

Universiteit Leiden

# Master Computer Science

Towards correlating radio astronomy signals on the Blackwell microarchitecture


| | |
|---|---|
| Name: | Tobias Hofstra |
| Student ID: | s3322874 |
| Date: | 14/08/2026 |
| Specialisation: | Advanced Computing and Systems |
| 1st supervisor: | Ben van Werkhoven |
| 2nd supervisor: | John Romein |


Master's Thesis in Computer Science


Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

**Abstract**

Modern radio telescopes typically come in the form of an array of antennas in order to achieve high sensitivity and resolution. The signals that these antennas receive must be combined (correlated), which is a computationally expensive task, especially if this is done in real time. Due to the high data rates, this task is performed on GPUs. The Tensor-Core Correlator is a highly performant and efficient correlator implementation that utilizes Tensor Cores on NVIDIA GPUs, but it is optimized for the Ampere microarchitecture and does not fully utilize the hardware features and software techniques of newer microarchitectures. In this thesis, we first develop an efficient kernel for computing General Matrix Multiply (GEMM), an operation closely related to correlation. In this kernel, we use recent GEMM kernel design principles and make full use of the hardware features of the new Blackwell microarchitecture. We then extend this normal GEMM kernel to multiple variants that perform complex multiplication. We find that the normal GEMM kernel performs very well, and achieves close to peak performance on an NVIDIA Jetson Thor. Furthermore, we find that the GEMM variants capable of performing complex multiplication offer valuable insights into efficiently performing the data transformations required for complex GEMM and correlation on the Blackwell microarchitecture, and that the best performing variant achieves 83.5% of the performance of the normal GEMM kernel.

# 1 Introduction

Graphical Processing Units (GPUs) are becoming more and more important with the increasing computational demands of AI models and, in particular, Large Language Models (LLMs). Accordingly, development of GPU hardware is becoming increasingly focused towards speeding up AI workloads. An example of this on modern NVIDIA GPUs are Tensor Cores [5], specialized hardware units capable of performing Matrix Multiply and Accumulate (MMA) operations at considerably higher throughput than regular cores. Although Tensor Cores can operate on multiple different integer and floating-point types, there has been a focus towards narrow-precision floating-point numbers due to the low precision demands of deep learning applications. The usage of Tensor Cores is critical for achieving optimal performance in implementations of GEMM, the algorithm that forms the computational backbone of the neural networks that support LLMs.

AI is not the only area where GPUs are used to perform computations. GPUs are of particular interest in the field of High Performance Computing (HPC), where they can accelerate simulations and computations in many different scientific disciplines. Radio astronomy is among those disciplines that see a high computational demand. This is in large part due to the trend of radio telescopes that consist of a large amount of geographically distributed antennas. Examples include the Low Frequency Array (LOFAR) [9], the Atacama Large Millimeter/Submillimeter Array (ALMA) [24], and the Square Kilometer Array (SKA) [8]. An important task in these radio telescopes is the correlation of the different signals that the receivers pick up in order to filter out noise that arises from having smaller, less powerful antennas. Performing this task in real time for a large amount of receivers is incredibly computationally intensive [22] and requires specialized hardware or GPUs, with the latter becoming a more frequent choice than the former for implementing correlator kernels.

The recent NVIDIA Blackwell architecture [6] introduces many new hardware features to accelerate AI and HPC workloads. Among these, the fifth generation Tensor Cores and Tensor Memory [7] are of particular interest. Although a correlator kernel that uses Tensor Cores already exists [17] (the Tensor-Core Correlator), these new features pose a large enough departure from previous generations of Tensor Cores in terms of functionality that a new correlator kernel is necessary in order to fully utilize Blackwell architecture GPUs.

There are two main challenges in adapting the Tensor-Core Correlator to the Blackwell architecture: support for complex number multiplication and the asynchronous execution of fifth generation Tensor Cores. There are more features of the Blackwell architecture that have to be accounted for, but these two challenges represent the main points that will be investigated for the implementation of the new correlator kernel.

This thesis will work towards adapting the Tensor-Core Correlator for the Blackwell architecture by (1) investigating the microarchitectural differences between the Ampere architecture targeted by the Tensor-Core Correlator and the Blackwell architecture, including the corresponding differences between the Tensor Cores and their usage in GEMM kernels, and by (2) investigating how to efficiently perform correlation on Blackwell Tensor Cores, including comparing multiple complex GEMM strategies.

The thesis is laid out as follows: Section 2 describes the background information and related work pertaining to signal correlation, GEMM, Tensor Cores, and the Tensor-Core Correlator; Section 3 describes the structure of an efficient non-complex GEMM kernel and multiple complex GEMM/correlation strategies; Section 4 describes the experiments on the different kernels and contains the results; Section 5 discusses the results; and finally Section 6 contains the conclusions and future work.

# 2 Background and Related Work

In Section 2.1, an overview of signal correlation is given, including how it can be computed, and an overview of related works that handle signal correlation. Section 2.2 gives a short overview of GEMM and how it relates to signal correlation. Sections 2.3, 2.4 and 2.5 each describe the microarchitectural details and a typical GEMM kernel of the Ampere, Hopper, and Blackwell architectures, respectively. Section 2.6 gives a complete overview of the Tensor-Core Correlator, including an overview of how Tensor Cores can be made to work with complex numbers. Section 2.7 describes CuTe, the library used in order to program Blackwell Tensor Cores.

## 2.1 Signal correlation

One of the main interests of radio astronomy is the ability to construct images of the sky from radio signals. The resolution of these images is in large part limited by the *angular resolution*. The angular resolution for a radio telescope with a single antenna is limited by the diameter of the telescope, which becomes impractical to keep increasing. As a result, radio telescopes consisting of an array of antennas have become increasingly popular in order to increase angular resolution and sensitivity. This introduces a pipeline where signals from the different antennas have to be filtered, compensated for delays (since the antennas are geographically distributed, the wavefront of a celestial source hits the antennas at different times), corrected, and, finally, correlated [2]. In this thesis, and in the Tensor-Core Correlator, only the correlation step is considered.

There are, broadly, two main types of correlators: XF correlators and FX correlators, where F denotes a Fourier transform that filters the data into frequency channels, and X denotes the cross correlation. An XF correlator, then, first performs the cross correlation, and then filters the correlated signals. The FX correlator, which is the type considered in this thesis, first filters the input data, and then performs the cross correlation on the filtered data signals. The main advantage of the FX correlator is that the filtered frequency channels can be processed independently, exposing ample parallelism that the GPU can take advantage of.

The signal can be expressed as $e^{2\pi ift}$, where $f$ is the frequency and $t$ is time. Thus, a sample can be expressed as a complex number of the form $e^{2\pi ift} = \cos(2\pi ft) + i\sin(2\pi ft)$. Since the receivers are dual polarized, they collect samples in orthogonal X and Y directions. The Fourier transform filters these samples into frequency channels. Then, for each frequency channel, and for each pair of receivers and for each possible combination of polarizations (so, XX, XY, YX, and YY), the samples are correlated and integrated over time. Specifically, the correlation of two samples is a complex multiplication where the sample of one receiver is multiplied by the complex conjugate of the other receiver, and integration is done by accumulating the result. Note that receivers are also paired with themselves (also called autocorrelation).

Pseudo-code of the correlation algorithm can be found in Listing 1. Note that this pseudo-code produces a square matrix. However, since a correlator computes correlations of pairs of receivers, the resulting matrix will be mirrored across the diagonal. Because of this, for

actual correlator algorithm implementations, an output triangle is considered. From the pseudo-code, it is visible how the different frequency channels, the different combinations of receivers and the different combinations of polarizations can all be processed independently.

```
for (channel = 0; channel < n_frequency_channels; channel++)
  for (receiver1 = 0; receiver1 < n_receivers; receiver1++)
    for (receiver2 = 0; receiver2 < n_receivers; receiver2++)
      for (polarization1 = 0; polarization1 < 2; polarization1++)
        for (polarization2 = 0; polarization2 < 2; polarization2++) {
          sum = 0 + i0;
          for (time = 0; time < n_times; time++) {
            sample1 = samples[channel][receiver1][polarization1][time];
            sample2 = samples[channel][receiver2][polarization2][time];
            sample2 = complex_conjugate(sample2);

            sum += sample1 * sample2;
          }
        correlation[channel][receiver1][receiver2][polarization1][polarization2] = sum;
        }
```

Listing 1: Correlation algorithm pseudo-code

One early approach of tackling the highly parallel task of correlating a large amount of receivers without the use of custom hardware is given by Romein et al. [18]. In this approach, a Blue Gene/L supercomputer is used to perform signal processing of the LOFAR radio telescope, including the correlation step. Blue Gene/L consists of many PowerPC 440 cores, each of which contains two 64-bit FPUs that can perform fused multiply-add instructions on complex numbers. Work is distributed by assigning each *subband* (a partition of the entire measured frequency spectrum) to a set of nodes. In each set, the sampled subband data is divided into separate seconds, and these are distributed to individual cores, which perform the entire signal processing pipeline for that second of data. The correlation step in this approach applies tiling on the output triangle: instead of correlating each pair of stations one-by-one, tiles of multiple stations are considered per iteration. This way, loaded samples are kept in registers and reused when computing each correlation in the tile. A tile size has to be chosen such that maximal reuse is possible, while still fitting in the register file.

Perhaps the first correlator kernel on the GPU is presented by Schaaf et al. [19], which implements a full FX correlator using the OpenGL graphics API on the NVIDIA 5900XT and NVIDIA 6800 Ultra GPUs. Harris et al. [11] and Wayth et al. [23] both present GPU implementations of just the correlation step, using CUDA, on the NVIDIA 8800 GTS and NVIDIA C1060 GPUs, respectively. Nieuwpoort and Romein [22] present correlator kernels on multiple different types of multi-core processors, including an implementation for the ATI 4870 GPU and the NVIDIA Tesla C1060. All of these implementations do not reach the peak performance of their respective GPU. Clark et al. [3] introduce the xGPU library for performing correlation on the GPU. Due to the combination of register tiling and shared memory tiling and the new Fermi microarchitecture, they achieve 79% of the peak performance of the NVIDIA GTX 480.

## 2.2 GEMM

GEMM is the following operation:

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

Where $\alpha$ and $\beta$ 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 $C$ and $D$ refer to the same matrix, meaning that the accumulation is done in-place. This definition of GEMM also allows for regular $A \cdot B$ matrix multiplication by setting $\alpha$ to 1 and $\beta$ to 0. A pseudo-code GEMM algorithm can be found in Listing 2. From this pseudo-code, the parallelism of GEMM becomes clear, as the different rows over $M$ and the columns over $N$ of the output matrix $D$ can be computed entirely independently. Both GEMM and correlation share a very similar inner-most loop that consists of many multiple-and-add operations. The most important distinctions between (complex) GEMM and correlation is the complex conjugate that correlation applies, and the triangle output considered by correlation, compared to the matrix output considered by GEMM.

Although it is trivial to divide rows and columns over different hardware cores/threads, special attention has to be given to data reuse in GEMM. Elements of the $D$ matrix that lie on the same row all require the same row from $A$ to be computed, and elements that lie on the same column of the $D$ matrix all require the same column from $B$ to be computed. Simply put, computing geometrically neighboring elements of $D$, usually formulated as two-dimensional *tiles*, results in reuse of data. Thus, both the tiling strategy as well as the memory architecture have to carefully considered when implementing GEMM algorithms.

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

        D[i][j] = alpha * sum + beta * C[i][j]
    }
```

Listing 2: GEMM algorithm pseudo-code

## 2.3 Volta/Turing and Ampere

### 2.3.1 Microarchitecture details

Tensor Cores were introduced in the Volta microarchitecture [4]. Tensor Cores are separate hardware cores present on each Streaming Multiprocessor (SM) that are capable of performing $D = A \cdot B + C$ matrix-multiply-accumulate (MMA) operations. Typically, $C$ and $D$ refer to the same matrix, in order to accumulate the result of multiple MMA operations. On Volta, Tensor Cores operate on $4 \times 4$ 16-bit floating-point (FP16) $A$ and $B$ matrices, and on 16- or 32-bit floating-point (FP32) $C$ and $D$ matrices, equivalent to a $4 \times 4 \times 4$ GEMM, albeit without the $\alpha$ and $\beta$ scaling factors. Large scale GEMMs can be performed by combining many of these smaller MMA operations.

On the Volta, Turing, and Ampere microarchitectures, Tensor Cores can be utilized by `mma` and `wmma.mma` instructions. These instructions are part of the Parallel Thread Execution (PTX) Instruction Set Architecture (ISA), a stable and machine-independent ISA available on NVIDIA GPUs. This means that the `mma` and `wmma.mma` PTX instructions can also be

used on later NVIDIA microarchitectures, and that the CUDA compiler may translate these instructions to different architecture instructions.

`mma` and `wmma.mma` are both instructions that operate on matrices larger than $4 \times 4 \times 4$. On the Volta microarchitecture, these instructions only operate on matrices of FP16 numbers, where `mma` performs a $8 \times 8 \times 4$-shaped MMA, and `wmma.mma` performs a $16 \times 16 \times 16$, $8 \times 32 \times 16$, or $32 \times 8 \times 16$-shaped MMA. The Turing microarchitecture introduces 8-, 4-, and 1-bit signed and unsigned integer support for Tensor Cores, as well as new shapes for the FP16 `mma`. The Ampere microarchitecture introduces Tensor Core support for double precision and for alternative floating-point precision formats in the form of `bf16` and `tf32`. Furthermore, the Ampere microarchitecture also introduces multiple new shapes for FP16 and integer `mma`, as well as support for sparse matrix fragments.

The Tensor Core instructions `mma` and `wmma.mma` are both instructions that operate across a *warp*, a group of 32 threads. This means that for an instruction to be executed, all threads in the warp must perform the same instruction. Each thread in the warp holds a small fragment of each of the matrices that are used for the MMA operation. Although the two instructions are very similar, only `wmma.mma` is exposed to C/C++ through the Warp Matrix Multiple Accumulate (WMMA) API. The WMMA API provides the `load_matrix_sync` and `store_matrix_sync` methods, which map to the `wmma.load` and `wmma.store` PTX instructions respectively. These operations are responsible for data movement between shared or global memory and registers, and ensures that the correct threads receive the correct fragments for the `wmma.mma` instruction (`mma_sync` in the WMMA API). The `mma` instruction does not offer specialized load and store instructions on the Volta microarchitecture. Instead, the exact mapping of matrix fragments to threads for `mma` is specified in the PTX documentation, which is not the case for `wmma.mma`.

The `mma` and `wmma.mma` instructions are both *synchronous*, meaning that the threads that execute the instruction are busy until the MMA is completed.

### 2.3.2 GEMM

GEMM implementations involving Tensor Cores almost always use tiling to divide the problem across different thread blocks, see Figure 1. Each thread block is then assigned one or more two-dimensional tiles of $\texttt{MMA_M} \times \texttt{MMA_N}$, unique partitions of the $C$ and $D$ matrices. Tile dimensions are chosen such that they comprise one or multiple MMA operations, as defined by the `mma` or `wmma.mma` dimensions. The thread blocks iterate over the $K$ dimension to repeatedly load tiles of shape $\texttt{MMA_M} \times \texttt{MMA_K}$ from the $A$ matrix and tiles of shape $\texttt{MMA_K} \times \texttt{MMA_N}$ from the $B$ matrix. These tiles are then multiplied and accumulated. The repeated loading and multiplication and accumulation of tiles over the $K$ dimension is referred to as the *mainloop*. When the mainloop has finished executing, the thread blocks load the corresponding $C$ tile. After applying the scale factors $\alpha$ and $\beta$ to the accumulation result and $C$ tile respectively, the two are added and stored to the $D$ matrix. The processing of the accumulation result and write-back to global memory is referred to as the *epilogue*.

One simple heuristic for maximizing GEMM performance is to keep the Tensor Cores as busy as possible. To this end, the kernel has to ensure that data is supplied to the Tensor Cores

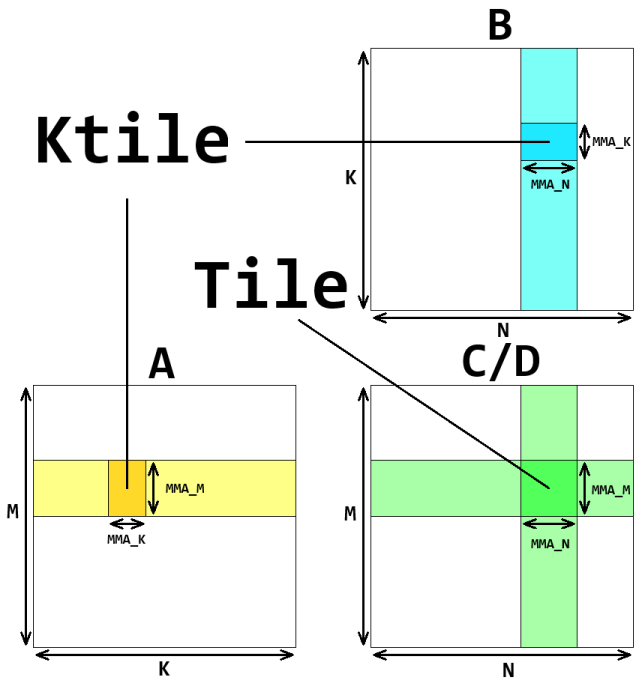



Figure 1: GEMM tiling

as fast as possible. In fact, Tensor Cores are so fast that GEMM kernels become memory-bound [26]. A common optimization is *software pipelining*, where data is prefetched one or more iterations before it is computed on, usually through the use of asynchronous copies introduced in Ampere. Asynchronous copies transfer data from global to shared memory and can be overlapped with the synchronous `mma` and `wmma.mma` instructions.

Furthermore, thread block and tiling dimensions have to be carefully chosen in order to maximize shared memory reuse and global memory bandwidth usage. Lastly, avoiding shared memory bank conflicts is critical for reaching maximal performance [13].

## 2.4 Hopper

### 2.4.1 Microarchitecture details

The Hopper microarchitecture introduces many new features, and two of them are key for performing efficient GEMMs on Hopper GPUs: fourth generation Tensor Cores, and the Tensor Memory Accelerator (TMA) [5].

The Hopper Tensor Cores do not operate on a warp level, but on a *warpgroup* level, a set of four contiguous warps. The Hopper Tensor Core operations are available through the `wgmma` instructions, which have to be executed by all threads in a warpgroup to function. The MMA is performed by executing the `wgmma.async` instruction, which, as the name implies, is *asynchronous*. This means that threads executing the `wgmma.async` instruction are free to resume execution before the Tensor Cores complete the MMA operation. The

completion of these asynchronous instructions can be tracked with the `wgmma.commit_group` and `wgmma.wait_group` instructions. Furthermore, Hopper Tensor Cores can only perform MMA operations of the form $D = A \cdot B + D$ or $D = A \cdot B$. The $B$ input to `wgmma.async` is sourced from shared memory, and the $A$ input may be sourced from either shared memory or from registers. The accumulation matrix $D$ is always sourced from registers. It is important to note that, unlike the `mma` and `wmma.mma` instructions, the `wgmma` instructions are available only on Hopper, and not on future microarchitectures.

Furthermore, the Hopper microarchitecture introduces support for 8-bit floating-point numbers (FP8) in Tensor Cores. There are two types of FP8 available: E4M3, which uses 1 sign bit, 4 exponent bits, and 3 mantissa bits, and E5M2, which uses 1 sign bit, 5 exponent bits, and 2 mantissa bits.

The TMA is a separate hardware unit that performs asynchronous data copies between shared and global memory. It is capable of copying large blocks of data, laid out in varying shapes and sizes, ranging from 1-dimensional to 5-dimensional tensors. TMA copies are issued by a single thread executing the `cp.async.bulk` instruction (for 1-dimensional contiguous arrays) or the `cp.async.bulk.tensor` instruction (for multi-dimensional arrays). The issuing thread only has to specify the starting address of the source and destination, and the amount of bytes to be transferred in the case of 1-dimensions arrays, and a copy descriptor in the case of multi-dimensional arrays. This copy descriptor is also called a *tensor map*, and describes the exact layout of the matrix to be copied. The TMA then takes care of the address and offset calculation involved in targeting the correct data to be copied. Before the Hopper microarchitecture, this computationally expensive task had to be performed by all individual threads. This functionality makes the TMA especially well suited for GEMM, which enjoys a considerable performance improvement as a result [12].

### 2.4.2 GEMM

The hardware features of the Hopper microarchitecture give rise to a new GPU programming technique: *warp specialization*. Typically, most GPU kernels are designed in such a way that all warps of a thread block perform the same computation, but on different data. This approach also holds for threads in a warp and for thread blocks in a grid. With warp specialization, warps are responsible for different tasks. On Hopper, the most common form of warp specialization is the *producer-consumer* model, where each thread block is divided into *producer* warps, which are responsible for preparing (producing) data, and *consumer* warps, which are responsible for computing on (consuming) the data [25].

Warp specialization is very advantageous for GEMM due to the asynchronous behavior of both the production method, which consists of performing asynchronous TMA copies in order to ready data from the $A$ and $B$ matrices in shared memory, and the consumption method, which consists of using the readied data in shared memory to perform asynchronous Tensor Core MMA operations. Without warp specialization, threads cannot wait for completion of both asynchronous methods at the same time, and run the risk that one asynchronous method has finished executing while threads are still waiting for the other method to complete. Although the latency hiding of warp specialization is similar to that of software pipelining,

warp specialization simplifies the work of the compiler, allowing it to generate an optimal instruction stream [1].

Typical Hopper GEMM kernels employ two warpgroups for performing the MMA operations, and one warpgroup for performing TMA operations [10]. These warps can communicate to each other through the `mbarrier` mechanism. These barriers are objects that reside in shared memory, which allows all warp(group)s in a thread block to communicate with each other by accessing the barrier object. Each `mbarrier` object maintains a phase, which advances upon completion. There are two ways to complete a phase of an `mbarrier`: performing a certain amount of *arrive-on* operations with `mbarrier.arrive`, or by specifying an amount of transaction bytes with `mbarrier.expect_tx` and having the barrier track asynchronous memory operations. Threads can check for phase completion with `mbarrier.test_wait`, which is a non-blocking operation that tests for completion, and with `mbarrier.try_wait`, which also tests for completion, but becomes a blocking operation if the phase has not yet completed, and calling threads resume execution only when the phase has completed. In case the `mbarrier` wait is carried out by a differing thread than the arrive-on, a parity phase bit is needed. This phase bit can either have an even phase of parity 0 or an odd phase of parity 1. This means that an `mbarrier.try_wait` operation with parity 0 will block on a newly initialized `mbarrier` until the phase of that barrier is completed (by another thread performing an arrive-on operation, for example). To wait for completion of the next phase, the phase bit must be flipped (to obtain parity 1), and it must be flipped again to wait for the phase after that, and so on.

Asynchronous TMA operations are tracked with the `mbarrier` mechanism. The producer warpgroup repeatedly performs TMA copies to shared memory buffers, so long as there are empty buffers to copy to. The consumer warpgroups wait on barriers specified by the TMA for completion, and initiate MMA operations on shared memory buffers as soon as they are full. As soon as an MMA operation is initiated, consumer warpgroups move on to the next full buffer, without waiting for the initiated MMA operation to finish. When an MMA has finished, the consumer warpgroups notify the producer warpgroups that the shared memory buffer used by the MMA operation is empty, after which the producer warpgroup can perform the next TMA copy on that buffer. After the mainloop finishes executing, the consumer warpgroup handles the epilogue. Since both the consumer and producer operate independently, they can target different buffers from each other at the same time, allowing the producer-consumer method to potentially a higher degree of pipelining than double-buffering.

Another possible technique is using *persistent kernels*, where thread blocks are launched to occupy as many SMs as possible and divide all $D$ tiles between them. This technique minimizes the overhead in launching thread blocks and the set-up that kernels have to perform before they can start the mainloop, sometimes referred to as the prologue. Processing multiple tiles per thread block also allows for some variations on the standard form of warp specializations. For instance, there may be separate producer warps responsible for loading the data needed for the epilogue. There can also be a tile scheduler warp, which is responsible for selecting the tiles that the other warps will work on.

## 2.5 Blackwell

### 2.5.1 Microarchitecture details

The Blackwell microarchitecture [6] brings two important features for GEMM kernels: Tensor Memory, and fifth generation Tensor Cores. Only a select amount of Blackwell GPUs have Tensor Memory: GPUs with compute capability 10.0 (NVIDIA (G)B200), 10.3 (NVIDIA (G)B300), and 11.0 (Jetson T4000 and T5000). Blackwell GPUs with a higher compute capability (e.g., 12.0) do not have Tensor Memory despite claiming to have fifth generation Tensor Cores.

Tensor Memory is separate on-chip memory that is present on each SM [7]. Tensor Memory is *word addressable*, meaning that storage units are 32 bits wide. Specifically, Tensor Memory has a two dimensional structure: it consists of 128 lanes, where each lane holds 512 columns of 32 bits. This means that on each SM, Tensor Memory can hold 256KB of data, making it have the same capacity as the register file. Tensor Memory must be dynamically allocated and deallocated with the `tcgen05.alloc` and `tcgen05.dealloc` instructions. Allocation happens column-wise, meaning that all lanes in a column are allocated. The number of allocated columns must be a power of 2 and in the range 32-512.

Tensor Memory can be accessed with the `tcgen05.ld` instruction, which loads Tensor Memory values into registers, with the `tcgen05.st` instruction, which stores register values to Tensor Memory, and with the `tcgen05.cp` instruction, which copies data from shared memory to Tensor Memory. Tensor Memory cannot be accessed without using these instructions, which must be executed by all threads in a warp. Certain access restrictions on `tcgen05.ld` and `tcgen05.st` exist: the rank of a warp within a warpgroup determines what Tensor Memory lanes it can access. An overview of Tensor Memory is given in Figure 2.

There are a handful of shapes that can be specified for `tcgen05.ld` and `tcgen05.st` which determine the data-to-thread mapping of one warp to the 32 lanes of Tensor Memory it can access. These shapes can be repeated along the columns such that one instruction performs multiple loads or stores. The standard `.32x32b` shape maps each thread to each lane with identical ID: thread 0 reads from lane 0, thread 1 reads from lane 1, etc. There are also the `.16x64b`, `.16x128b`, and `.16x256b` shapes which access only 16 lanes and have more complex data-to-thread mappings, and where the latter two shapes also read/write more than one value per thread.

The Tensor Memory is primarily used by the fifth generation Tensor Cores. Just like Hopper Tensor Cores, the Blackwell Tensor Cores compute $D = A \cdot B + D$ MMA, and matrices $A$ and $B$ are sourced from shared memory. However, the matrix $D$ resides in Tensor Memory, and matrix $A$ can optionally be sourced from Tensor Memory instead of shared memory. Since the large accumulation matrix $D$ is stored in Tensor Memory, the register pressure of the GEMM kernel is greatly reduced. Blackwell Tensor Core MMA operations are performed with the `tcgen05.mma` instruction, which is asynchronous and executed by only a single thread. Completion of asynchronous MMA operations can be tracked with `tcgen05.commit`, which makes a specified `mbarrier` responsible for tracking any previously issued `tcgen05.mma` instructions, similarly to how `mbarrier` objects track the completion of TMA copies. For




Figure 2: Overview of Tensor Memory

example, if four `tcgen05.mma` instructions are followed by one `tcgen05.commit` instruction, performing an `mbarrier.wait` on the barrier specified by the commit will only return when all four MMA operations have completed.

Blackwell Tensor Cores support all data types of previous Tensor Core generations with the exception of double precision floating-point, and introduce support for 6- and 4-bit floating-point numbers. Tensor Core MMA dimensions have expanded significantly on Blackwell. For dense FP8, $K$ is fixed at 32, $M$ can be chosen as either 64 or 128, and $N$ ranges from 8 all the way to 256, resulting in a largest possible MMA shape of $128 \times 256 \times 32$.

Furthermore, on Blackwell two SMs can form a pair and cooperate with each other when performing Tensor Core MMA operations. This is done by partitioning the MMA in the $M$ dimension, which can be two times as large in the case of two SMs. An overview of two SM MMA on Blackwell is given in Figure 3. Each SM holds a portion of the $A$, $B$, and $D$ matrices, and both portions of the $B$ matrix are used by the Tensor Cores of both SMs. Thread blocks must be launched in a cluster formation of at least two thread blocks large in order to make use of two SM Tensor Core MMA. Only one SM (the `leader SM`) in a pair needs to execute the `tcgen05.mma` instruction in order to exercise the Tensor Cores on both SMs.

### 2.5.2 GEMM

Typical Blackwell GEMMs still employ the same type of warp specialization introduced in the Hopper microarchitecture. However, since the Tensor Cores no longer require an entire warpgroup to perform the MMA instructions and hold partitions of the accumulation tile in their registers, those threads are now free to perform other work. The mainloop is handled

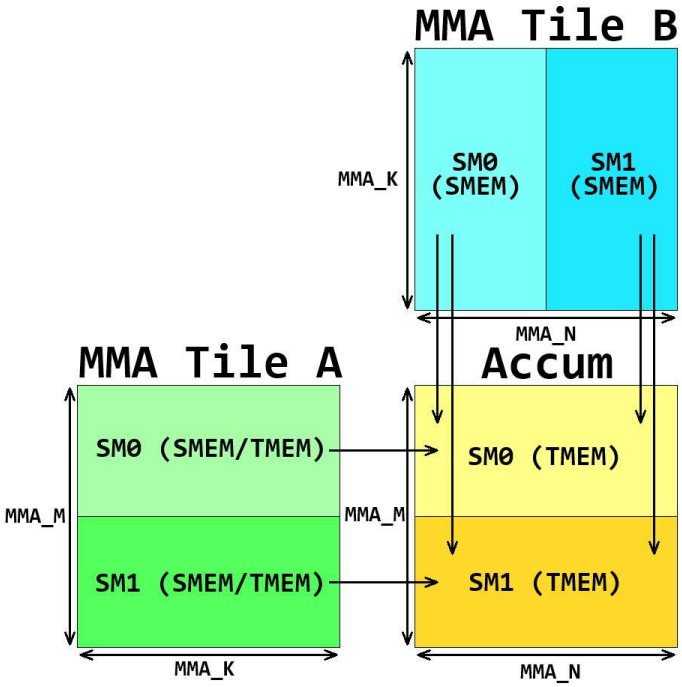



Figure 3: Blackwell Tensor Core MMA with two SMs

by only two warps that each elect leader threads: one producer warp that performs TMA copies from global to shared memory, and one consumer warp that performs MMA operations by calling `tcgen05.mma`. The epilogue is then handled by entirely separate warps, usually four warps in total, when keeping the Tensor Memory access restrictions in mind. This is possible due to the fact that the accumulation result lies in Tensor Memory, which permits threads to access the accumulation result without participating in the MMA operations. This property is the biggest factor that differentiates GEMM kernels on Blackwell from previous generations. Blackwell GEMMs are usually persistent kernels such that the mainloop of one $D$ tile can fully overlap with the epilogue of another tile.

## 2.6 The Tensor-Core Correlator

The Tensor-Core Correlator [17] is perhaps the first GPU correlator to use Tensor Cores. Correlation can, just like GEMM, be reduced to MMA operations suitable for Tensor Cores by using appropriate tiling of the output data structure. However, Tensor Cores do not support complex numbers, and the output in correlation takes the shape of a triangle, while Tensor Cores operate on matrices.

### 2.6.1 Complex numbers support

Multiplication of two complex numbers $a + bi$ and $c + di$ is defined as:

$$(a + bi) \cdot (c + di) = ac - bd + (ad + bc)i \tag{2}$$

Complex numbers and complex multiplication is supported on the CPU through the C++ `<complex>` library. Newer versions of CUDA have an analogous `<cuda/std/complex>` library. Without this library, complex number support can be emulated relatively easily on the regular GPU cores by using built-in vector types (`float2`, `double2`, etc.) to represent the real and imaginary parts of a complex number and by writing custom functions that multiply two vector types according to Equation 2. Calculating the real part consists of two multiplications ($a \cdot c$ and $b \cdot d$), one subtraction, and one addition to accumulate the result. Calculating the imaginary part consists of two multiplications ($a \cdot d$ and $b \cdot c$) and two additions. Special attention has to be given to the subtraction, which presents a problem for Tensor Cores as they can exclusively perform multiply-and-add operations.

The Tensor-Core Beamformer [14] approaches this problem by first separating the real and imaginary parts of the *interleaved* complex numbers (i.e., the real and imaginary component of a complex number lie next to each other in memory) out into separate real-valued and imaginary-valued matrices (also called a *planar* representation of complex numbers). Thus, for input matrices $A$ and $B$ and output matrix $C$, it uses 6 matrices: $A_{Re}$, $A_{Im}$, $B_{Re}$, $I_{Im}$, $C_{Re}$, and $C_{Im}$, where the subscripts $Re$ and $Im$ represent the real and imaginary parts of the matrix, respectively. Complex matrix multiplication is then done in 5 steps:

1) $C_{Re}$ += $A_{Re} \cdot B_{Re}$
2) $C_{Im}$ += $A_{Re} \cdot B_{Im}$
3) $B_{Im} = -B_{Im}$
4) $C_{Re}$ += $A_{Im} \cdot B_{Im}$
5) $C_{Im}$ += $A_{Im} \cdot B_{Re}$

Step 3 is performed in registers, and all other steps are performed by Tensor Core MMA operations. This approach works well when using `mma` or `wmma.mma` instructions, but poses a problem for Blackwell due to the asynchronicity of `tcgen05` instructions and because the $B$ matrix is not sourced from registers.

The Tensor-Core Correlator uses a different approach to work around the absence of subtraction operations in Tensor Cores. Namely, it embeds the negative values into the matrices itself. Each column in $B$ is duplicated: the duplicated column will have the real and imaginary parts of each complex number swapped, and the values in the original column are complex conjugated, see Figure 4. From the figure we see that, for two complex numbers $a+bi$ and $c+di$, embedded in the aforementioned way in the $A$ and $B$ matrix respectively in one K iteration, the desired outcome is obtained in the $C$ matrix, as it matches Equation 2.

It has to be noted that this is only complex matrix multiplication, not correlation. In correlation, the first sample is multiplied by the complex conjugate of the second sample. Thus, when correlating with this method, the original columns in B remain unchanged, and the duplicated columns contain values that are both swapped and complex conjugated.

This method embeds the transformation (i.e., either the conjugate in the original column and swap in the duplicated column for complex matrix multiplication, or the conjugate and swap in the duplicated column for correlation) in an interleaved manner, as the duplicated columns lie next to the original columns, and the MMA result also contains interleaved complex numbers. It is also possible to embed this transformation in a planar manner, see

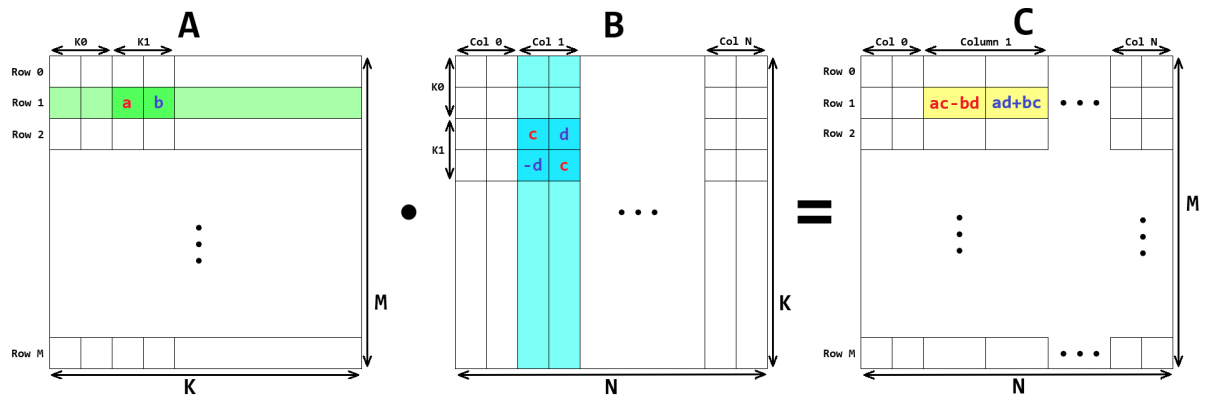



Figure 4: Complex GEMM with the interleaved method: embedding the complex transformation in adjacent columns of the B matrix. Multiply-add is shown for only one K iteration. Red numbers are real components, blue numbers are imaginary components

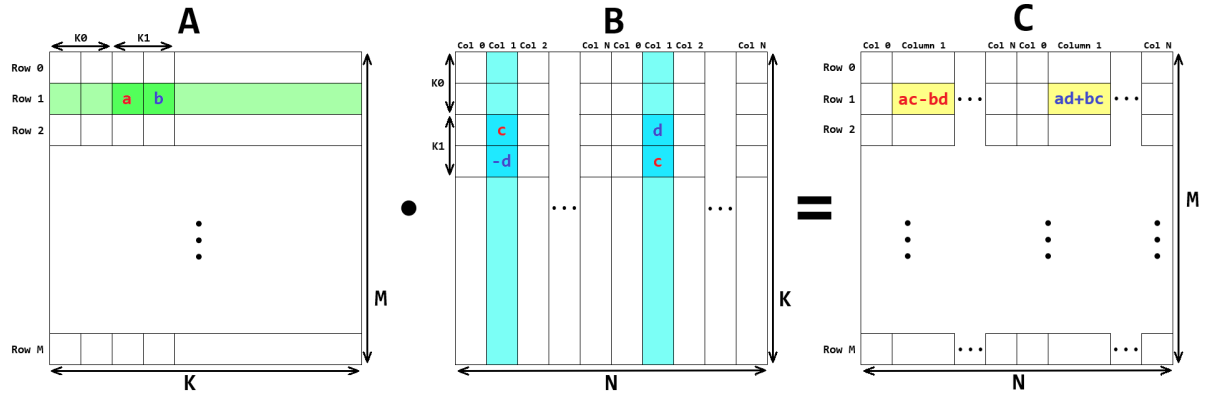



Figure 5: Complex GEMM with the planar method: embedding the complex transformation in separated columns of the B matrix

Figure 5. The MMA result with the planar method results in an output matrix in planar format, which has to be taken into account during the epilogue phase. We distinguish between these two transformation methods since they might offer different advantages and drawbacks, but they ultimately compute the same result.

Another approach to perform complex matrix multiplication with Tensor Cores is by using emulation [20]. The Ozaki-I scheme was originally developed for emulating high-precision matrix multiplication by using better performing low-precision Tensor Core matrix multiplication. The Ozaki-II scheme [15] improves the performance of this scheme and can be combined with the Karatsuba method to efficiently emulate complex arithmetic.

The Tensor-Core Correlator was originally designed for Volta Tensor Cores and uses the `mma` instructions. Since these instructions source the input matrices from registers, perform the complex conjugate and swap operations on the columns of $B$ is fast and efficient. Therefore,

only a small amount of overhead is incurred when performing Tensor Core MMA operations. This is not the case on the Blackwell microarchitecture, where the input matrices are sourced from shared memory. Furthermore, the Tensor-Core Correlator uses the software pipelining approach described in Section 2.3.2, which, at least for GEMM kernels, is insufficient to keep the asynchronous Blackwell Tensor Cores as busy as possible.

## 2.7 CuTe

One major challenge in implementing a GEMM kernel on Blackwell is the programmability of Blackwell Tenor Cores. The corresponding `tcgen05` instructions are available only through inline assembly code in CUDA programs, as the WMMA interface only exposes the `wmma.mma` instructions in a programmer-friendly way. Therefore, we choose to use the CuTe C++ library, part of the broader CUTLASS[1] project. CUTLASS is a C++ library developed by NVIDIA that implements fast and efficient GEMM kernels in CUDA. By making extensive use of template abstractions, users can create custom kernels out of hierarchical components. CuTe sits at the bottom of the hierarchy, and is responsible for the lowest-level of data decomposition. Specifically, it defines layouts of data and how threads move or compute on that data. Both CUTLASS and CuTe are header-only libraries that can easily be integrated in a CUDA program.

The most important concepts in CuTe are the `Tensor` and `Layout` object. The `Layout` specifies the `Shape` of a data object and its `Stride`, both of which can be multi-dimensional and hierarchically structured. For example, simple two-dimensional $(M, N)$ shapes are possible, but also more complex shapes like $(M, (N, K), L)$ are possible. The `Stride` describes the index to coordinate mapping of the data object. `Tensor` objects combine a `Layout` with backing data storage, which can take the form of registers, global memory, shared memory, or Tensor Memory. This allows `Tensor` objects to fully represent multi-dimensional arrays, which are present in every stage of GEMM decomposition. Furthermore, CuTe defines swizzle functors to create swizzled layouts, which are vital to avoid shared memory bank conflicts [16]. Even though the TMA is capable of copying non-swizzled data in global memory to shared memory in a swizzled pattern, and that the Blackwell Tensor Cores can operate directly on swizzled layouts, CuTe can be very helpful in abstracting away the complexities of swizzling when data in shared memory has to be accessed in between the TMA and Tensor Cores.

Besides the `Tensor` and `Layout` concepts, CuTe also provides the `cute::gemm` and `cute::copy` functions. When used together with CuTe's `TiledMMA` concept, `gemm` can wrap around the `tcgen05.mma` PTX instruction and abstract away its complexities such as instruction and matrix descriptors, allowing the programmer easy access to Tensor Core MMA operations. Likewise, the `copy` function, when used with some helper functions, can use the TMA and abstract away its complexities, such as creating the tensor map.

Finally, many CuTe objects and functions use `constexpr` specifiers such that they can be resolved at compile time. This also means that most of the thread to data mapping can be done at compile time, leaving minimal work for the threads to perform at runtime.

[1] https://github.com/nvidia/cutlass

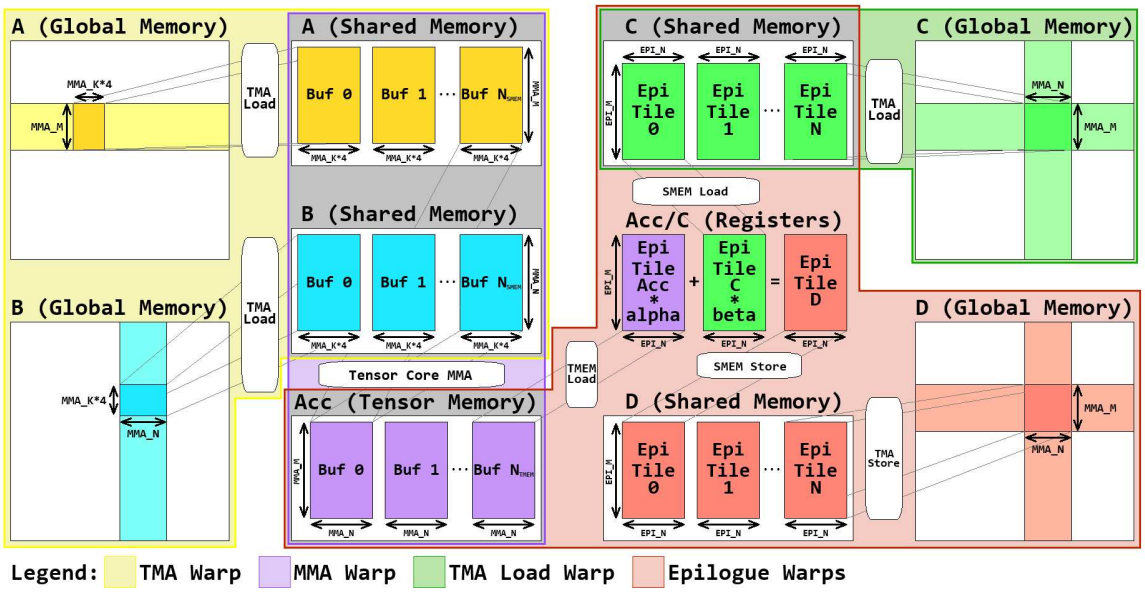



Figure 6: An overview of the normal GEMM kernel. The TMA warp loads $A$ and $B$ tiles from global to shared memory, the MMA warp consumes tiles in shared memory and multiplies-and-accumulates in Tensor Memory, the TMA load warp loads $C$ tiles from global to shared memory, and the epilogue warps consume accumulator tiles in Tensor Memory and $C$ tiles in shared memory to produce $D$ tiles, which it then stores to global memory.

# 3 Methodology

In Section 3.1, a complete description of a performant non-complex GEMM kernel is given, including an overview of the mainloop and epilogue and the warp specialization. In Section 3.2, multiple different variants of complex GEMM that extend the normal GEMM kernel are given.

## 3.1 Normal GEMM

A complete overview of warp-level data movement and computation of the normal GEMM kernel is given in Figure 6.

### 3.1.1 Host side setup

Some setup is required in order to construct the appropriate CuTe objects that are required to call the `gemm` and `copy` functions, which exercise the Tensor Cores and TMA, respectively. Due to the compile time nature of CuTe objects, it is not strictly necessary to perform these on the host, but is more convenient. A `TiledMMA` object is required to perform `tcgen05.mma` operations, and it can be constructed by specifying the data types of the $A$, $B$, and $C$ matrices, the Tensor Core MMA dimensions, the major dimension of the $A$ and $B$ matrices, whether to source the $A$ Tensor Core input from shared memory or Tensor Memory, and whether to use one or two SMs per MMA. For now, we restrict data types to use FP8 for both

the $A$ and $B$ matrices, and to use 32-bit floating-point for accumulation and the $C$ and $D$ matrices. We also use two SMs per MMA such that a larger MMA tile can be chosen, allowing for more reuse of data and higher performance. With two SMs, the chosen MMA dimensions are $256 \times 128 \times 32$ (i.e., selecting the largest possible `MMA_M` and second largest `MMA_N`, where `MMA_K` is fixed at 32 for FP8). An `MMA_M` of 256 maximizes Tensor Memory throughput, as each SM in the pair uses all $256/2 = 128$ lanes. An `MMA_N` of 256 would maximize data reuse for one tile, but severely limit the number of shared memory buffers, overall negatively impacting performance. The GEMM kernel assumes a transposed $B$ matrix, corresponding to the "K-Major" mode in `tcgen05` instructions. Finally, both $A$ and $B$ are sourced from shared memory, as the Tensor Cores can directly consume the TMA copy results.

We consider a persistent kernel that launches as many thread blocks as SMs. These thread blocks are launched in a thread block cluster of two thread blocks each, such that each thread block is paired with another one. The output $D$ matrix is divided into `MMA_M` $\times$ `MMA_N` tiles, which are statically distributed across the thread block pairs. The $A$, $B$, $C$, and $D$ matrices, as well as the various shared memory buffers that hold tiles of these matrices, are represented using `Tensor` objects, whose `Layout`s are defined on the host and passed as parameters to the kernel.

The most important part of the host side setup is the creation of the tensor maps that are required for performing TMA operations, which is also the only part that is required to do on the host side. CuTe offers methods that completely hide the managing of tensor maps, abstracting them away under `Copy_Atom` objects. Using CuTe, these objects are created for performing TMA loads on the $A$, $B$, and $C$ matrices, and TMA stores on the $D$ matrix.

### 3.1.2 Prologue

Before the mainloop can begin, some initialization is required. First, all the `mbarrier` objects used in the kernel have to be initialized. Tensor Memory has to be explicitly allocated. This is done by a single warp, which simply allocates all of the available Tensor Memory as only one thread block will be scheduled per SM. Thread block cluster pairs have to synchronize in order to ensure that both thread blocks can communicate to each other through `mbarrier` objects.

After synchronization, different warps assume their specialization. The kernel uses four different specializations:

1. One TMA warp, which issues global-to-shared memory TMA copies on the $A$ and $B$ matrices
2. One MMA warp, which issues Tensor Core operations that consume data from shared memory and produce accumulation results in Tensor Memory
3. One epilogue load warp, which issues global-to-shared memory TMA copies on the $C$ matrix
4. Four epilogue warps, which consume accumulation results in Tensor Memory and $C$ matrix tiles in shared memory to compute the epilogue and store the result to the $D$ matrix with shared-to-global memory TMA copies

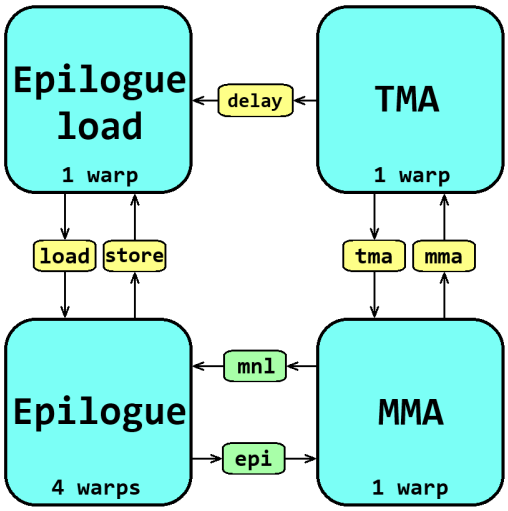



Figure 7: Warp specialization and associated barrier objects for normal GEMM. Outgoing arrows indicate performing arrive on operations, ingoing arrows indicate wait operations. Green colored barriers preside over Tensor Memory, yellow colored barriers preside over shared memory.

An overview of the different warps and the `mbarrier` objects they use to communicate with each other is given in Figure 7. After being divided into different specializations, each warp begins iterating over the same set of statically scheduled $D$ tiles.

### 3.1.3 Mainloop

The mainloop comprises two warps: the MMA warp and the TMA warp. The two warps preside over shared memory buffers that hold the $A$ and $B$ MMA tiles, which are used as inputs to Tensor Core MMA operations.

MMA tiles are themselves organized in *K-tiles*. This is because the TMA support multiple swizzling modes: 32B, 64B, and 128B, where the largest swizzling mode performs the best. To use the 128B swizzling mode, the TMA has to perform 128-byte wide copies. However, `tcgen05.mma` instructions have a fixed $K$ dimension of 32 bytes. Thus, for both $A$ and $B$ (since $B$ is transposed), each K-tile consists of four 32-byte wide MMA tiles repeated in the $K$ dimension, such that each K-tile is 128-bytes wide and is allowed to use the 128B swizzling mode in the TMA. Both the TMA warp and MMA warp iterate over the $K$ dimension of $A$ and $B$ in the form of K-tiles. The TMA warp copies entire K-tiles to shared memory, and the MMA warp views these K-tiles as four subsequent MMA tiles. Thus, for $A$, K-tiles are shaped $\texttt{MMA_M} \times (\texttt{MMA_K} \cdot 4)$ and MMA tiles are shaped $\texttt{MMA_M} \times \texttt{MMA_K}$, and, for $B$, K-tiles are shaped $\texttt{MMA_N} \times (\texttt{MMA_K} \cdot 4)$ and MMA tiles are shaped $\texttt{MMA_N} \times \texttt{MMA_K}$.

The K-tiles for $A$ and $B$ are buffered in shared memory such that K-tiles from multiple iterations can be held at the same time, see Figure 8. K-tiles are buffered as much as the

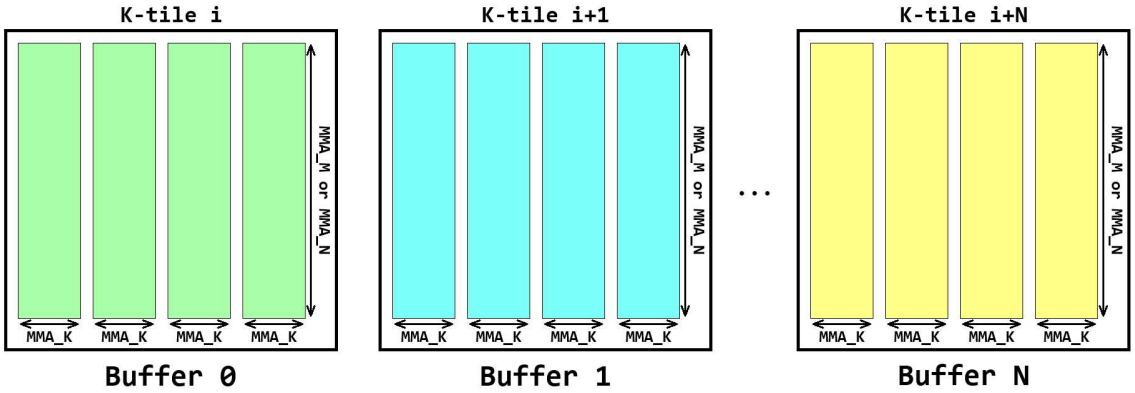



Figure 8: $N$ shared memory buffers for holding $A$ or $B$ K-tiles from iteration $i$ to iteration $i + N$. Each K-tile consists of 4 MMA tiles.

shared memory capacity allows, while keeping in mind that the epilogue also reserves some shared memory. The MMA and TMA warps communicate to each other through the `tma_bar` and `mma_bar` `mbarrier` objects, in order to ensure a correct ordering of shared memory accesses. There is one `tma_bar` barrier and one `mma_bar` for each time $A$ and $B$ are buffered.

The TMA warp iterates over the $K$ dimension, and in each iteration uses the TMA to copy a K-tile from $A$ and $B$ from global memory to shared memory. In each iteration, it first selects a shared memory buffer to copy to in a circular buffer fashion. In order to ensure that the selected shared memory buffer is not currently being used in an MMA operation (and therefore is ready to receive data again), the TMA warp calls `cute::wait_barrier` (which wraps the `mbarrier.try_wait` PTX instruction) on the corresponding `mma_bar` barrier. For the first tile that an SM will process, all the shared memory barriers are by definition empty. Because of this, the parity phase bit used to wait on the `mma_bar` barriers will start flipped, in order to skip the first wait and not deadlock. Once the TMA warp is sure that a shared memory buffer is empty, the appropriate amount of transaction bytes is set on the corresponding `tma_bar` barrier, and `cute::copy` is called to perform TMA copies for $A$ and $B$, using the same `tma_bar` to track the completion of the asynchronous operation. While the TMA copies are still in-flight, the TMA warp moves on to the next K-tile to repeat the process, until all K-tile copies have been issued.

On Blackwell, the `cp.async.bulk.tensor` PTX instruction (which `cute::copy` wraps to perform TMA copies) can specify an `mbarrier` object on a different SM. This feature allows the leader SM in a cluster pair to track the completion of TMA operations issued by both SMs in the cluster pair. Specifically for this kernel, the two TMA warps in a cluster pair both specify the `tma_bar` on the leader SM when performing TMA copies, such that the MMA warp on the leader SM is aware when all of the data required for a two SM MMA operation is ready.

Likewise, the MMA warp also iterates over K-tiles in the $K$ dimension. However, since the Tensor Core MMA operations only have to be performed by one SM, an MMA warp not on the leader SM returns immediately. An MMA warp that is on the leader SM selects a

shared memory buffer each iteration in a circular buffer fashion. In order to ensure that the selected shared memory buffer contains loaded K-tiles from $A$ and $B$, the MMA warp waits on the corresponding `tma_bar`. Once the wait is completed, the MMA warp iterates over the four MMA tiles that make up a K-tile, and calls `cute::gemm` on each MMA tile. Then, `tcgen05.commit` is executed such that the issued Tensor Core MMA operations are tracked by the `mma_bar` barrier that corresponds to the selected shared memory buffer. After committing the issued MMAs, the MMA warp moves on to the next shared memory buffer and repeats the process until the MMAs issued on the last K-tile have been committed.

### 3.1.4 Mainloop-Epilogue pipelining

Since the kernel is persistent and each thread block computes multiple tiles over its lifetime, there exists the opportunity to overlap the epilogue of one tile with the mainloop of the next tile. This is exploited by allocating multiple Tensor Memory buffers, which hold the mainloop accumulation result of multiple tiles. For each accumulation result buffer, there is one `epi_bar` and one `mnl_bar`. For MMA operations with shape $256 \times 128$ and two SMs, the mainloop accumulation result can be buffered in Tensor Memory four times. The MMA warp commits the last asynchronous `tcgen05.mma` operations per output tile to both the corresponding `mnl_bar` (the Tensor Memory accumulation result buffer produced by the MMA) and to the corresponding `mma_bar` (the shared memory $A/B$ K-tile buffers consumed by the MMA).

Before starting to execute the MMA loop that iterates over all K-tiles, MMA warps wait on the `epi_bar` corresponding to the selected Tensor Memory buffer that will hold the accumulation result of the current tile. Like how the TMA warp waits on `mma_bar`, the parity phase bit used in the wait on `epi_bar` is initialized to 1, in order to skip the wait in the first iteration and not deadlock.

Additionally, for the very first tile that each thread block computes, the epilogue load warp is intentionally delayed using the `delay_bar` barrier. This is done to guarantee the TMA warp goes first, ensuring that the Tensor Cores do not idle while the TMA load from $C$ blocks TMA loads from $A$ and $B$. Specifically, for the first tile, the TMA warp issues TMA copies on all shared memory buffers for $A$ and $B$, and then performs an arrive-on operation on `delay_bar`, which signals the epilogue load warp that it may start issuing TMA copies.

### 3.1.5 Epilogue

On each thread block, one warp will be responsible for epilogue loads (the *epilogue load warp*), and four warps will be responsible for computing the epilogue and storing the result (the *epilogue warps*). These two warp types preside over shared memory buffers that are dual-use: they are used both for holding loads from the $C$ matrix, and for holding output data that will be stored to the $D$ matrix. These $C/D$ tiles are themselves further divided into subtiles, which is done for two reasons. The first reason is that loading the large $C$ tiles and accumulation results at once into registers may exceed the per-thread register limit, causing some values to be offloaded to local memory (thread-local global memory), significantly decreasing performance. The second reason is that, since the large $C/D$ tiles put a heavy

strain on shared memory usage, typically only one shared memory buffer is allocated to hold a $C/D$ tile. By dividing it into subtiles, the epilogue load warp and the epilogue warps can overlap their functionality on a single $C/D$ tile buffer by targeting different subtiles.

Subtiles are tiled horizontally (in other words, only over the $N$ dimension of $C/D$ tiles), as vertical tiling would result in uneven work distribution due to Tensor Memory access restrictions. Each subtile of each $C/D$ tile has one `store_bar` barrier and one `load_bar` barrier to synchronize shared memory accesses between the epilogue load warp and the epilogue warps.

The epilogue load warp functions very similarly to the TMA warp. For the first tile in each thread block, a wait on `delay_bar` is performed. Afterwards, the epilogue load warp iterates over all the subtiles of the $C$ tile, and in each iteration first waits on the corresponding `store_bar` (again with a parity phase bit initialized to 1, as the shared memory buffers for $C$ are by definition empty when the kernel starts). When the wait is complete, the appropriate amount of transaction bytes are set on the corresponding `load_bar` barrier, which is then used to track the asynchronous TMA copy started by `cute::copy`.

The epilogue warps will be responsible for computing the epilogue and storing the result to the $D$ matrix. Since these warps have to perform Tensor Memory loads, four warps per thread block are required to access the entire Tensor Memory. Before the epilogue warps can begin processing the accumulation result of a given tile, they must first wait on the corresponding `mnl_bar`. Afterwards, the epilogue warps iterate over all the subtiles of the current tile. For each subtile, the epilogue warps first load the subtile of Tensor Memory to registers. For the last subtile, an arrive-on is performed on the corresponding `epi_bar` to signal that the Tensor Memory buffer is empty and ready to be used again by the mainloop. After the Tensor Memory load finishes, the epilogue warps can then wait on the `load_bar` that corresponds to the subtile. When the wait completes, the epilogue warps load the $C$ subtile from shared memory to registers. With a subtile from both the accumulation result and the $C$ tile in registers, the epilogue warps can finally compute the epilogue by scaling the accumulation result with alpha and the $C$ tile with beta, and adding both together. The result is stored back to the same shared memory subtile buffer that holds the $C$ tile.

Once the epilogue result has been stored to shared memory, it can be stored to global memory using the TMA. Participating threads must execute `fence.proxy.async` to establish an ordering of the synchronous shared memory writes and the subsequent asynchronous TMA operations. TMA stores are also asynchronous, and can unfortunately not be tracked using the `mbarrier` mechanism. Checking for completion can only be done by committing issued TMA stores with the `cp.async.bulk.commit_group` PTX instruction, and by waiting with the `cp.async.bulk.wait_group` PTX instruction, which blocks the executing thread until all but a specified number of committed TMA stores have completed. Thus, after the epilogue result for a given subtile has been computed and stored to shared memory, the epilogue warps elect one thread to perform TMA stores with `cute::copy` and commit them. Before moving on the next subtile, the elected thread uses `cp.async.bulk.wait_group` to allow only a specified amount of TMA stores to be in-flight, waiting for the rest. Provided that the wait is skipped an amount of times that is equal to the number of allowed in-flight TMA stores, we know that when a wait returns, a TMA store has completed. Thus, whenever the elected

| Variant | Specialized transformation warps | Transformed input matrix | $A$ matrix sourced from | Separate $A/B$ TMA | Interleaved method supported | Planar method supported |
|---|---|---|---|---|---|---|
| 1 | ✗ | $B$ | SMEM | ✗ | ✓ | ✓ |
| 2 | ✓ | $B$ | SMEM | ✗ | ✓ | ✓ |
| 3 | ✓ | $A$ | SMEM | ✗ | ✗ | ✓ |
| 4 | ✓ | $A$ | TMEM | ✗ | ✗ | ✓ |
| 5 | ✓ | $A$ | TMEM | ✓ | ✗ | ✓ |

Table 1: Overview of different complex GEMM variants

thread is done waiting, it performs an arrive-on operation on the `store_bar` corresponding to the stored subtile.

## 3.2 Complex GEMM

We developed several different variants that extend the normal GEMM kernel, which all perform complex GEMM/correlation instead of a regular GEMM. These variants differ mainly in the matrix where the transformation is performed and if that matrix is sourced from shared memory or Tensor Memory, but they also implement different optimizations. An overview of the different complex GEMM variants is given in Table 1. All kernel variants work with real-valued numbers (instead of working with `<complex>` numbers) and implicitly view two adjacent numbers as the real and imaginary parts of one complex number. Thus, the $N$ dimension of the $C/D$ matrices (in terms of real-valued numbers) is two times as large as the $N$ dimension of the $B$ matrix. Since all kernel variants perform complex matrix multiplication/correlation through interleaved or planar transformations of the $A$ or $B$ matrix, each MMA produces tiles of shape $\texttt{MMA_M} \times (\texttt{MMA_N} \cdot 2)$ out of $\texttt{MMA_M} \times \texttt{MMA_K}$ shaped loads from the $A$ matrix and $\texttt{MMA_K} \times \texttt{MMA_N}$ shaped loads from the $B$ matrix.

Complex GEMM is computed by performing either *complex transformations*, and correlation is computed by performing *correlation transformations*, see Figure 9. Both transformations take one complex value and produce two, which corresponds to the doubling of rows in $A$ or columns in $B$. The complex transformation complex conjugates the input to produce the first output value, and swaps the real and imaginary parts of the input to produce the second output value. The correlation transformation copies the input without changes to produce the first output value, and complex conjugates and swaps the input to produce the second output value.

### 3.2.1 Variant 1

The first variant represents the most naive approach to performing either complex matrix multiplication or correlation, where the transformation is performed right before the Tensor Core MMA operations, like in the Tensor Core Correlator. Instead of using one warp on one SM in a cluster pair for performing the MMA, both SMs use four warps for performing the transformation and the MMA. Since both SMs in a cluster pair now have MMA warps (instead of only the leader SM having an MMA warp), the TMA warps in a cluster pair have

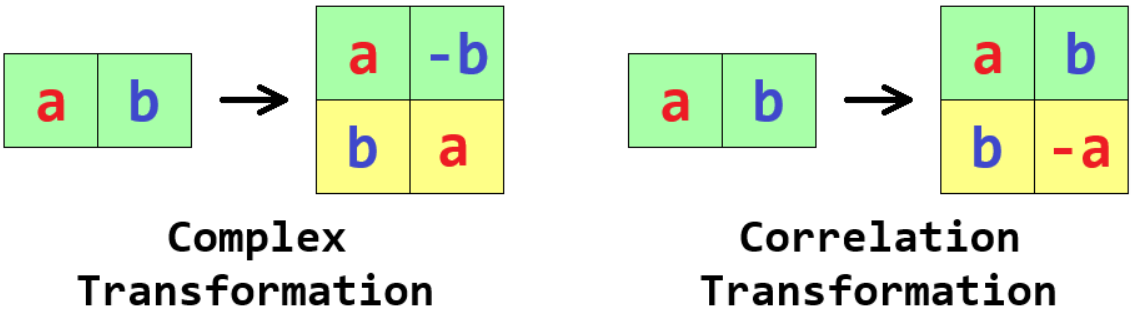



Figure 9: Complex and correlation transformation of one complex value

their respective MMA warp track TMA copies, instead of having the leader SM track the TMA copies of both TMA warps. Like in the normal GEMM kernel, the MMA tiles that make up the K-tiles of both $A$ and $B$ are sourced from shared memory.

The complex/correlation transformations are applied in K-tiles from the $B$ matrix. Since the four subsequent Tensor Core MMA operations (one for each MMA tile in a K-tile) consume an entire $\texttt{MMA_N} \times (\texttt{MMA_K} \cdot 4)$-shaped post-transformation K-tile of $B$, the TMA warp performs $(\texttt{MMA_N}/2) \times (\texttt{MMA_K} \cdot 4)$-shaped TMA copies. Thus, each shared memory buffer for holding K-tiles of $B$ can be seen as two halves, where the top half will contain TMA copy results.

Like in the normal GEMM kernel, the MMA warps continuously iterate over K-tiles of $A$ and $B$ and wait on `tma_bar` barriers for shared memory buffers to be filled by the TMA warp. When the buffers (or in the case of $B$, the top-half of the buffer) are filled, the MMA warps load values from shared memory to registers, such that each of the 128 threads holds a small fragment of the K-tile of $B$. Such a fragment corresponds to (part of) an entire row of the loaded $B$ K-tile. With the values in registers, the transformation can be performed. At this point, the type of transformation (complex multiplication or correlation) and the method (interleaved or complex) determine what the threads do next.

If using the interleaved transformation method, each thread must always write the transformed data to two adjacent rows (not columns, since $B$ is transposed) of shared memory, and thus allocates an additional fragment of registers for that second row. For the complex transformation, the loaded values are first complex conjugated in the first fragment, and then swapped in the second fragment. For the correlation transformation, the loaded values are complex conjugated and swapped in the second fragment. After performing the transformation, all threads in the four MMA warps synchronize (since they may overwrite shared memory locations that other threads are reading from) and write the transformed values back to the same shared memory buffer where the values were loaded from. Figure 10 shows the interleaved transformation method, using the complex transformation.

Since we work with 8-bit floating-point numbers in 32-bit registers, we transform four values at a time by implicitly packing two complex values with an 8-bit real part and 8-bit imaginary

part in one 32-bit integer. The complex conjugate can be computed efficiently with a bit mask that flips the sign bit on each of the two 8-bit floating-points packed in one 32-bit integer. The swap is performed with the `__byte_perm` intrinsic, which allows individual bytes of a 32-bit integer to be reordered.

If using the planar method, the complex/correlation transformation happens in the same manner as with the interleaved method. However, we can use the fact that if only one row is transformed, which is the case for the correlation transformation, the top-half of the shared memory buffer for $B$ does not need to be changed. Because of this, only one fragment of registers needs to be allocated and stored back to shared memory, resulting in half as many shared memory stores when compared to interleaved correlation. After performing either the complex or correlation transformation, the threads can write the transformed values back to shared memory without synchronizing, as threads can only write to shared memory addresses that they read from themselves. Figure 11 shows the planar transformation method using correlation transformation.

Unlike with the interleaved method, the two halves of a K-tile for $B$ are multiplied separately by $A$ when using the planar method. This is done with two separate `cute::gemm` calls and in two different accumulator tiles in Tensor Memory. This is because in two SM MMA each SM in a cluster pair holds one half of the $B$ MMA tile across the $N$ dimension (see Figure 3). Thus, if each of those two halves are further divided into two halves, and then multiplied as one whole MMA tile, the resulting accumulation matrix in Tensor Memory would be divided into four parts: two separated real parts and two separated imaginary parts. By multiplying the two halves of the $B$ MMA tiles separately, all real parts and all imaginary parts can be accumulated in two separate Tensor Memory buffers, even in two SM MMA.

After the transformation has completed and the warps have synchronized, all MMA warp threads must execute the `fence.proxy.async` instruction to order the synchronous shared memory reads/writes and the asynchronous MMA operations. After fencing and synchronizing warps, both SMs in a cluster pair must synchronize, since the Tensor Core MMA operations must still be executed by one SM. Synchronization between SMs is done with the `mbarrier` mechanism, since both SMs in a cluster pair are able to access each others shared memory. The leader SM waits for the other SM to perform an arrive-on on the `tsf_bar mbarrier`. The other SM then waits on the `tsf_bar` in its own shared memory. When the leader SM observes the arrive-on, it means that both SMs have finished the transformation, at which point the leader SM elects one warp to call `cute::gemm`. When the leader SM has initiated all MMA operations for the current K-tile, it performs an arrive-on on the `tsf_bar` of the other SM, such that the MMA warps on both SMs can continue executing and move on to the next K-tile while the Tensor Cores are busy.

In the planar method, the epilogue warps have to account for the two separate real and imaginary tiles by loading from both Tensor Memory buffers for each subtile iteration. Since the Tensor Memory loads happen through registers, the planar format can be implicitly converted to an interleaved format by carefully selecting which registers are used to load from the real valued accumulator tile and which registers are used to load from the imaginary valued accumulator tile. Thus, after performing Tensor Memory loads in both the interleaved and planar methods, the epilogue warps contain an interleaved representation of complex

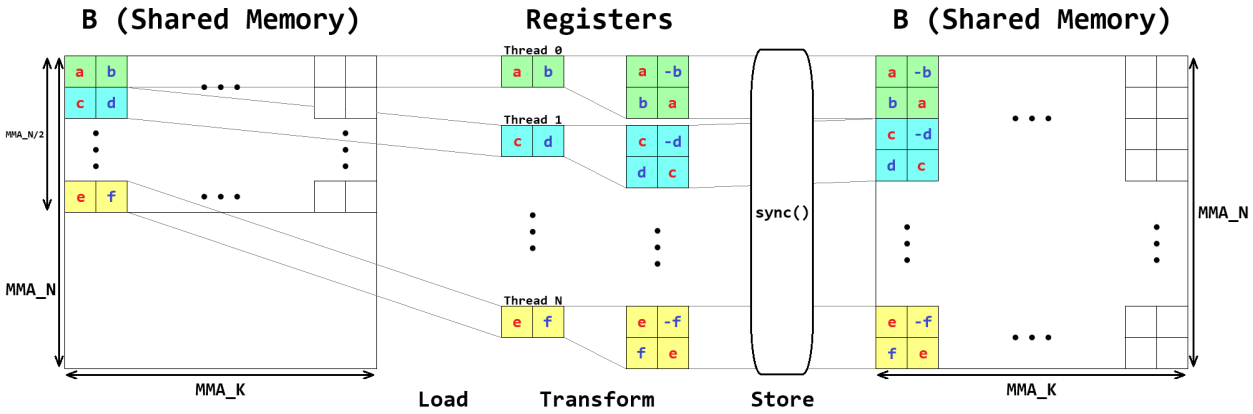



Figure 10: The complex transformation with the interleaved method on shared memory

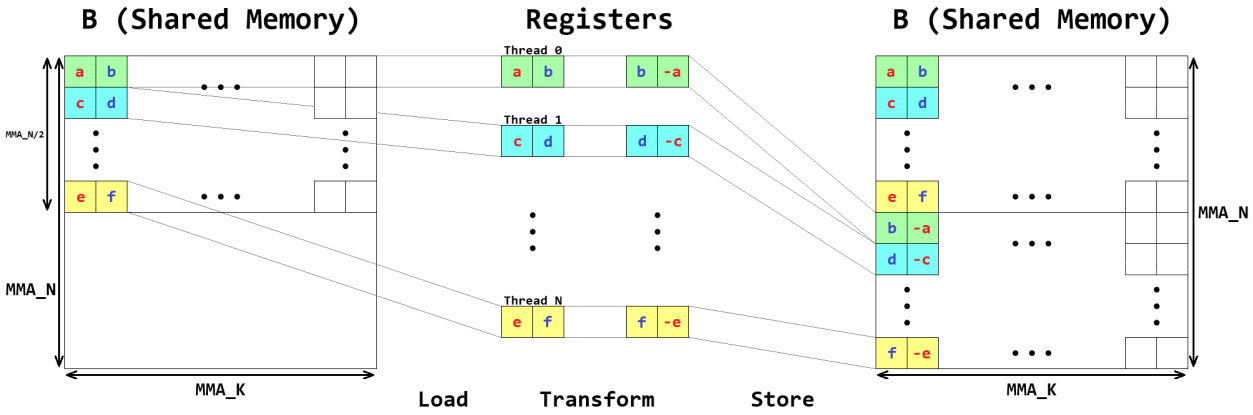



Figure 11: The correlation transformation with the planar method on shared memory

values in registers, which can then be stored to shared memory and subsequently to $D$ in global memory in the same manner as normal GEMM.

### 3.2.2 Variant 2

The second variant works in mostly the same way as the first variant, but with one important optimization. Namely, instead of using four MMA warps on both SMs of a cluster pair, the second variant uses just one MMA warp on the leader SM, and four *transformation warps* on each SM. The MMA warp, TMA warp, and transformation warps all use the same shared memory buffers for $A$ and $B$. Figure 12 shows how the warp specializations in the second variant communicate with each other. For each shared memory buffer that holds a K-tile from $A$ and $B$, there is one `mma_bar`, one `tma_bar`, and one `tsf_bar`. Compared to normal GEMM, the TMA warp remains unchanged: it waits on the `mma_bar` to check if a shared memory buffer is empty and ready to be used for TMA loads, and then uses the `tma_bar` to track the completion of TMA loads.

The `tma_bar` barriers are waited on by the transformation warps, instead of the MMA warp.

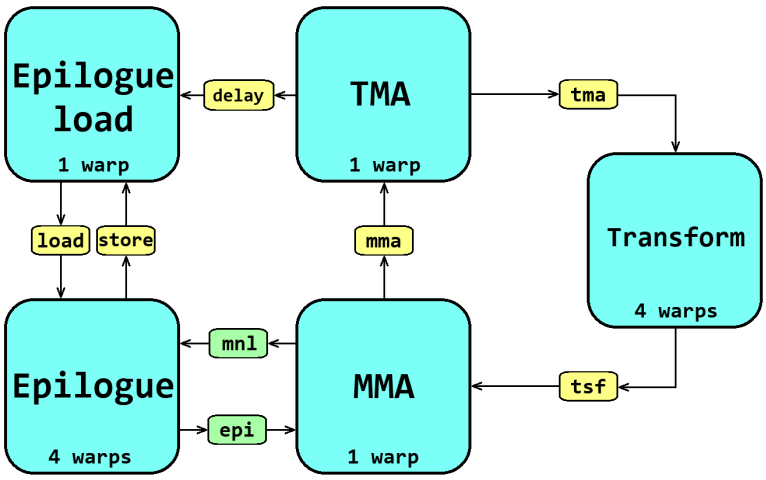



Figure 12: Warp specialization and associated barrier objects for the second variant of complex GEMM

After completing a wait, transformation warps perform, in the shared memory buffer for $B$, either the complex or correlation transformation, with either the interleaved or planar method, in the same way as the MMA warps in the first complex GEMM variant. They only differ in what happens after the transformation has been performed. Instead of synchronizing both SMs in a cluster pair with each other, the transformation warps perform an arrive-on operation on the corresponding `tsf_bar` on the leader SM. By initializing the `tsf_bar` barriers with an arrive count of 256 (four warps of 32 threads on two SMs), the phase of a `tsf_bar` barrier for a given shared memory buffer only completes when all warps in both SMs have finished transforming the corresponding K-tile and performed an arrive-on operation. After signaling the `tsf_bar` barrier, the transformation warps can immediately move on to transforming the next K-tile of $B$ in shared memory without having to wait on each other.

The MMA warp functions the same as in normal GEMM, with the exception that it waits on the `tsf_bar` barriers instead of the `tma_bar` barriers. After calling `cute::gemm` on each MMA tile of a given K-tile in a shared memory buffer, the MMA operations are committed to the corresponding `mma_bar` as usual.

### 3.2.3 Variant 3

The third variant performs the transformation in the K-tile of $A$ instead of $B$, and otherwise functions the same as the second variant. Since $A$ and $B$ MMA tiles both have $K$ as the leading dimension and since $B$ is transposed, loading from/storing to shared memory and transforming the values works in the exact same way. The only difference is that shared memory buffers for $A$ are doubled and used by the transformation warps, instead of $B$.

This variant can not make use of the interleaved method. When performing the interleaved transformation in $B$, the extra columns appear along the $N$ dimension of $B$. Since the Tensor

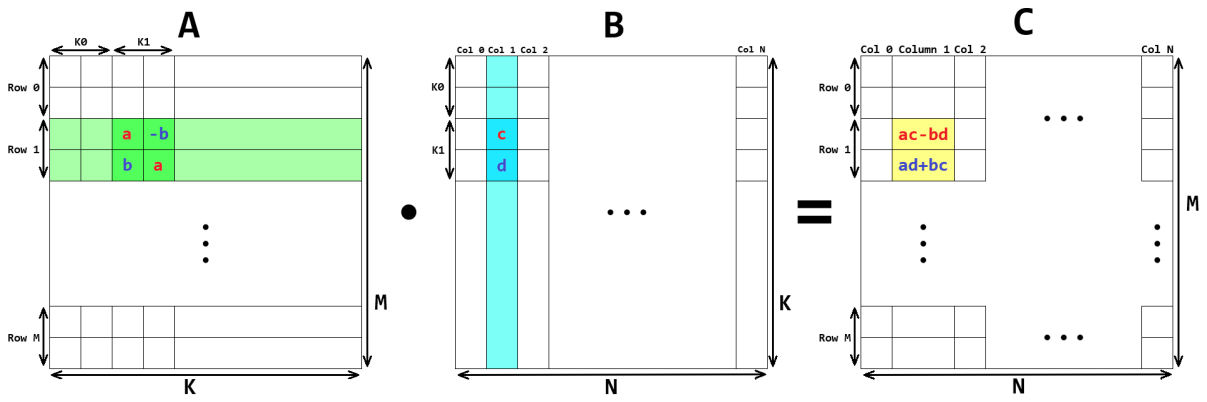



Figure 13: Complex GEMM with the interleaved method: embedding the complex transformation in adjacent rows of the A matrix.

Memory accumulation matrix is $N$-major, the real and imaginary parts of each complex number end up in adjacent columns and thus next to each other in Tensor Memory, allowing one or more complex numbers to be loaded into registers with a single Tensor Memory load. When performing the interleaved transformation in $A$, the extra rows appear along the $M$ dimension of $A$. As a result, the real and imaginary parts of a given complex number end up in adjacent rows of the resulting accumulation matrix, see Figure 13. The accumulation matrix lies in Tensor Memory, and there are no Tensor Memory load shapes that map two adjacent lanes to the same thread. Threads would thus have to load only real or only imaginary values and then read/write shared memory in an interleaved format, leading to a memory access pattern not optimized for the swizzle mode, which likely results in shared memory bank conflicts.

All variants that perform the transformation in $A$ can thus only make use of the planar method. The planar method for transformations in $A$ works the same way as with $B$: the two halves of a K-tile of $A$ are multiplied and accumulated separately, again resulting in one Tensor Memory accumulation tile that contains the real parts and one tile that contains the imaginary parts.

### 3.2.4 Variant 4

The fourth variant also performs the transformation in $A$, but sources the $A$ matrix in `tcgen05.mma` operations from Tensor Memory instead of shared memory. One major advantage of this is that the K-tiles for $A$ no longer have to be doubled in size to account for the extra rows required to hold transformed values, allowing for more shared memory buffers. Tensor Memory is large enough for the transformed K-tiles from $A$ to be buffered multiple times, and this variant buffers the Tensor Memory an equal amount to shared memory. In other words, for every shared memory buffer that holds one $A$ and $B$ K-tile, there is one corresponding buffer in Tensor Memory that holds the complex or correlation transformed K-tile for $A$. Similarly to how each shared memory buffer in the planar method can be divided into two halves, each Tensor Memory buffer consists of two fragments: one fragment

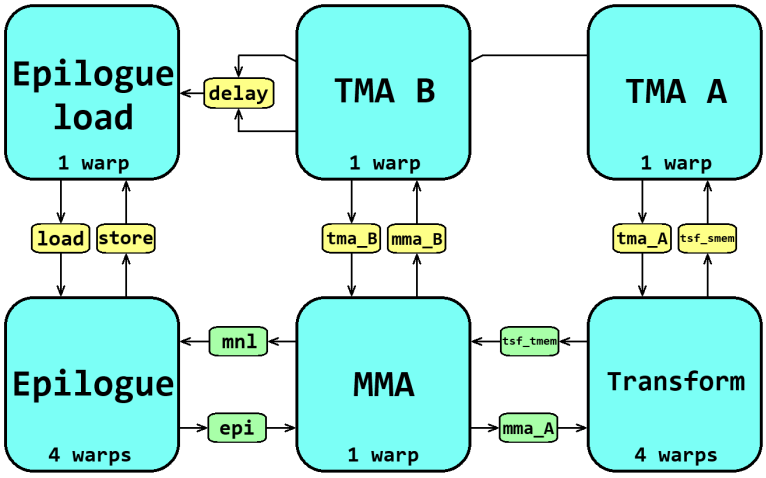



Figure 14: Warp specialization and associated barrier objects for the fifth variant of complex GEMM

for holding transformed values that compute the real parts, and one fragment for holding transformed values that compute the imaginary parts.

The transformation warps have to account for storing to Tensor Memory instead of to shared memory. Notably, when performing correlation, the original rows are not transformed, but must still be stored to Tensor Memory. The transformation warps allocate three fragments in registers for performing the transformation: one fragment for holding the original values from shared memory, one two fragments for the transformation. After waiting on the `tma_bar`, threads load from shared memory as usual. Then, the first half of the transformation is performed and stored in the first fragment used for holding transformed values. For complex multiplication, this means complex conjugating each original value, and for correlation, each original value is copied without any changes. Then, the first transformation fragment is written to the first Tensor Memory fragment of a given buffer with `tcgen05.st`. These store instructions are asynchronous, which allows the threads of the transformation warps to immediately perform the second half of the transformation while the first half is being stored. For complex multiplication, the second half of the transformation involves performing swaps on each original value and storing the result to the second transformation fragment. For correlation, each original value is complex conjugated and swapped instead. The second transformation fragment is then stored to the second Tensor Memory fragment. Afterwards, the warps synchronize and wait on the Tensor Memory stores to complete with `tcgen05.wait::st`. When the wait completes, arrive-on operations are performed on `tsf_bar` such that the MMA warp can observe that the transformed data is ready in Tensor Memory.

### 3.2.5 Variant 5

In the fifth variant, we aim to optimize the buffering and pipelining of the fourth variant. By allocating a Tensor Memory buffer for each shared memory buffer that holds a K-tile of $A$, the fourth variant is easy to implement and does not require the communication between warps to be changed. However, if Tensor Memory can hold more buffers than shared memory or vice-versa, the number of buffers has to be set to the lowest of the two, causing either Tensor Memory or shared memory to be underutilized. Furthermore, we observe that although a shared memory buffer holding a K-tile from $A$ can be reused for the next TMA copy as soon as the transformation warps finish loading from it, in the fourth variant it will remain unused until at least the corresponding transformation and Tensor Core MMA operations have finished.

Therefore, in this variant we choose to use two TMA warps for loading K-tiles from $A$ and $B$ separately, and then to decouple the Tensor Memory buffers from shared memory buffers. This allows all three buffers to have varying sizes. Figure 14 shows the warp specialization of the fifth variant, and how the warps communicate with each other. The TMA B warp and the MMA warp use the `tma_B_bar` and `mma_B_bar` barriers to communicate accesses of the shared memory buffers for K-tiles of $B$. The TMA A warp and the transform warps use the `tma_A_bar` and `tsf_smem_bar` barriers to communicate accesses of the shared memory buffers for K-tiles $A$. Finally, the MMA warp and the transform warps use the `mma_A_bar` and the `tsf_tmem_bar` barriers to communicate access of the Tensor Memory buffers for the transformed K-tiles of $A$.

The TMA A warp is responsible for copying K-tiles from $A$, and the TMA B warp is responsible for copying K-tiles from $B$. Both TMA B warps in a cluster pair specify the `tma_B_bar` barrier on only the leader SM to track TMA loads, since the MMA warp now waits for transformed $A$ K-tiles and the $B$ K-tiles independently. Both TMA A warps in a cluster pair specify the `tma_A_bar` barrier on their own SM for tracking completion of $A$ K-tile copies. This is because this barrier is waited on by the transform warps, which are present on both SMs of a cluster pair. The transformation warps iterate over the shared memory buffers that hold K-tiles of $A$. In each iteration, they wait on `tma_A_bar`, collectively load the entire corresponding K-tile from shared memory to registers, and then perform an arrive-on on the corresponding `tsf_smem_bar` barrier. By waiting on this barrier, the TMA A warp ensures that shared memory buffers for $A$ are reused for TMA loads as soon as possible.

Since the Tensor Memory buffers are now decoupled from shared memory buffers, the transformation warps no longer have a corresponding Tensor Memory buffer (which is guaranteed to be empty) to write to when loading from shared memory. Therefore, after loading a K-tile from $A$ into registers, the transformation warps perform both halves of the transformation (so, the transformation for the first fragment which computes the real values, and for the second fragment which computes the imaginary values), wait on `mma_A_bar` for the next Tensor Memory buffer to be free, and then store both fragments to Tensor Memory. Afterwards, an arrive-on operation is performed on `tsf_tmem_bar` of the leader SM in a cluster pair to signal the MMA warp that the transformed K-tile of $A$ is ready.

The MMA warp waits, for each K-tile, on `tsf_tmem_bar` for the transformed K-tile of $A$,

and then on `tma_B_bar` for the K-tile of $B$. When both waits complete, `cute::gemm` is called to multiply the MMA tiles as usual. After initiating the Tensor Core MMA operations, they are committed to both the `mma_A_bar` and `mma_B_bar` barriers, such that both the transformation warps and the TMA B warp can wait for the Tensor Memory and shared memory buffers to be empty.

# 4 Results

For evaluating both the normal and complex GEMM kernels, we will use the NVIDIA Jetson AGX Thor Developer Kit, which is equipped with a Blackwell GPU, and CUDA version 13.0. This Blackwell GPU has 20 SMs and a compute capability of 11.0, which means it can use `tcgen05` instructions.

## 4.1 Normal GEMM

We benchmark the performance of the normal GEMM kernel (`normal-gemm`) and compare it against cuBLAS. We consider a GEMM problem with $M = N = 4096$ and a varying $K$ dimension, where the $A$ and $B$ matrices are of type FP8, and the $C$ and $D$ matrices are of type FP32. Performance measurements were taken by running the kernel $100 - 1000$ times with $10 - 100$ warm-up runs depending on GEMM problem size, and then repeated and averaged 50 times to reduce variance.

The cuBLAS kernel was launched with `cublasLtMatmul`. With Nsight Compute we learn that cuBLAS does not launch a CUTLASS kernel, but instead launches a Jetson-specific kernel (`nvjet_qqsss_128x256_128x6_2x1_2cta_v_bz_TNT`). Although no further information is available about these Jetson kernels, from the name and the assembly (SASS) source code we determine that this kernel (at least in the case of $K = 4096$) uses `tcgen05` and TMA instructions, two SM MMA, MMA tile dimensions of $128 \times 256$, a K-tile width of 128 bytes (buffered 6 times in shared memory), warp specialization, and persistent kernel tile scheduling through Cluster Launch Control (CLC). CLC is a feature introduced in the Blackwell architecture that allows thread blocks that are not yet executing to be canceled, enabling their work to be stolen by other thread blocks. CLC can thus be used to implement a persistent kernel that dynamically schedules tiles to work on.

### 4.1.1 Performance

The performance in TFLOPS/s of the normal GEMM and cuBLAS kernels when performing $D = A \cdot B + C$ can be found in Figure 15. The cuBLAS kernel outperforms the `normal-gemm` kernel for all $K$ dimensions except 512. The difference in performance between the two kernels is relatively minor except for $K = 16384$ and especially $K = 8192$, where the cuBLAS kernel achieves almost twice the TFLOPS/s of the `normal-gemm` kernel. For `normal-gemm`, the peak performance can be found at $K = 4096$, while the peak performance for cuBLAS can be found at $K = 8192$.

The performance measurement results for the computation of $D = A \cdot B$ can be found in Figure 16. We observe that for both kernels, the performance is significantly higher when

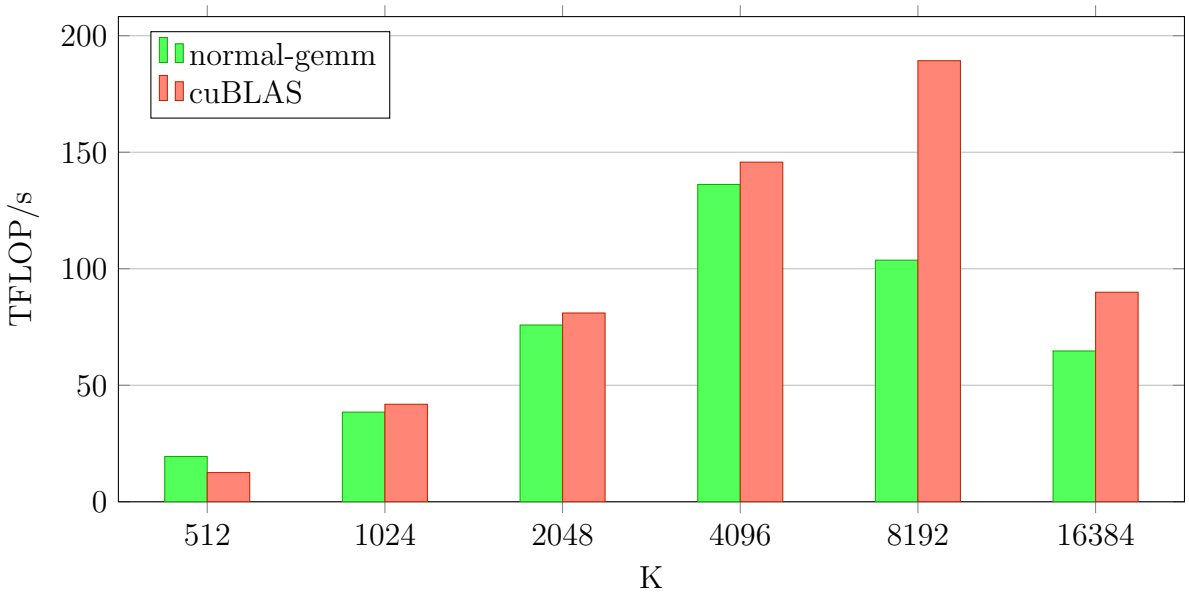



Figure 15: Average performance of computing $D = A \cdot B + C$ in TFLOPS/s of the normal GEMM kernel compared against cuBLAS

there is no $C$ matrix to be added. In the most extreme case, `normal-gemm` with $K = 2048$ achieves a 2.7× speedup when not adding the $C$ matrix. We suspect that this difference has to do with the LPDDR5X memory of the Jetson Thor. This memory is very energy efficient, but offers a relatively low bandwidth of 273 GB/s. Since the $C$ matrix is of type FP32, it is actually larger than the FP8 $A$ and $B$ matrices combined for $K < 8192$. Although each SM has to load entire rows of $A$ and columns of $B$ for each (much smaller in comparison) tile of $C$, the data from $A$ and $B$ can often be fetched from the L2 cache instead of the low-bandwidth memory. Since individual $C$ tiles are not used by multiple SMs, each load for a $C$ tile will incur loads from off-chip memory, vastly increasing the amount of traffic between the L2 cache and the low bandwidth memory. Additionally, the extra traffic incurred by loading $C$ tiles substantially lowers the L2 cache hitrate.

For $D = A \cdot B$, the `normal-gemm` kernel outperforms cuBLAS in most cases, only falling behind for $K = 8192$ and $K = 16384$. Again, the `normal-gemm` kernel peaks at $K = 4096$ and the cuBLAS kernel peaks at $K = 8192$. We theorize that this has to do with the tile scheduling: `normal-gemm` statically schedules tiles across SMs in a grid-stride fashion, while the cuBLAS kernel uses CLC, and additionally may use a different mapping of thread blocks to tiles. The tile scheduling is important for $A/B$ matrix reuse and hitrate in the L2 cache, and the static scheduling of `normal-gemm` happens to perform well at $K = 4096$ but is not optimal for other sizes, while the dynamic CLC scheduling of cuBLAS performs better with larger matrices.

### 4.1.2 Profiler analysis

We profile the `normal-gemm` kernel on $K = 4096$ with Nsight Compute in order to identify any bottlenecks on performance. Several metrics obtained by Nsight Compute are presented

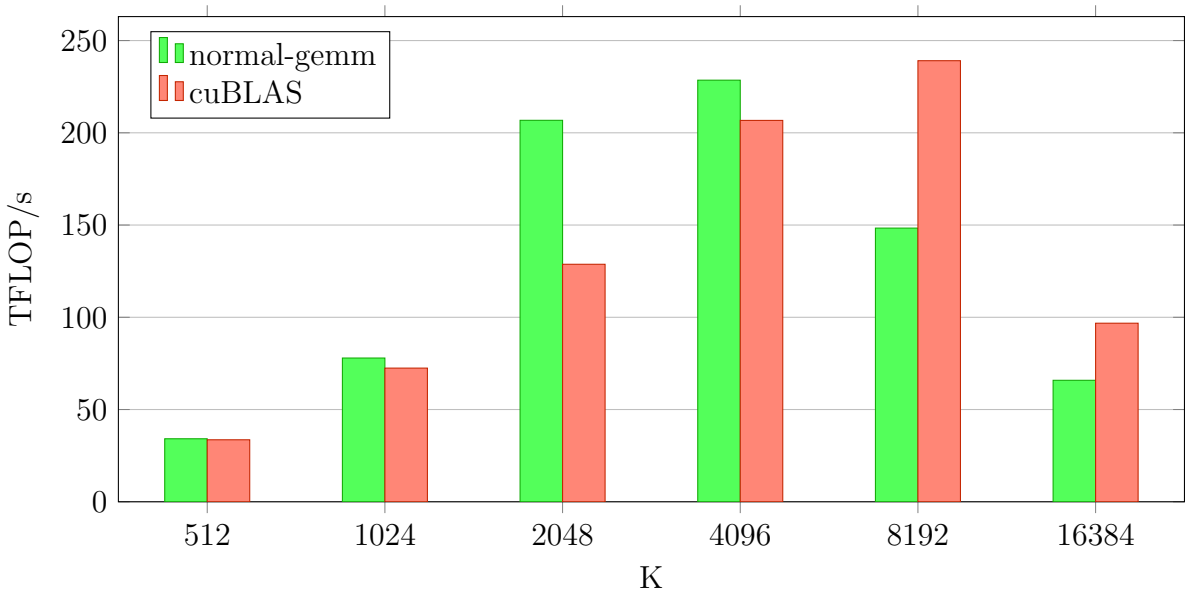



Figure 16: Average performance of computing $D = A \cdot B$ in TFLOPS/s of the normal GEMM kernel compared against cuBLAS

| Statistic | normal-gemm |
|---|---|
| Compute Throughput[1] | 51.42% |
| Tensor Core Pipe Active[2] | 41.72% |
| Memory Throughput / L2 cache throughput | 85.12% |
| L2 Hit Rate | 86.16% |
| Maximum Bandwidth[3] | 79.14% |

[1] `sm__pipe_tc_cycles_active.avg.pct_of_peak_sustained_elapsed`
[2] `sm__pipe_tensor_cycles_active.avg.pct_of_peak_sustained_active`
[3] `lts__lts2xbar_cycles_active.avg.pct_of_peak_sustained_elapsed`

Table 2: Several kernel metrics obtained with Nsight Compute of the normal GEMM kernel with problem shape $M = N = K = 4096$

in Table 2. The Compute Throughput measures the amount of cycles that the `tc` pipe was active, which is responsible for all `tcgen05` instructions except `tcgen05.ld`, `tcgen05.st`, and `tcgen05.wait`. Tensor Core Pipe Active measures the `tensor` pipe, which is only responsible for `tcgen05.mma` instructions. Since the Tensor Cores are on average active only 41.72% of the time, the kernel is not compute bound. The high Memory Throughput and Maximum Bandwidth indicate that the L2 cache is utilized well, and the high L2 hit rate indicates that the L2 cache is utilized efficiently.

Nsight Compute also allows for roofline analysis. For FP8 input, FP32 output, and dense multiplication, each SM can perform 16384 floating-point operations with the Tensor Cores per cycle[2]. With 20 SMs and a max GPU frequency of 1.57 GHz, the maximum performance is $16384 \cdot 20 \cdot 1.57 \cdot 10^9 = 514.5$ TFLOPS/s. In practice, this theoretical maximum will be limited

[2] Obtained from `sm__ops_path_tensor_op_utcqmma_src_fp4_fp6_fp8_dst_fp32_sparsity_off.max.peak_sustained`

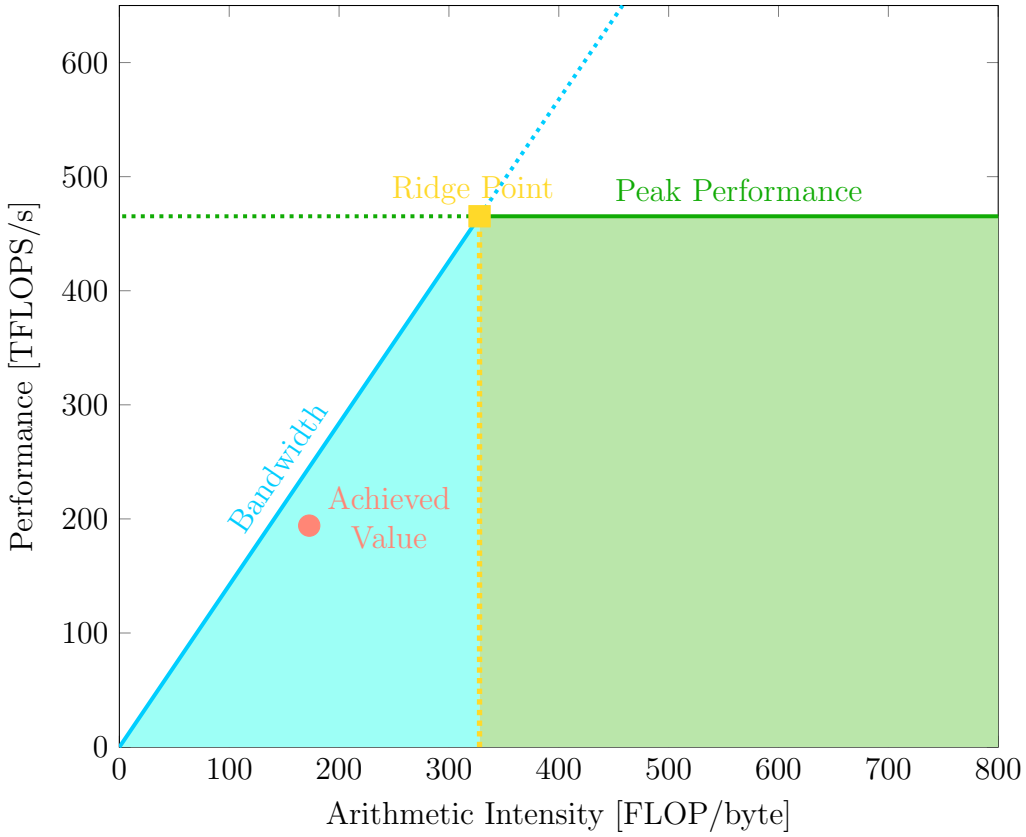



Figure 17: Roofline plot of the normal GEMM kernel with $M = N = K = 4096$, using the L2 cache for bandwidth and arithmetic intensity

by the availability of data, and thus the throughput of the L1 data cache (shared memory), the L2 data cache, and main memory. Nsight Compute reports that the kernel achieves an arithmetic intensity of 172.83 FLOPS/byte (in terms of L2 traffic) and a performance of 194.1 TFLOPS/s, keeping in mind that Nsight Compute limits clock frequency at the base frequency of 1.39 GHz. At this arithmetic intensity, Nsight Compute reports that the L2 cache throughput limits performance at 245.2 TFLOPS/s, see Figure 17. Thus, although some extra performance can be gained without changing arithmetic intensity, the kernel sits firmly in the memory-bound region of the roofline model.

In order to move into the compute-bound region of the roofline model, the arithmetic intensity must be increased. This can be done by better utilizing the L2 cache, which increases memory throughput, and by using larger tiles, which leads to more data reuse. Since the L2 cache is already well-utilized, the only practical way to increase arithmetic intensity is by increasing the tile size. Increasing the `MMA_N` dimension from 128 to 256 (for $D = A \cdot B$ with $M = N = K = 4096$) decreases performance from 228.52 TFLOPS/s to 183.46 TFLOPS/s. This is likely due to the fact that larger tiles result in a smaller amount of shared memory buffers, inhibiting the kernel's ability to hide data movement latencies, which is especially important considering the low-bandwidth memory on the Jetson Thor.

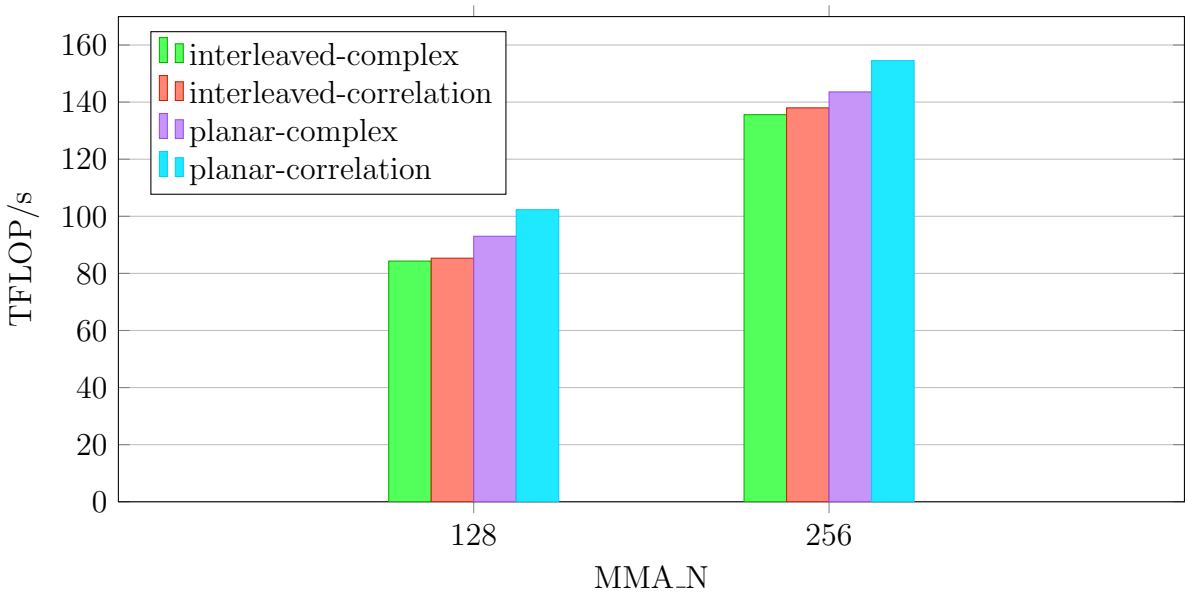



Figure 18: Average performance of variant 1 in TFLOPS/s for differing MMA_N

## 4.2 Complex GEMM

We benchmark the various complex GEMM variants by considering all possible combinations of transformation type (complex or correlation) and transformation method (interleaved or planar) for each variant. In the results for the normal GEMM kernel we observed that adding the $C$ matrix significantly reduces the performance, even though the mainloop does not change. Because of this, we only consider the computation of $D = A \cdot B$ when benchmarking complex GEMM variants, in order to more accurately assess any differences between variants. We use $A$ and $B$ matrices with dimensions $4096 \times 4096$ and a $D$ matrix with resulting dimensions $4096 \times 8192$, where $A$ and $B$ are FP8 and $D$ is FP32, like in the normal GEMM benchmarks. Performance measurements were taken by running the kernels 100 times with 10 warm-up runs, and then repeated and averaged 50 times to reduce variance.

During performance benchmarks, we also measured the wattage of the kernel runs with a PowerSensor3 [21], a power measurement tool. For a given kernel run, the measured FLOPS/s can be divided by the average wattage to obtain the energy efficiency in terms of FLOPS/J. We note that the PowerSensor3 is connected to the Jetson Thor over USB, which means that the measured energy usage concerns the entire device and not just the GPU.

### 4.2.1 Variant 1

The performance in TFLOPS/s for each combination of transformation method and type for differing `MMA_N` for variant 1 can be found in Figure 18. What is immediately observable is that an `MMA_N` of 256 vastly outperforms an `MMA_N` of 128, which was not the case in a normal GEMM kernel. We attribute this observation to two reasons. First, since TMA copies for $B$ will only load one-half of a shared memory buffer for $B$ (since the other half is reserved for the complex/correlation transformation), increasing the `MMA_N` from 128 to 256 will incur

half as many extra global memory loads in this complex GEMM variant than in the normal GEMM kernel. Second, we note that each transformation of a shared memory buffer brings with it an overhead that consists of synchronization, communication, and fencing for memory operations. Of these, we notice that the memory fence operations required for establishing an ordering of synchronous shared memory writes with asynchronous MMA operations are especially hindering with regards to performance. Thus, larger MMA tiles are beneficial for performance in complex GEMM variants as fewer tiles have to be transformed in total, resulting in fewer memory fence operations.

For both values of `MMA_N`, we observe that correlation performs better than complex multiplication for both the interleaved and planar methods, even though both transformation types consist of the same amount of bit mask and `__byte_perm` operations for each complex value. We suspect that since the correlation transformation performs these operations on only one row, register dependencies are eased and the compiler is able to generate a more optimized instruction stream capable of hiding the latency of shared memory accesses.

Furthermore, we observe that planar transformation performs better than interleaved transformation for both complex multiplication and correlation. For correlation, this result is expected since fewer shared memory stores are required, as was described in Section 3.2.1. Planar correlation thus performs much better, achieving a 1.12× speedup over interleaved correlation with an `MMA_N` of 256. The fact that the planar method also is demonstrably faster than the interleaved method for complex multiplication is a bit surprising. The shared memory loads and transformations transpire in the exact same way, the only difference lies in how the transformed data is stored back to shared memory, how the `tcgen05.mma` operations are performed, and how data is loaded from Tensor Memory in the epilogue. The epilogue warps accounts for the planar format by performing two half-sized Tensor Memory loads (in order to load real and imaginary values separately), which is typically slower and can thus not explain the performance increase. The fact that one `tcgen05.mma` operation is performed for each half of the transformed MMA tile of $B$ (in the case of planar transformation) instead of one `tcgen05.mma` operation for both halves of the transformed MMA tile of $B$ (in the case of interleaved transformation) is unlikely to lead to a measurable difference in performance, as the Tensor Core MMA operations should be enqueued efficiently in the asynchronous execution model. Thus, we theorize that the speedup achieved by the planar method over the interleaved method lies in the difference in how transformed values are stored to shared memory, namely, that the planar method makes better use of the swizzling format in order to avoid shared memory bank conflicts.

### 4.2.2 Variant 2

The performance in TFLOPS/s for each combination of transformation method and type for a differing number of shared memory buffers for variant 2 can be found in Figure 19. We again observe that correlation performs better than complex multiplication, and that the planar method performs better than the interleaved method. However, the specific case of planar correlation does not perform much better than the other combinations of transformation method and types. In general, for all different numbers of shared memory buffers, the different combinations have become much more similar to each other in terms of performance than

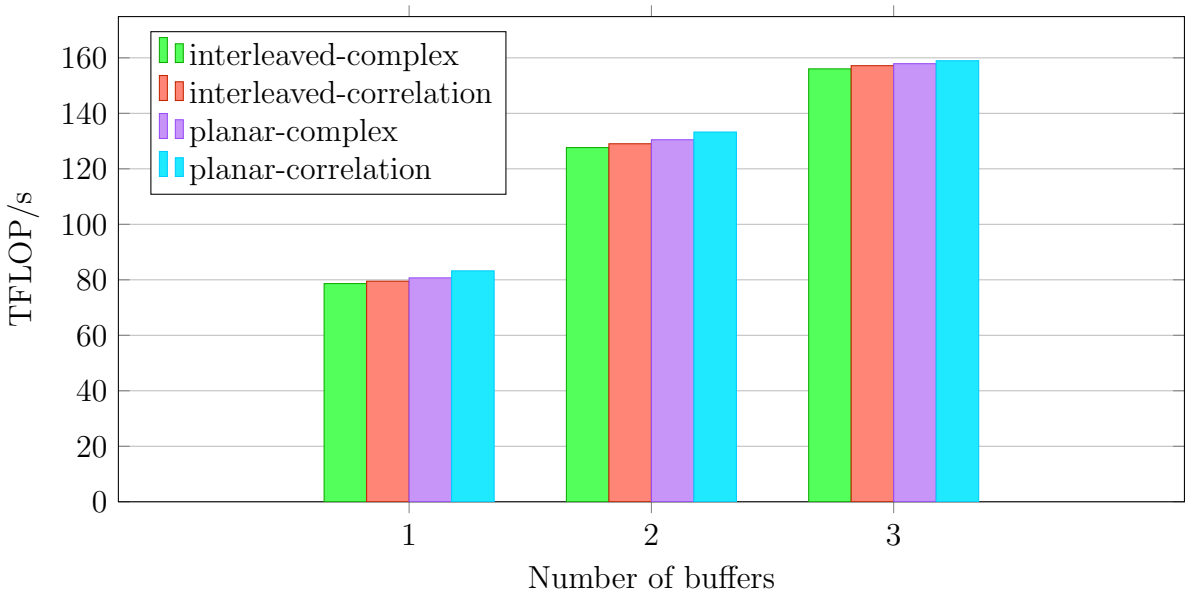



Figure 19: Average performance of variant 2 in TFLOPS/s for differing number of shared memory buffers

in variant 1. Thus, since the worst cases are improved more than the best cases, we can conclude that separating the transformation warps from the MMA warps allows the latency of a transformation (i.e., the time it takes for data that has arrived in shared memory to be used in an MMA operation) to hidden better, while still being a bottleneck to the overall performance of the kernel.

Furthermore, we observe that increasing the number of shared memory buffers significantly increases the performance, demonstrating that buffering is critical for hiding data movement and data transformation latencies.

### 4.2.3 Variants 3 & 4

Since variant 3 doubles the larger `MMA_M` of 256 instead of the smaller `MMA_N` of 128 to make space for the transformed values, the size of each buffer becomes larger and as a result variant 3 only has two shared memory buffers to work with. We note that there is a way to use all 128 lanes of Tensor Memory on each SM even with an `MMA_M` of 128 in two SM MMA (where the SM-local `MMA_M` would thus be only 64), but that this method requires a substantial redesign of the kernel and will thus not be considered in this work.

The performance in TFLOPS/s for variant 3 and for variant 4 with 2 and 4 shared memory buffers can be found in Figure 20. We observe that the performance of variant 3 is very poor compared to variants 1 and 2, which we attribute to the fact that variant 3 only has two shared memory buffers to hold data from $B$ and (transformed) data from $A$. In fact, the performance of variant 3 is comparable to the performance of variant 2 with two shared memory buffers. Thus, there does not seem to be an inherent advantage in choosing to perform the transformation in $A$ over performing it in $B$. However, performing the transformation in $A$

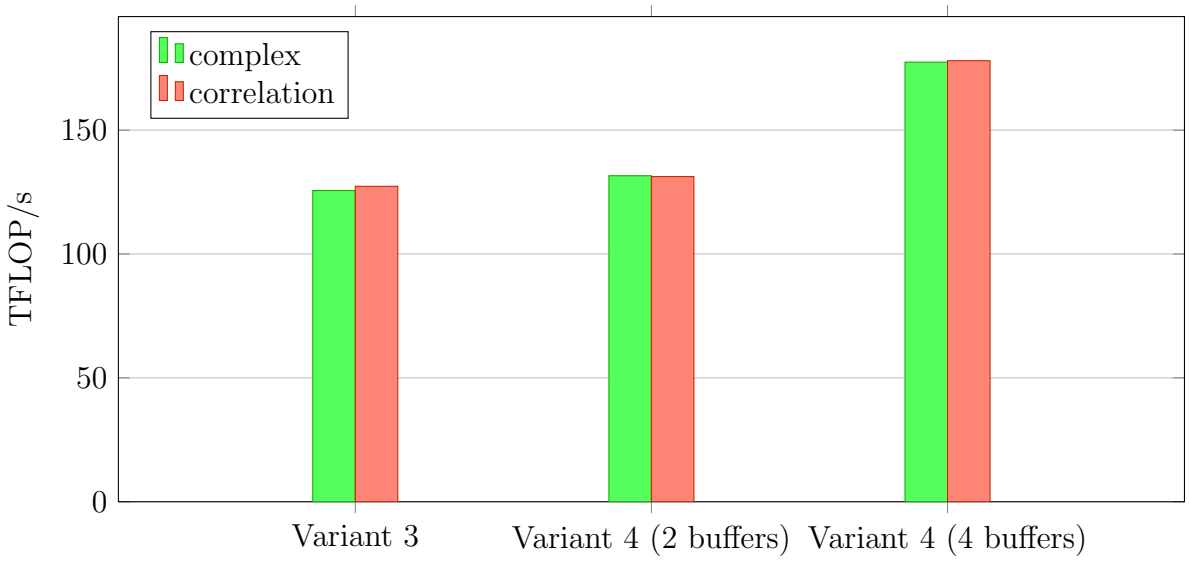



Figure 20: Average performance in TFLOPS/s of variant 3 and variant 4 with differing number of shared memory buffers

does offer the option to store the result back to Tensor Memory. By comparing variant 3 to variant 4 with two buffers, we can measure the difference in performance between storing to shared memory and to Tensor Memory. We observe a slight advantage in storing to Tensor Memory, which is likely because of the fact that threads do not have to fence for shared memory writes when writing to Tensor Memory. Additionally, Tensor Memory writes may also have a lower latency than shared memory accesses, although this would have to be verified experimentally.

The main advantage of writing the transformed $A$ matrix to Tensor Memory instead of shared memory is that shared memory buffers for $A$ no longer have to be doubled in size. This allows for more shared memory buffers for $A$ and $B$: four instead of two. With an `MMA_N` of 128, one accumulator tile for the real parts and one accumulator tile for the imaginary parts together take up $128 \cdot 2 = 256$ columns of Tensor Memory (since columns are 32-bit wide and accumulation happens in FP32), which leaves 256 columns for holding transformed values of $A$. Each buffer in Tensor Memory that holds transformed $A$ values is 64 columns wide, as each buffer holds two 128-byte wide (= 32 Tensor Memory columns) K-tiles (one for computing the real parts and one for computing the imaginary parts), which allows for 4 Tensor Memory buffers, fortunately the same as the amount of shared memory buffers. These extra shared memory buffers are of immense help for performance: with 4 buffers, variant 4 achieves a speedup over variant 3 of 1.41× for complex multiplication, and 1.40× for correlation.

### 4.2.4 Variant 5

In the fifth variant, we can experiment with different buffer configurations, as the shared memory buffers for $A$ and $B$ and the Tensor Memory buffers for transformed values of $A$ have been decoupled. The number of Tensor Memory buffers depends on the MMA tile

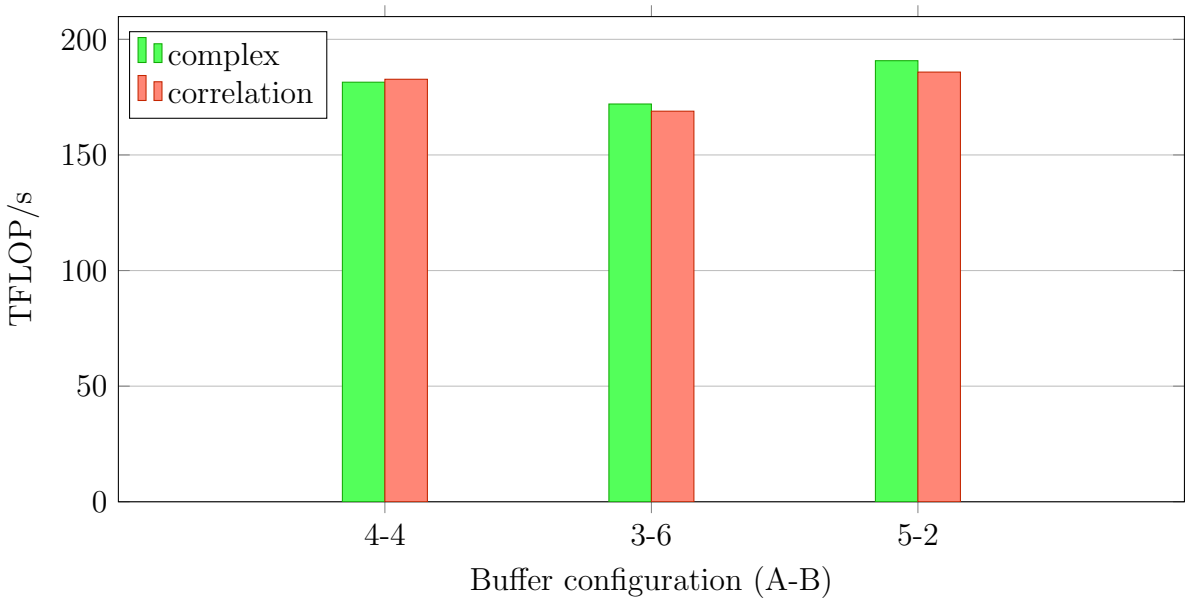



Figure 21: Average performance of variant 5 in TFLOPS/s with different configurations of number of shared memory buffers for $A$ and for $B$

dimensions and will thus always be 4 for each configuration, while the number of $A$ and $B$ buffers can be shifted to favour one or the other, keeping in mind that one $A$ buffer is as large as two $B$ buffers. Figure 21 shows the performance in TFLOPS/s for three buffer configurations: 4 $A$ buffers and 4 $B$ buffers (the same as in variant 4), 3 $A$ buffers and 6 $B$ buffers, and 5 $A$ buffers and 2 $B$ buffers.

The buffer configurations with 4 $A$ buffers and 4 $B$ buffers (the 4-4 configuration) for variant 5 performs slightly better than variant 4: variant 5 achieves 181.5 TFLOPS/s for complex multiplication and 182.7 TFLOPS/s for correlation, and variant 4 achieves 177.4 TFLOPS/s for complex multiplication and 178.0 TFLOPS/s for correlation. Thus, we can conclude that separating the TMA warp into a TMA $A$ warp and a TMA $B$ warp results in a small improvement of buffer utilization.

The buffer configuration with 3 $A$ buffers and 6 $B$ buffers (the 3-6 configuration) performs slightly but noticeably worse than the 4-4 configuration. In a similar vein, the 5-2 buffer configuration performs slightly but noticeably better than the 4-4 configuration. Therefore, a larger amount of $A$ buffers can help in hiding the transformation latency. Since the increase in performance is very slight, it is likely that the latency of the transformation can be (almost) completely hidden in the 5-2 configuration. However, a major downside is that only 2 buffers for $B$ remain available, which limits the ability to hide data movement latency.

### 4.2.5 Comparison of variants

Figure 22 shows the performance in TFLOPS/s of the best performing configuration of each variant (including the transformation method) for both complex multiplication and correlation. We observe that variant 5 comfortably outperforms all other variants. Variant

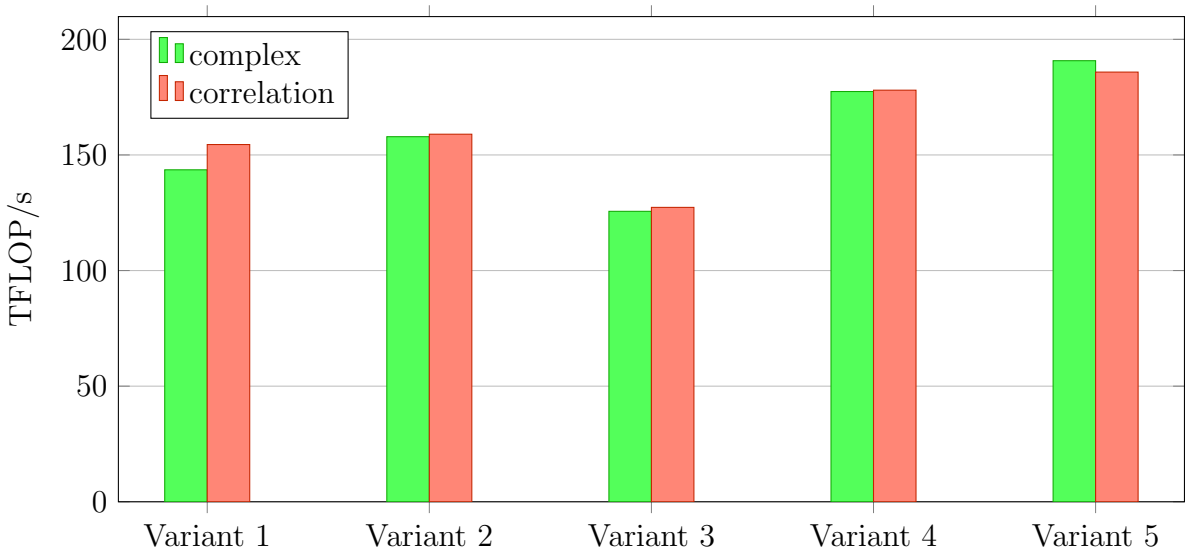



Figure 22: Average performance in TFLOPS/s of best performing kernel for each variant

5 with complex multiplication achieves 83.5% of the maximum performance obtained by the normal GEMM kernel, and likewise correlation with variant 5 achieves 81.3% of the performance of the normal GEMM kernel.

Figure 23 shows the respective energy efficiency in GFLOPS/J of each best performing configuration. As mentioned earlier, the measured energy usage concerns the entire Jetson Thor, including the idle power draw of the CPU and storage device. Therefore, measurements might not fully reflect the actual energy efficiency of a kernel, and differences between kernels might appear less significant in the graph. Generally, the energy efficiency of a given kernel matches closely to the achieved performance of the kernel. This follows intuition, since a faster time-to-finish of a given kernel run results in a smaller period of time where the GPU is active and consuming energy. The one noticeable exception is correlation with variant 1, which is only less energy efficient than variant 5, even though it is also outperformed by variants 2 and 4. Planar correlation performs fewer shared memory stores than the other combinations of transformation types and methods, and this lowered usage of shared memory might explain the high energy efficiency. However, since we do not observe the same result for variant 2, this could also be the result of random variance in measuring.

We repeated this experiment for integer data types, where $A$ and $B$ consist of 8-bit signed integers, and $D$ consists of 32-bit signed integers. The performance in TOPS/s of each best-performing variant on integer matrices can be found in Figure 24. Overall, the performance of the integer types is slightly worse than the floating-point data types. Since Tensor Core integer throughput is lower than the floating-point throughput, a reduction in performance is not surprising. Since the decrease in performance is very slight, we can conclude that the floating-point complex GEMM variants can not fully hide the transformation latency, and that the performance is not close to being bounded by the Tensor Core throughput.

The 8-bit integer energy efficiency is shown in Figure 25. Overall, the efficiency of integer Tensor Core multiplication is far higher than that of floating-point Tensor Core multiplica-

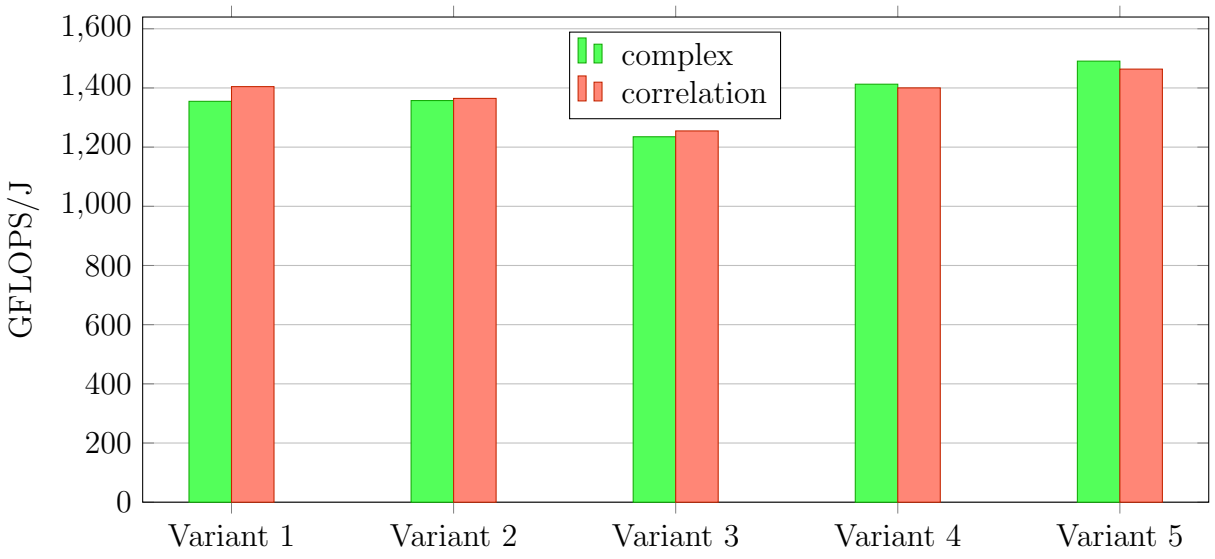



Figure 23: Average energy efficiency in GFLOPS/J of best performing kernel for each variant

tion. In the case of variant 5 correlation, integer multiplication is 1.54× more energy efficient than floating-point multiplication. We suspect that since integer multiplication is in general simpler than floating-point multiplication, the associated Tensor Core pipeline constitutes fewer transistors and thus uses less energy to perform multiplications.

# 5 Discussion

In this thesis, we set out to adapt the Tensor-Core Correlator to the Blackwell microarchitecture. To this end, we investigated the microarchitectures from Volta to Blackwell, and how each microarchitecture enables specific techniques for Tensor Core powered GEMM kernels, and how to efficiently perform data transformations required for correlation. We have developed an efficient GEMM kernel that uses many of these techniques and optimizations: `tcgen05.mma` instructions to accumulate MMA results in Tensor Memory, TMA instructions to perform efficient memory copies between global and shared memory, shared memory swizzling to avoid bank conflicts, warp specialization for pipelining and optimized instruction streams, two SM MMA for greater data reuse, epilogue tiling to reduce register pressure and overlap TMA loads of $C$ and TMA stores of $D$, and persistent tile scheduling to reduce kernel launch overhead and overlap the mainloop with the epilogue.

The normal GEMM kernel struggles to keep up with the performance of cuBLAS when adding a $C$ matrix. When only computing $D = A \cdot B$, our GEMM kernel performs much better, and even outperforms cuBLAS in some cases. The performance of our GEMM kernel is highly dependent on the problem shape, and peaks at $M = N = K = 4096$, achieving a performance of 228.5 TFLOPS/s. In contrast, cuBLAS achieves a slightly higher peak performance of 239.1 TFLOPS/s, with a problem shape of $M = N = 4096$ and $K = 8192$. We note that our GEMM kernel does not change any parameters depending on the GEMM problem size, but that cuBLAS is capable of selecting parameters or even a different kernel

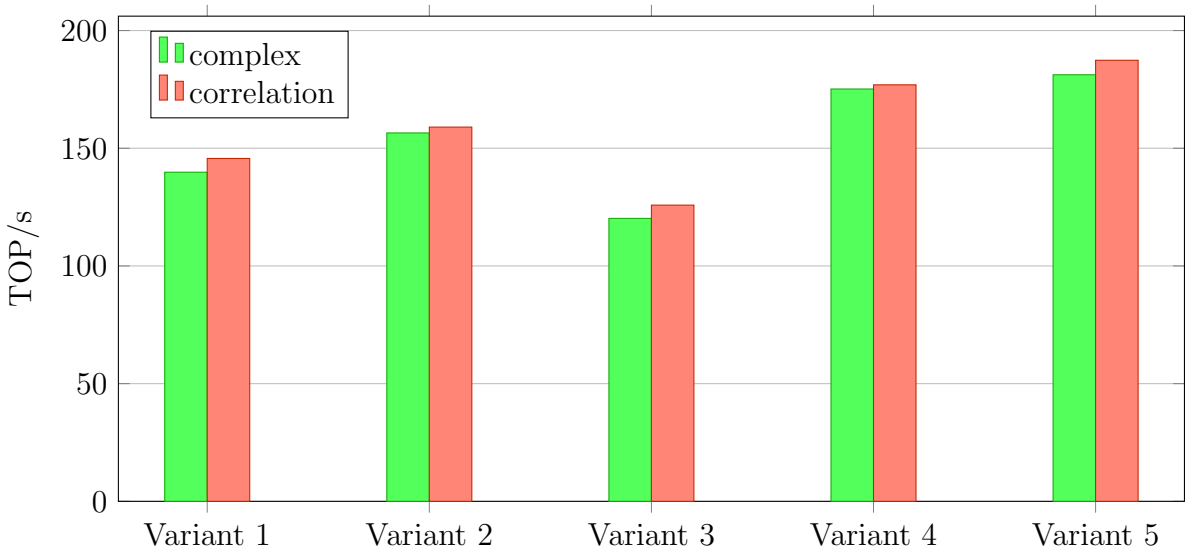



Figure 24: Average INT8 performance in TOPS/s of best performing kernel for each variant

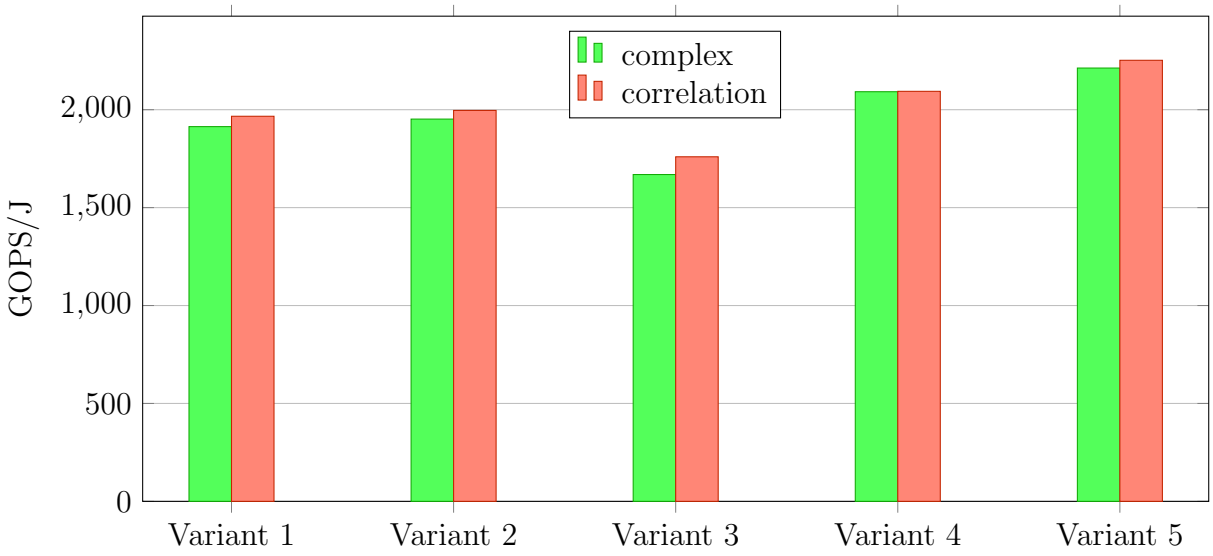



Figure 25: Average INT8 energy efficiency in GOPS/J of best performing kernel for each variant

depending on the GEMM problem size. We believe that each kernel at their respective peaks achieves (close to) maximum achievable performance on the Jetson Thor, both being extremely memory-bound due to the low-bandwidth LPDDR5X memory. Furthermore, we believe that our GEMM kernel's inability to perform well across all GEMM problem shapes is the result of the tile static scheduling. Dynamic tile scheduling (through CLC) and more complex thread block to tile mappings (see for example the `swizzle_and_rasterize()`[3] method in CUTLASS) can be used to achieve better work balancing between SMs and L2 cache data reuse across multiple GEMM problem shapes.

We used the normal GEMM kernel to develop multiple different variants that are able to perform complex matrix multiplication and correlation. These variants all have to perform shared memory loads and transform data in the mainloop, which for all variants decreases the achieved performance when compared to the normal GEMM kernel. Performance improvements across variants were achieved with two methods: warp specialization, which allowed variant 2 to outperform variant 1 and allowed variant 5 to outperform variant 4, and using Tensor Memory to store transformed data, which allowed variant 4 to outperform variant 2. Both of these methods ultimately improve the utilization of the available shared memory buffers, which helps towards hiding the latency that is incurred by the transformation of data. Efficiently performing data transformations in a GEMM kernel on the Blackwell microarchitecture becomes a challenge in trying to hide the latencies of both transformation and data movement. Fulfilling both of these objectives at once requires smart usage of the available shared memory.

Although the developed complex GEMM variants are capable of performing correlation, they do not take the triangle shape into account, and use simple rectangular matrices. The Tensor-Core Correlator does compute a triangle shape, and uses a more complicated input data format which explicitly contains the polarizations and frequency channels. Furthermore, the Tensor-Core Correlator typically works with a number of receivers that would translate into significantly smaller $M$ and $N$ dimensions for the input matrices (to our complex GEMM variants) than the dimensions considered in this work. Smaller $A$ and $B$ matrices might require smaller MMA tiles in order to fully saturate all SMs of the GPU, and in this work we have considered MMA tile dimensions with the assumption that the $M$ and $N$ dimensions are sufficiently large. Still, an early preliminary comparison to the Tensor-Core Correlator can be made. With 8-bit integer multiplication and 32-bit integer accumulation, the Tensor-Core Correlator achieves a peak performance of 69.7 TOPS/s at 690 GOPS/J, and variant 5 (the fastest complex GEMM variant of this work) achieves a peak performance of 187.4 TOPS/s at 2252.3 GOPS/J. Thus, keeping in mind that the measured performance of variant 5 is much more advantageous than it would be in a proper correlation kernel in a realistic scenario, we are still positive about these results, as they show that a significant increase in performance and energy efficiency can be gained with `tcgen05`.

[3] https://github.com/NVIDIA/cutlass/blob/main/include/cutlass/gemm/kernel/sm100_tile_scheduler.hpp#L697

# 6 Conclusions and future research

In this thesis, we have provided valuable insights into efficiently correlating radio astronomy signals on the Blackwell microarchitecture by designing and benchmarking an efficient normal GEMM kernel, and multiple GEMM variants capable of complex matrix multiplication through efficiently transforming input data. Many optimizations and techniques such as warp specialization, `tcgen05` matrix multiplication, and global-to-shared memory TMA copies are essential for an efficient and performant GEMM kernel on the Blackwell microarchitecture. Such a normal GEMM kernel can then be extended by performing data transformations inside the mainloop in order to compute a complex matrix multiplication or correlation, albeit by computing a full square instead of a correlation triangle. Warp specialization is essential to efficiently performing these transformations, and using Tensor Memory to store the transformation results allows maximal usage of the available shared memory in order to hide latencies incurred by data movement and transformations.

This work only represents the first step towards efficiently correlating radio astronomy signals on the Blackwell microarchitecture, as the developed complex GEMM variants are not capable of performing (efficient) correlation in a practical setting. Future research into developing a complete Blackwell correlator kernel that builds on the results of this work will encounter multiple challenges.

Blackwell Tensor Core MMA dimensions have increased significantly compared to previous microarchitectures due to the introduction of Tensor Memory. While larger blocks are beneficial for data reuse and thus overall GEMM performance, they are not well-fitted for approximating the triangle that a correlator computes, leading to an increase in the amount of redundantly computed elements. Other MMA tile configurations and the possibility to not use two SM MMA would have to be investigated, and, depending on the increase in redundancy, it may be worthwhile to compare the performance of the smaller-dimensional `mma` PTX instructions to the performance of `tcgen05.mma`.

Persistent GEMM kernels unlock overlap between the mainloop and epilogue, allowing for more parallelism and increased performance. However, we found that the persistent tile scheduling of the GEMM kernels we developed do not perform well under all GEMM problem dimensions. Future complex GEMM kernels can improve on this work by considering dynamic tile scheduling using Cluster Launch Control, introduced in Blackwell. Additionally, the Tensor-Core Correlator uses two kinds of tiles in order to reduce redundantly computed elements of the triangle structure, and the dynamic tile scheduling may even be essential in achieving high performance on persistent correlator kernels.

While we have conducted some energy consumption measurements, and found that complex GEMM variants are significantly more energy efficient when using integer data types instead of floating-point types at a minimal loss in performance, more work can be done in the area of energy efficiency. Since we used a Jetson Thor with a PowerSensor3, we were only able to measure the energy usage of the entire device, and future work on a Jetson platform would have to use the `tegrastats` utility to measure the energy usage of the individual components. Furthermore, in this thesis we have optimized for performance, and then measured the energy efficiency of the best performing variants. Future work could use an approach that optimizes

for energy efficiency first, and performance second.

Kernels in this work were implemented entirely in C++ with features from CuTe. This requires a lot of low-level programming in order to handle warp specialization and software pipelining, communication and synchronization, and computation and data movement. Even with a library like CuTe, implementing these common features is time consuming and error prone, and the result becomes hard to maintain, especially across different microarchitectures. Domain Specific Languages (DSLs) such as Triton, cuTile Python, and CuTe DSL offer high-level abstractions that implement these features, and can be used to implement complex GEMM/correlation kernels in the future. Furthermore, the new CUDA Tile C++ (introduced with CUDA version 13.3 in May 2026, well after the start of this thesis) can be used to program on a tile level in C++, and also provides abstractions for the TMA and for Tensor Cores.

# References


[1] Michael Bauer, Henry Cook, and Brucek Khailany. CudaDMA: Optimizing GPU memory bandwidth via warp specialization. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2011.

[2] Yoshihiro Chikada. Correlators for interferometry – Today and Tomorrow. *International Astronomical Union Colloquium*, 131:26–32, 1991.

[3] M.A. Clark, PC La Plante, and L.J. Greenhill. Accelerating radio astronomy cross-correlation with graphics processing units. *The International Journal of High Performance Computing Applications*, 27(2):178–192, 2013.

[4] NVIDIA Corporation. NVIDIA Tesla V100 GPU architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017.

[5] NVIDIA Corporation. NVIDIA H100 Tensor Core GPU Architecture. https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c, 2022.

[6] NVIDIA Corporation. NVIDIA Blackwell Architecture Tenchincal Brief. https://resources.nvidia.com/en-us-blackwell-architecture, 2024.

[7] NVIDIA Corporation. Inside NVIDIA Blackwell Ultra: The Chip Powering the AI Factory Era. https://developer.nvidia.com/blog/inside-nvidia-blackwell-ultra-the-chip-powering-the-ai-factory-era/, 2025.

[8] Keith Grainge, Alachkar Bassem, Shaun Amy, et al. Square Kilometre Array: The radio telescope of the XXI century. *Astronomy reports*, 61(4):288–296, 2017.

[9] M. P. van Haarlem, M. W. Wise, A. W. Gunst, et al. LOFAR: The LOw-Frequency ARray. *Astronomy & Astrophysics*, 556:A2, 2013.

[10] Bastian Hagedorn and Vinod Grover. It's about Time: Temporal Abstractions for Asynchronous GPU Tensor Computations. In *Proceedings of the 35th ACM SIGPLAN International Conference on Compiler Construction*, CC '26, page 14–26, New York, NY, USA, 2026. Association for Computing Machinery.

[11] Chris Harris, Karen Haines, and Lister Staveley-Smith. GPU accelerated radio astronomy signal convolution. *Experimental Astronomy*, 22(1-2):129–141, 2008.

[12] Weile Luo, Ruibo Fan, Zeyu Li, Dayou Du, Hongyuan Liu, Qiang Wang, and Xiaowen Chu. Dissecting the NVIDIA Hopper Architecture through Microbenchmarking and Multiple Level Analysis, 2025.

[13] Matt Martineau, Patrick Atkinson, and Simon McIntosh-Smith. Benchmarking the NVIDIA V100 GPU and Tensor Cores. In *Euro-Par 2018*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes

in Bioinformatics), pages 444–455. Springer Verlag, jan 2019.

[14] Leon Oostrum, Bram Veenboer, Ronald Rook, Michael Brown, Pieter Kruizinga, and John W. Romein. The tensor-core beamformer: A high-speed signal-processing library for multidisciplinary use. In *2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 582–592, 2025.

[15] Katsuhisa Ozaki, Yuki Uchino, and Toshiyuki Imamura. Ozaki Scheme II: A GEMM-oriented emulation of floating-point matrix multiplication using an integer modular technique, 2025.

[16] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. Swizzle inventor: Data movement synthesis for gpu kernels. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 65–78. Association for Computing Machinery, 2019.

[17] John W Romein. The Tensor-Core Correlator. *Astronomy & Astrophysics*, 656:A52, 2021.

[18] John W. Romein, P. Chris Broekema, Ellen van Meijeren, Kjeld van der Schaaf, and Walther H. Zwart. Astronomical real-time streaming signal processing on a Blue Gene/L supercomputer. In *Proceedings of the Eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '06, page 59–66. Association for Computing Machinery, 2006.

[19] Kjeld Schaaf and Ruud Overeem. Cots Correlator Platform. *Experimental Astronomy*, 17:287–297, 06 2004.

[20] Yuki Uchino, Qianxiang Ma, Toshiyuki Imamura, Katsuhisa Ozaki, and Patrick Lars Gutsche. Emulation of complex matrix multiplication based on the chinese remainder theorem. In *ISC High Performance 2026 Research Paper Proceedings (41st International Conference)*, page 1–12. IEEE, 2026.

[21] Steven van der Vlugt, Leon Oostrum, Gijs Schoonderbeek, Ben van Werkhoven, Bram Veenboer, Krijn Doekemeijer, and John W. Romein. PowerSensor3: A Fast and Accurate Open Source Power Measurement Tool. In *2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 213–226. IEEE Computer Society, 2025.

[22] Rob V Van Nieuwpoort and John W Romein. Using many-core hardware to correlate radio astronomy signals. In *Proceedings of the 23rd international conference on Supercomputing*, pages 440–449, 2009.

[23] Randall B. Wayth, Lincoln J. Greenhill, and Frank H. Briggs. A GPU-based Real-time Software Correlation System for the Murchison Widefield Array Prototype. *Publications of the Astronomical Society of the Pacific*, 121(882):857–865, 2009.

[24] Alwyn Wootten and A. Richard Thompson. The Atacama Large Millimeter/Submil-

limeter Array. *Proceedings of the IEEE*, 97(8):1463–1471, 2009.

[25] Rohan Yadav, Michael Garland, Alex Aiken, and Michael Bauer. Task-based tensor computations on modern gpus. *Proc. ACM Program. Lang.*, 9(PLDI), 2025.

[26] 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, 05 2020.