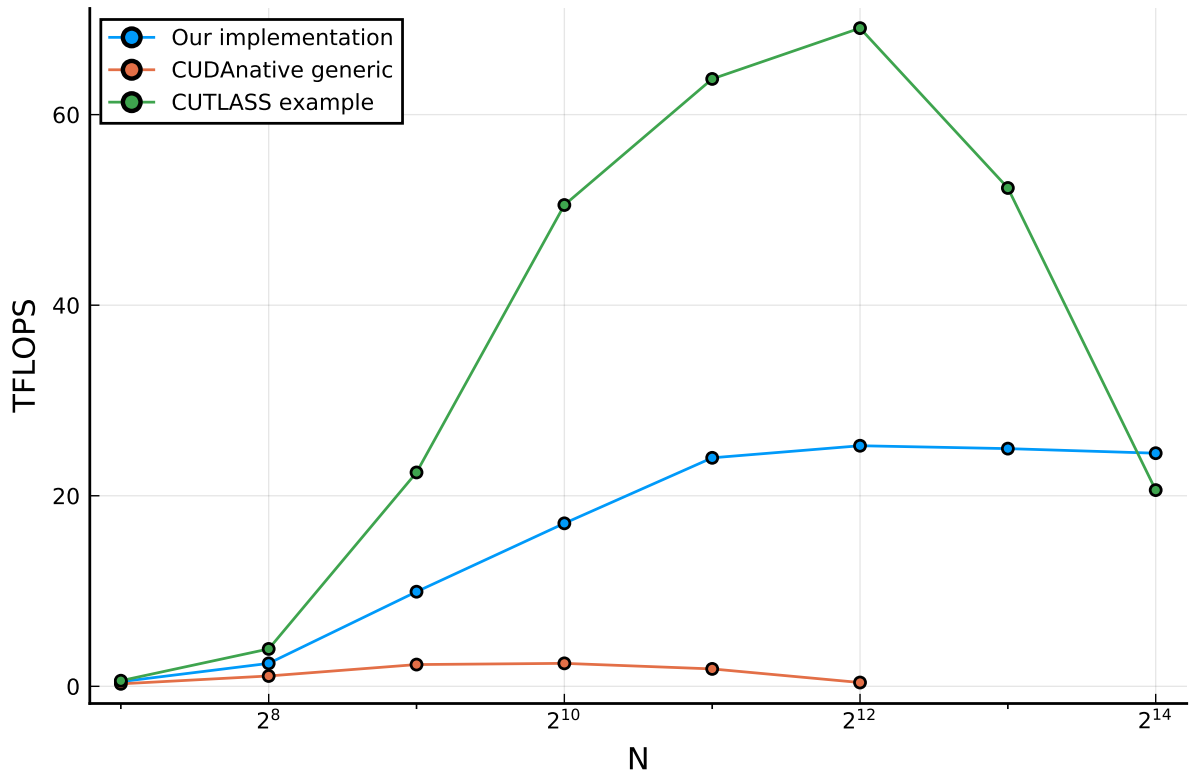


Figure 6.7: An illustration of the performance of our mixed-precision complex GEMM on the NVIDIA V100 GPU.

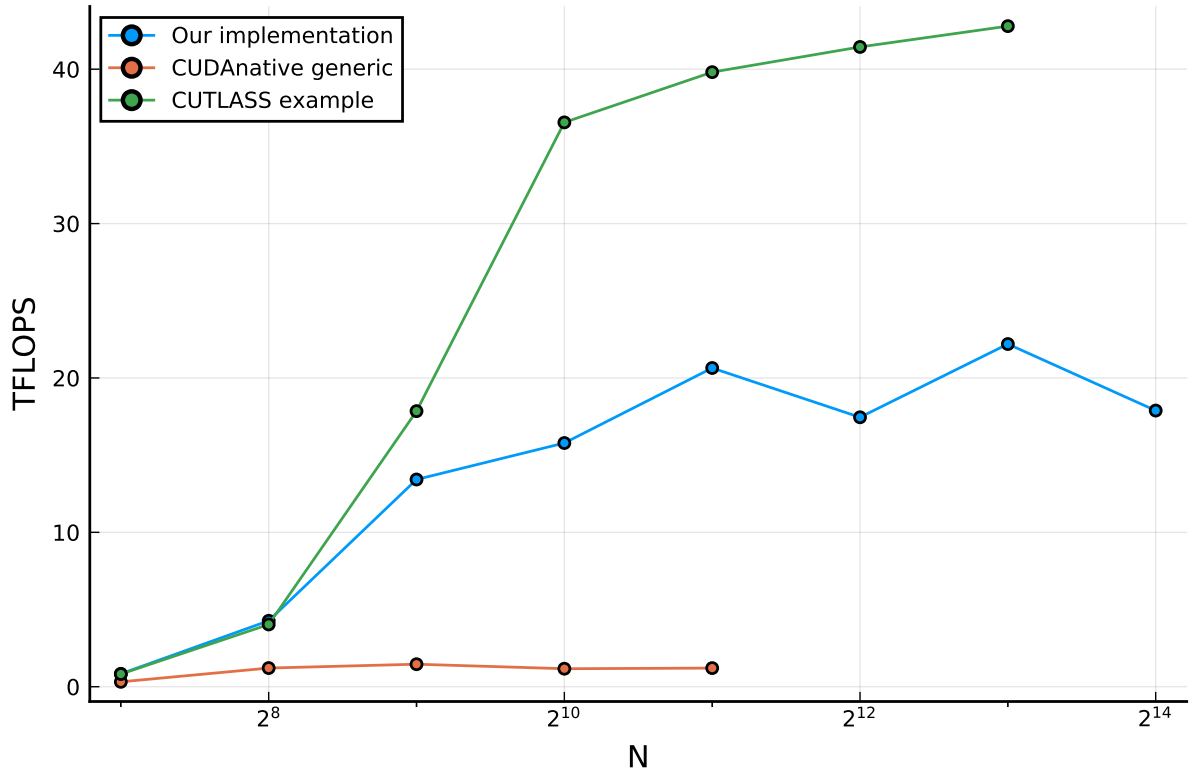


Figure 6.8: An illustration of the performance of our mixed-precision complex GEMM on the NVIDIA RTX 2080 Ti GPU.

dual numbers is thus defined as $(a+\varepsilon b)+(c+\varepsilon d)=(a+c)+\varepsilon(b+d)$, and multiplication as $(a+\varepsilon b)\cdot(c+\varepsilon d)=ac+\varepsilon(ad+bc)$. One important application of dual numbers in scientific computing is automatic differentiation of functions. In Julia, this method for automatic differentiation is implemented in the package ForwardDiff.jl [64].

Because of the similarity between complex numbers and dual numbers, most of the discussion of the previous example applies to this example as well. As a result, we can simply reuse the interleaved and split layouts for matrices of dual numbers. The only difference is in the operator building block. Instead of four WMMA multiplications, we only need to perform three: `A.real * B.real`, `A.real * B.dual`, and `A.dual * B.real`. The use cases of complex and dual matrices are thus an excellent illustration of the reusability of the building blocks in our GEMM API.

Performance-wise, we observe the same behaviour as the complex GEMM. This is not surprising, as the only difference between complex and dual GEMMs is the operator used in the inner loop. Note that neither CUTLASS nor cuBLAS include GEMM kernels for dual numbers. Thus, multiplying two dual matrices in Julia will dispatch to CUDAnative's generic implementation, which is many orders of magnitude slower than peak device performance. This means that using our GEMM API already results in a significant speedup for dual matrices, even though we can still improve the performance of our complex and dual GEMMs.

6.5.2 Portability

To conclude the evaluation of our GEMM API, we will discuss the portability of GEMM kernels using our API across different GPU architectures. In the previous section, we have evaluated GEMM on a Volta and a Turing GPU, the only architectures that have Tensor Cores. We were able to achieve similar performance results for both architectures, without explicitly tuning our GEMM kernel to a specific architecture. This already indicates that the general structure of our GEMM kernels is portable to different GPU architectures.

Additionally, there are also two building blocks in our API that improve the portability of kernels. First, the params building block determines the tiling sizes for each stage of GEMM. To port a GEMM kernel to a GPU with different hardware properties, such as the size of shared memory, we can simply change the relevant tiling size. Second, operators allow us to cover the different computational capabilities of GPUs. For example, our WMMA operator only works on GPUs that have Tensor Cores. To support GEMM on GPUs of older generations, we can add another operator that computes the GEMM using FPUs instead. Recall that Tensor Cores can also be programmed using the generation-specific `mma` PTX instructions. These `mma` instructions can be mapped to different operators, one for each GPU generation.

6.6 Conclusion

In this chapter, we have designed an API for flexible GEMM kernels. We have proposed a set of customisable building blocks that together implement a GEMM. We have illustrated how these components interact using three use cases: a normal mixed-precision GEMM, a complex mixed-precision GEMM, and a dual mixed-precision GEMM. We also compared the performance of our kernels with state-of-the-art implementations in cuBLAS and CUTLASS. In most cases, the performance of our GEMM kernels is similar to CUTLASS's kernels with the same parameters.

The GEMM API and the tiling API of the previous chapter were bundled in one pull request to CUDAnative, that is currently under review[3]. The GEMM API part of this pull request consists of approximately 1800 lines of Julia source code. For each of the three use cases we discussed, the pull request contains the necessary components (such as layouts and operators), and an example that illustrates how to instantiate a GEMM that uses these components. The pull request also contains a set of Bash scripts that compare the performance of our GEMM kernels with cuBLAS and CUTLASS.

[3]Pull request available at `https://github.com/JuliaGPU/CUDAnative.jl/pull/629`.

7 Conclusion and future work

In this thesis, we first described the relevant aspects of CUDA's programming model for GPGPU computing, the Julia programming language, and the Julia package CUDANATIVE.JL. We then explained the concept of mixed-precision arithmetic, and how it relates to Tensor Cores in the latest NVIDIA GPUs. Next, we focused on matrix multiplication kernels. We highlighted why flexibility is an important criterion for GEMM, and how the state-of-the-art fares in this regard. We identified two important libraries, CUBLAS and CUTLASS, that served as benchmarks for the GEMM kernels we developed.

The main goal of this thesis is to build a flexible mixed-precision GEMM using NVIDIA Tensor Cores. To that end, we have designed, implemented, and evaluated three different APIs. We started with an API to program Tensor Cores from within the Julia programming language. This API is inspired by CUDA C++'s API for Tensor Cores, WMMA, but deviates from it to improve readability, or to make the resulting API fit better in the Julia programming language.

We then moved on to an API for recursive blocking. This API is used to facilitate writing algorithms that use tiling techniques to improve performance, such as matrix multiplications or tensor contractions. Given our goal of mixed-precision GEMM, we illustrated the use of this tiling API using three different steps in a mixed-precision matrix multiplication kernel.

Finally, we proposed a set of components that together implement a GEMM kernel. Each of these components can be customised by the user of our API, thereby increasing the

flexibility of the kernel. To demonstrate the flexibility of our proposed scheme, we have instantiated the necessary components for three different use cases. We first applied our API to the case of a mixed-precision GEMM, where we were able to achieve performance similar to NVIDIA's CUTLASS library. Next, we introduced the necessary components to also support multiplications of complex-valued matrices. Finally, we adapted this complex GEMM so it also supports matrices of dual numbers.

The abstractions as described in this thesis serve as a starting point for further research and development in the area of flexible GEMMs. In particular, we see several possible directions for future work or research:

1. Our WMMA API only supports the Volta-generation datatypes and shapes, since only those were supported by LLVM at the time we implemented our API. In the future, the WMMA API can be extended to also include support for the datatypes and shapes introduced in Turing and Ampere.

2. In this thesis, our main focus was on mixed FP16-FP32 GEMMs using Volta-style Tensor Cores. In the future, we can instantiate the necessary components to support other GEMM datatypes, such as FP32 or FP64, as well. We can even use Tensor Cores to accelerate this matrix product, since Tensor Cores of the Ampere generation add support for these datatypes.

3. We can look into other applications of flexible GEMM, such as tensor contractions, in more detail. By implementing those applications using our GEMM API, we can see if our proposed scheme works well for these cases, or if adaptations to the GEMM components are needed.

4. Starting from Ampere, NVIDIA GPUs support asynchronous copy instructions from global to shared memory [63]. The upcoming 2.2 release of CUTLASS will use these instructions to construct efficient software pipelines to hide the latency of global loads [28]. A possible direction for future research is to add support for these asynchronous copy instructions to our layout components.

5. There is still a performance gap between our mixed-precision GEMM kernels and cuBLAS. Bridging this gap will likely come down to replacing WMMA with the generation-specific `mma` instructions, and using a permuted layout in shared memory instead of padding.

6. Over the course of this thesis, we implemented some optimisations manually, such as the separation of memory addresses in a base and offset, and vectorisation. To increase programmer productivity, it would be better if these optimisations were applied automatically by the compiler. We discussed that CUDAnative `addrspacecast`s pointers before load or store instructions. This hampers various optimisations in LLVM, because `addrspacecast`s are treated as a black box by most optimisation passes. Recently, CUDAnative changed the way it stores pointers, thus eliminating the need for `addrspacecast`. It would be interesting to see if this change is sufficient, or if more work is needed to let the compiler handle these optimisations.

References

[1] Martin Abadi et al. 'TensorFlow: A system for large-scale machine learning'. In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 265–283. URL: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

[2] A. Abdelfattah, S. Tomov and J. Dongarra. 'Fast Batched Matrix Multiplication for Small Sizes Using Half-Precision Arithmetic on GPUs'. In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2019, pp. 111–122.

[3] A. Abdelfattah, S. Tomov and J. 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)*. 2019, pp. 17–24.

[4] Ahmad Abdelfattah et al. 'Performance, Design, and Autotuning of Batched GEMM for GPUs'. In: *High Performance Computing*. Ed. by Julian M Kunkel, Pavan Balaji and Jack Dongarra. Cham: Springer International Publishing, 2016, pp. 21–38. ISBN: 978-3-319-41321-1.

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

[6] E. Aprà, M. Klemm and K. Kowalski. 'Efficient Implementation of Many-Body Quantum Chemical Methods on the Intel® Xeon Phi Coprocessor'. In: *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2014, pp. 674–684.

[7] Alexander Auer et al. 'Automatic code generation for many-body electronic structure methods: The tensor contraction engine'. In: *Molecular Physics, R. J. Bartlett Festschrift Special Issue* 104 (Jan. 2006). DOI: 10.1080/00268970500275780.

[8] Brett W. Bader and Tamara G. Kolda. 'Algorithm 862: MATLAB Tensor Classes for Fast Algorithm Prototyping'. In: *ACM Transactions on Mathematical Software* 32.4 (Dec. 2006), pp. 635–653. DOI: 10.1145/1186785.1186794.

[9] Paul Barham and Michael Isard. 'Machine Learning Systems Are Stuck in a Rut'. In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS '19. Bertinoro, Italy: Association for Computing Machinery, 2019, pp. 177–183. ISBN: 9781450367271. DOI: 10.1145/3317550.3321441. URL: https://doi.org/10.1145/3317550.3321441.

[10] T. Besard, C. Foket and B. De Sutter. 'Effective Extensible Programming: Unleashing Julia on GPUs'. In: *IEEE Transactions on Parallel and Distributed Systems* 30.4 (2019), pp. 827–841.

[11] Tim Besard. *LLVM.jl: Julia wrapper for the LLVM C API*. 2020. URL: https://github.com/maleadt/LLVM.jl.

[12] Tim Besard et al. 'Rapid software prototyping for heterogeneous and distributed platforms'. In: *Advances in Engineering Software* 132 (2019), pp. 29–46.

[13] Valentin Churavy. *GPUifyLoops.jl: Support for writing loop-based code that executes both on CPU and GPU*. 2020. URL: https://github.com/vchuravy/GPUifyLoops.jl.

[14] BLAS contributors. *BLAS (Basic Linear Algebra Subprograms)*. 2017. URL: http://www.netlib.org/blas/.

[15] Andy Ferris. *Statically sized arrays for Julia*. 2016. URL: https://github.com/JuliaArrays/StaticArrays.jl.

[16] Geetika Gupta. *Using Tensor Cores for Mixed-Precision Scientific Computing*. 2019. URL: https://devblogs.nvidia.com/tensor-cores-mixed-precision-scientific-computing/.

[17] Azzam Haidar et al. 'Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers'. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. SC '18. Dallas, Texas: IEEE Press, 2018. DOI: 10.1109/SC.2018.00050. URL: https://doi.org/10.1109/SC.2018.00050.

[18] Azzam Haidar et al. 'Harnessing Tensor Cores FP16 Arithmetic to Accelerate Linear Solvers and HPC Scientific Applications'. NVIDIA GPU Technology Conference. 2018. URL: http://on-demand.gputechconf.com/supercomputing/2018/video/sc1826-harnessing-tensor-cores-fp16-arithmetic-accelerate-linear-solvers-hpc-scientific-applications.html.

[19] Geoffrey Hinton, Sara Sabour and Nicholas Frosst. 'Matrix capsules with EM routing'. In: *International Conference on Learning Representations*. 2018.

[20] Tim Holy. *TiledIteration.jl: A Julia package to facilitate writing multithreaded, multidimensional, cache-efficient code*. 2020. URL: https://github.com/JuliaArrays/TiledIteration.jl.

[21] Jianyu Huang, Chenhan D. Yu and Robert A. van de Geijn. *Implementing Strassen's Algorithm with CUTLASS on NVIDIA Volta GPUs*. 2018. arXiv: 1808.07984 [cs.MS].

[22] Tsuyoshi Ichimura et al. '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: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. SC '18. Dallas, Texas: IEEE Press, 2018.

[23] Intel. *Intel Math Kernel Library*. 2020. URL: https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html.

[24] Zhe Jia et al. *Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking*. 2018. arXiv: 1804.06826 [cs.DC].

[25] JuliaLang.org. *Julia Micro-Benchmarks*. 2020. URL: https://julialang.org/benchmarks.

[26] JuliaLang.org. *The Julia Language*. 2020. URL: https://julialang.org.

[27] JuliaLang.org. *The Julia Language Official Documentation*. 2020. URL: https://docs.julialang.org/en/v1.

[28] Andrew Kerr. 'Developing CUDA kernels to push Tensor Cores to the absolute limit on NVIDIA A100'. May 2020. URL: https://developer.nvidia.com/gtc/2020/video/s21745.

[29] Andrew Kerr et al. 'CUTLASS: CUDA Template Library for Dense Linear Algebra at all levels and scales'. Mar. 2018. URL: http://on-demand.gputechconf.com/gtc/2018/presentation/s8854-cutlass-software-primitives-for-dense-linear-algebra-at-all-levels-and-scales-within-cuda.pdf.

[30] Andrew Kerr et al. 'Programming Tensor Cores: Native Volta Tensor Cores with CUTLASS'. Mar. 2019. URL: https://developer.nvidia.com/gtc/2019/video/S9593.

[31] Khronos Group. *OpenCL: An open standard for parallel programming of heterogeneous systems*. 2020. URL: https://www.khronos.org/opencl.

[32] Khronos Group. *OpenGL: The Industry's Foundation for High Performance Graphics*. 2020. URL: https://www.opengl.org.

[33] Jinsung Kim et al. 'A Code Generator for High-Performance Tensor Contractions on GPUs'. In: *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*. CGO 2019. Washington, DC, USA: IEEE Press, 2019, pp. 85–95. ISBN: 9781728114361.

[34] Ronny Krashinsky et al. *NVIDIA Ampere Architecture In-Depth*. May 2020. URL: https://devblogs.nvidia.com/nvidia-ampere-architecture-in-depth/.

[35] J. Lai and A. Seznec. '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)*. 2013, pp. 1–10.

[36] J. Li et al. 'An input-adaptive and in-place approach to dense tensor-times-matrix multiply'. In: *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2015, pp. 1–12.

[37] LLVM contributors. *The LLVM Compiler Infrastructure Project*. 2020. URL: https://llvm.org.

[38] LLVM contributors. *The LLVM Target-Independent Code Generator*. 2020. URL: https://llvm.org/docs/CodeGenerator.html.

[39] LLVM contributors. *User Guide for the NVPTX Back-end*. 2020. URL: https://llvm.org/docs/NVPTXUsage.html.

[40] Wenjing Ma et al. 'GPU-Based Implementations of the Noniterative Regularized-CCSD(T) Corrections: Applications to Strongly Correlated Systems'. In: *Journal of Chemical Theory and Computation* 7.5 (2011), pp. 1316–1327. DOI: 10.1021/ct1007247. URL: https://doi.org/10.1021/ct1007247.

[41] Stefano Markidis et al. 'NVIDIA tensor core programmability, performance & precision'. In: *Proceedings - 2018 IEEE 32nd International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2018* (2018), pp. 522–531. DOI: 10.1109/IPDPSW.2018.00091.

[42] Devin A. Matthews. *High-Performance Tensor Contraction without Transposition*. 2016. arXiv: 1607.00291 [cs.MS].

[43] Vishal Mehta. 'Getting Started with Tensor Cores in HPC'. NVIDIA GPU Technology Conference. 2019. URL: https://on-demand.gputechconf.com/supercomputing/2019/video/sc1909-getting-started-with-tensor-cores-for-hpc.

[44] Paulius Micikevicius et al. 'Mixed Precision Training'. In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL: https://openreview.net/forum?id=r1gs9JgRZ.

[45] Microsoft. *Compute Shader Overview*. May 2018. URL: https://docs.microsoft.com/en-us/windows/win32/direct3d11/direct3d-11-advanced-stages-compute-shader?redirectedfrom=MSDN.

[46] Microsoft. *DirectX graphics and gaming*. May 2018. URL: https://docs.microsoft.com/en-us/windows/win32/directx.

[47] Edoardo [Di Napoli] et al. 'Towards an efficient use of the BLAS library for multilinear tensor contractions'. In: *Applied Mathematics and Computation* 235 (2014), pp. 454–468. ISSN: 0096-3003. DOI: https://doi.org/10.1016/j.amc.2014.02.051. URL: http://www.sciencedirect.com/science/article/pii/S0096300314002902.

[48] Rajib Nath, Stanimire Tomov and Jack Dongarra. 'An Improved MAGMA GEMM For Fermi Graphics Processing Units'. In: *International Journal of High Performance Computing Applications* 24.4 (Nov. 2010), pp. 511–515. ISSN: 1094-3420. DOI: 10.1177/1094342010385729. URL: http://dx.doi.org/10.1177/1094342010385729.

[49] T. Nelson et al. 'Generating Efficient Tensor Contractions for GPUs'. In: *2015 44th International Conference on Parallel Processing*. 2015, pp. 969–978.

[50] NVIDIA. *Automatic Mixed Precision for Deep Learning*. 2020. URL: https://developer.nvidia.com/automatic-mixed-precision.

[51] NVIDIA. *cuBLAS: CUDA Toolkit Documentation*. 2020. URL: https://docs.nvidia.com/cuda/cublas/index.html.

[52] NVIDIA. *CUDA C++ Programming Guide*. 2020. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide.

[53] NVIDIA. *cuDNN Developer Guide: NVIDIA Deep Learning SDK Documentation*. 2020. URL: https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html.

[54] NVIDIA. *cuTENSOR: A High-Performance CUDA Library for Tensor Primitives*. 2020. URL: https://docs.nvidia.com/cuda/cutensor/index.html.

[55] NVIDIA. *CUTLASS: CUDA Templates for Linear Algebra Subroutines*. 2020. URL: https://github.com/NVIDIA/cutlass.

[56] NVIDIA. *Deep Learning Performance Guide*. June 2019. URL: https://docs.nvidia.com/deeplearning/sdk/pdf/Deep-Learning-Performance-Guide.pdf.

[57] NVIDIA. *NVIDIA Turing Architecture whitepaper*. 2018. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[58] NVIDIA. *NVIDIA Unveils CUDA – The GPU Computing Revolution Begins*. Nov. 2006. URL: https://www.nvidia.com/object/IO_37226.html.

[59] NVIDIA. *NVIDIA V100*. 2020. URL: https://www.nvidia.com/en-us/data-center/v100.

[60] NVIDIA. *Parallel Thread Execution ISA Version 6.5*. 2020. URL: https://docs.nvidia.com/cuda/parallel-thread-execution.

[61] Adam Paszke et al. 'PyTorch: An Imperative Style, High-Performance Deep Learning Library'. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[62] Roman Poya, Antonio J Gil and Rogelio Ortigosa. 'A high performance data parallel tensor contraction framework: Application to coupled electro-mechanics'. In: *Computer Physics Communications* 216 (2017), pp. 35–52. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2017.02.016. URL: http://www.sciencedirect.com/science/article/pii/S0010465517300681.

[63] Pramod Ramarao. *CUDA 11 Features Revealed*. May 2020. URL: https://devblogs.nvidia.com/cuda-11-features-revealed/.

[64] J. Revels, M. Lubin and T. Papamarkou. 'Forward-Mode Automatic Differentiation in Julia'. In: *arXiv:1607.07892 [cs.MS]* (2016). URL: https://arxiv.org/abs/1607.07892.

[65] Jarrett Revels. *Cassette.jl: Overdub your Julia Code*. 2020. URL: https://github.com/jrevels/Cassette.jl.

[66] Norman Rink et al. 'CFDlang: High-level code generation for high-order methods in fluid dynamics'. In: *Real World Domain Specific Languages Workshop 2018*. Feb. 2018, pp. 1–10. DOI: 10.1145/3183895.3183900.

[67] E. Solomonik et al. 'Cyclops Tensor Framework: Reducing Communication and Eliminating Load Imbalance in Massively Parallel Contractions'. In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 2013, pp. 813–824.

[68] Paul Springer and Paolo Bientinesi. *Design of a high-performance GEMM-like Tensor-Tensor Multiplication*. 2016. arXiv: 1607.00145 [cs.MS].

[69] Paul Springer and Paolo Bientinesi. *The Landscape of High-Performance Tensor Contractions*. Feb. 2017. URL: http://www.netlib.org/utk/people/JackDongarra/WEB-PAGES/Batched-BLAS-2017/talk13-springer.pdf.

[70] Paul Springer and Chen-Han Yu. 'cuTENSOR: High-Performance CUDA Tensor Primitives'. In: *NVIDIA GPU Technology Conference 2019*. Mar. 2019.

[71] Field G. Van Zee and Robert A. van de Geijn. 'BLIS: A Framework for Rapidly Instantiating BLAS Functionality'. In: *ACM Trans. Math. Softw.* 41.3 (June 2015). ISSN: 0098-3500. DOI: 10.1145/2764454. URL: https://doi.org/10.1145/2764454.

[72] R. C. Whaley and J. J. Dongarra. 'Automatically Tuned Linear Algebra Software'. In: *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. 1998, pp. 38–38.

[73] Zhang Xianyi. *OpenBLAS: An optimized BLAS library*. 2020. URL: https://www.openblas.net.

[74] Da Yan, Wei Wang and Xiaowen Chu. 'Demystifying Tensor Cores to Optimize Half-Precision Matrix Multiply'. To appear in: *Proceedings of the 34th IEEE International Parallel and Distributed Processing Symposium*. 2020. URL: https://www.cse.ust.hk/~weiwa/papers/yan-ipdps20.pdf.

[75] Xiuxia Zhang et al. 'Understanding the GPU Microarchitecture to Achieve Bare-Metal Performance Tuning'. In: *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '17. Austin, Texas, USA: Association for Computing Machinery, 2017, pp. 31–43. ISBN: 9781450344937. DOI: 10.1145/3018743.3018755. URL: https://doi.org/10.1145/3018743.3018755.

FACULTY OF ENGINEERING AND ARCHITECTURE

Flexible matrix multiplication kernels on GPUs

Thomas Faingnaert
Student number: 01506418

Supervisor: Prof. dr. ir. Bjorn De Sutter
Counsellor: Dr. Tim Besard

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Academic year 2019-2020

GHENT UNIVERSITY