# Flexible Performant GEMM Kernels on GPUs

Thomas Faingnaert, Tim Besard, Bjorn De Sutter, *Member, IEEE*

**Abstract**—General Matrix Multiplication or GEMM kernels take centre place in high performance computing and machine learning. Recent NVIDIA GPUs include GEMM accelerators, such as NVIDIA's Tensor Cores. Their exploitation is hampered by the two-language problem: it requires either low-level programming which implies low programmer productivity or using libraries that only offer a limited set of components. Because rephrasing algorithms in terms of established components often introduces overhead, the libraries' lack of flexibility limits the freedom to explore new algorithms. Researchers using GEMMs can hence not enjoy programming productivity, high performance, and research flexibility at once. In this paper we solve this problem. We present three sets of abstractions and interfaces to program GEMMs within the scientific Julia programming language. The interfaces and abstractions are co-designed for researchers' needs and Julia's features to achieve sufficient separation of concerns and flexibility to easily extend basic GEMMs in many different ways without paying a performance price. Comparing our GEMMs to state-of-the-art libraries cuBLAS and CUTLASS, we demonstrate that our performance is in the same ballpark of the libraries, and in some cases even exceeds it, without having to write a single line of code in CUDA C++ or assembly, and without facing flexibility limitations.

**Index Terms**—matrix multiplication, graphics processors, high-level programming languages

## 1 Introduction

GEMM (General Matrix Multiplication) kernels form the core of many computations in the fields of HPC (High Performance Computing) and ML (Machine Learning). In HPC, GEMM is at the core of linear algebra [1], including dense linear algebra [2], [3], and is used for earthquake simulation [4], plasma visualisation [5], and weather and climate prediction [6]. In ML they are used to train neural networks including fully connected layers in traditional neural networks, convolutional neural networks, long short-term memory cells, and natural language processing [7], [8]. To accelerate their computations, researchers in the mentioned domains have relied on the massively parallel computing resources of GPUs (Graphics Processing Units).

To answer the demand for more efficient GEMMs, recent GPUs include matrix multiplication accelerators, such as NVIDIA's TCs (Tensor Cores) [9]. Researchers can exploit these resources in two ways. They can express their algorithms in high-level PLs (Programming Languages) such as Python and express them in terms of established GEMM variants for which efficient implementations are available in third-party libraries such as cuBLAS or CUTLASS. This approach offers high research productivity, at the cost of being limited to the APIs (Application Programming Interfaces) and GEMM implementations available in the libraries. In many domains, this lack of flexibility is problematic. When non-standard, more generalised GEMMs as needed in neural networks [10], convolutional networks [8], fluid dynamics [11], electromechanics [12], computational chemistry [13], or any other computation on multidimensional tensors [14], [15], [16], [17], [18], [19], [20], [21], [22], [23] are rephrased in terms of standard GEMM kernels available in libraries, additional custom kernels need to be launched in between the GEMM kernels for things such as precision conversions, layout conversions (transpositions), type conversions, bias operations, element-wise operations, etc. These extra kernels introduce huge overheads because they have a massive impact on the traffic to the very slow global memory.

Alternatively, researchers can rewrite the most demanding parts of their software in lower-level PLs such as CUDA C/C++ [24] or OpenCL [25]. This decreases their productivity, however, and they now require much more PL and GPU programming model knowledge outside their own application domain. In short, many researchers working with GEMM-like algorithms suffer from the *two-language problem*. They cannot achieve high performance, high research productivity, and algorithmic flexibility together.

The scientific PL Julia is designed to overcome the two-language problem [26]. It offers a high-level syntax, dynamic typing, managed memory, meta-programming, multiple dispatch, and other features that increase programmer productivity. Julia's compiler is based on type inference and just-ahead-of-time compilation, which allows it to generate code devoid of much of the run-time overhead (e.g. in the form of dynamic type checks) that other PLs pay for supporting the mentioned features. On CPUs (Central Processing Units), Julia code is comparable in performance to C, C++, and Fortran code [27]. Through the CUDA.jl package, it is possible to program NVIDIA GPUs directly in Julia, at a high abstraction level of arrays or at the lower-level of CUDA-like kernels [28], [29]. Before our research, TCs were not supported in CUDA.jl, however. The package and its high-level APIs were hence of limited use to many researchers.

In our research, we set out to overcome this issue in three steps. First, we developed support for TCs in the Julia compiler and libraries through a WMMA API of wrapper functions around so-called compiler intrinsics. This allows for exploiting TCs in kernels written with the lower-level support in CUDA.jl. This low-level API only focuses on the

• *T. Faingnaert and B. De Sutter are with the Department of Electronics and Information Systems, Ghent University, Belgium. T. Besard works for Julia Computing.*
*thomas.faingnaert@ugent.be;tim@juliacomputing.com*
*Corresponding author: bjorn.desutter@ugent.be*

*Manuscript received X, ; revised X.*

WMMA operation. It does not free the programmer from the cumbersome task of coordinating the memory traffic in the memory hierarchy to move data to and from the TCs. So secondly, we developed a tiling API in Julia that allows programmers to coordinate the memory traffic to and from TCs at a high abstraction level, and, importantly, without paying a price in terms of performance. The low-level API and the tiling API enable efficient use of TCs and the GPU memory hierarchy, but to reach good performance, the GEMM computations themselves, possibly with fused additional computations, then still need to be programmed at a rather low level of abstraction requiring a lot of expertise. In the final step, we therefore developed a high-level GEMM API in Julia that allows programmers to express and combine a range of extensions of basic GEMM computations in an abstract, intuitive way, without having to pay an unacceptable price in performance. Combined, these three APIs solve the two-language problem to a great extent with respect to hardware resources such as TCs.

Our main result is that we get performance in the same ballpark of hand-tuned libraries and in some cases even much better performance, without having to write a single line of code in a lower-level PL and without being limited to the specific GEMM versions supported by the libraries.

This paper focuses on the tiling and GEMM APIs. After providing the necessary background in Section 2, Section 3 discusses requirements for a tiling API, presents our novel way for abstracting tiling and the Julia API we designed based on that abstraction, and demonstrates and evaluates the API on a number of stages in GEMM computations. Section 4 discusses the requirements for flexibility in GEMMs in more detail. We present the different building blocks at the basis of our Julia GEMM API that provide that flexibility in an intuitive manner, and we demonstrate the API on a number of examples. In Section 5, our final contribution is a performance evaluation of multiple variants of GEMM computations, showing that we get relatively close to the performance of hand-tuned libraries like CUBLAS, CUTLASS, and CUTENSOR without having to write any single line in a lower-level PL. The paper then ends with a discussion of some related work in Section 6, the availability of our artefacts in Section 7, and with a conclusion and a look forward in Section 8.

## 2 Background

### 2.1 GPU programming

The main difference between programming GPUs versus CPUs is their underlying programming model. 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 [24]. Since our main interest is in NVIDIA GPUs, we limit our discussion to NVIDIA's CUDA programming model. *Threads* are the smallest unit of execution in the hierarchy. The hardware groups them into sets of 32 threads called *warps*. Threads in the same warp execute in a SIMT (Single Instruction Multiple Thread) fashion. These threads must hence execute the same instruction at the same time, possibly on different data. Threads are also grouped by the programmer into *blocks*. Threads in the same block can communicate efficiently, so that they can cooperate on a common task. Finally, 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, which correspond directly to levels in the thread hierarchy. The *register* file is the fastest type of memory. Each thread typically has access to 255 registers. Each block has its own set of *shared memory*, that may be used by threads in the same block to communicate. Finally, *global memory* can be accessed by all threads on the device, regardless of which block they belong to. Global memory has the largest capacity, but also has much higher latency and lower throughput.

To fully exploit the available resources on a GPU, programmers can either use low-level PLs like CUDA C/C++ [24] or OpenCL [25] to program their own kernels, or they can use the foreign function interface of high-level PLs such as Python to invoke kernels in libraries. The former option requires quite some knowledge in GPU programming models, forces the programmers to write quite some boilerplate code to manage data in memories and configure the kernels, and offers little performance portability, so manual (re)tuning of code is necessary when porting the code to different devices. Popular libraries such as CUBLAS contain kernel versions tuned for many different devices to overcome the performance portability issue.

### 2.2 Julia Programming Language

The open-source PL Julia features a high-level syntax [30]. A central paradigm in its design is the way it handles dispatch, the process by which the compiler chooses 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.

Julia's type system is *dynamic*, meaning that the types of expressions are not necessarily known statically. However, Julia inherits some of the advantages of static type systems through several features of its compiler. For one, the Julia compiler applies type inference to deduce the types of values used by the program. Code is then *specialised* based on this information, e.g. function calls are devirtualised, dynamic type checks are removed, etc. This style of compilation, dubbed *just-ahead-of-time*, has the performance of ahead-of-time compiled PLs with the flexibility of a just-in-time compiled one. We rely on this design to seamlessly compose a GEMM computation from all involved components, i.e. beyond what normal layering of libraries at different layers of abstraction allows as is typically done with other PLs.

Julia's compiler is built on top of LLVM, a compiler infrastructure project commonly used in research and industry [31]. 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.

The Julia package CUDA.JL reuses part of the Julia compilation process to allow executing kernels written in

Julia on NVIDIA GPUs [28]. In particular, the aforementioned compilation pipeline is run to the point where Julia IR is lowered to LLVM IR. The LLVM IR is intercepted and sent to the LLVM NVPTX backend instead of the backend of the host architecture. This NVPTX backend converts the IR to PTX (Parallel Thread Execution) instructions, the virtual instruction set of NVIDIA GPUs.

With CUDA.JL, it is possible to program NVIDIA GPUs at the lower-level of CUDA-like kernels and at the higher abstraction level of arrays [28]. The former involves less boilerplate and verbosity, and makes reusing code easier compared to programming in CUDA C/C++. The latter enables much more productive programming [29].

Julia's multiple dispatch enables transparent exploitation of performance-optimised functionality from popular GPU libraries. For example, for any function from the CUBLAS library, a Julia package can contain a generic implementation in pure Julia code that operates for all (numeric) data types and that hence accepts all arguments of type `Number`. In addition, the package can contain wrappers that each only accept a more concrete argument type such as `Float32` and that invoke the corresponding CUBLAS function for that type. Users of the package can then invoke the function on any type they want. If it is supported by the CUBLAS library, they will get optimal performance "for free".

### 2.3 Tensor Cores

Each TC performs a matrix multiply-accumulate expression of the form $D = A \cdot B + C$. TCs support a limited set of possible data types for these matrices. 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 TCs in C++ in the so-called WMMA (Warp Matrix Multiply Accumulate) API. 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, called a *fragment*. In the remainder of this paper, unless stated differently, we will assume 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. 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 $D$.
3) Store the resultant $D$ fragment to memory using a WMMA STORE operation.

In CUDA C++ each step corresponds to an overloaded C++ function. Calls to these functions are mapped one-to-one onto the corresponding WMMA PTX instruction by the compiler.

To add support for WMMA to CUDA.JL, we reused the pre-existing WMMA PTX intrinsics in the NVPTX backend. This necessitated adaptations to Julia's compiler, in particular to the code generation process. Our WMMA API consists of two different layers. The lowest layer consists of Julia wrapper functions that are mapped one-to-one to these intrinsics. The second layer is a high-level interface, similar to CUDA C++'s version of WMMA. It consists of `load_a`, `load_b`, `load_c`, `mma`, and `store_d` functions, which call the intrinsic wrapper corresponding to the argument types.

At its launch with the Volta architecture in 2017, WMMA only supported $16 \times 16 \times 16$ multiply-accumulates of FP16 matrices. More recent GPU architectures extend the interface with new data types and shapes. Turing's second generation TCs, introduced in 2018, add support for 8-bit, 4-bit, and 1-bit data types, along with new WMMA shapes depending on the data type used. The most recent version of WMMA includes support for FP64, bfloat16, and TF32 data types, and was launched in May 2020 with the introduction of Ampere.

TCs can also be used through libraries instead of WMMA. NVIDIA's CUDNN library contains TC kernels for common ML algorithms. ML frameworks such as TensorFlow, PyTorch, and MXNet use CUDNN for training and inference. CUBLAS, CUBLASLT, and CUTLASS contain optimised GEMM kernels for HPC applications. NVIDIA's CUTENSOR builds on CUTLASS and contains Tensor-Core-accelerated kernels for tensor computations.

## 3 Abstractions for Recursive Blocking

### 3.1 Requirements

Matrix multiplication is rich in data reuse. For example, multiplying square matrices of size $N$ requires $\mathcal{O}(N^3)$ floating point operations, but only $\mathcal{O}(N^2)$ storage, so each element is reused roughly $\mathcal{O}(N)$ times. To exploit this reuse, data needs to be re-accessed as much as possible in faster memories. When all data does not fit into the fastest memories, the transfers between different memories in the hierarchy need to be coordinated carefully to maximise reuse.

For GEMMs, the general idea is to copy tiles of the input matrices up into the memory hierarchy: from global memory to shared memory and from there to registers. The size of the tiles in each step is chosen such that they fit in the available memory. As computations of different tiles of the resultant matrix are independent, those can be performed completely in parallel to maximise the resource utilisation of massively parallel GPUs. Because of the one-to-one mapping between levels of threads and the memory hierarchy, each of the tiled copy operations is also performed cooperatively, by all threads in the relevant part of the thread hierarchy.

Consider again the GEMM of $D = A \cdot B + C$. With tiling, this GEMM will consist of the following stages:

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 dimension $K$, stride = block tiling size.
   a) Copy a tile of $A$ from global memory to shared memory, cooperatively by all threads in a block.
   b) Do the same for a tile of $B$.
   c) Iterate over dimension $K$, stride = warp tiling size.
      i) Copy a tile of $A$ from shared memory to registers, cooperatively by all threads in a warp.
      ii) Do the same for a tile of $B$.
      iii) Compute a tile of $D$, given the $A$, $B$, and $C$ tiles, cooperatively by all threads in a warp.

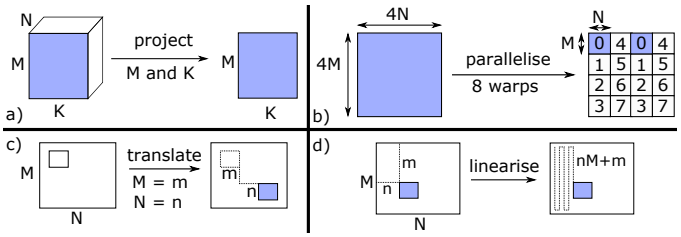


Figure 1. Projection, parallelisation, translation and linearisation of tiles.

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.

For a WMMA GEMM, stages 2, 3.c.i, and 3.c.ii correspond to WMMA LOAD operations, stage 3.c.iii to MMA operations, and stage 4 to WMMA STORE operations.

This form of recursive blocking is an absolute requirement to achieve good performance, but it is also complex to program. Tile sizes need to be chosen in function of the hardware and the number of dimensions and the sizes of the data, indexing of the matrices depends on their layouts, optimal tiling parameters can differ between memory and computational stages. Determining the tiling parameters is complex, and encoding all address computations in the actual code is cumbersome, error-prone, and results in code that is hard to comprehend, port, and maintain. To make it easier to program general GEMM computations with recursive blocking, we developed a novel API with which the tiling computations can be abstracted to a much higher level. The requirements we put forward for this API are the following:

- *Code readability* to ease writing kernels that use blocking.
- *Zero performance cost* compared to manually expressed address computations of all tiles.
- *Support for multiple dimensions* (>2) to support TCTs (Tensor ConTractions) and batched GEMMs.
- *Recursive blocking* for the different levels of the memory hierarchy through independent tiling parameters.
- *WMMA-compatibility* to exploit WMMA.

### 3.2 Abstract Operations

To meet the requirements, we propose a novel abstraction consisting of four different operations on tiles.

*Projection* is the first abstraction. 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 1a. The projection abstraction thus allows us to easily reduce the original three-dimensional tile to a tile containing only the relevant dimensions.

*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 thread blocks or warps. Consider the example in Figure 1b. 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. The set of all cooperating warps do not need to cover the entire tile. In the 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.

*Translation* moves a tile over a specified distance in each dimension. In the example of Figure 1c, 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* is used to convert a tile's location from a Cartesian index to a linear index. This is needed to calculate the offset of a tile in memory, relative to the base pointer of the parent tile. In the example of Figure 1d, we consider a subtile at a Cartesian offset of $(m, n)$ from its parent tile with size $(M, N)$. Linearisation results in the linear offset of this tile, relative to the top-left corner of the parent tile. 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 $nM + m$.

### 3.3 A tiling API for Julia

To overcome challenges in developing a concrete API based on the four abstractions while meeting all requirements, we relied on some high-level Julia features.

First, a tile is fully determined by its position and its size. Our tiling API contains a `Tile` struct that stores this information. Storing the size in a field of this struct does, however, not suffice to meet the zero-cost requirement. In Julia, each function is JIT-compiled once for each combination of argument types occurring during the execution of the program. During each such JIT-compilation, no specialisation takes place based on the values of the arguments. In order for the compiler to generate high quality code for the different stages in tiled GEMMs, it needs to know how many registers are needed when transferring slices into registers. In other words, the sizes of the tiles need to be available at compile time, such that specialised code can be generated per tile size. Moreover, no dynamic type checking should be necessary in the generated code. To obtain the required specialisation, yet avoid that any dynamic type checks are needed, we defined `Tile` to be a parameterised type, where one of the type parameters (rather than a field) is the size of the tile. Julia's type inference can then obtain all the necessary information to enable specialised code generation in the JIT compiler without running into type instability issues [30].

Secondly, we observe that when we want to implement GEMM using tiling, we typically do not think in terms of

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

Listing 1: The definition of a Tile in the Julia tiling API.

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 $M$ and $K$ dimensions. Rather than writing the position as `pos[1]`, we can increase readability by naming the dimensions, so that we may write `pos.M`. This form of syntactic sugar can easily be achieved with the existing Julia type `NamedTuple`, which we use to store both the position and size of a tile.

Thirdly, we observed a form of structural bias in the Julia-LLVM tool flow with respect to address computations such as those typically occurring in recursive blocking code. The problem is that in many stages of the tiled computation, different threads operate on tiles at different positions in the input matrices or tensors. If the position stored in the `Tile` is simply a single position, unnecessarily complex PTX code is generated. However, if the position is split into a thread-dependent base index and a thread-independent offset, the compiler generates code that efficiently exploits the available register + constant addressing mode.

The final definition of the parameterised `Tile` type is shown in Listing 1. With this definition, 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 element-wise sum of the original tile's base and the argument `dist`. This essentially moves the multidimensional tile over the distance specified by the argument.

The first argument of `linearise(coord, dim)` represents the coordinate of the tile. 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 `dim` 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 number of dimensions.

One option to project tiles is to define a function `project(tile, dims)`, where `dims` contains a list of the dimensions to keep. A projection of a tile to the $M$ and $N$ dimension could then be written as `project(tile, (:M, :N))`. We instead opted to use Julia's extensibility. In Julia, the syntactic construct `a.b` is converted to a call to `Base.getproperty(a, :b)` [30]. Through the multiple dispatch mechanism, we override this function such that one can express the project operation as `tile.MN` instead of `project(tile, (:M, :N))`.

Listing 2 shows 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

```
1  @inline Base.getproperty(tile::Tile{size, names, T},
   ↪  sym::Symbol) where {size, names, T} =
   ↪  getproperty_impl(tile, Val(sym))
2
3  @generated function getproperty_impl(
   ↪  tile::Tile{size, names, T}, ::Val{sym})
   ↪  where {size, names, T, sym}
4    if sym == :base || sym == :offset
5      # standard fields
6      return quote
7        getfield(tile, sym)
8      end
9    else
10     # tile projection
11     sym_str = String(sym)
12     new_names = ntuple(
       ↪  i -> Symbol(sym_str[i]), length(sym_str))
13
14     return quote
15       # create new NamedTuples with the correct dimensions
16       new_base   = ...
17       new_offset = ...
18       new_size   = ...
19
20       # return projected tile
21       return Tile{new_size, new_names, ...}(
         ↪  new_base, new_offset)
22     end
23   end
24 end
```

Listing 2: Tile projection overview in our tiling API.

immutable and only one copy of each distinct value is stored [30]. 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 set 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 at type-inference time and depending on the argument types, as shown on line 3 of Listing 2. Since we moved the field name to the type domain, we can thus generate a different, specialised implementation for each projection. 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`. Lines 4–8 handle this by checking if the passed symbol is `base` or `offset`. If so, we just return the value of the field by calling `getfield`. Julia's `@generated` functions must return an `Expr`, which is a block of code to be compiled. Such blocks are surrounded with the `quote ... end` construct, as shown in lines 6–8.

The projection itself is implemented in lines 10–22. Line 11 converts the symbol representing the field name to a `String`, which line 12 then converts 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 projected tile, and returns that tile.

The parallelise operation is exposed as a function call `parallelise(tile, tiling_size, index, count)`. The `tile` argument of type `Tile` is the parent tile that will be subdivided and parallelised over a set of entities that can be 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 integer from 0 to `count - 1`, and determines the identifier of the currently executing entity.

Figure 2 shows an example parallelisation. It starts with a parent tile of size $4m \times 2n$, divides it in subtiles of size $m \times n$, and parallelises them across 2 warps. The 0/1 in each subtile indicates the warp responsible for it. We write the operation as `parallelise(Tile(M = 4 * m, N = n), Tile(M = m, N = n), warpId, 2)`, where `warpId` is either 0 or 1, i.e. the id of the currently executing warp.

To generalise the parallelisation operation 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 `parallelise` 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. 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.

All operations on `Tile`s in our API are built on top of Julia interfaces that work for any 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. This supports higher dimensions as required.

### 3.4 Example Usage

To illustrate the use, readability and zero cost of the API, we consider three representative stages of the tiled GEMM.

#### 3.4.1 Copying a tile of C from global to shared memory

To copy a tile of $C$ from global to shared memory in step 1 of the complete GEMM, Listing 3 implements the approach illustrated in Figure 3. Each block copies a separate tile, and we launch the GEMM kernel with enough blocks to fully cover the $C$ matrix. The tile size is determined by the `block_tile` variable. It initially has three dimensions, so we first project it to the $M$ and $N$ dimension using `block_tile.MN` on line 1 in Listing 3.

Next, we divide `block_tile` in subtiles and parallelise the resulting `warp_tile`s over a set of `WARPS_PER_BLOCK` cooperating warps in the block, also on line 1. The `@unroll` macro from the Julia package GPUIFYLOOPS.JL [32] 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)`.

Typically, `MEM_CD_WARP.N` is 1, so that the resulting `warp_tile` is highly rectangular. This is necessary to access

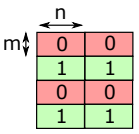


Figure 2. Parallelisation over 2 warps each handling a $4 \times 2$ set of subtiles.

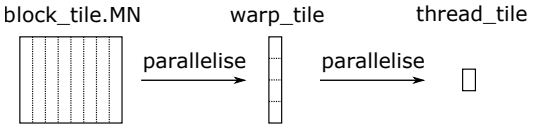


Figure 3. Copying a tile of the $C$ matrix from global to shared memory.

global memory efficiently, as this guarantees that the threads in one warp access adjacent memory locations. The hardware is then able to coalesce these memory accesses into fewer memory transactions, thus increasing memory throughput. This is commonly referred to as *global memory coalescing*.

Similarly, we parallelise the `warp_tile` over the set of 32 threads in a warp on line 2. The integer variable `laneId` 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 (Single Precision Floating Point) $C$ 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, such that the GPU can issue one 128-bit load/store, the largest memory transaction size supported by the GPU, thus maximally vectorising the memory accesses.

The positions of all 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 5–8 convert the base and 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 10–11, 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.

Listing 4 is equivalent to Listing 3, but does not use our tiling API. The `BLOCK_M` and `BLOCK_N` variables on line 1 are constants that correspond to the size of `block_tile` in Listing 3. The outer loop on lines 1–6 corresponds to the first parallelisation, the inner loop on lines 8–13 is the equivalent of the second parallelisation. In both loops the tile bases and offsets are calculated manually. Lines 15–18 convert the bases and offsets to linear indices, and are thus the equivalent of the linearisations in Listing 3. The translation is handled by the addition of the translation offsets `block_i` and `block_j` on line 15. Clearly the use of our tiling API in Listing 3 is less verbose and more maintainable.

Listing 5 shows the CUDA PTX to which Listing 3 is compiled. First, each thread's base addresses are computed in registers `%rd20` and `%rd13` for shared and global memory, respectively. The loads and stores are vectorised, as indicated

```
@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)

    value = vloada(Vec{MEM_CD_THREAD, Float32}, global_ptr, global_linear_offset)
    vstorea!(Vec{MEM_CD_THREAD, Float32}, shared_ptr, value, shared_linear_offset)
  end
end
```

Listing 3: Copying a tile of the $C$ matrix from global to shared memory using our tiling API.

```
@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)

    value = vloada(Vec{MEM_CD_THREAD, Float32}, global_ptr, global_linear_offset)
    vstorea!(Vec{MEM_CD_THREAD, Float32}, shared_ptr, value, shared_linear_offset)
  end
end
```

Listing 4: Implementing the first stage in GEMM using manual calculation of addresses.

by the suffix `v4.f32`. The stores to shared memory are on lines 6, 12, and 20. As the shared memory size is known at compile time, the code exploits the register plus constant addressing modes as discussed in Section 3.3. By contrast, the compiler does not know the size of the matrix in global memory, so it does not know the linearised offset either, 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 bit shift `shl.b64`), and an addition.

The code in Listing 5 is identical to the code that the Julia-LLVM tool flow generates for Listing 4. We conclude that no superfluous instructions are generated because of the use of our tiling API, for both the loads from global memory and the stores to shared memory.

We can use similar code for steps 2, 3-a, 3-b, 3-c-i, and 3-c-ii of the GEMM, with independently chosen tile configurations for each of them. This way, recursive double-sided blocking is supported.

```
1  // Calculate the base addresses in %rd13 and %rd20...
2  shl.b64             %rd22, %rd13, 5;
3  add.s64             %rd23, %rd17, %rd22;
4  cvta.to.global.u64 %rd24, %rd23;
5  ld.global.v4.f32    {%f5, %f6, %f7, %f8}, [%rd24];
6  st.shared.v4.f32    [%rd20+4096], {%f5, %f6, %f7, %f8};
7
8  shl.b64             %rd25, %rd13, 6;
9  add.s64             %rd26, %rd17, %rd25;
10 cvta.to.global.u64 %rd27, %rd26;
11 ld.global.v4.f32    {%f9, %f10, %f11, %f12}, [%rd27];
12 st.shared.v4.f32    [%rd20+8192], {%f9, %f10, %f11, %f12};
13
14 // ... repetition of similar blocks due to unrolling
15
16 mul.lo.s64          %rd64, %rd13, 480;
17 add.s64             %rd65, %rd17, %rd64;
18 cvta.to.global.u64 %rd66, %rd65;
19 ld.global.v4.f32    {%f61, %f62, %f63, %f64}, [%rd66];
20 st.shared.v4.f32    [%rd20+61440], {%f61, %f62, %f63, %f64};
```

Listing 5: The PTX code generated for Listing 3.

### 3.4.2 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 4. 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. Let us consider the case where a `block_tile` has size $(M, N, K) = (128, 128, 16)$. This means that each block calculates an $M \times N = 128 \times 128$ tile of $D$, by multiplying all $M \times K = 128 \times 16$ tiles in a row of $A$ with all $K \times N = 16 \times 128$ tiles in a column of $B$. These tiles of $D$ 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, each block contains 8 warps, in a $2 \times 4$ arrangement. Each warp calculates a $64 \times 32$ tile of $D$ in each iteration, by multiplying a $64 \times 16$ tile of $A$, and a $16 \times 32$ 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. In the case where the matrices are stored in column-major, the warps are assigned to tiles in the order

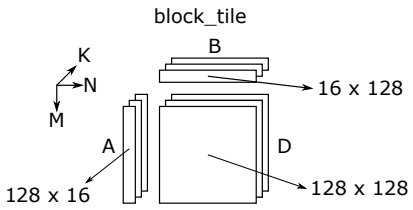


Figure 4. 3D iteration space in the inner loop of the matrix product.

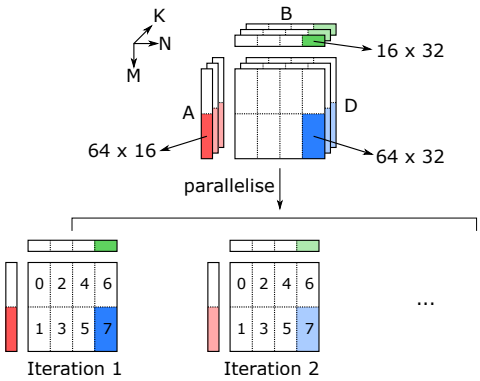


Figure 5. Computation of the matrix product in the innermost loop.

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. 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, so with this choice of tiling size, the parallelisation operation implicitly iterates over the $K$ dimension.

Line 1 of Listing 6 shows this parallelisation operation. It 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 gives a tile of size $(M, K) = (64, 16)$, which thus consists of four $16 \times 16$ WMMA fragments. To load those, 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 $64 \times 32$ product of $D$ using the `WMMA.mma` function from our WMMA API.

This example is perhaps the best illustration of the tiling API, as it combines all four operations on tiles: parallelisation, projection, translation, and linearisation. Using these four operations significantly improves readability compared to writing the necessary address calculations by hand.

We omit the PTX code generated for this listing because it provides no additional value, but we confirm similar observations as on the first example: the base addresses of $A$ and $B$ for each warp are calculated once, and stored in registers. The code in Listing 6 is converted to a set of `wmma.load.a`, `wmma.load.b`, and `wmma.mma` instructions,

```
1  @unroll for warp_tile = parallelise(block_tile,
   ↪  Tile(M=64, N=32, K=16), warpId, 8)
2    @unroll for i = 1 : 4
3      a_tile = translate(warp_tile.MK, (M=(i-1)*16, K=0))
4      linear_base = linearise(a_tile.base, ...)
5      linear_offset = linearise(a_tile.offset, ...)
6      a_frags[i] = WMMA.load_a(...)
7    end
8    @unroll for j = 1 : 2
9      b_tile = translate(warp_tile.KN, (K=0, N=(j-1)*16))
10     linear_base = linearise(b_tile.base, ...)
11     linear_offset = linearise(b_tile.offset, ...)
12     b_frags[j] = WMMA.load_b(...)
13   end
14   @unroll for i = 1 : 4
15     @unroll for j = 1 : 2
16       acc_frags[i, j] = WMMA.mma(a_frags[i], b_frags[j],
         ↪  acc_frags[i, j], ...)
17     end
18   end
19 end
```

Listing 6: Matrix product computation using our tiling API.

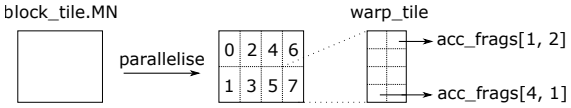


Figure 6. Copying a tile of $D$ from registers to shared memory.

and 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.

### 3.4.3 *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 stage, 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 6. `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 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 three dimensions $M$, $N$, and $K$. Figure 6 uses the same tiling sizes as our previous example: `block_tile` is a $128 \times 128$ matrix, and is parallelised across `WARPS_PER_BLOCK` = 8 warps, each handling a `COMPUTE_WARP.M` $\times$ `COMPUTE_WARP.N` = $64 \times 32$ subtile. The corresponding parallelisation operation returns a `warp_tile`, and is shown on line 1 of Listing 7. Note that the for loop of line 1 only has 1 iteration in this case, since 8 warps fully cover the `block_tile`.

Finally, this `warp_tile` is divided in a $4 \times 2$ 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.

```
@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 7: Copying a $D$ tile from registers to shared memory.

Again, we omit the generated PTX code, but we can confirm our earlier observations. 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.

## 4 Flexible GEMM kernel abstractions

In the previous section, we designed tiling abstractions to implement performant GEMM kernels. In this section, we add the necessary flexibility to this GEMM, such that users can instantiate a wide range of GEMM variants.

### 4.1 Requirements

The Google Brain DL (Deep Learning) research team provides an excellent overview of why this flexibility is needed [10]. They focus on Capsule networks, a novel neural network ML idea where the neurons are matrix-valued rather than scalars [33]. In short, they observed that the inflexibility of existing ML frameworks TensorFlow [34] and PyTorch [35] forced the researches to rephrase their computations in terms of the limited set of GEMM kernels already supported by these frameworks. They had to insert multiple data transposition and matrix materialisation stages that introduced detrimental amounts of memory access overhead. They also had to insert separate kernels in between their network layers to perform simple but not-yet-established operations on the matrix elements. Not being able to fuse those operations in the GEMM kernels themselves, this again introduced massive amounts of overhead. Clearly, for advanced research in a domain such as ML, libraries providing only established GEMM functionality do not suffice.

A flexible GEMM also needs to support a multitude of different memory layouts. Basic GEMMs involve only row-major and column-major layouts. Convolutions, which are also implemented with GEMMs, involve more dimensions than matrices, so more layouts need to be considered. For images, e.g. ML frameworks typically use four dimensions: a batch of $N$ images with $C$ channels, each consisting of $W \times H$ features. Among the many possible choices, NCHW and NHWC are most common [8].

Next, we consider the generalisation of matrix multiplications to multidimensional TCTs, which are common in several scientific fields, such as fluid dynamics [11], electromechanics [12], and computational chemistry [13]. Whereas a matrix-matrix multiplication has three different indices $\{m, n, k\}$, TCTs involve an arbitrarily large set of indices. Matrix transpositions are extended to arbitrary permutations of those indices. The number of possible data layouts for TCTs is hence much higher. For example, a TCT of 4D tensors has a total of $4! \times 4! \times 4! = 13824$ different memory layouts.

Given the importance of TCTs, a lot of research has been done to implement efficient support for the large number of possible cases. Springer and Bientinesi classify the traditional approaches to TCTs in three main categories [14]: loop nesting [15], [36], [37], [38], LoG [17], [18], and TTGT (Transpose-Transpose-GEMM-Transpose) [19], [20]. All three of them suffer from serious performance issues due to bad data reuse or the need to insert data reshuffling and transposition kernels.

In 2016, Springer and Bientinesi proposed another method for TCTs, GETT (GEMM-like Tensor-Tensor contraction) [38], that has since been adopted by other TCT implementations [21], [22]. GETT is based on the principles of TTGT, but implicitly reorganises tensors while loading them to avoid separate transpositions. GETT can therefore be seen as a variant of TTGT, where the transposes are fused into the GEMM kernel. Clearly, this fusion requires that the underlying GEMM kernel is flexible.

While the tiling and WMMA APIs introduced in previous sections allow that flexibility, programming against them would still be quite cumbersome. We hence propose a higher-level API based on a high-performance GEMM kernel that can easily be customised through a set of higher-level abstractions that are as intuitive as possible to researchers from, e.g. the domains of ML and DL. We put forward the following main requirements for this high-level GEMM API:

- *Flexibility* with respect to data layouts, on-the-fly data transpositions, and fused operations are prime objectives. Support for other, more complex data types such as complex numbers [2] or dual numbers as used in automatic differentiation [39] is also of interest.
- *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. This obviously implies that our GEMM should be WMMA-compatible and support double-sided recursive blocking, i.e. independent tiling parameters need to be supported for the data transfer stages at different levels of the memory hierarchy and for the computational stages.
- *Portability*: GEMM kernels built with our API should perform well on a range of devices. We should hence make as few assumptions about the underlying hardware as possible. For example, our API needs to be able to handle different shared memory sizes, as well as GPUs with and without TCs of different generations.

### 4.2 A flexible, abstract GEMM API for Julia

Our strategy is to implement the general structure of a performant GEMM kernel beforehand. To make it flexible, we split this GEMM in a small set of building blocks with a predetermined interface. Concretely, the GEMM contains calls to a set of functions with a predetermined name. To extend the basic implementation, it will suffice to implement new versions of the called functions that customise the behaviour based on their input types. Julia's

just-in-time type inference and compilation flow enables us to perform this split without introducing performance overhead. Furthermore, Julia's multiple dispatch allows us to make the split orthogonally, which results in more intuitive building blocks and eases code reuse and hence programmer productivity.

#### 4.2.1 Params

We of course 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, others specify the kernel's launch configuration, such as the number of warps per block. 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 TCs, but future work could expand these heuristics to other cases as well.

#### 4.2.2 Layouts

The positions of tiles at different levels in the memory hierarchy 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 knows 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. With these functions, users 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.

For a classic GEMM kernel, the most obvious instantiations of the layout building block are `RowMajor` and `ColumnMajor`. As mentioned before, each of these 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 parameterised 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`.

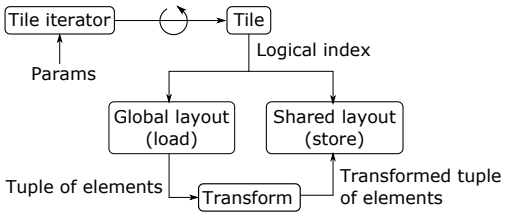


Figure 7. Copying an $A$ tile from global to shared memory.

The layout building block can also be used to create a GEMM with a more complicated mapping between logical indices and physical offsets. For example, GETT's reinterpretation of multidimensional tensors as matrices 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 materialised 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 or diagonal 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.

#### 4.2.3 Transforms

The next building block is that of *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 in the GEMM. By having a transform after every load and before every store, element-wise operations to the input and result matrices can be applied consistently in our API.

Transforms can serve for element-wise operations, such as a simple scaling in the case of GEMM, and for 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.

Figure 7 illustrates how params, layouts, and transforms 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.

#### 4.2.4 Operators

The previous building blocks together copy tiles from global to shared memory. The purpose of the next building block, called *operators*, is to load tiles from shared memory, perform the matrix multiplication, and store the resulting tile back to shared memory. To do so, this 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`.

The listed functions map one-to-one onto the steps of the WMMA API mentioned in Section 2.3. This is no coincidence, as both the operator building block and the WMMA API are warp-level matrix multiply-accumulate operations. It is hence fairly easy to define an implementation of the operator building blocks that uses TCs 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, it 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 or dual numbers. The operator building block also improves the portability of the GEMM kernel. For example, the WMMA operator may be parameterised with the WMMA shape, so that we can select the WMMA shape that is optimal for our GPU. Alternatively, we can define an alternative operator that calculates the matrix product using the traditional FPUs (Floating Point Units) instead of TCs on devices that lack the latter.

```
@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 8: Copying a $C$ tile from global to shared memory.

#### 4.2.5 Epilogues

While the already discussed transform abstraction already allows performing element-wise operations on the $D$ matrices, we add an epilogue abstraction to our API to enable customisation of the way global memory is updated at the last stage of GEMM. This enables, e.g. to apply a reduction operation across all threadblocks.

In the general GEMM implementation, our epilogue building block only has one purpose: to copy tiles of the resultant matrix from shared memory to global memory. 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.

### 4.3 Example Uses

#### 4.3.1 Fully-featured interface

As a first example of how to use the presented building blocks and API, we consider the first step in a GEMM kernel: copying a tile of the $C$ matrix from global to shared memory. Listing 3 showed an implementation of this step using our tiling API. In our GEMM API, this first step is implemented as shown in Listing 8. The code has a similar structure to Listing 3, 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.

Now 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 9. 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 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,

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

conf = GemmKernels.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}
                        )

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

Listing 9: Matrix product $D_{ij} = \max(\sum_k A_{ik} \cdot B_{kj} + C_{ij}, 0)$.

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

alpha = rand(Float32)
beta  = rand(Float32)

GemmKernels.BLAS.gemmEx!('N', 'N', alpha, a, b, beta, c)
```

Listing 10: A matrix product using our BLAS-like interface.

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 using this configuration is executed in lines 13–16. The transform that should be applied when copying tiles of the resultant $D$ matrix from the register file to shared memory is determined by the argument `transform_regs_to_shared_d`. The call to `GemmKernels.matmul` will execute each step in the GEMM kernel, using the components given by the user. For example, Listing 8 will be executed with `GLOBAL_C_LAYOUT = Layout.AlignedColMajor{Float32}`. We conclude that we can instantiate and launch customised GEMM kernels easily, without sacrificing flexibility.

#### 4.3.2 BLAS-like interface

The interface of the previous section exposes maximal flexibility to the user, but differs from the interface used by CUBLAS. We also provide a more familiar BLAS-like interface which can be used if not all flexibility is needed. This interface supports all operations of CUBLAS's `gemmEx`, i.e. linear scaling and transposition of the input matrices, but, importantly, with support for many more input types.

To ease the transitioning process, this BLAS-like interface has the same signature as CUDA.JL's `gemmEx` wrapper. To use our GEMM kernels in existing code, it suffices to replace `CUDA.CUBLAS.gemmEx!` by `GemmKernels.gemmEx!`.

Using the BLAS interface, there is no need to specify each component manually. Instead, they are derived from the types of the arguments. For example, Listing 10 calculates the matrix product $C := \alpha \cdot A \cdot B + \beta \cdot C$. Based on the combination of the types of the $A$, $B$, and $C$ matrix, our implementation of the BLAS-like interface instantiates a GEMM kernel with `operator = Operator.WMMAOp{16, 16, 16}, global_a_layout = Layout.AlignedColMajor{Float16}`, etc.

Using the BLAS interface in library code allows extending this library with new types, without changing the library code. For example, the library code in Listing 11 is called

```
function library_code(a::CuArray, b::CuArray, c::CuArray)
  # ...
  GemmKernels.BLAS.gemmEx!('N', 'N', alpha, a, b, beta, c)
  # ...
end
```

Listing 11: Library code making use of our BLAS-like API.

```
for f in (:load_a, :load_b, :load_c, :store_d)
  @eval @inline $f(op, ::Type{Layout.Padded{L, P}}, args...)
  ↪  where {L, P} = $f(op, L, args...)
end
```

Listing 12: Redirecting operator calls for padded layouts to the underlying layout, using Julia's metaprogramming.

for GPU arrays, but does not impose any restrictions on the element type. Using custom element types is as simple as adding support for them in our GEMM framework, and calling the library code with this new type.

### 4.4 Discussion

For our GEMM framework design, we have strived for orthogonality of different components. For example, epilogues and operators only interact via shared memory and can be combined arbitrarily as long as they both support the same shared memory layout. Transforms use broadcast expressions that work for different data types and array lengths, and can hence be combined with different layouts or params.

Nevertheless, some inevitable coupling between different components remains. Most prominently, layouts are coupled to epilogues (e.g. a bias epilogue can require different logic for row-major and column-major layouts), and to operators (e.g. WMMA supporting padded and non-padded layouts). Luckily, we can reduce the impact on code verbosity and reuse through several features of Julia. Epilogues can contain layout-agnostic code, and rely on fine-grained method overloading for layout-specific code paths. Metaprogramming can be used to redirect operator calls for padded layouts to the underlying layout, as illustrated in Listing 12.

Another point that merits some discussion is the extent to which our APIs and abstractions are Julia-specific, i.e. whether or not parts of them can be used for similar APIs in other PLs. While none of our proposed abstractions are Julia-specific, implementations in other PLs would suffer from reduced code reuse, or increased overhead and verbosity. Julia's unique combination of multiple dispatch, type inference, and JIT compilation allows us to compose GEMM operations from different components, without incurring any run time overhead. Due to the parametric nature of Julia's type system, tile sizes can be moved to the type domain, such that specialised code can be generated per tile size. Julia's metaprogramming capabilities prove extremely powerful to improve code reuse and reduce code verbosity.

It is also possible to call Julia functions from C code, using Julia's C API. Since most PLs can call C functions, our framework's kernels can be used in C++, Python, C#, and other high-level PLs, and hence also in ML frameworks written in these PLs, such as TensorFlow and PyTorch.

Finally, we should discuss the issue of portability. So far, we focused on flexible GEMMs for CUDA-enabled GPUs. Nevertheless, the abstractions in our tiling API and

flexible GEMM API are vendor-agnostic, and we expect they can be reused for AMD and Intel GPUs. More concretely, porting our framework necessitates two changes. First, our WMMA operator needs to be replaced with an operator using traditional floating point hardware. Secondly, our template kernel contains CUDA-specific concepts such as `threadIdx`, and hence needs to be ported to OpenCL. Code duplication can be avoided using the package KernelAbstractions.jl that allows writing vendor-agnostic GPU kernels [40].

## 5 Evaluation

To evaluate the performance and flexibility of our APIs, we created the necessary components for five GEMM variants as discussed below. Run times were measured on an NVIDIA RTX 2080 Ti with NVIDIA Nsight Compute and with BenchmarkTools.jl, which continues sampling until the standard deviation becomes small enough. We compare the performance of our kernels to CUTLASS 2.2, cuTENSOR 1.3.0 and cuBLAS 11.2. We set the latter's math mode to `CUBLAS_TENSOR_OP_MATH` and call `cublasGemmEx`. We use CUDA 11.0, CUDA.jl 2.0, and Julia 1.5.

### 5.1 Mixed-precision GEMM

Our first example is a normal mixed-precision GEMM, i.e. a computation of the form $D = A \cdot B + C$. This operation is directly supported by NVIDIA's cuBLAS library. To use TCs in our GEMM framework, we create an operator that simply calls the correct WMMA functions in our WMMA API for each step in the GEMM's inner loop.

In a GEMM, the $A$ and $B$ matrices may be stored in a column-major memory layout (N), or a row-major memory layout (T). We hence implemented both a `ColMajor` and `RowMajor` layout component. These layouts are suitable for global memory, but lead to inefficient memory accesses in shared memory. On NVIDIA GPUs, shared memory is split into a set of *memory banks*. Memory accesses to addresses that map to the same bank, so-called bank conflicts, are serialised.

The simplest way to reduce these bank conflicts is to add padding to every column or row, such that the mapping of matrix elements to banks is changed. To achieve this, we use a `PaddedLayout` component to store matrices in shared memory. This layout serves as a wrapper for other layouts, e.g. `PaddedLayout{ColMajor, 8}` is a column-major layout, where every column is padded by 8 elements.

Out of the three functions associated with layouts, only `size` needs to be specialised for each type of padded layout. This is necessary because padding differs for, e.g. row-major layouts vs. column-major layouts. Calls to `load` or `store` are automatically redirected to the underlying layout. As a result, supporting padding only required adding 14 lines of source code. To use padded layouts for a GEMM, it suffices to set, e.g. `shared_a_layout = Layout.PaddedLayout{Layout.ColMajor, 8}`, similarly to Lines 9–10 in Listing 9.

The epilogue for mixed-precision GEMM simply copies a tile from shared memory to global memory.

Figure 8 compares the performance of our mixed-precision GEMM to CUTLASS and cuBLAS. The different lines and markers represent the four combinations of the

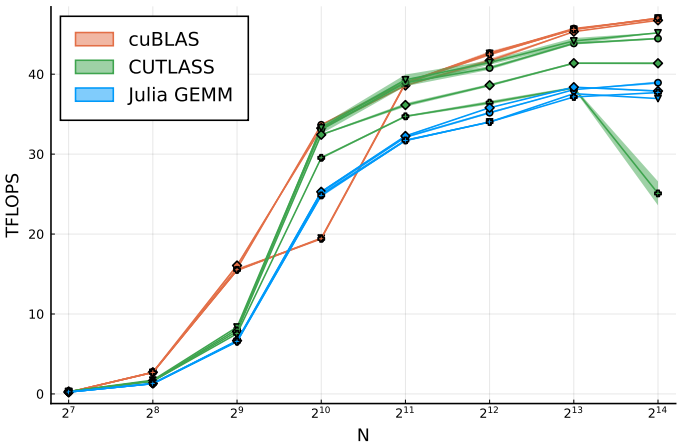


Figure 8. Performance of mixed-precision GEMM.

data layouts of the $A$ and $B$ matrices, the shaded regions represent error margins. We have no explanations for the two anomalous results in the measurements for CUTLASS and cuBLAS. They occurred consistently over many experiments. For the most interesting, larger matrices with N=2048 and more, the performance of our kernels ranges between 82% and 86% of cuBLAS, the best performing library. We conclude that our kernels achieve reasonably good performance, despite being written completely in Julia. To the best of our knowledge, no existing implementation purely written in a single higher-level PL comes close.

The performance difference between our kernel and the state-of-the-art in CUTLASS and cuBLAS can be attributed to two factors. First, our implementation does not yet contain data swizzling to avoid bank conflicts in shared memory. A device-dependent layout is necessary for swizzling optimally for a GPU's shared memory implementation, which we have not yet explored. Secondly, cuBLAS does not use WMMA, but accesses TCs directly, allowing the use of custom memory layouts in shared memory that perform better than our padded layouts. Implementing this in our framework necessitates a new layout and a new operator component. This custom operator would use the `mma` family of instructions instead of WMMA. Contrary to WMMA, `mma` does not have load or store instructions, so distribution of matrix elements to different threads needs to be done explicitly. Use of the `mma` operator would hamper portability, though, because some `mma` instructions are optimised for a particular GPU architecture. Engineering this is future work.

### 5.2 Diagonal matrices

Next, we focus on a mixed-precision GEMM where the $A$ or $B$ matrix is a diagonal matrix. In Julia, diagonal matrices are represented using the `Diagonal` parametric type. This type acts as a wrapper for a one-dimensional array, which contains the diagonal elements. References to elements on the diagonal are redirected to loads or stores of this underlying array, whereas references off the diagonal return 0 without performing any actual memory access.

We leveraged Julia's multiple dispatch to provide a GEMM implementation that is specialised for diagonal matrices by means of two optimisations. First, we replace the layouts from Section 5.1 with a `Diagonal` layout. Similar

Table 1
Performance of our diagonal matrix GEMM and the equivalent CUBLAS implementation, for $N = 4096$; run times are for 100 iterations.

| | cuBLAS | Ours |
|---|---|---|
| **Run time (ms)** | $322.77 \pm 0.08$ | $50.68 \pm 0.05$ |
| **#Global mem. accesses** | 3410K per kernel | 490K per kernel |
| **#Tensor Core instr.** | 67 109K per kernel | 3113K per kernel |

to Julia's `Diagonal` wrapper, this layout simply returns 0 if the accessed element lies off the diagonal, and otherwise accesses an array. Second, we extended our template GEMM kernel with a customisable predicate that determines for each inner loop iteration if it should be executed or skipped. By default, this predicate is constant `true`, but we specialise it for diagonal matrices so that iterations that perform computations on elements off the diagonal are skipped.

Table 1 compares the performance of this specialised GEMM kernel with CUBLAS. As CUBLAS does not include GEMM kernels specialised for diagonal matrices, the CUBLAS version of our code had to materialise the diagonal matrix before calling the standard CUBLAS GEMM kernel. This is in line with the common practice as discussed in Section 4.1. The first optimisation results in a reduction of 89% in global memory traffic. The second optimisation reduces the number of TC operations by over 95%. Together, they lead to a GEMM that is more than 6 times faster than what we can obtain with CUBLAS's inflexible kernels.

Adding support for diagonal matrices required adding 23 lines of source code to our existing framework. We conclude that specialisation of kernels in our framework requires minimal effort, while at least in some cases still obtaining massive performance improvements.

### 5.3 Operator fusion

CUBLAS fuses linear scaling into its GEMM computation, i.e. its GEMM is of the form $D = \alpha \cdot AB + \beta \cdot C$. Other computations cannot be fused in CUBLAS's kernels and require a separate kernel launch. We consider two examples of GEMM computations that can exploit operation fusion: custom element-wise operations and adding a bias vector.

In custom element-wise operations, the linear scaling with $\alpha$ and $\beta$ is replaced by any arbitrary function. We implemented this in our GEMM framework using an `ElementwiseTransform` component. It has an arbitrary function as a parameter, which is applied to every element.

In bias computations, a one-dimensional bias vector is added to every row of the matrix product. To add support for bias in our GEMM framework, we created a custom epilogue that loads the bias vector from global memory, and adds it to the matrix product in shared memory, before writing the result back to global memory.

Table 2 compares the run times of six GEMMs with and without element-wise operations on input and/or output matrices, and with and without bias vectors. For the purpose of this experiment, we use additions with a constant and ReLU, a popular function in the domain of ML, as the element-wise operations, but similar results are obtained with other ones that are not supported in CUTLASS, and for which researchers would have to fall back on CUBLAS or our solution. For this reason, we only compare to CUBLAS.

Table 2
Run times of CUBLAS that lacks fusion capabilities and our GEMMs that exploit operation fusion, for $N = 4096$ and 100 iterations.

| | cuBLAS [ms] | Ours [ms] |
|---|---|---|
| **GEMM** | $260.82 \pm 1.51$ | $312.22 \pm 1.61$ |
| **GEMM + ReLU on D** | $286.20 \pm 1.10$ | $313.00 \pm 1.53$ |
| **GEMM + bias** | $287.72 \pm 1.80$ | $315.56 \pm 1.57$ |
| **GEMM + bias + ReLU on D** | $288.19 \pm 2.15$ | $315.54 \pm 1.60$ |
| **GEMM + bias + ReLU on C & D** | $313.88 \pm 1.22$ | $315.40 \pm 1.95$ |
| **GEMM + bias + ReLU on C & D + addition operation on A & B** | $340.38 \pm 1.21$ | $316.73 \pm 1.84$ |

With CUBLAS, combining the GEMM with element-wise operations or bias vectors results in additional kernel launches. While we have fused the operations as much as possible, i.e. the application of ReLU on $D$ and the addition of the bias vector are fused in one kernel, the GEMM and the other operations each still correspond to separate kernels which cannot be fused any further. By contrast, our framework seamlessly fuses all operations in the GEMM kernel instead. The effect is clearly visible in the results in the table. While the standard GEMM in CUBLAS is faster than with our framework, in line with the results in Section 5.1, our framework catches up as more fusable operations are added. Whereas the extra operations require almost no additional time in our framework (bias vectors need to be loaded, hence their small cost), each operation requiring a separate kernel costs approximately 10% in performance with CUBLAS. Ultimately, our GEMM becomes about 7% faster. This clearly illustrates the need for operator fusion, and how effective our approach is. The fact that our approach fuses the operations seamlessly also implies that any future optimisation of our default implementation of the mixed-precision GEMM, through swizzling or use of mma instead of WMMA, will automatically benefit GEMMs with fusable operations as well. Our GEMM will then outperform CUBLAS even more.

Note that, although our Julia GEMM implementation can easily be invoked from within other languages and ML frameworks, as discussed in Section 4.4, this advantage of fused element-wise operations can only be obtained if they are expressed in Julia, because that fusion depends on the Julia-specific features also discussed in Section 4.4.

### 5.4 Complex and dual numbers

Standard mixed-precision GEMMs differ from mixed-precision GEMMs of complex numbers in two ways. 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`. Second, complex GEMMs use different memory layouts in global and shared memory. In global memory, complex matrices are typically stored in an interleaved layout, where the real and imaginary parts are stored contiguously. This layout is incompatible with WMMA, so in shared memory, we use a split layout instead, where the real and imaginary parts are stored separately. In our GEMM framework, these two differences correspond to a new operator `WMMAComplexOp`, and two new layouts `InterleavedComplex` and `SplitComplex`, respectively.

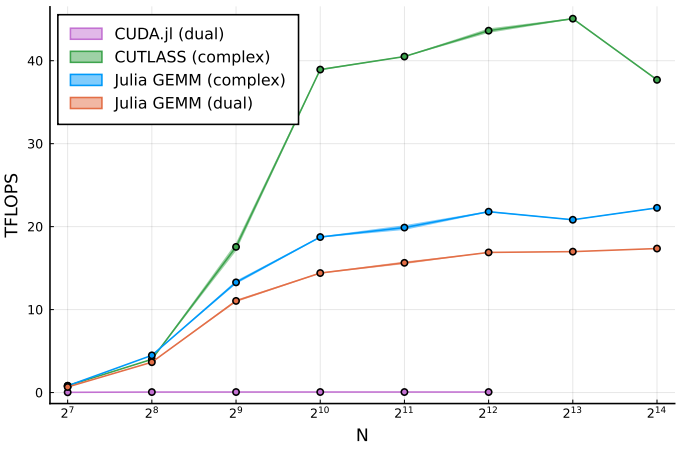


Figure 9. Performance of complex and dual GEMMs.

Dual numbers differ slightly from complex numbers. The imaginary unit $i$ is replaced by $\varepsilon$, and $\varepsilon^2 = 0$ whereas $i^2 = -1$. As such, we need an additional `WMMADualOp` operator component, but we can reuse the split and interleaved layouts we developed for complex matrices.

Figure 9 shows the performance of four GEMM kernels using complex or dual numbers. As CUBLAS supports neither complex numbers using TCs nor dual numbers, we did not include it in the comparison. CUBLASLT does support complex numbers, but uses CUTLASS's kernels. CUTLASS only supports complex numbers. So we compare the performance of our two kernels to CUTLASS for complex numbers and to the generic CUDA.JL kernel that is invoked when our API is not used to compute a mixed-precision GEMM on dual numbers. Our complex number implementation achieves a peak performance of 59% of CUTLASS's peak performance. We conclude that, despite the fact that our kernels are written completely in Julia, and do not contain optimisations specific to complex GEMM, we are still able to reach reasonably good performance.

The difference in performance between our complex GEMM and CUTLASS's can partly be attributed to CUTLASS's use of a split layout in both global and shared memory, eliminating the overhead of changing layouts in the global-to-shared memory stream. Our kernels use an interleaved layout in global memory, as this is the format that Julia uses. This comes with some overhead, but eliminates the need for extra interleaved-to-split and split-to-interleaved kernel launches that would be necessary for CUTLASS.

While CUDA.JL contains wrappers for CUBLAS's GEMM kernels, it supports only a limited number of data types. Calling these wrappers with unsupported types, such as dual numbers, falls back to a generic implementation that is many orders of magnitude slower, as is clear from the CUDA.JL line in Figure 9.

Adding support for complex and dual numbers to our framework required 169 lines of source code, of which 85 are common to both data types. We conclude that extending our framework with custom data types requires minimal effort, especially compared to the alternative of writing a performant GEMM kernel for these data types from scratch.

```
abstract type LayoutA{T} <: Layout.AlignedColMajor{T} end

@inline function Layout.load(::Type{LayoutA{T}},
    workspace, tile::Tile{size}) where {T, size}
  NUMEL = 16 ÷ sizeof(T)

  M = tile.base.M + tile.offset.M
  K = tile.base.K + tile.offset.K
  d = K
  a = M ÷ Base.size(workspace, 1)
  b = M % Base.size(workspace, 1)

  offset = 1 + b + d * Base.size(workspace, 1) +
    a * Base.size(workspace, 1) * Base.size(workspace, 2)
  Layout.vloada(Layout.Vec{NUMEL, T}, pointer(workspace),
    offset)
end
```

Listing 13: Layout for tensor $A$.

Table 3
Performance of TCT and the equivalent CUTENSOR implementations, for $N_a = 64, N_b = 32, N_c = 2048, N_d = 2048$; run times are in µs.

| cuTENSOR TTGT | cuTENSOR GETT | Ours |
|---|---|---|
| $351.14 \pm 2.46$ | $1238.81 \pm 9.39$ | $433.82 \pm 5.26$ |

### 5.5 Tensor Contraction

Implementing and evaluating fully-optimised general TCTs is out-of-scope of this paper. However, we do want to demonstrate that our GEMM functionality does provide the necessary support for common TCTs. We hence evaluated the functionality for one TCT, namely the first benchmark of the Tensor Contraction Code Generator benchmarks [16]. In Einstein notation, this TCT is written as $D_{abc} = A_{bda} \cdot B_{dc}$.

Using our GEMM APIs, we created custom global memory layouts for matrices $A$ to $D$. The layout for $C$ always returns zero without performing loads to global memory. The layouts for $A$, $B$, and $D$ provide the fused transpositions of GETT [38]. As the contraction index $d$'s position is almost perfect for an NN-GEMM, the $A$, $B$, and $D$ layouts started from the `AlignedColMajor{T}` layouts. Custom mappings from elements in global memory to shared memory were implemented such that GEMM in shared memory is equivalent with TCT in global memory. We did not extensively search for optimal memory layouts, but simply tried to use as many stride-1 accesses in global memory as possible, to facilitate vectorisation and global memory coalescing. As an example, the $A$ layout code is shown in Listing 13. With this layout, $A_{bda}$ in global memory becomes $A_{bad}$ in shared memory, which is then interpreted as the 2D matrix $A_{mk}$, by mapping of dimensions $(ba) \to m$ and $d \to k$. $B$ is $B_{dc}$ in global memory and stays the same in shared memory, with mapping $d \to k$ and $c \to n$. The GEMM in shared memory then computes $A_{mk}B_{kn} = A_{bad}B_{dc} = D_{bac}$, and the custom $D$ layout ensures shuffling is performed to $D_{abc}$ while copying data from shared to global memory.

We compare the performance of our implementation to that in CUTENSOR, a hand-optimised tensor library from NVIDIA. We observed that CUTENSOR by default does not use the GETT approach for the evaluated TCT, as it executes two kernels, similar to the TTGT approach [19], [20]. We can however force it to use the GETT approach as well.

Table 3 lists the performance results. Our kernels achieve 81% of the performance of CUTENSOR TTGT, the best performing configuration. Interestingly, CUTENSOR's GETT

kernel performs $3.5\times$ worse than TTGT, and we outperform it by a factor of $2.9$.

The $B$ and $D$ layouts look similar to Listing 13; the $C$ layout always returns $0$ and elides the indexing logic and memory operations, but otherwise has the same structure. In total, these four layouts add up to 58 lines of code.

We conclude that our framework can support (at least some variants of) TCTs, and achieves performance in the same ballpark as that of state-of-the-art kernels in cuTENSOR, while requiring minimal programming effort.

## 6 Related work

In Section 4.1, we already discussed the need for flexible and performant GEMMs on GPUs in various domains. As discussed in Section 2.3, several libraries exist to deliver the performance. We compared the performance of our GEMM API with some of them in Section 5.

NVIDIA's CUTLASS template library contains components to instantiate performant GEMMs on GPUs [41]. As it is written in C++, it does not solve the two-language problem and its impact on programmer productivity. It did serve, however, as an inspiration for the abstractions in our GEMM API. For example, CUTLASS also has the notion of layouts that map logical indices to physical memory offsets, and epilogues for custom post-processing. Some components have a slightly different purpose, however. In CUTLASS, transforms only apply to the global-to-shared memory stream of the $A$ and $B$ matrices. Element-wise transforms on the resultant matrix are handled in a custom epilogue. Adding support for custom transformations requires significant effort, as CUTLASS epilogues are typically 150–200 lines long. In our GEMM API, transforms are applied after every load and before every store, ensuring that element-wise operations to the input and resultant matrices can be applied more easily and consistently.

The CUTLASS codebase is extensive and contains many components, making it more difficult for end users to get started extending it. Each GEMM typically involves quite a few layered template instantiations, impacting code comprehension. Most templates are heavily specialised for different memory layouts, computations, etc., reducing orthogonality and code reuse. For example, CUTLASS contains different epilogues for GEMMs exploiting TCs and GEMMs using FPUs. Our GEMM API abstractions offer better separation of concerns and hence more reusability.

Like our GEMM API, CUTLASS contains both a BLAS-like interface, and an interface exposing all its flexibility. Launching a CUTLASS kernel using the latter requires more boilerplate compared to our approach in Listing 9, e.g. because CUTLASS users need to explicitly allocate a workspace that is used internally in CUTLASS.

NVIDIA's cuBLASLt is a lightweight BLAS library dedicated to GEMM, with a more flexible API than cuBLAS. This flexibility comes in the form of support for more matrix layouts, input and compute data types, and algorithmic implementations. It is available on CUDA 10.1 or later.

Like our GEMM API, launching a kernel in cuBLASLt is also a two-step process. First, a "plan" must be created that determines the options for the GEMM computation. Second, this plan is used to launch one or more GEMM kernels.

cuBLASLt features concepts similar to CUTLASS, such as epilogues that post-process the resultant matrix and layouts describing how matrices are stored in memory. Each of these corresponds to an enumeration that lists the legal values, hence limiting flexibility. For example, it only includes bias and ReLU as possible epilogues, and offers no support for custom layouts such as diagonal matrices, or custom data types such as dual numbers. While cuBLASLt's interface is an improvement over that of cuBLAS, its closed-source nature still results in limited extensibility.

BLIS is a framework that facilitates the instantiation of an entire BLAS library for new architectures, and hence has a larger scope than just GEMM [42]. It achieves this by rephrasing all BLAS operations in terms of a limited set of kernels. Their focus is on CPUs rather than GPUs, however. Similar to our GEMM kernel, BLIS contains a set components that can be reused for new BLAS-like operations. BLIS's GEMM kernels offer support for more memory layouts and data types than traditional BLAS libraries. Nevertheless, BLIS's flexibility is mainly aimed at developers of the BLIS library, instead of its users. For example, extending BLIS with support for complex numbers required significant effort, that warranted a separate paper describing its implementation [43].

Section 3 presented a tiling API that allows programmers to coordinate memory transfers to improve data locality. Automated tools based on polyhedral optimisation exist that can automatically generate tiled code from nested loops [44], [45], [46]. Basic approaches only reorder memory accesses, but more advanced ones can also exploit parallelism. For example, Polly can exploit inter-tile parallelism using the OpenMP interface [47]. The framework by Baskaran et al. is even capable of automatically adding padding for shared memory accesses to reduce bank conflicts [48].

Recent work by Bondhugula uses the polyhedral utilities in MLIR to generate performant GEMM kernels [49]. His approach achieves a performance that is within 9% of state-of-the-art CPU GEMMs in BLIS and MKL. It still offers limited flexibility, however. Incorporating domain-specific optimisations, such as diagonal matrices, is significantly harder than in our approach, as it requires adaptations to the MLIR code base and/or TableGen rules.

The Diesel DSL (Domain-Specific Language) uses polyhedral techniques to compile high-level expressions to performant GPU kernels [50]. Bhaskaracharya et al. extend Diesel with support for TCs and fused kernels that combine matrix multiplications with ReLU and bias [51]. Their work focuses only on Volta TCs, however, and does not address other forms of GEMM flexibility such as support for more complex data types. Additionally, only a limited number of element-wise operations are supported.

Halide, a DSL for image processing, was also extended with TC support by Sioutas et al. [52]. Their approach features a fixed kernel skeleton, similar to our template GEMM kernel. Their kernel also makes use of WMMA, and achieves performance results similar to ours. It still has limited flexibility, however. For example, it offers no support for complex data types, and only handles one combination of memory layouts for the $A$ and $B$ matrices, which necessitates explicit transposition kernels.

Note that BLIS, the polyhedral techniques (in so far as they support GPUs), DIESEL and HALIDE all involve statically compiled, statically typed PLs. For the general-purpose PLs, this by itself foregoes the productivity advantages of rapid-prototyping PLs such as Julia or Python. For the DSLs, it implies that code reuse across domains is limited. Those solutions also by construction do not consider flexibility beyond the data types, layouts, and operations typical for their domains. Our solution does not suffer from these drawbacks, yet obtains performance in the same ballpark.

## 7 Availability

Our contributions are open source and available in the relevant GitHub repositories. Support for Tensor Cores using WMMA was merged into CUDA.JL, and is available in the latest stable version. The required adaptations to the Julia compiler were sent to the developers, and have been merged upstream. Our tiling and flexible GEMM APIs are bundled in one Julia package GEMMKERNELS.JL. This package is available at https://github.com/thomasfaingnaert/GemmKernels.jl, and can easily be installed using Julia's built-in package manager. It contains all instantiations of our API abstractions of all experiments and listings in this paper, ready for out-of-the-box re-use.

## 8 Conclusions and Future Work

In this paper, we first presented tiling abstractions with which programmers can use tiling techniques, which are necessary to achieve high performance for many computations, at a high level of abstraction.

We then discussed a flexible GEMM API where the kernel consists of a set of orthogonal components. Each of these components corresponds to a set of Julia functions that can be specialised for different GEMM variants. We demonstrated the flexibility of this approach by instantiating the necessary components for 5 variants of GEMM computations: a normal mixed-precision GEMM, computations using diagonal matrices, computations exploiting operator fusion, GEMMs on complex and dual numbers, and a tensor contraction. We argued how specific features of the Julia compiler, such as multiple dispatch, type inference, and just-ahead-of-time compilation, allow for this flexibility without run-time overhead.

An experimental evaluation showed that the performance of our GEMM kernels written entirely in Julia is in the same ballpark as, and in some cases even exceeds, the state of the art in the manually tuned CUBLAS, CUTLASS, and CUTENSOR.

We presented two interfaces to use our flexible GEMM API: a fully-featured interface and a BLAS-like interface. The former exposes the full flexibility of our framework, the latter extends BLAS's GEMM with support for more data types such as dual numbers. We demonstrated our APIs for CUDA-enabled GPUs, but our abstractions are vendor-agnostic and can be ported to other GPU architectures.

In the future, we plan to port our framework to other GPUs such as those of AMD and Intel, and to add support for the `mma` family of instructions as well as data swizzling, as used in CUTLASS and CUBLAS, to improve performance. At the moment, the matrix inputs to our kernels must be zero-padded such that their size is a multiple of the GEMM tile sizes. The engineering required to support arbitrary matrix dimensions, e.g. through the use of predicated memory accesses as done by CUTLASS, is also future work.

## Acknowledgements

This work was funded by the Research Foundation Flanders (Fonds voor Wetenschappelijk Onderzoek), grant number 3G051318.

## References

[1] BLAS contributors. (2017) BLAS (Basic Linear Algebra Subprograms).
[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 *IEEE/ACM 10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, 2019, pp. 17–24.
[3] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, "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*, ser. SC '18. IEEE Press, 2018.
[4] T. Ichimura, K. Fujita, T. Yamaguchi, A. Naruse, J. C. Wells, T. C. Schulthess, T. P. Straatsma, C. J. Zimmer, M. Martinasso, K. Nakajima, M. Hori, and L. Maddegedara, "A fast scalable implicit solver for nonlinear time-evolution earthquake city problem on low-ordered unstructured finite elements with artificial intelligence and transprecision computing," in *Proc. Int'l Conference for High Performance Computing, Networking, Storage, and Analysis*, 2018.
[5] A. Haidar, H. Bayraktar, S. Tomov, and J. Dongarra, "Harnessing Tensor Cores FP16 arithmetic to accelerate linear solvers and HPC scientific applications," 2018, nVIDIA GPU Technology Conference.
[6] V. Mehta, "Getting started with Tensor Cores in HPC," 2019, nVIDIA GPU Technology Conference.
[7] D. Yan, W. Wang, and X. Chu, "Demystifying Tensor Cores to optimize half-precision matrix multiply," in *Proc. 34th IEEE International Parallel and Distributed Processing Symposium*, 2020.
[8] NVIDIA. (2019, 6) Deep learning performance guide.
[9] ——. (2020) NVIDIA V100.
[10] P. Barham and M. Isard, "Machine learning systems are stuck in a rut," in *Proc. Workshop on Hot Topics in Operating Systems*, 2019, pp. 177—-183.
[11] N. Rink, A. Susungi, J. Castrillón, J. Stiller, and C. Tadonki, "CFDlang: High-level code generation for high-order methods in fluid dynamics," in *Real World Domain Specific Languages Workshop 2018*, 02 2018, pp. 1–10.
[12] R. Poya, A. J. Gil, and R. Ortigosa, "A high performance data parallel tensor contraction framework: Application to coupled electro-mechanics," *Computer Physics Communications*, vol. 216, pp. 35–52, 2017.
[13] A. Auer, G. Baumgartner, D. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, G. Xiaoyang, R. Harrison, S. Krishnamoorthy, S. Krishnan, C.-C. Lam, Q. Lu, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov, "Automatic code generation for many-body electronic structure methods: The tensor contraction engine," *Molecular Physics*, vol. 104, 01 2006.
[14] P. Springer and P. Bientinesi, "The landscape of high-performance tensor contractions," in *Workshop on Batched, Reproducible, and Reduced Precision BLAS*, 2017.
[15] T. Nelson, A. Rivera, P. Balaprakash, M. Hall, P. D. Hovland, E. Jessup, and B. Norris, "Generating efficient tensor contractions for GPUs," in *2015 44th International Conference on Parallel Processing*, 2015, pp. 969–978.
[16] P. Springer and P. Bientinesi, "Design of a high-performance GEMM-like tensor–tensor multiplication," *ACM Transactions on Mathematical Software (TOMS)*, vol. 44, no. 3, pp. 1–29, 2018.
[17] E. D. Napoli, D. Fabregat-Traver, G. Quintana-Ortí, and P. Bientinesi, "Towards an efficient use of the BLAS library for multilinear tensor contractions," *Applied Mathematics and Computation*, vol. 235, pp. 454–468, 2014.

[18] J. Li, C. Battaglino, I. Perros, J. Sun, and R. Vuduc, "An input-adaptive and in-place approach to dense tensor-times-matrix multiply," in *Proc. Int'l Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.
[19] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel, "Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions," in *27th Int'l Symposium on Parallel and Distributed Processing*, 2013, pp. 813–824.
[20] B. W. Bader and T. G. Kolda, "Algorithm 862: MATLAB tensor classes for fast algorithm prototyping," *ACM Transactions on Mathematical Software*, vol. 32, no. 4, pp. 635–653, December 2006.
[21] J. Kim, A. Sukumaran-Rajam, V. Thumma, S. Krishnamoorthy, A. Panyala, L.-N. Pouchet, A. Rountev, and P. Sadayappan, "A code generator for high-performance tensor contractions on GPUs," in *Proc. IEEE/ACM Int'l Symposium on Code Generation and Optimization*, 2019, p. 85–95.
[22] D. A. Matthews, "High-performance tensor contraction without transposition," *SIAM Journal on Scientific Computing*, vol. 40, no. 1, pp. C1–C24, 2018.
[23] C. Psarras, L. Karlsson, J. Li, and P. Bientinesi, "The landscape of software for tensor computations," 2021.
[24] NVIDIA. (2020) CUDA C++ programming guide.
[25] Khronos Group. (2020) OpenCL: An open standard for parallel programming of heterogeneous systems.
[26] JuliaLang.org. (2020) The Julia language.
[27] ——. (2020) Julia micro-benchmarks.
[28] T. Besard, C. Foket, and B. De Sutter, "Effective extensible programming: Unleashing Julia on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 4, pp. 827–841, 2019.
[29] T. Besard, V. Churavy, A. Edelman, and B. De Sutter, "Rapid software prototyping for heterogeneous and distributed platforms," *Advances in Engineering Software*, vol. 132, pp. 29 – 46, 2019.
[30] JuliaLang.org. (2020) The Julia language official documentation.
[31] LLVM contributors. (2020) The LLVM compiler infrastructure project.
[32] V. Churavy. (2020) GPUifyLoops.jl: Support for writing loop-based code that executes both on CPU and GPU.
[33] G. Hinton, S. Sabour, and N. Frosst, "Matrix capsules with EM routing," in *International Conference on Learning Representations*, 2018.
[34] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *Proc. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 265–283.
[35] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, 2019, pp. 8024–8035.
[36] E. Aprà, M. Klemm, and K. Kowalski, "Efficient implementation of many-body quantum chemical methods on the Intel® Xeon Phi coprocessor," in *Proc. Int'l Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 674–684.
[37] W. Ma, S. Krishnamoorthy, O. Villa, and K. Kowalski, "GPU-Based Implementations of the Noniterative Regularized-CCSD(T) Corrections: Applications to Strongly Correlated Systems," *Journal of Chemical Theory and Computation*, vol. 7, no. 5, pp. 1316–1327, 2011.
[38] P. Springer and P. Bientinesi, "Design of a high-performance GEMM-like tensor-tensor multiplication," 2016.
[39] J. Revels, M. Lubin, and T. Papamarkou, "Forward-mode automatic differentiation in Julia," *arXiv:1607.07892 [cs.MS]*, 2016.
[40] V. Churavy. (2020) KernelAbstractions.jl: Heterogeneous programming in Julia. [Online]. Available: https://github.com/JuliaGPU/KernelAbstractions.jl
[41] NVIDIA. (2020) CUTLASS: CUDA templates for linear algebra subroutines.
[42] F. G. Van Zee and R. A. van de Geijn, "BLIS: A framework for rapidly instantiating BLAS functionality," *ACM Trans. Math. Softw.*, vol. 41, no. 3, Jun. 2015.
[43] F. G. Van Zee, "Implementing high-performance complex matrix multiplication via the 1M method," *SIAM Journal on Scientific Computing*, vol. 42, no. 5, pp. C221–C244, 2020.
[44] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proc. 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 101–113.
[45] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gomez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for CUDA," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 1–23, 2013.
[46] P. Di, D. Ye, Y. Su, Y. Sui, and J. Xue, "Automatic parallelization of tiled loop nests with enhanced fine-grained parallelism on GPUs," in *Proc. 41st Int'l Conference on Parallel Processing*, 2012, pp. 350–359.
[47] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly—performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, p. 1250010, 2012.
[48] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "A compiler framework for optimization of affine loop nests for GPGPUs," in *Proc. 22nd Annual Int'l Conference on Supercomputing*, 2008, pp. 225–234.
[49] U. Bondhugula, "High performance code generation in MLIR: An early case study with GEMM," *preprint arXiv:2003.00532*, 2020.
[50] V. Elango, N. Rubin, M. Ravishankar, H. Sandanagobalane, and V. Grover, "Diesel: DSL for linear algebra and neural net computations on GPUs," in *Proc. 2nd ACM SIGPLAN Int'l Workshop on Machine Learning and Programming Languages*, 2018, pp. 42–51.
[51] S. G. Bhaskaracharya, J. Demouth, and V. Grover, "Automatic kernel generation for Volta Tensor Cores," *arXiv preprint arXiv:2006.12645*, 2020.
[52] S. Sioutas, S. Stuijk, T. Basten, L. Somers, and H. Corporaal, "Programming tensor cores from an image processing DSL," in *Proc. 23th Int'l Workshop on Software and Compilers for Embedded Systems*, 2020, pp. 36–41.

**Thomas Faingnaert** is a PhD student at Ghent University in the Computer Systems Lab. He obtained his MSc degree in Computer Science Engineering from Ghent University's Faculty of Engineering and Architecture in 2020. His research focuses on software protection, and high-level abstractions for GPU programming in Julia.

**Tim Besard** is a software engineer at Julia Computing. He obtained his MSc in Computer Engineering from University College Ghent in 2011, and his PhD in Computer Science Engineering from Ghent University in 2019. He is currently the lead maintainer of several GPU back-ends for the Julia programming language.

**Bjorn De Sutter** is associate professor at Ghent University in the Computer Systems Lab. He obtained his MSc and PhD degrees in Computer Science from Ghent University's Faculty of Engineering in 1997 and 2002. His research focuses on the use of compiler techniques to aid programmers with non-functional aspects of their software, such as performance, code size, reliability, and security.