Universiteit Leiden
The Netherlands

Opleiding Informatica

Developing an auto-tunable GEMM kernel that utilizes Tensor Cores

Tobias Hofstra

Supervisors:
Dr. B.J.C. van Werkhoven
Dr. K.F.D. Rietveld

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

24/8/2024

Abstract

With the rising popularity of deep learning, GEMM, the core computation in training AI models, is becoming more and more important to optimize. To this end, NVIDIA has developed tensor cores for usage in their GPUs, which significantly speed up GEMM computations. This work aims to combine tensor cores with application level auto-tuning to develop a tunable kernel that makes use of tensor cores, something which until now has not been realized. We showcase that for a GEMM kernel that makes use of tensor cores, auto-tuning increases performance by finding optimal configurations that would be hard to find by hand. However, simply applying auto-tuning is not enough to beat state-of-the-art kernels for a problem as explored as GEMM.

Contents

1 Introduction

Graphical Processing Units (GPUs) have become essential in graphics processing, scientific computing, and recently, deep learning applications. This is in large part due to the parallel structure of GPUs, which allows them to exploit the parallel nature of the aforementioned problems, while more traditional computing platforms like Central Processing Units (CPUs) struggle in this area. The two leaders in the area of GPU development are NVIDIA and AMD, whose GPUs can be programmed through the parallel programming platforms of CUDA and HIP respectively. In this thesis, NVIDIA GPUs and CUDA will be the primary focus.
CUDA is used to describe the programming platform of NVIDIA GPUs as a whole, but for the purposes of this thesis it can be thought of as a programming language that extends C++. An application in CUDA consists of *host* code, which is performed by the CPU, and *device* code, which is performed by the GPU. Another name for the device code of a GPU application is a *kernel*.
Kernels have proven difficult to optimize by hand due to the large set of optimization parameters and the need to understand the underlying hardware [RRS+08, vWPS20]. This difficulty has given rise to the automatic tuning of kernels, also called auto-tuning [LDT09], which easily allows performance optimizations to be made on kernels [GL11, TNLD10]. Another benefit of auto-tuning is portability: performance improvements can be found for any hardware platform.
Deep learning applications have led to NVIDIA implementing so-called tensor cores in their GPUs. These tensor cores significantly speedup deep learning applications, and were first introduced with the Volta GPU microarchitecture in 2017 [MCL+18]. Tensor cores can work with a variety of data types, but due to relatively high classification accuracy in deep learning applications with low precision integers [GAGN15], 16-bit numbers (also called half precision) see the most use in tensor cores. Usually, 16-bit numbers are used inside the tensor cores, and 32-bit numbers outside of the tensor cores, to achieve so-called *mixed precision*, which will be the precision considered in this thesis. General Matrix Multiply (GEMM) is a core computation used in many GPU applications [YSL+23, YWC20] and can likewise enjoy significant performance improvements from tensor cores [NVI17].
Due to the importance of GEMM and the impact of tensor cores, there is a clear need for optimized GEMM implementations that use tensor cores. NVIDIA has provided such GEMM kernels in the cuBLAS [NVIa] and CUTLASS [NVIc] libraries, but these are not tuned to specific hardware platforms, and thus better kernel configurations may exist than those used in these kernels. Likewise auto-tunable GEMM kernels already exist, but none of them use tensor cores and therefore fall behind in performance compared to GEMM kernels that do use tensor cores.

1.1 Thesis overview

This thesis aims to amend the aforementioned issue by developing an auto-tunable GEMM kernel that utilizes tensor cores. The resulting kernel will then be auto-tuned and benchmarked against existing GEMM kernels to measure its performance on the DAS-6 compute cluster [BEdL+16]. Section 2 contains the background information and the related work; Section 3 describes how the kernel is designed; Section 4 goes into detail about the auto-tuning process and discusses the results from auto-tuning; Section 5 contains the results and discussions of the benchmarking of the different kernels; Section 6 describes conclusions from this work and further research. Notably, Section 6 reveals a very unique optimal configuration for the A100 GPU. Moreover, the tunable

tensor core kernel outperforms tunable kernels that do not make use of tensor cores, but falls behind in performance compared to state-of-the-art kernels that make use of tensor cores. This bachelor thesis was written for the Computer Science Bachelor program at Leiden University and supervised by Ben van Werkhoven and Kristian Rietveld at the Leiden Institute of Advanced Computer Science (LIACS).

2 Background and related work

This section contains all the background information for this thesis, which includes:

1. The definition of GEMM and some basic GEMM optimizations.
2. Tensor cores, and the three main ways to program them: CUTLASS, CuTe, and the WMMA API.
3. An overview of auto-tuning and existing auto-tuners.
4. Related work.

2.1 GEMM

GEMM is the following operation:

$$D = \alpha \cdot A \cdot B + \beta \cdot C \tag{1}$$

Where α and β are constant scalars, A is a matrix of dimensions $M \times K$, B is a matrix of dimensions $K \times N$, and C and D are matrices of dimensions $M \times N$. The total GEMM problem dimensions are then defined as $M \times N \times K$. In some computation use cases the matrix D equals C (accumulation). This definition also allows for regular $A \cdot B$ matrix multiplication by setting α to 1 and β to 0 or C to an all-zero matrix.
Because the individual elements of the output matrix D are independent from each other, GEMM is a prime candidate for usage in parallel execution paradigms, realized by GPUs. For this thesis, CUDA will be considered. NVIDIA has provided a general overview of a hierarchically blocked structure used in CUTLASS [KMDT, NVId], which forms a solid foundation for GEMM implementations. The high-level hierarchy of a whole GEMM operation in CUTLASS can be seen in Figure 1. This figure details the different levels of the algorithm and which part of the GPU is used at every level. A naive triple loop matrix multiplication implementation can be found in code Listing 1. In this naive implementation, every element of the output matrix is computed one-by-one. The problem here is that the elements of matrices A and B get loaded multiple times. Due to the limited sizes of on-chip caches, this often means the matrix elements will not be present in the cache when needed again. The hierarchically blocked structure solves this problem by moving the inner-most loop to the outer-most loop. This can be seen in Listing 2.

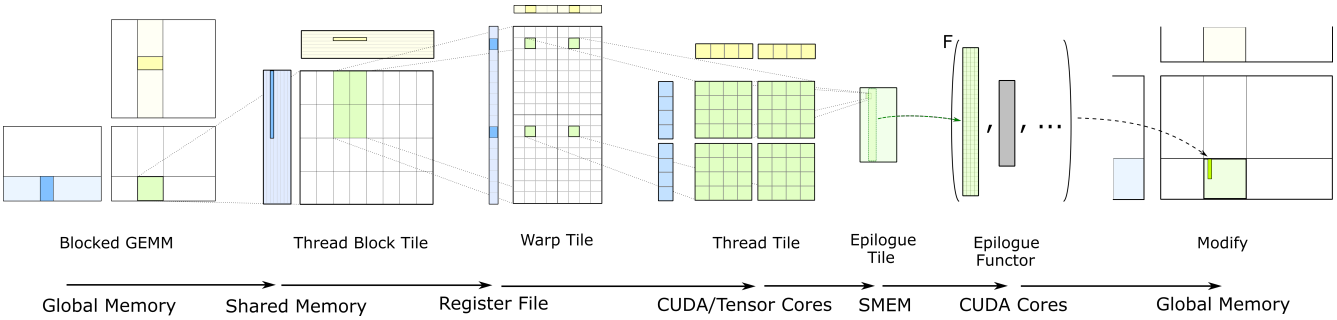


Figure 1: CUTLASS GEMM hierarchy. Image from [NVIe]

Listing 1: Naive matrix multiplication

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

Listing 2: Better matrix multiplication

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

In this structure, the outer-most loop selects the kth column of the A matrix and the kth row of the B matrix. The product of these elements are then computed and *accumulated* in the C matrix. Once an iteration in the outer-most loop is finished, the selected column of the A matrix and row of the B matrix are not used again.
Since this optimization works best for small matrices, the C matrix is partitioned into tiles, which are then distributed across thread blocks (thread block tiles). Within these tiles, the workload can further be divided across groups of 32 threads, also called warps (warp tiles).

2.2 Tensor cores

In recent years the main application of GEMM has been in training of AI models. Fully connected layers in feed-forward neural networks that use backpropagation can be directly computed with matrix multiplication, and convolutional layers in convolutional neural networks can also be reduced to a GEMM problem [KC06]. Because of this any optimizations targeting neural network training performance will likely be centered around speeding up GEMM computation. To this end, many companies started developing hardware platforms specifically for neural network workloads, such as neural processing units (NPUs), field programmable gate arrays (FPGAs) and application-specific integrated circuit (ASICs) like Google's Tensor Processing Units [JYP+17], AMD's matrix cores [AMD20], and NVIDIA's tensor cores [NVI17]. In this thesis, tensor cores will be considered. Tensor cores can be used through the Warp Matrix Functions instructions in the PTX ISA [NVI24], but NVIDIA has provided three programmer-friendly ways to work with tensor cores: the CUTLASS library [NVIc], the CuTe library [NVIb], and the WMMA API [AY].

2.2.1 CUTLASS

CUTLASS stands for CUDA Templates for Linear Algebra Subroutines and is a state-of-the-art library developed by NVIDIA for implementing high performance GEMM kernels. As the name suggests, CUTLASS makes extensive use of C++ templates, allowing for some degree of freedom when using a CUTLASS kernel, making it tunable to a degree. As mentioned earlier in Section 2.1, CUTLASS implements the hierarchically blocked structure. For example, thread block tile dimensions and warp tile dimensions can both be chosen at will. Furthermore, the matrices A, B, and C and D can all be specified with different data types and layouts (whether the data is stored row or column major). Critically relevant for this thesis, the user can also specify if tensor cores should be used or not when using a CUTLASS kernel.

Matrix A	Matrix B	Accumulator	Matrix Size
__half	__half	float	16x16x16
__half	__half	float	32x8x16
__half	__half	float	8x32x16
precision::tf32	precision::tf32	float	16x16x8
double	double	double	8x8x4
precision::u4	precision::u4	int	8x8x32
precision::b1	precision::b1	int	8x8x128

Table 1: Some of the supported data types and matrix sizes in WMMA.

CUTLASS can be used to construct GEMM kernels on five different levels, ordered from high to low:

1. Device: on this level a whole CUTLASS kernel is called from inside a CUDA program, similarly to calling kernels from cuBLAS or other libraries.
2. Kernel: on this level the main computation loops and data structuring/movement is managed.
3. Collective: here the main loops which perform the intensive multiplications are defined.
4. Tiled (MMA and Copy)/Atom: these two levels are the lowest levels and execute the high-performance data copies and matrix multiply and accumulate (MMA) instructions. These two levels are not defined in the main CUTLASS library, but in CuTe.

When using a CUTLASS kernel from the *device* level, the user does not have a lot of control in how the computation will be performed with regards to tunable parameters. For example, usage of hardware features such as shared memory is largely up to the CUTLASS implementation itself, even though these features can impact performance and, in particular, energy efficiency [SVvWB22]. Furthermore, without an easy way to obtain optimal parameters, an auto-tunable GEMM kernel can theoretically gain some advantages over CUTLASS.

2.2.2 CuTe

CuTe is a very new library within CUTLASS itself, introduced early in 2023 with CUTLASS 3.0.0. CuTe provides templates for defining layouts of threads and data, which can then be used in a GEMM using tensor cores. This design of abstractions allows the programmer to not worry too much about the exact partitioning of threads and data, making it easier to implement specific algorithms. Due to a lack of extensive documentation as of time of writing, CuTe will not be used in this thesis, although it could prove useful for developing kernels that use tensor cores in the future.

2.2.3 WMMA API

The Warp Matrix Multiply Accumulate (WMMA) API is built into CUDA through the `nvcuda::wmma` namespace. Using this API, the programmer can perform matrix multiply and accumulation (MMA) with tensor cores on fixed (sub)matrix sizes by letting every thread in a warp execute the WMMA instructions.

First, the (sub)matrices have to be declared as so-called fragments with `wmma::fragment<>`. These `fragments` can be fragments of the A (`wmma::matrix_a`) and B (`wmma::matrix_b`) matrices or accumulators (`wmma::accumulator`) for holding parts of the C matrix and the intermediate result of the matrix-multiply computation. These fragments can be declared with different shapes, data types, and for A and B fragments whether the matrix is row or column major. Table 1 contains some of the combinations of data types and matrix sizes WMMA supports. For example, the declaration of a fragment of a column major A matrix with $8 \times 32 \times 16$ shaped operations and half-precision elements could look like:

```
wmma::fragment<wmma::matrix_a, 8, 32, 16, half, wmma::col_major>
```

A WMMA operation of shape $8 \times 32 \times 16$ entails multiplying a 8×16 submatrix of A by a 16×32 submatrix of B, and accumulating the result in an 8×32 submatrix of C.
After the fragments have been declared, they need to be initialized with `wmma::fill_fragment` or loaded with data with `wmma::load_matrix_sync`. The data in these fragments is then stored in registers. Here it is critical that all threads within a warp execute these functions with the same arguments, otherwise this is undefined behavior. After the fragments contain the necessary data, they can be multiplied with `wmma::mma_sync`.

2.3 Auto-tuning

Auto-tuning can be divided into two methodologies: code generation auto-tuning with compilers, and application-level auto-tuning. Research into auto-tuning was first introduced to target CPU applications, with GPU applications being considered only more recently. As such, examples of auto-tuners given in this section may be both CPU and GPU auto-tuners.
The first methodology of auto-tuning is based on code generation. The user can specify some high-level input, and the compiler will generate multiple code variants and select the best one. Common optimizations and transformations made by the compiler include loop tiling/blocking, loop unrolling, loop permutation, prefetching, software pipelining [BDG+18], data copy, iteration space splitting [CCH07], and polyhedral transformations [HNS09, BHRS08]. Some examples of compiler-directed auto-tuners are ATLAS [WD98], FFTW [FJ98], SPIRAL [PMJ+05], CHiLL [CCH07], Orio [HNS09] and TVM [CMJ+18], which focuses on GEMM.
The second methodology of auto-tuning works on application level. Here, algorithms include *tunable parameters*, which can take on different values to represent different code variants. It is then up to the programmer to provide a list of values all the parameters can take on, and ensure the implementation of the algorithm is generic enough so that it can function with as many different configurations as possible. Examples include OpenTuner [AKV+14], CLTune [NC15], Kernel Tuning Toolkit (KTT) [FPB17], Auto-Tuning Framework (ATF) [RSSG21], and, the auto-tuner used in this thesis, Kernel Tuner [vW19].

2.3.1 KTdashboard

Using the tool KTdashboard, which is part of the Kernel Tuner ecosystem, the results of the tuning process can be analyzed in greater detail. KTdashboard displays a scatter plot of all the configurations and allows the user to select the x and y axes and colour the configurations by the tunable parameters. The colours range from yellow (high value for specified parameter) to purple (low value for specified parameter).

2.4 Related work

As illustrated in Section 2.2, GEMM is an incredibly important operation. As such, many of the aforementioned auto-tuners in Section 2.3 include or work with GEMM kernels. However, none of the GEMM kernels in the application-level auto-tuners make use of tensor cores, which deliver significant performance improvements.

Yu et al. [YSL+23] introduce *CUTLASS-tailor*, a machine learning framework for predicting CUTLASS kernel parameters to achieve the highest performance. Using supervised learning the researchers demonstrate significant speedup compared to cuBLAS by choosing the right CUTLASS parameters. Although auto-tuning was not used to achieve these results, the work nevertheless demonstrates the importance of selecting the right tunable parameters.

3 Methodology

This section contains the full description of the tunable tensor core GEMM kernel, following the hierarchally blocked structure described in Section 2.1. A brief overview of every tunable parameter used in this new kernel can be found in Table 2. One important thing to note is that the kernel itself assumes that the A matrix is not transposed and that the B matrix is transposed, while, for the sake of simplicity, the explanation below assumes neither matrix is transposed.

Name	Description	Permitted values
WMMA_M	M dimension of the WMMA operation	8, 16, 32
WMMA_N	N dimension of the WMMA operation	8, 16, 32
TILE_COLS	Width of a thread block tile	2^i for $i \in \{3, 4, 5, \ldots\}$
TILE_ROWS	Height of a thread block tile	2^i for $i \in \{3, 4, 5, \ldots\}$
TILES_PER_CTA	Number of thread block tiles computed per CTA	2^i for $i \in \{0, 1, 2, \ldots\}$
BLOCK_INDEX	Block indexing mode	0, 1, 2
SEQUENTIAL_TILES	Boolean for stride between tiles in the same CTA	0, 1
WMMA_COLS	Amount of WMMA operations per CTA, horizontally	2^i for $i \in \{0, 1, 2, \ldots\}$
WMMA_ROWS	Amount of WMMA operations per CTA, vertically	2^i for $i \in \{0, 1, 2, \ldots\}$
TILE_SHMEM	Boolean for storing tiles of C in shared memory	0, 1
FRAG_A_SHMEM	Boolean for storing fragments of A in shared memory	0, 1
FRAG_B_SHMEM	Boolean for storing fragments of B in shared memory	0, 1

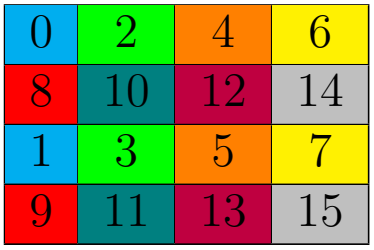
Table 2: An overview of the tunable parameters

3.1 Thread block tiles

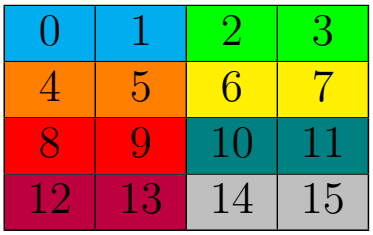
First, the D matrix gets divided into *thread block tiles*. These thread block tiles are of size `TILE_ROWS`×`TILE_COLS`. Since the thread block tiles will later be divided into WMMA operations, whose valid dimensions for mixed-precision operations consist of 8, 16, and 32, the dimensions of the thread block tiles are powers of 2, with 8 being the minimum width and length. The tunable parameters `TILES_PER_CTA`, `BLOCK_INDEX`, and `SEQUENTIAL_TILES` control how the thread block tiles, once defined, get divided across the thread blocks, also called Cooperative Thread Arrays (CTAs).

`TILES_PER_CTA` indicates the number of thread block tiles every CTA computes. In the case where this parameter is larger than 1, the parameter `SEQUENTIAL_TILES` controls how the multiple tiles are laid out. If `SEQUENTIAL_TILES` is set to 1, tiles belonging to the same CTA will be next to each other, and the CTA will compute them from left-to-right. If `SEQUENTIAL_TILES` is set to 0, there will be a stride in between tiles, see Figure 2. Furthermore, the parameter `BLOCK_INDEX` affects the indexing of CTAs to tiles. The indexing mode affects how thread block tiles get mapped to the CTAs. The indexing modes are cartesian, diagonal, and transpose, which are represented through pre-processor defines by 0, 1, and 2, respectively. Figure 3 visually demonstrates how these indexing modes function. The parameters `TILES_PER_CTA`, `SEQUENTIAL_TILES` and `BLOCK_INDEX` can be used with each other to create a large variety of global memory access patterns.

After the thread block tile dimensions and indexing has been defined, the parameter `TILE_SHMEM` indicates whether or not a tile from the C matrix is first loaded into shared memory. The advantage

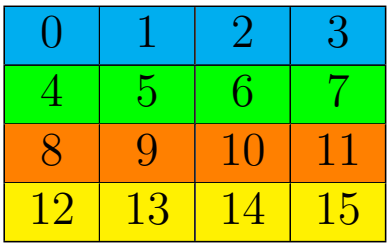


(a) SEQUENTIAL_TILES = 0

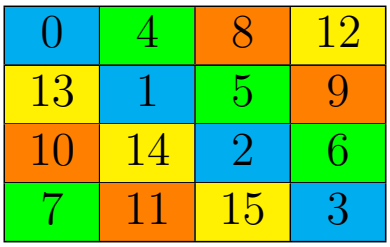


(b) SEQUENTIAL_TILES = 1

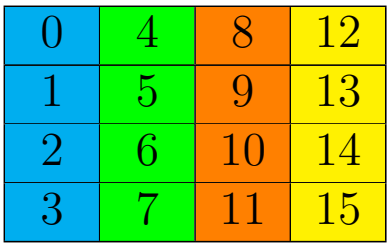
Figure 2: The difference between SEQUENTIAL_TILES values, where tiles of the same colour belong to the same CTA and CTAs compute the numbered tiles from lowest to highest



(a) Cartesian indexing



(b) Diagonal indexing



(c) Transpose indexing

Figure 3: The different CTA to tile indexing modes, where tiles of the same colour belong to the same CTA and CTAs compute the numbered tiles from lowest to highest, and SEQUENTIAL_TILES is set to 1

is that the whole CTA can uniformly copy elements from global memory to shared memory, preventing random global memory accesses that the individual warps would cause. A downside of using shared memory for the thread block tile is that a lot of shared memory is demanded per CTA, since these tiles can grow large and consist of 4-byte floating-point numbers.

3.2 Warp tiles

Once the thread block tiles are fully defined, the work has to be divided across all the warps that make up the CTA. Every warp will be responsible for a *warp tile*, which consists of 1 or more WMMA operations. The parameters `WMMA_M` and `WMMA_N` are tunable in order to generate every mixed-precision WMMA operation, see Table 1. Since the K dimensions is always 16 in these WMMA operations, `WMMA_K` is not tunable. The amount of WMMA operations in a warp tile is determined by the parameters `WMMA_COLS` and `WMMA_ROWS`. Namely, a warp tile is a `WMMA_ROWS`×`WMMA_COLS` 2-dimensional array of WMMA operations, as illustrated in Figure 4.
Next, the warps will load their respective warp tile of the C matrix into `wmma::accumulator` type `wmma::fragment`s and multiply them by (β/α), effectively storing the result of $(\beta/\alpha) \cdot C$ in registers. In order to save on register budget, these fragments holding the thread block tiles will also be used to hold the intermediate result of the matrix multiplication and accumulation. The result at the end will be multiplied by α. This will still produce a mathematically correct result, since:

$$\alpha \cdot A \cdot B + \beta \cdot C = \alpha \cdot (A \cdot B + (\beta/\alpha) \cdot C) \tag{2}$$

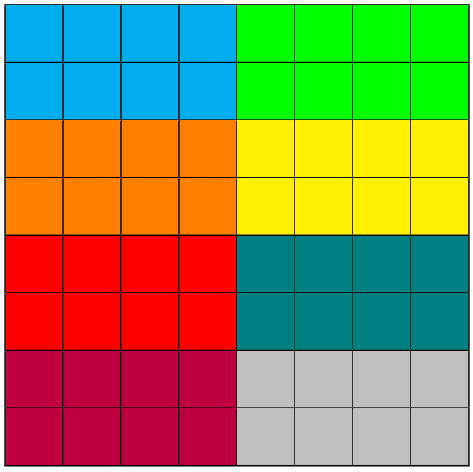


Figure 4: A 128x128 thread block tile, with 16x16x16 WMMA operations and WMMA_COLS = 4, WMMA_ROWS = 2, where each colour is a separate warp tile

This will result in a slight loss of precision, but this can be disregarded due to the general margin of error accepted in mixed-precision computations.
The `wmma::fragments` for A and B are declared as arrays of lengths `WMMA_ROWS` and `WMMA_COLS`, respectively.

3.3 Main loop

Once the elements of C have been loaded into the registers, the main loop can begin. A pseudocode algorithm of the main loop can be found in Algorithm 1. In the main loop, the warps will fully compute the matrix product of the warp tile they are responsible for. This is done by iterating over the K dimension that both the A and B matrices have in common. This iteration is done in steps of `WMMA_K` (16), equal to the K dimension of the WMMA operation. At every iteration step, a double for loop performs the matrix multiplication and accumulation by loading fragments of A and B into registers and calling `wmma::mma_sync`. The A matrix fragment is of shape `WMMA_K` $\times$ `TILE_ROWS` and the B matrix fragment is of shape `TILE_COLS` $\times$ `WMMA_K`.
The outer for loop iterates over the `WMMA_ROWS` of the warp tile and at every iteration loads the fragment needed for that row of WMMA operations into registers. These fragments correspond to the green column in Figure 5 and are from the A matrix. The inner for loop iterates over the `WMMA_COLS` and loads the fragments needed for the columns of the WMMA operations, which correspond to the blue row in Figure 5 and come from the B matrix. In the inner loop the fragments only need to be loaded in the first iteration of the outer loop; when the outer loop iterates over the next row of the warp tile, all the fragments from the B matrix are still present in the registers. Once the inner loop has loaded in the fragment from B, it gets multiplied with the fragment from A that was loaded in by the outer loop and accumulated with the result in the accumulator fragment.
In the kernel, loading the fragments into registers can be done using shared memory. This is configurable by the parameters `FRAG_A_SHMEM` and `FRAG_B_SHMEM`, which indicate if shared memory is used to load fragments from A and B, respectively. Usage of shared memory here indicates that instead of letting the warps load directly from global memory in the 2 inner for loops, the CTA will load all the required fragments into shared memory at the beginning of a main loop iteration, and

Algorithm 1 Main loop for computing matrix multiplication and accumulation

Declare WMMA fragments a, b, acc
for $k = 0, 16, 32, \ldots, K$ **do**
  for $i = 0, 1, 2, \ldots, WMMA\_ROWS$ **do**
    Load fragment from matrix A in $a[i]$
    for $j = 0, 1, 2, \ldots, WMMA\_COLS$ **do**
      if $i = 0$ **then**
        Load fragment from matrix B in $b[j]$
      end if
      $acc[i][j] \leftarrow acc[i][j] + a[i] \times b[j]$
    end for
  end for
end for

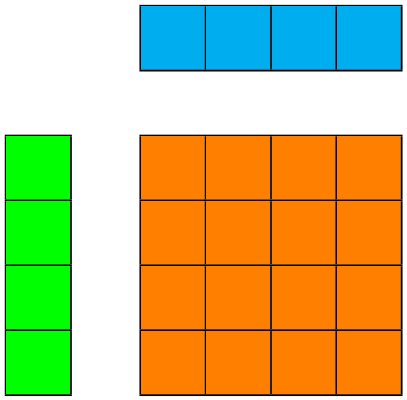


Figure 5: One iteration of the main loop, where the thread block tile is in orange, and the green column and blue row are fragments of the A and B matrices, respectively. In the next iteration, the green column would move right by 16 elements, and the blue row would move down by 16 elements.

Iteration	0	1	2	3	4	...	$n-1$	n
Stage 1	M	C	M	C	M		C	I
Stage 2	I	M	C	M	C	...	M	C

Figure 6: Iterations in a 2 stage pipeline loop, where I = Idle, M = copying Memory, C = Compute

then let the warps load from shared memory. This way there will be fewer global memory accesses in total, since multiple warps can load the same fragment from shared memory.
The main loop then uses a 2-stage pipeline featuring overlapping global-to-shared memory copy and compute using the Cooperative Groups CUDA extension. An overview of the pipeline can be found in Figure 6. For this pipeline, the shared memory buffer has to be doubled in size. The first stage will work with fragments stored in the first half of the shared memory buffer, and the second stage will work with the second half.
In the first iteration, just before the main loop begins, global-to-shared memory copies for stage 1 will be requested using `memcpy_async()`, and stored in the *first* half of the shared memory buffer. This function uses hardware acceleration to directly copy from global memory to shared memory, instead of transferring data through registers, which is the case in regular memory copies. Furthermore, this function is non-blocking, which means that the calling threads can continue executing instructions while the data is being copied.
After the first memory copies have been requested, the kernel enters the main loop and requests the memory copies for the *next* stage. In other words, `memcpy_async()` gets called again, requesting the fragments for the next iteration of the main loop, storing it in the *second* half of the shared memory buffer. After these copies have been requested, the threads will call `wait_prior<X>()`. This function blocks calling threads until the previously requested memory copies are finished, but allows for the last X requests to not have finished. In the first iteration this means that the threads will wait until the memory copies requested *before* the main loop have finished. The warps can then start computing the MMA product of the current stage, while memory copies for the next stage are still in progress. When the warps then finish computing the MMA product of the first iteration of the main loop, they can, in the next iteration, quickly request memory copies and then wait only a short while before they can start computing the next MMA product.
Because memory copies are only requested for the next stage in the main loop, the last MMA product has to be computed outside of the main loop. Once this is done, all that remains is the epilogue, which in this case is simply multiplying the result by α and storing that result in the D matrix.

4 Auto-Tuning

The tunable tensor core kernel will be tuned with Kernel Tuner [vW19]. Multiple GEMM problem dimensions will be considered, as well as multiple GPUs. This section details the search space restrictions and discusses the results of the auto-tuning by examining the performance distribution of all configurations and the impact of individual parameters.

4.1 Restrictions

In order to ensure that the generated configurations are correct, a number of restrictions have to be imposed on the search space:

- The only valid combinations of `WMMA_M` and `WMMA_N` are 8×32, 16×16, and 32×8, and these have to be given as a restriction.
- The warp tile dimensions should not be larger than the thread block tile dimensions, so, `WMMA_N` $\cdot$ `WMMA_COLS` has to be less or equal than `TILE_COLS` (and conversely for `WMMA_M`, `WMMA_ROWS` and `TILE_ROWS`).
- Since the diagonal and transpose thread block indexing modes are only applicable to square tiles, `TILE_COLS` has to equal `TILE_ROWS` whenever the thread block indexing mode is not cartesian.
- Whenever `TILES_PER_CTA` is equal to 1, both `SEQUENTIAL_TILES` options will result in the same kernel, so arbitrarily choose to only generate configurations with `SEQUENTIAL_TILES` $= 1$.
- Due to the way `memcpy_async()` works, it is not desirable to have a group of threads request multiple memory copies in a loop (`memcpy_async()` cannot perform a strided copy). Therefore, ensure that the number of threads in a CTA is equal to or greater than `TILE_COLS` and `TILE_ROWS`, so that 1 thread will at most copy a full 16 element wide row/column of a fragment.

Furthermore, in order to generate a search space that can be worked through in a reasonable amount of time, only a select amount of values can be considered for each parameter. For `WMMA_M` and `WMMA_N`, the value pairs of 8×32, 16×16, and 32×8 will be used. For the thread block tile dimensions `TILE_COLS` and `TILE_ROWS` the values 32, 64, 128, and 256 will be considered. A `TILES_PER_CTA` of 1, 2 and 4 will be used, and all the previously discussed block indexing modes will be considered. With the exception of `TILE_SHMEM`, all boolean parameters will be considered with 0 and 1. During testing, `TILE_SHMEM` reduced performance in every case, and therefore it will not be used to find the optimal configuration. Lastly, `WMMA_COLS` and `WMMA_ROWS` can take on the values of 1, 2, 4, 8, and 16.

4.2 Experiment setup

The experiments for this section and for Section 5 will be ran on the DAS-6 clusters. Namely, the VU and ASTRON clusters will be used, with CUDA versions 12.3 and 12.2.1, respectively. Three GPUs will be considered: the A4000, an Ampere architecture card with "Third-Generation Tensor

GPU and GEMM dimensions	WM	WN	TC	TR	TPC	BI	ST	WC	WR	TS	FAS	FBS
A4000, 4096x4096x4096	16	16	128	64	1	C	1	2	4	0	1	0
A4000, 1024x1024x1024	8	32	128	64	1	C	1	1	8	0	1	0
A4000, 512x1024x128	16	16	64	64	1	C	1	2	2	0	1	0
A100, 4096x4096x4096	16	16	128	128	4	D	0	8	2	0	0	1
A4000 Ada, 4096x4096x4096	8	32	128	128	1	C	1	2	8	0	0	0

Table 3: Best performing configuration for each experiment

Cores" that is readily available on most of the DAS-6 clusters, the A100, a higher-end Ampere card with similar architecture to the A4000 but with increased performance and more advanced memory, and the A4000 Ada, an Ada Lovelace architecture card with "Fourth-Generation Tensor Cores". Due to the availability of the A4000, it will be used with multiple GEMM problem sizes, while the A100 and A4000 Ada will only be considered with a problem size of $4096 \times 4096 \times 4096$. The A4000 and A100 will be used on the VU cluster and the A4000 Ada will be used on the ASTRON cluster.

4.3 Results

All the experiments and their respective best tuning configurations can be found in Table 3. This table contains the following parameters: `WMMA_M` (WM), `WMMA_N` (WN), `TILE_COLS` (TC), `TILE_ROWS` (TR), `TILES_PER_CTA` (TPC), `BLOCK_INDEX` (BI), `SEQUENTIAL_TILES` (ST), `WMMA_COLS` (WC), `WMMA_ROWS` (WR), `TILE_SHMEM` (TS), `FRAG_A_SHMEM` (FAS), and `FRAG_B_SHMEM` (FBS).
Even though the optimal configurations for the experiments ran on the A4000 differ from each other, the optimal values for the parameters themselves are more or less the same for each experiment. This also means that KTdashboard results for the A4000 experiments look very similar. As a result, KTdashboard results that are shown for a specific GEMM problem size on the A4000 are also representative of the results for the other GEMM problem sizes on the A4000.
The first noticeable result, which is generally observed in auto-tuning, is that the optimal configurations all differ from each other, despite the fact that the experiments are ran on similar GPU architectures or even on the same GPU. This confirms the complex nature of tuning GPU kernels: even changing the problem size results in a different optimal configuration. The challenge of tuning GPU kernels is further illustrated by Figures 7 and 8 where it is visible that there are a handful of configurations that perform significantly better than the rest. Without thoroughly exploring the search space of tunable parameters, these would be difficult to find.

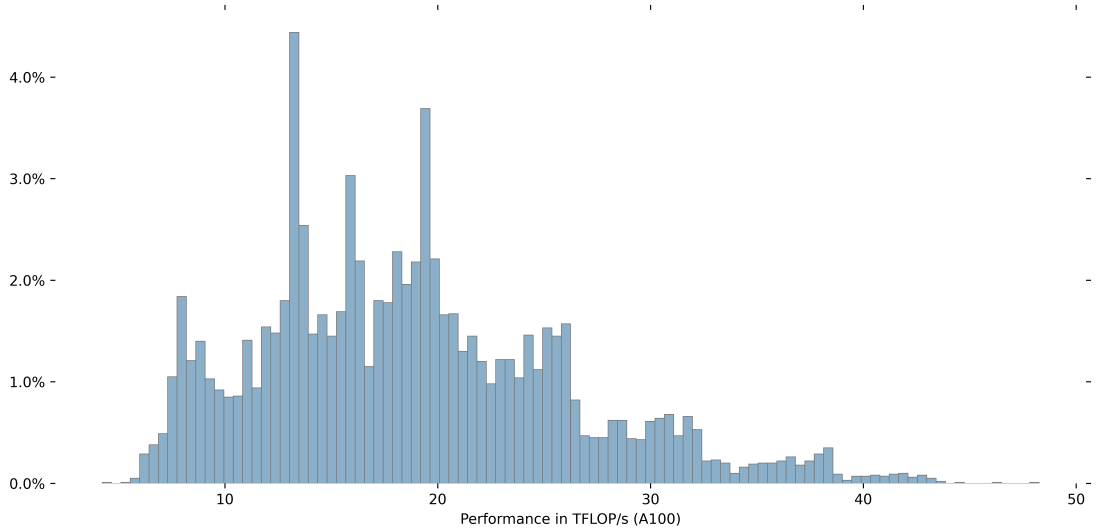


Figure 7: Histogram of the performance of all generated configurations on the A100, GEMM problem size of 4096x4096x4096

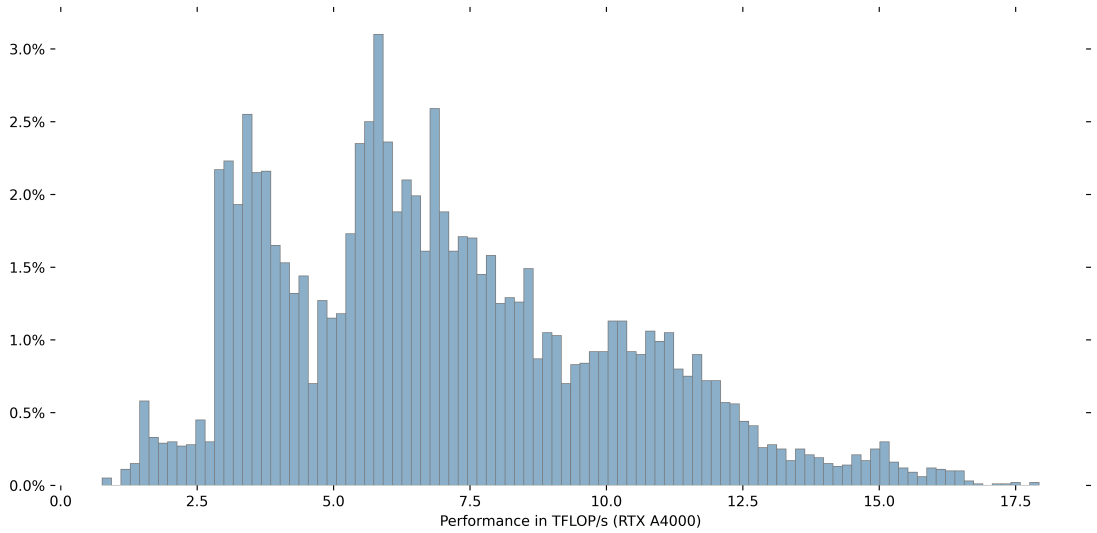
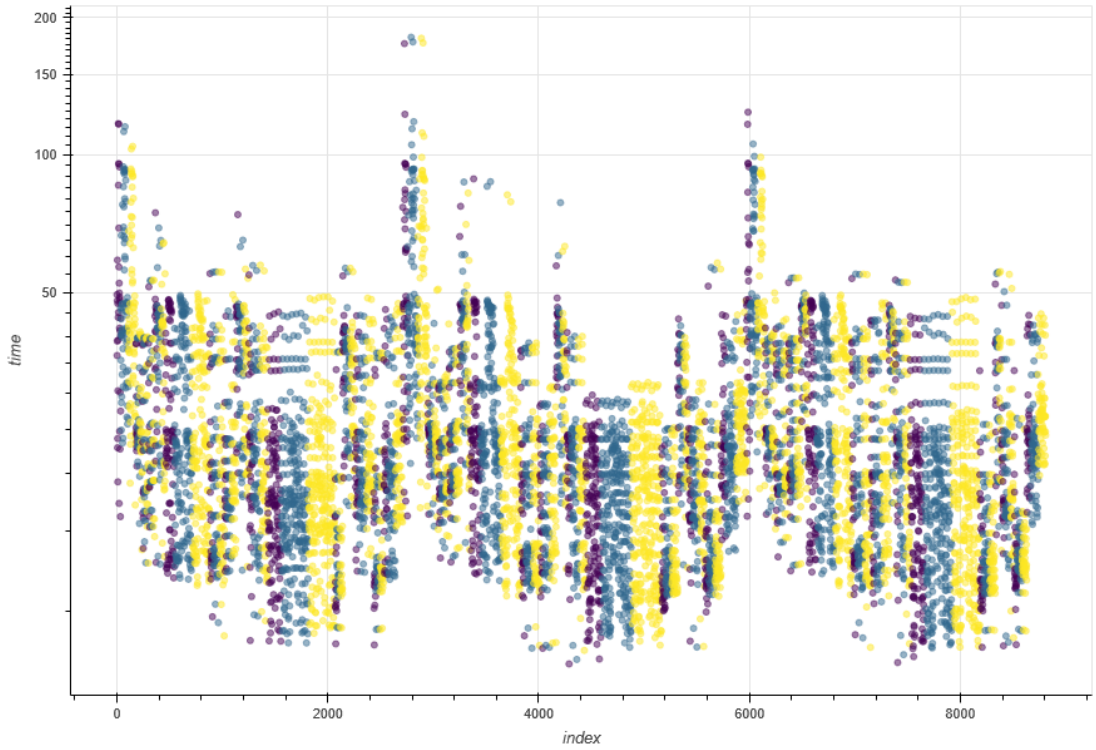
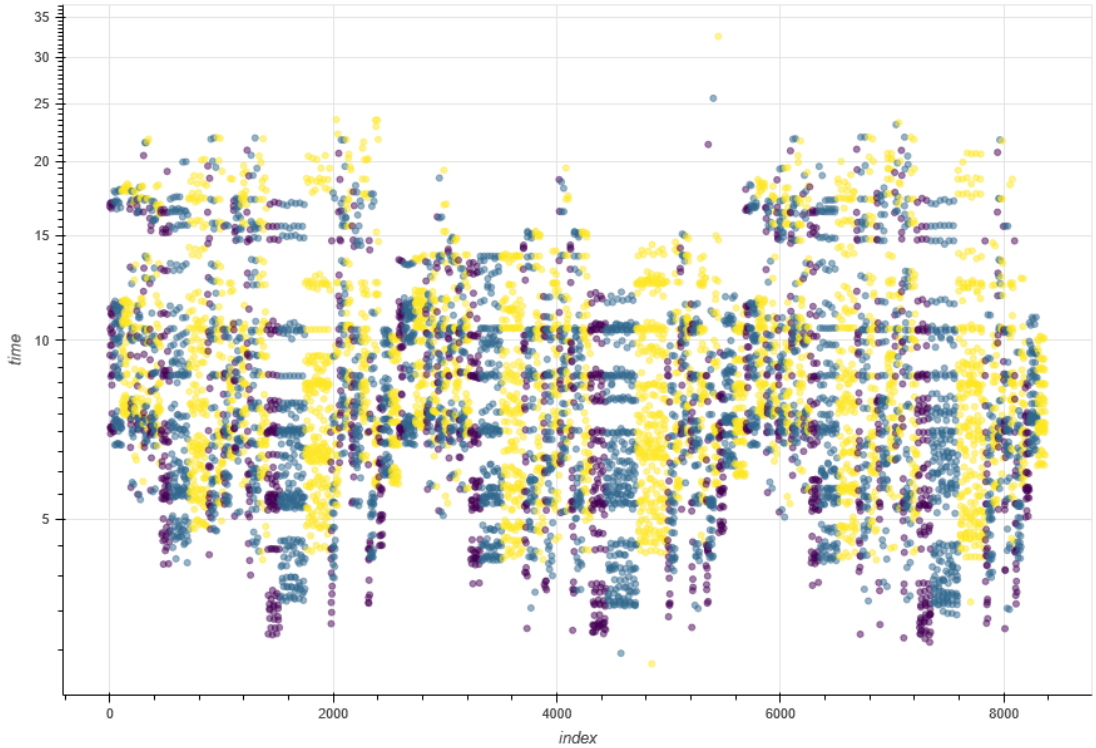


Figure 8: Histogram of the performance of all generated configurations on the A4000, GEMM problem size of 4096x4096x4096

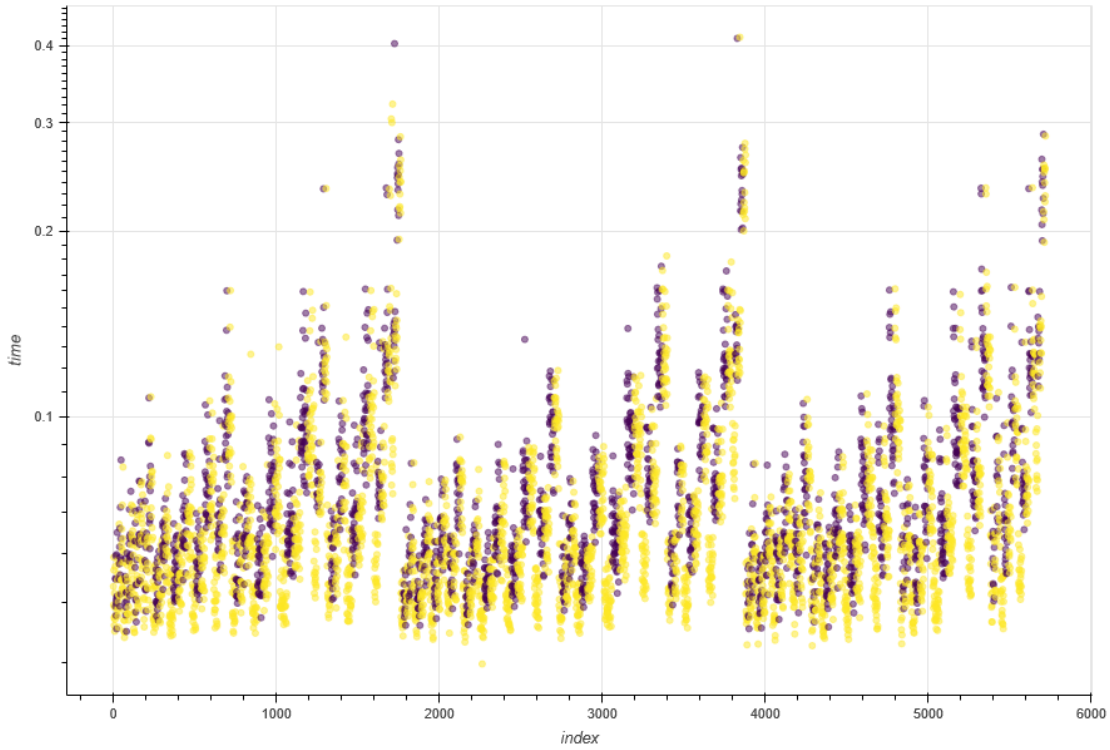


(a) Results for the A4000, GEMM problem size of 4096x4096x4096

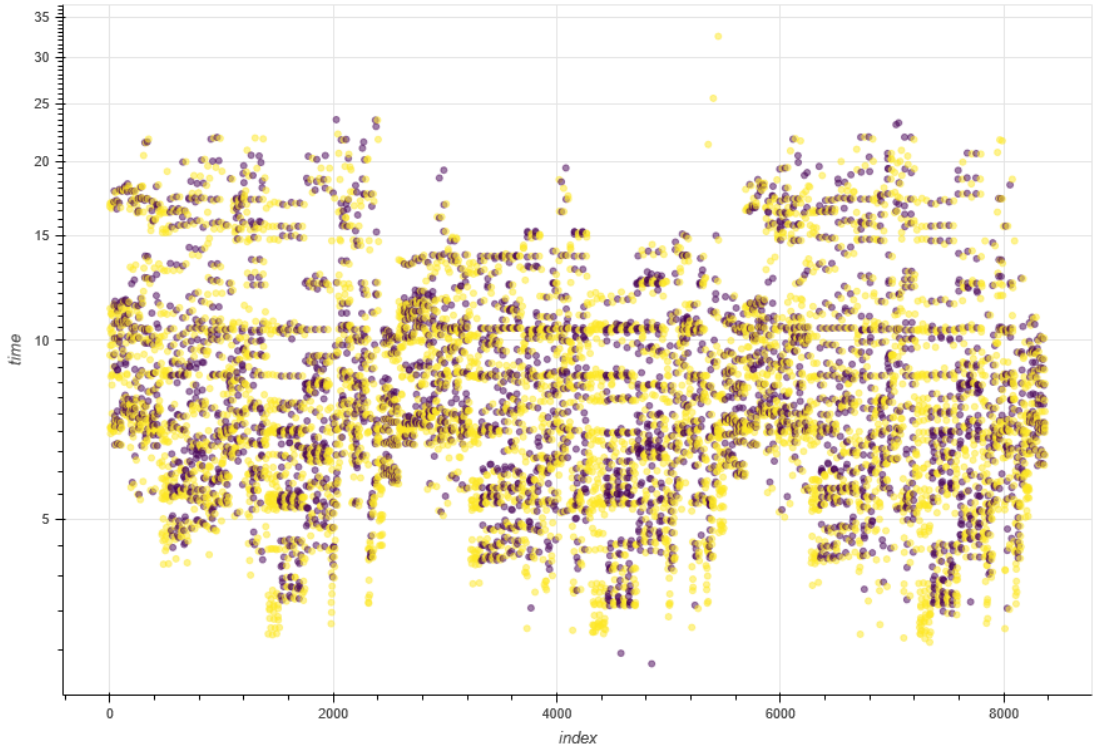


(b) Results for the A100, GEMM problem size of 4096x4096x4096

Figure 9: KTdashboard results, colouring for TILES_PER_CTA, where purple, blue, and yellow represent 1, 2, and 4 tiles per CTA, respectively

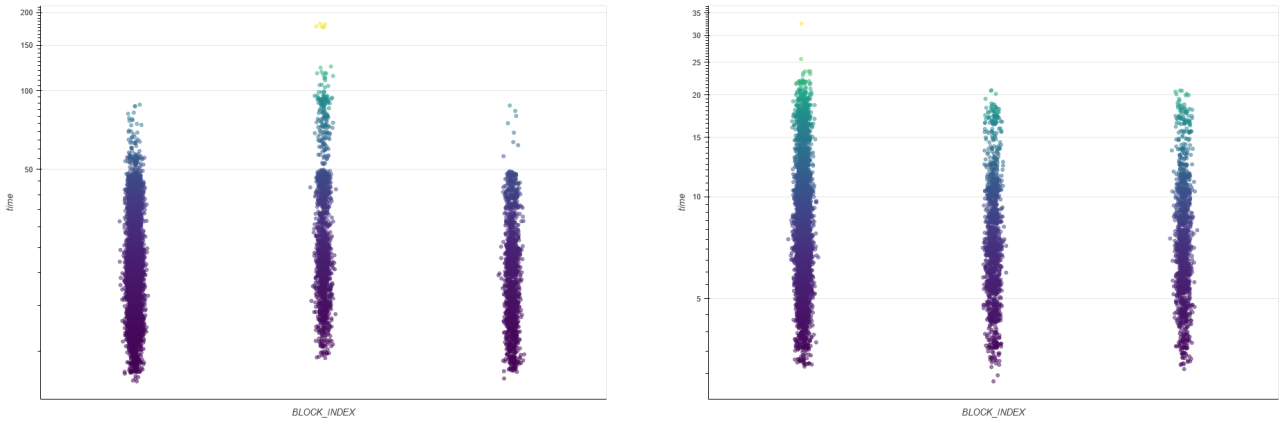


(a) Results for the A4000, GEMM problem size of 512x1024x128



(b) Results for the A100, GEMM problem size of 4096x4096x4096

Figure 10: KTdashboard results, colouring for SEQUENTIAL_TILES, where yellow represents SEQUENTIAL_TILES=1 and purple represents SEQUENTIAL_TILES=0



(a) A4000, GEMM problem size of 4096x4096x4096 (b) A100, GEMM problem size of 4096x4096x4096

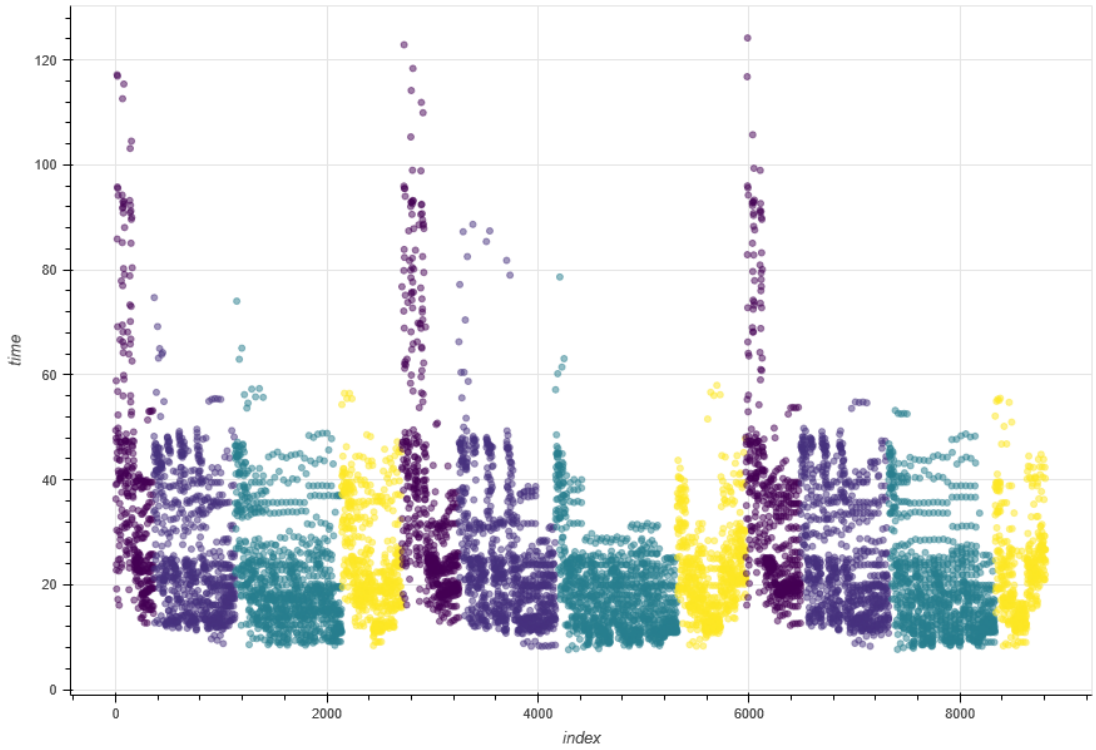
Figure 11: KTdashboard results with BLOCK_INDEX as the x axis. From left to right: cartesian, diagonal, transpose. Colouring for kernel execution time in milliseconds

4.3.1 Memory access patterns

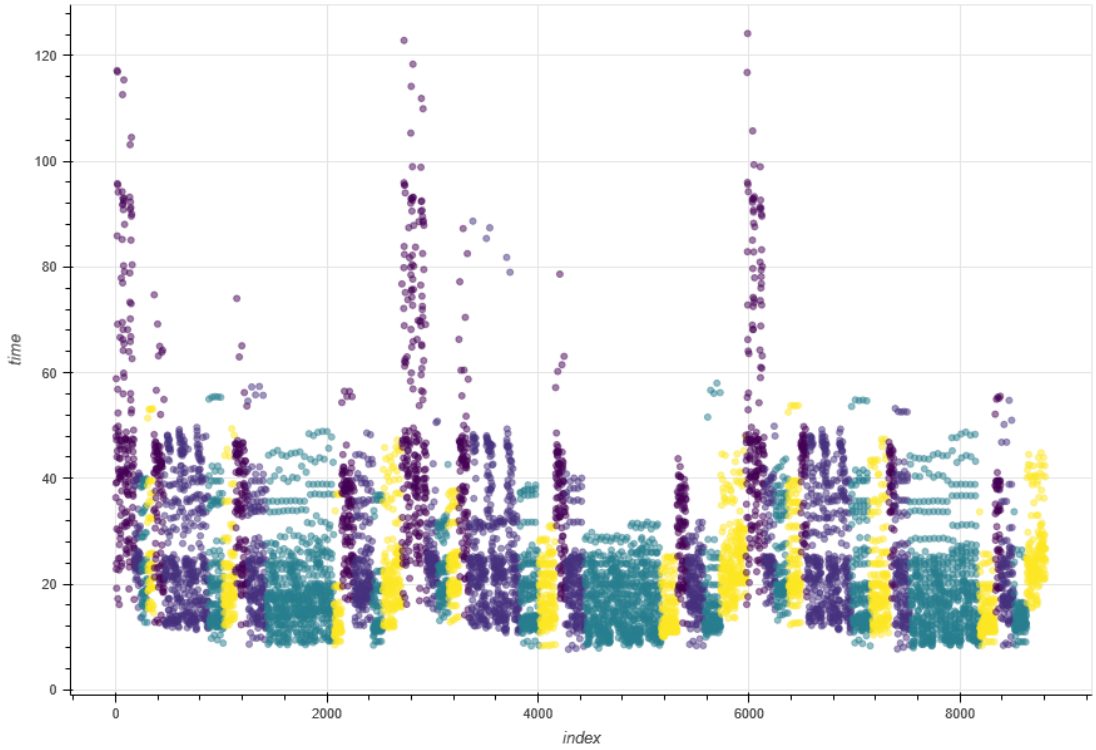
Of the two histograms, the A100 displays a much greater difference between the average and optimal configurations. In Table 3 the optimal configuration for the A100 also differs much more from the rest, using multiple thread block tiles per CTA, non-sequential tiles, and diagonal block indexing. Since these three parameters mainly affect the global memory access pattern, the difference in optimal values is likely caused by the A100's unique memory layout. The performance of `TILES_PER_CTA` parameter values for the A4000 and for the A100 with a GEMM problem size of $4096 \times 4096 \times 4096$ can be found in Figure 9. An interesting observation is that although the best configuration for the A100 uses a `TILES_PER_CTA` of 4, all the other configurations just behind it in performance use 1 or 2 tiles per CTA, and that a `TILES_PER_CTA` of 4 performs worse overall for both GPUs.
A similar observation can be made with regards to `SEQUENTIAL_TILES`, in Figure 10. Despite the two best configurations for the A100 using non-sequential tiles, a `SEQUENTIAL_TILES` of 1 performs better overall. Conversely, for the A4000 with a GEMM problem size of $512 \times 1024 \times 128$, all of the best configurations use a `SEQUENTIAL_TILES` of 1.
The KTdashboard results for the `BLOCK_INDEX` are visible in Figure 11. Here it is clearly visible that for the A4000, the diagonal block indexing performs very poorly, and that the cartesian and transpose block indexing are similar in performance, with the cartesian block indexing performing slightly better. For the A100, the opposite is true, and the diagonal block indexing performs better than the cartesian and transpose block indexing.
This result reveals that the best tuning configuration for a given kernel is not necessarily a combination of the best performing parameter values, but can instead be a combination of specific parameter values that work well with each other.

4.3.2 Thread block tile dimensions

In every single configuration the thread block tile dimensions take on values of either 64 or 128, which are very often used values for thread block tiles in GEMM algorithms. The KTdashboard results for the thread block tile parameters for the A4000 with a GEMM problem size of $4096 \times 4096 \times 4096$



(a) Colouring for TILE_COLS



(b) Colouring for TILE_ROWS

Figure 12: KTdashboard results for the A4000, GEMM problem size of 4096x4096x4096, colouring for the thread block tile dimensions

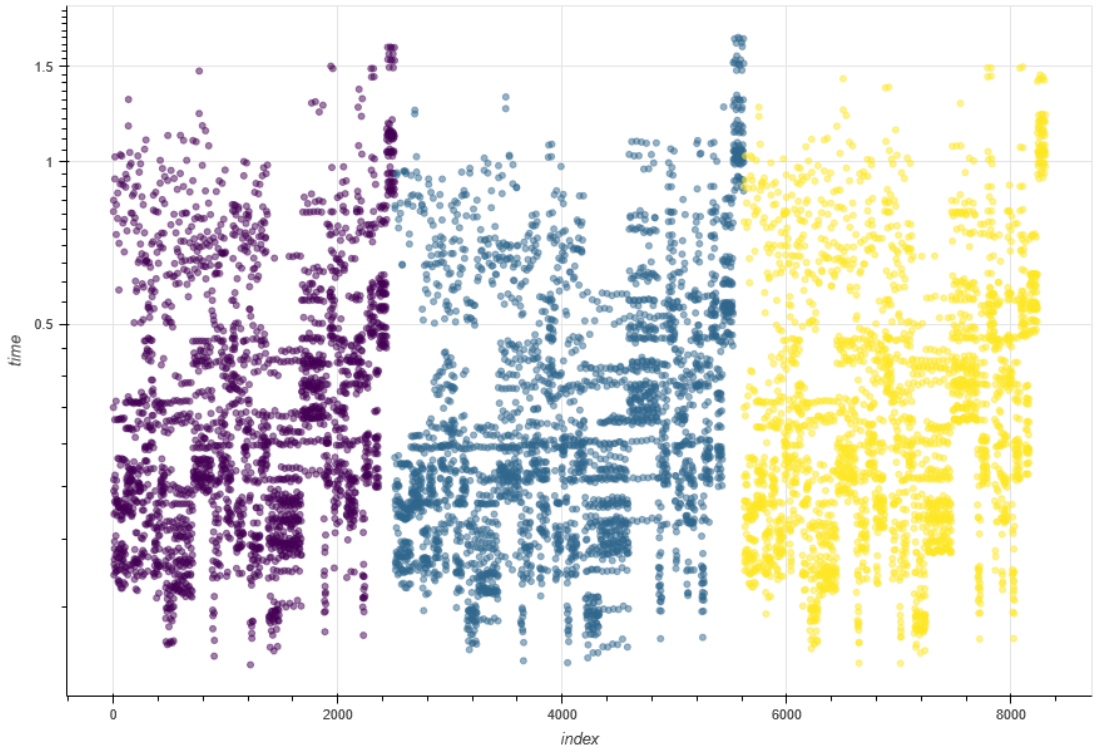


Figure 13: KTdashboard results for the A4000, GEMM problem size of 1024x1024x1024, colouring for WMMA dimensions. From left to right: 8x32, 16x16, 32x8

can be found in Figure 12. Here it can be observed that the optimum lies somewhere between the blue colours, which represent values of 64 and 128. Overall, both the very large thread block dimensions of 256 (in yellow) and the small thread block dimensions of 32 (in purple) perform poorly.
The optimal configuration of the A4000 with a GEMM problem size of $512 \times 1024 \times 128$ uses thread block tile dimensions of 64×64, the smallest of all the configurations. This is likely because the GEMM problem size is also the smallest of all experiments, indicating that, for this problem size, maximum occupancy can only be reached with smaller thread block tiles.

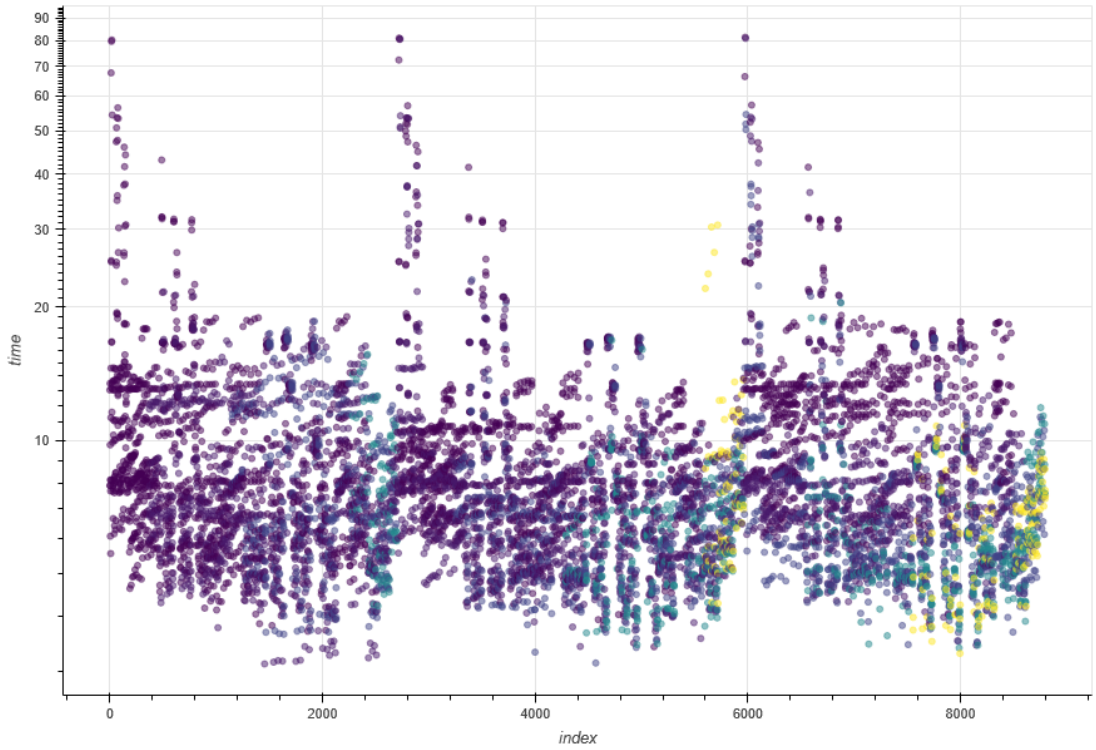
4.3.3 Warp tile dimensions

In Table 3, the optimal `WMMA_M` and `WMMA_N` parameter value pairs are either 16 and 16 or 8 and 32, respectively. Figure 13 shows that for the A4000, with a GEMM problem size of $1024 \times 1024 \times 1024$, the performance of the 3 different WMMA operation are largely identical. This would indicate that there is no inherent advantage in any of the WMMA operation shapes. After all, a warp tile with WMMA shape $8 \times 32 \times 16$, a `WMMA_ROWS` of 8, and a `WMMA_COLS` of 2, has the exact same size as a warp tile with WMMA shape $32 \times 8 \times 16$, a `WMMA_ROWS` of 2, and a `WMMA_COLS` of 8. Then, the only difference between computing these two warp tiles is how the WMMA API performs memory loads for the different WMMA shapes.
The KTdashboard results for the warp tile dimensions for the A4000 and A4000 Ada can be found in Figures 14 and 15, respectively. Recalling Figure 13, the scatter plot can be divided in 3 sections

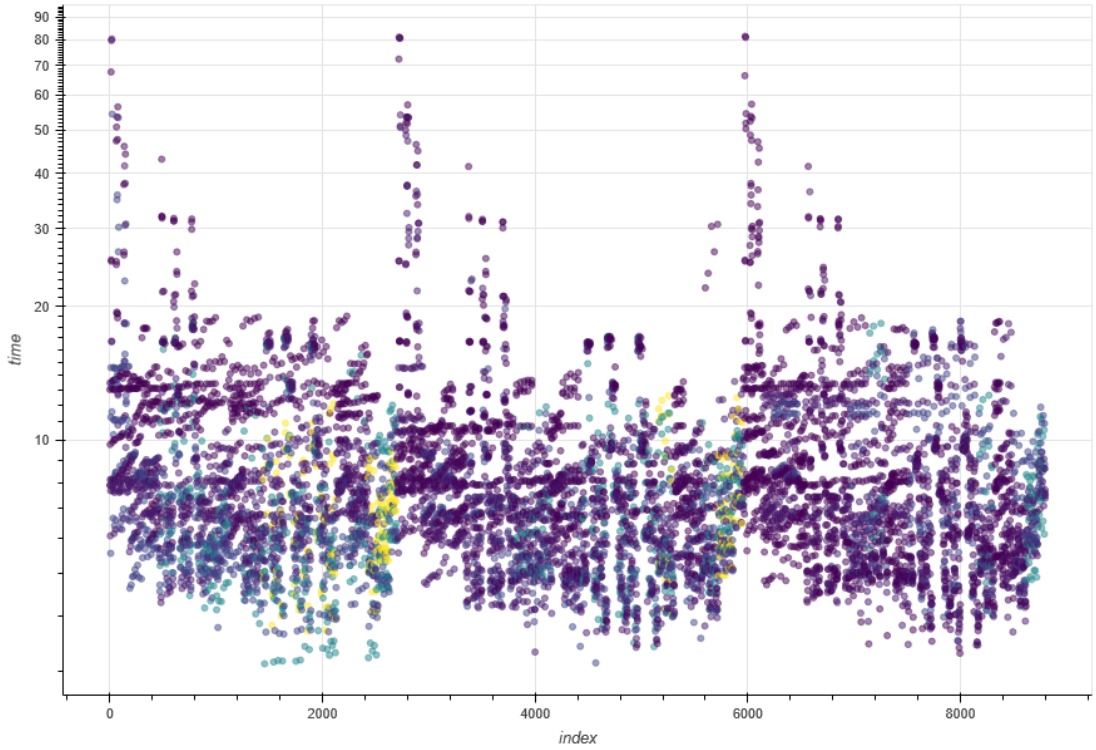
(a) Colouring for WMMA_COLS

(b) Colouring for WMMA_ROWS

Figure 14: KTdashboard results for the A4000, GEMM problem size of 4096x4096x4096, colouring for the warp tile dimensions



(a) Colouring for WMMA_COLS



(b) Colouring for WMMA_ROWS

Figure 15: KTdashboard results for the A4000 Ada, GEMM problem size of 4096x4096x4096, colouring for the warp tile dimensions

of WMMA shapes, due to the order in which the parameters get explored by Kernel Tuner. From left to right, the WMMA dimensions are: $8 \times 32 \times 16$, $16 \times 16 \times 16$, and $32 \times 8 \times 16$. For both GPUs, it is visible that for the $32 \times 8 \times 16$ shaped WMMA operation, a higher `WMMA_COLS` value performs better, and vice-versa for $8 \times 32 \times 16$ and `WMMA_ROWS`. This indicates a tendency for square shaped warp tiles in optimal configurations.

4.3.4 Shared memory

Another interesting observation is the usage of shared memory. Because shared memory usage reduces the number of global memory accesses, it theoretically improves performance. Therefore it is surprising to see that the optimal configurations only use shared memory for matrix A *or* matrix B, instead of for both matrices. This is not an issue with shared memory buffer size since configurations that use shared memory for both matrices do get generated, they just perform worse. Furthermore, the best configuration for the A4000 Ada does not use shared memory at all.

5 Benchmarking

Now that the optimal kernel configurations have been found for the tunable tensor core kernel, the kernel will be compared against other kernels in this section. These kernels will fall into either of two categories:

- The kernel uses tensor cores, but is not tunable.
- The kernel is tunable, but does not use tensor cores.

The experiment setup for this section is the same for Section 4, which is described in Section 4.2.

5.1 Other kernels

For this experiment, three other GEMM kernels will be considered and benchmarked against the tunable tensor core kernel. The first kernel is an OpenCL implementation in the CLBlast library [Nug18]. This kernel is tunable, but does not make use of tensor cores. For this experiment, this kernel has been tuned for half-precision to be in accordance with the mixed-precision computation of the tunable tensor core kernel. Like the tunable tensor core kernel, the CLBlast kernel will be tuned seperately for each experiment with Kernel Tuner. The second kernel that will be considered is from the cuBLAS library [NVIa]. In particular, the `cublasSgemmEx()` function will be considered to enable mixed-precision matrix multiplication. cuBLAS is not tunable by the user, but does make use of tensor cores. The third kernel that will be considered will be a kernel from the CUTLASS library [NVIc]. As mentioned in Section 2.2.1, CUTLASS makes use of tensor cores and is partially tunable. Since the optimal parameters for a CUTLASS kernel are difficult to find, the native parameters of the CUTLASS kernel will be used. In other words, no parameters other than usage of tensor cores and the necessary definitions of the GEMM problem will be specified. Since default parameters are not specified for newer architectures, the performance of the CUTLASS kernel may be further impacted.
When using a profiler, it is observed that for this problem the cuBLAS function actually calls a CUTLASS kernel. Regardless, in the discussion below cuBLAS and CUTLASS will be considered as two seperate kernels.

5.2 Results

The results of the benchmarking process can be found in Figure 16. The tunable tensor core, CUTLASS, and cuBLAS kernels were ran and measured 10.000 times in a separate CUDA program, and then averaged to take the result. The CLBlast results come from the reported performance of the best performing configuration by Kernel Tuner, and may therefore deviate a bit from the measurements reported by CUDA.
The first observation to make is that cuBLAS easily outperforms every other kernel, including the CUTLASS kernel. This is to be expected, since CUTLASS kernels are meant to be hand-tuned to achieve maximum performance, and cuBLAS is meant to have the best performing out-of-the-box GEMM kernels from NVIDIA. CLBlast performs the worst in every category, reinforcing that tensor cores are critical to achieving high GEMM performance. CUTLASS is the 2nd best performer for every experiment except for the A4000 with a GEMM problem size of $512 \times 1024 \times 128$,

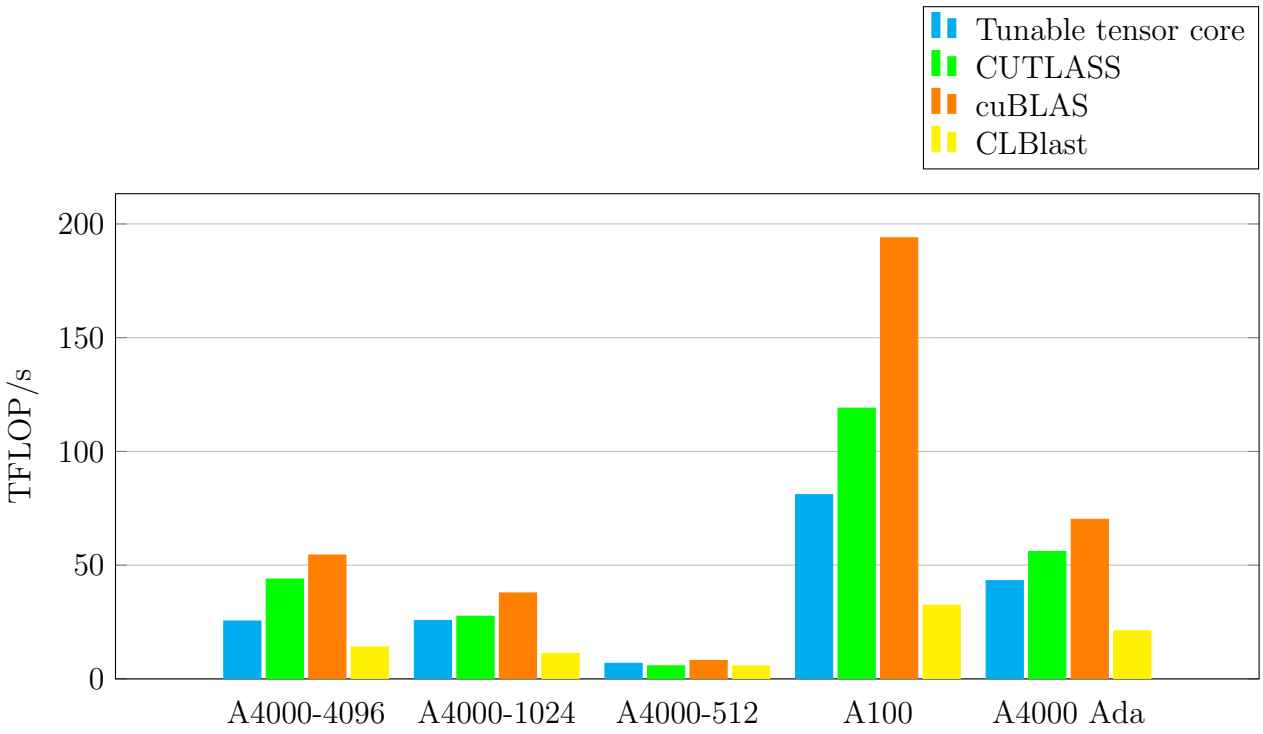


Figure 16: Average performance in TFLOP/s of 10.000 runs for each kernel, for each experiment

Statistic	Tunable tensor core	cuBLAS
Compute throughput	50.97%	90.43%
Memory throughput	84.76%	58.82%
L1 cache throughput	86.35%	36.36%
L2 cache throughput	61.98%	58.82%
Pipe Tensor Cycles Active	50.97%	90.43%
Executed instructions	318.783.488	71.132.209

Table 4: Nsight Compute statistics for the tunable tensor core kernel and for the cuBLAS kernel

where the tunable tensor core kernel overtakes it. The tunable tensor core and CLBlast kernels have an advantage for this problem because the auto-tuning enables them to adapt better to the unusual GEMM dimensions, and because the other kernels are optimized for large matrices. The performances of all kernels are a lot closer to each other for the A4000 with a GEMM problem size of $512 \times 1024 \times 128$ since the K dimension is so small, which causes the main loop to be a lot shorter.

5.3 Profiler results

Using NVIDIA Nsight Compute, a profiler for CUDA programs, many statistics about the kernel execution can be gathered in order to analyze the kernel performance in depth. Table 4 contains statistics obtained with Nsight Compute for the tunable tensor core kernel and for the cuBLAS kernel, on the A4000 with a GEMM problem size of $4096 \times 4096 \times 4096$. Here, throughput is

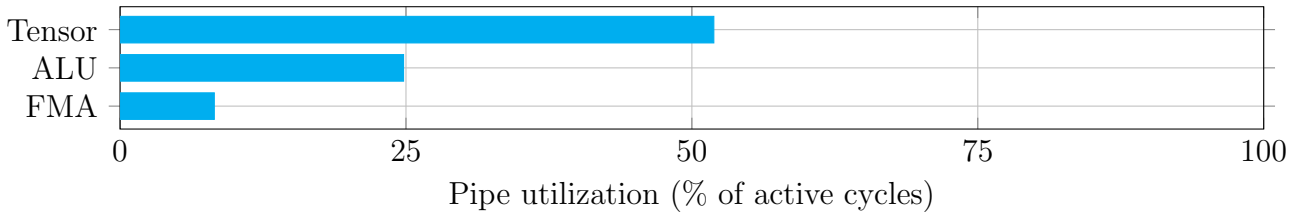


Figure 17: Pipeline utilization for the tunable tensor core kernel

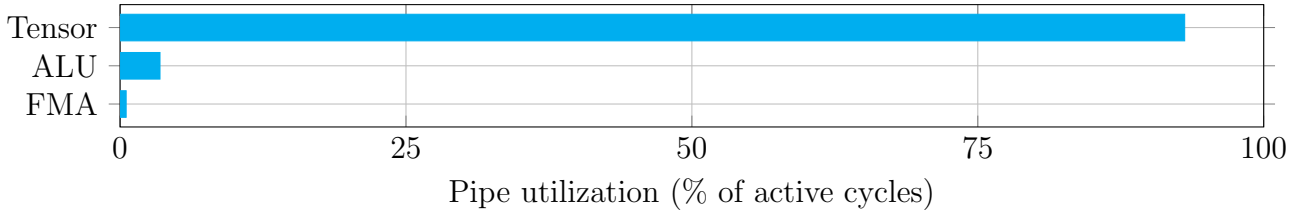


Figure 18: Pipeline utilization for cuBLAS

defined as the percent of the peak sustained rate achieved during all elapsed cycles of the kernel execution. Usually, compute throughput is the most important statistic, and is desired to be as high as possible.

The first observation to make is that the cuBLAS kernel shows a very high compute throughput and a lower memory throughput. The tunable tensor core kernel shows the opposite, with a high memory throughput but a lower compute throughput. The tunable tensor core also has a much higher L1 cache throughput. This indicates that the tunable tensor core kernel spends too many cycles executing memory instructions, and too little cycles in compute instructions. The "pipe tensor cycles active" statistic is equal to the compute throughput, meaning that the peak compute performance happens whenever the tensor cores are active. Therefore, compute throughput can be increased by ensuring that the kernel spends the most amount of time possible using tensor cores to perform MMA operations.

Figures 17 and 18 illustrate the pipeline utilization of the different hardware units on the streaming multiprocessors. Once again, the cuBLAS kernel makes much greater use of the tensor cores than the tunable tensor core kernel. This indicates that cuBLAS has a much more optimized main loop, where data copies and computes are better overlapped, and that cuBLAS has a better memory access pattern, having to wait less for memory loads to finish. The 10 most executed instructions for the tunable tensor core kernel can be found in Figure 19 and for the cuBLAS kernel in Figure 20. For the tunable tensor core kernel, the most executed instruction is integer-multiply-and-add (IMAD), which is executed by the FMA. This is why the tunable tensor core kernel has a greater utilization of the FMA than the cuBLAS kernel. Matrix-multiply-and-accumulate (HMMA) is the cuBLAS kernel's most and tunable tensor core kernel's second most executed instruction, and is the instruction that gets executed by the tensor cores. For both kernels, the number of HMMA instructions is the same. Most of the IMAD instructions for the tunable tensor core kernel are used to calculate memory addresses. Since the difference in the number of executed instructions is not proportional to the difference in performance, the large number of IMAD instructions is likely not the main reason that the tunable tensor core kernel performs poorly, but probably still causes a performance bottleneck.

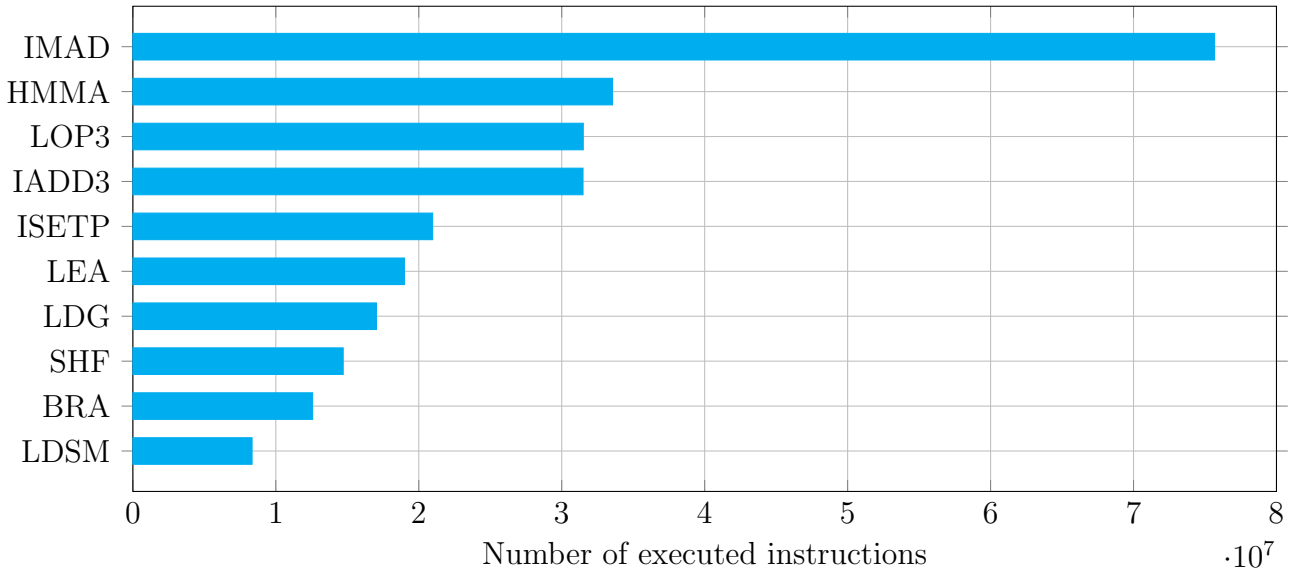


Figure 19: The 10 most executed instructions and their respective instruction count for the tunable tensor core kernel

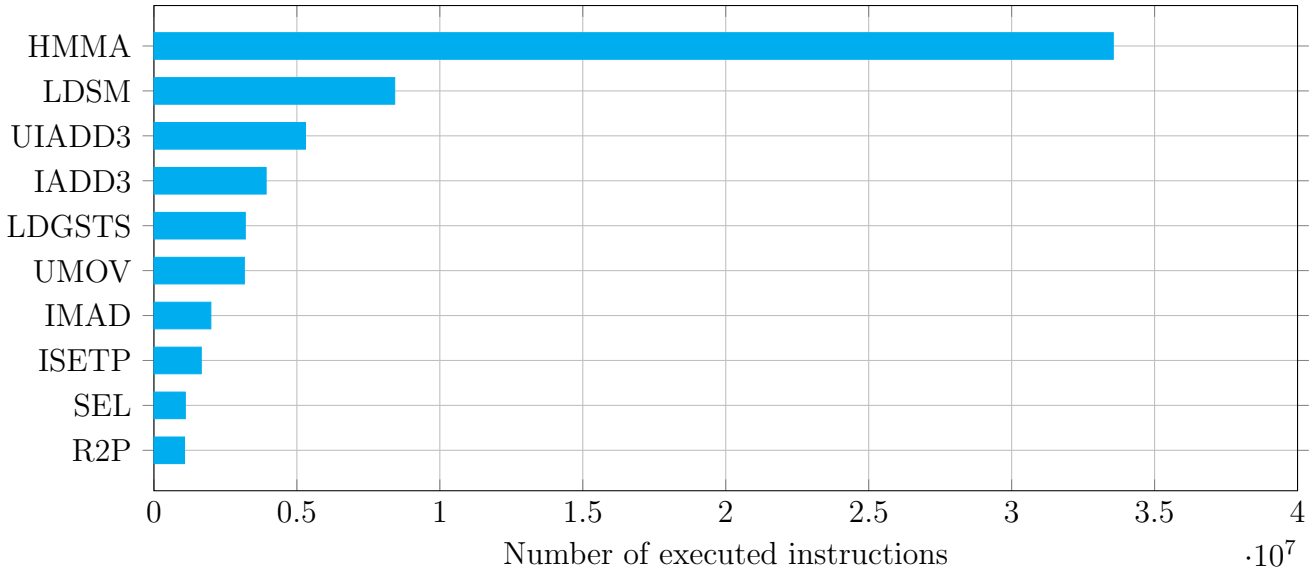


Figure 20: The 10 most executed instructions and their respective instruction count for the cuBLAS kernel

6 Conclusions and further research

Overall, even though the tunable tensor core kernel outperforms CLBlast, a tunable GEMM kernel that does not use tensor cores, the tunable tensor core kernel performs quite poorly compared to existing kernels that do make use of tensor cores. Still, the tunable tensor core kernel can be used in auto-tuning research, serving as an alternative to or possibly replacing the much older CLBlast kernel. Furthermore, it is shown that auto tuning does help in developing a GEMM kernel that uses tensor cores, with a handful of configurations per experiment being greatly ahead in performance compared to the rest. By hand, these optimal configurations would be hard to find. In conclusion, auto-tuning can help a lot with developing GEMM kernels, but only auto-tuning is not enough to develop kernels that match performance of state-of-the-art GEMM kernels.
In the future, a tunable tensor core kernel may be developed that aims to fix the performance problems identified in Section 5.3. Some of these problems can possibly be resolved by implementing more optimizations that CUTLASS uses, such as "warp-scoped matrix fragments", an optimization similar to the overlapping memory copy and compute, but instead targeting registers. Another tunable tensor core kernel can be developed with other methods of accessing tensor cores, such as using CuTe, or by directly targeting PTX instructions. It could also incorporate hardware features of newer GPUs, such as Tensor Memory Accelerators in the new Hopper architecture. More features can also be made tunable, such as choosing whether or not to take the transpose of the A or B matrices, the number of stages in the pipeline of the main loop, and integer data types.
Lastly, developing an energy efficient GEMM kernel is especially interesting. Since GEMM forms the backbone of many neural networks, it becomes very important to consider the energy usage of developing deep learning applications, and possibly shifting the focus of GEMM kernels from FLOPS performance to energy efficiency, since these two performance metrics usually do not overlap [SVvWB22].

References

[AKV+14] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 303–315, 2014.

[AMD20] AMD. Amd cdna architecture. https://www.amd.com/content/dam/amd/en/documents/instinct-business-docs/white-papers/amd-cdna-white-paper.pdf, 2020. Accessed: 28-6-2024.

[AY] Jeremy Appleyard and Scott Yokim. Programming tensor cores in cuda 9. https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/.

[BDG+18] Prasanna Balaprakash, Jack Dongarra, Todd Gamblin, Mary Hall, Jeffrey K. Hollingsworth, Boyana Norris, and Richard Vuduc. Autotuning in high-performance computing applications. *Proceedings of the IEEE*, 106(11):2068–2083, 2018.

[BEdL+16] Henri Bal, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. A medium-scale distributed system for computer science research: Infrastructure for the long term. *Computer*, 49(5):54–63, 2016.

[BHRS08] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 101–113. Association for Computing Machinery, 2008.

[CCH07] Chun Chen, Jacqueline Chame, and Mary W. Hall. Chill : A framework for composing high-level loop transformations. 2007.

[CMJ+18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning, 2018.

[FJ98] M. Frigo and S.G. Johnson. Fftw: an adaptive software architecture for the fft. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)*, volume 3, pages 1381–1384 vol.3, 1998.

[FPB17] Jiří Filipovič, Filip Petrovič, and Siegfried Benkner. Autotuning of opencl kernels with global optimizations. In *Proceedings of the 1st Workshop on AutotuniNg and ADaptivity AppRoaches for Energy Efficient HPC Systems*. Association for Computing Machinery, 2017.

[GAGN15] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International conference on machine learning*, pages 1737–1746. PMLR, 2015.

[GL11] Dominik Grewe and Anton Lokhmotov. Automatically generating and tuning gpu code for sparse matrix-vector multiplication from a high-level representation. In *GPGPU-4: Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, page 12, 03 2011.

[HNS09] Albert Hartono, Boyana Norris, and P. Sadayappan. Annotation-based empirical performance tuning using orio. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–11, 2009.

[JYP+17] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, and Jonathan Ross. In-datacenter performance analysis of a tensor processing unit. In *44th International Symposium on Computer Architecture (ISCA)*, 2017.

[KC06] Patrice Simard Kumar Chellapilla, Sidd Puri. High performance convolutional neural networks for document processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition, Université de Rennes 1*, oct 2006.

[KMDT] Andrew Kerr, Duane Merrill, Julien Demouth, and John Tran. Cutlass: Fast linear algebra in cuda c++. https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/.

[LDT09] Yinan Li, Jack Dongarra, and Stanimire Tomov. A note on auto-tuning gemm for gpus. In *Proceedings of the 9th International Conference on Computational Science: Part I*, pages 884–892, 05 2009.

[MCL+18] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. Nvidia tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531, 2018.

[NC15] Cedric Nugteren and Valeriu Codreanu. Cltune: A generic auto-tuner for opencl kernels. *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, pages 195–202, 2015.

[Nug18] Cedric Nugteren. Clblast: A tuned opencl blas library. In *Proceedings of the International Workshop on OpenCL*, IWOCL '18. ACM, May 2018.

[NVIa] NVIDIA. cublas. https://developer.nvidia.com/cublas.

[NVIb] NVIDIA. Cute. https://github.com/NVIDIA/cutlass/blob/main/media/docs/cute/00_quickstart.md.

[NVIc] NVIDIA. Cutlass. https://github.com/NVIDIA/cutlass.

[NVId] NVIDIA. Cutlass: Cuda template library for dense linear algebra at all levels and scales. https://on-demand.gputechconf.com/gtc/2018/presentation/s8854-cutlass-software-primitives-for-dense-linear-algebra-at-all-levels-and-sc pdf.

[NVIe] NVIDIA. gemm-hierarchy-with-epilogue.png. https://github.com/NVIDIA/cutlass/blob/main/media/images/gemm-hierarchy-with-epilogue.png.

[NVI17] NVIDIA. Nvidia tesla v100 gpu architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017. Accessed: 19-2-2024.

[NVI24] NVIDIA. *Parallel Thread Execution ISA Version 8.5*, 2024. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#warp-level-matrix-multiply-accumulate-instructions.

[PMJ+05] M. Puschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, Jianxin Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson, and N. Rizzolo. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[RRS+08] Shane Ryoo, Christopher Rodrigues, Sam Stone, Sara Baghsorkhi, Sain-Zee Ueng, John Stratton, and Wen-mei Hwu. Program optimization space pruning for a multithreaded gpu. In *Proceedings of the 2008 CGO - Sixth International Symposium on Code Generation and Optimization*, pages 195–204, 04 2008.

[RSSG21] Ari Rasch, Richard Schulze, Michel Steuwer, and Sergei Gorlatch. Efficient auto-tuning of parallel programs with interdependent tuning parameters via auto-tuning framework (atf). *ACM Trans. Archit. Code Optim.*, 18(1), jan 2021.

[SVvWB22] Richard Schoonhoven, Bram Veenboer, Ben van Werkhoven, and Kees Joost Batenburg. Going green: optimizing gpus for energy efficiency through model-steered auto-tuning, 2022. https://arxiv.org/abs/2211.07260.

[TNLD10] Stanimire Tomov, Rajib Nath, Hatem Ltaief, and Jack Dongarra. Dense linear algebra solvers for multicore with gpu accelerators. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8, 04 2010.

[vW19] Ben van Werkhoven. Kernel tuner: A search-optimizing gpu code auto-tuner. *Future Generation Computer Systems*, 90:347–358, 2019.

[vWPS20] Ben van Werkhoven, Willem Jan Palenstijn, and Alessio Sclocco. Lessons learned in a decade of research software engineering gpu applications. In *International Conference on Computational Science, ICCS 2020*, pages 399–412, jan 2020.

[WD98] R.C. Whaley and J.J. Dongarra. Automatically tuned linear algebra software. In *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, pages 38–38, 1998.

[YSL+23] Yongseung Yu, Donghyun Son, Younghyun Lee, Sunghyun Park, Giha Ryu, Myeongjin Cho, Jiwon Seo, and Yongjun Park. Tailoring cutlass gemm using supervised learning. In *2023 IEEE 41st International Conference on Computer Design (ICCD)*, pages 465–474, 2023.

[YWC20] Da Yan, Wei Wang, and Xiaowen Chu. Demystifying tensor cores to optimize half-precision matrix multiply. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 634–643, 2020.