applied sciences

Article

Benchmarking GPU Tensor Cores on General Matrix Multiplication Kernels through CUTLASS

Xuanteng Huang , Xianwei Zhang *, Panfei Yang * and Nong Xiao

School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou 510006, China; huangxt57@mail2.sysu.edu.cn (X.H.); xiaon6@mail.sysu.edu.cn (N.X.)

* Correspondence: zhangxw79@mail.sysu.edu.cn (X.Z.); yangpanfei@ceprei.com (P.Y.)

Abstract: GPUs have been broadly used to accelerate big data analytics, scientific computing and machine intelligence. Particularly, matrix multiplication and convolution are two principal operations that use a large proportion of steps in modern data analysis and deep neural networks. These performance-critical operations are often offloaded to the GPU to obtain substantial improvements in end-to-end latency. In addition, multifarious workload characteristics and complicated processing phases in big data demand a customizable yet performant operator library. To this end, GPU vendors, including NVIDIA and AMD, have proposed template and composable GPU operator libraries to conduct specific computations on certain types of low-precision data elements. We formalize a set of benchmarks via CUTLASS, NVIDIA's templated library that provides high-performance and hierarchically designed kernels. The benchmarking results show that, with the necessary fine tuning, hardware-level ASICs like tensor cores could dramatically boost performance in specific operations like GEMM offloading to modern GPUs.

Keywords: GPU; GEMM; benchmark; tensor core

Citation: Huang, X.; Zhang, X.; Yang, P.; Xiao, N. Benchmarking GPU Tensor Cores on General Matrix Multiplication Kernels through CUTLASS. *Appl. Sci.* **2023**, *13*, 13022. https://doi.org/10.3390/app132413022

Academic Editors: Rong Gu and Mande Xie

Received: 26 September 2023
Revised: 16 November 2023
Accepted: 22 November 2023
Published: 6 December 2023

1. Introduction

The last decade has witnessed GPUs acting as a rising star in a myriad of domains, including scientific computing, big data analysis and machine learning. Programmers writing such workloads tend to offload performance-critical calculations to the GPUs [1–3] while leaving the CPUs only for control flow and inter-process communication management. These operations include general matrix multiplication (GEMM) and Convolution (Conv); both take a large portion of the layers in recent eye-catching big data and deep learning applications. Offloading GEMM and Conv operations to GPUs could obtain ten- to thousand-fold performance increases.

In certain scenarios, there is no need to use data types with a wider bitwidth in calculations. For example, in the first few loops of iterative numeric algorithms like the Euler method, it is unnecessary to use high-precision data in calculations due to the existence of a large difference between the current numeric solution and the real target. In addition, for some inference tasks like classification, the final goal is to distinguish the category corresponding to the maximal value in the score vector. Thus, using high-precision data types like `fp64` or `fp32` would be wasteful for both off-chip and on-chip resources including memory bandwidth, registers and ALU usage. Harnessing mixed-precision data (Figure 1) brings the opportunities of deploying real-time tasks on resource-limited devices [4,5].

Recently, quantized neural network models have been proposed to reduce the model size and required computation, thus improving the overall performance. From an architectural perspective, NVIDIA has embedded a transformer engine into its latest data center, the *Hopper* GPU architecture [6], to accelerate emerging transformer calculations for language models. In addition, special floating point formats like `tf16` and `tf32` (also called e8m7 and e8m10, respectively, where tf stands for tensor format) are designed to boost the quantized NN workloads as they require much lower transferring bandwidths, registers and on-chip memory resources for storage while achieving negligible errors compared with the traditional IEEE-754 version. Exploiting these new data formats in some language model workloads like BERT has achieved significant performance increases in recent NVIDIA GPUs [7].

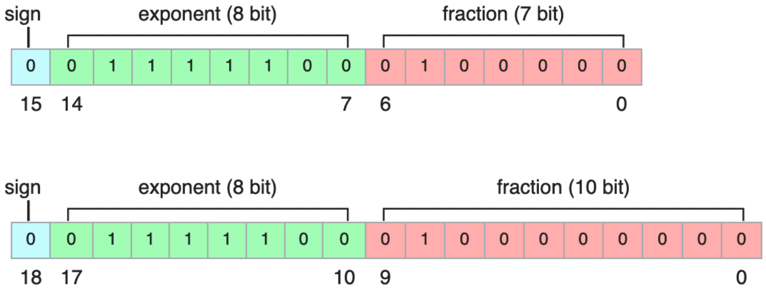


Figure 1. Floating point number representation formats of tf16 (top, also called e8m7) and tf32 (bottom, also called e8m10).

To meet these emerging requirements for both the algorithm design and the workload characteristics, hardware vendors like NVIDIA and AMD both integrate application-specific integrated circuits (ASICs) into their latest products. NVIDIA announced their first-generation Tensor Core (TC) together with the release of the *Volta* data center GPU architecture in 2017 and then revised the TC over several generations. AMD followed a technical route, naming their analogous component the Matrix Core (MC) in their MI100 series data center GPU. This algorithm–architecture co-design marked a huge success with the fact that mainstream deep learning frameworks like PyTorch have embraced these designs with the help of vendor-provided high-performance libraries like cuDNN, cuBLAS and MIOpen [8]. Other vendors like Google and Tesla have also presented proprietary ASIC accelerators like TPU [9] and Dojo [10], aiming to accelerate the quantized workloads by exploiting special hardware components to calculate low-precision types of data elements.

However, most of these vendor libraries are proprietary. NVIDIA makes them opaque to advance the first-party software ecosystem. Although AMD has made their implementation of MIOpen and the corresponding rocWMMA (Wave-level Matrix Multiply Accumulation) for the HIP software stack open-source [11], it is not well studied yet, since matrix cores are only available on its CDNA architecture realized in MI100 and later products. When using APIs from opaque libraries, the runtime will automatically select the "best" algorithm based on the user-provided problem size and data layout. This automatic mechanism works well in most cases, but outliers still exist whereby the chosen configuration is not suitable for the user's input. For instance, in big data analysis and processing, the programmer needs various custom kernels to handle incoming data streams and extract key features from them; to this end, one may wish to have a library containing kernels with higher customizability while maintaining performance with minimal programming and maintenance efforts.

Realizing the need for a community-driven high-performance GPU kernel library with user customizability, NVIDIA announced CUTLASS, an open-source codebase to provide concepts, primitives, kernels and programming models on top of CUDA. CUTLASS is a template C++ library targeting GEMM, SpGEMM (sparse GEMM) and convolution kernels that are widely used in both AI and HPC applications. With the C++ template, CUTLASS implements zero-cost abstraction and compile-time polymorphism that supports varying data types. More importantly, CUTLASS offers a system of concepts, interfaces and approaches that allows users to easily split and distribute work items into different granularities of resource organizations. Figure 2 demonstrates a representative workflow pipeline in a CUTLASS GEMM kernel. Initially, the whole problem is parallelized by splitting the output matrix into multiple tiles. Each CTA (Cooperative Thread Array, also known as a *thread block*) is responsible for loading, calculating and then storing each tile in the output matrix. Inside each CTA, 32 consecutive threads are organized as *warps* and get executed on the *SM* (Streaming Multiprocessor) in the lock-step fashion. In each iteration, each warp will try to load a tile of input matrix elements from global memory to shared memory, and synchronize at the CTA level to ensure all required data in current iteration are ready in shared memory. Then, the required CTA data in shared memory are further split into warp-level tiles to load into per-thread registers for calculation. Right after the data preparation is complete, the result tile will be computed depending on which hardware units are used (traditional CUDA Core for the SIMT fashion, Tensor Core for WMMA fashion). When the primary calculation is accomplished, some extra steps including scaling, clamping or rounding need to be conducted (called *epilogue*) before the output elements are stored back to global memory. As we can see, the whole pipeline including work item distribution and parallel execution is hierarchically organized corresponding to the hardware architecture, which is the core design of CUTLASS and other GPU-based accelerating algorithms.

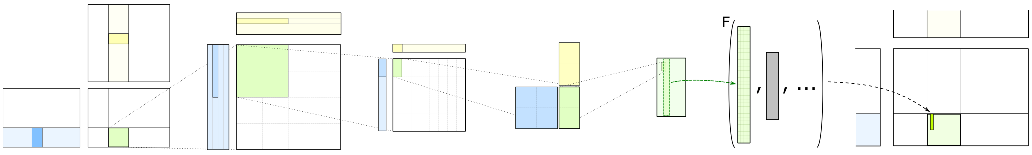


Figure 2. CUTLASS pipeline phases and design abstraction.

2. Materials and Methods

2.1. GPU Architecture and Programming Model

The concept of a general purpose GPU was introduced in 2006. Figure 3 illustrates the SM architecture in NVIDIA's *Ampere* GPU. As we can see, each SM contains four sub-processors, with integer and floating point units (CUDA Cores) inside. Besides, as aforementioned, NVIDIA has added Tensor Cores into the SM since the *Volta* generation. CTAs are scheduled to sub-processors for execution based on the granularity of warp (32 consecutive threads in CUDA). Instructions from the warp may be issued and executed on CUDA Cores, Tensor Cores or the LD/ST units to fetch data from shared memory or global memory. Warps are swapped out and switched to the stall state while they are waiting for the necessary and dependent data from lower cache levels or off-chip memory. The warp scheduler is responsible for switching between the ready and stalled warps to hide memory access latency of LD/ST instructions, making the GPU a throughput-oriented processor compared to the CPU, which is latency-oriented.

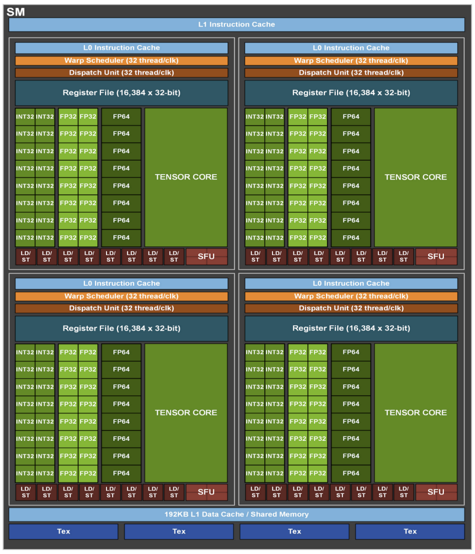

SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
LD/ST
SFU
192KB L1 Data Cache / Shared Memory
Tex

Figure 3. The architecture of SM in NVIDIA's *Ampere* GPU from NVIDIA's official whitepaper [12].

CUDA provides three warp-level collaborative APIs and specific C++ templated types for programmers to exploit the WMMA function. Listing 1 is the brief C++ statement sequence using WMMA APIs.

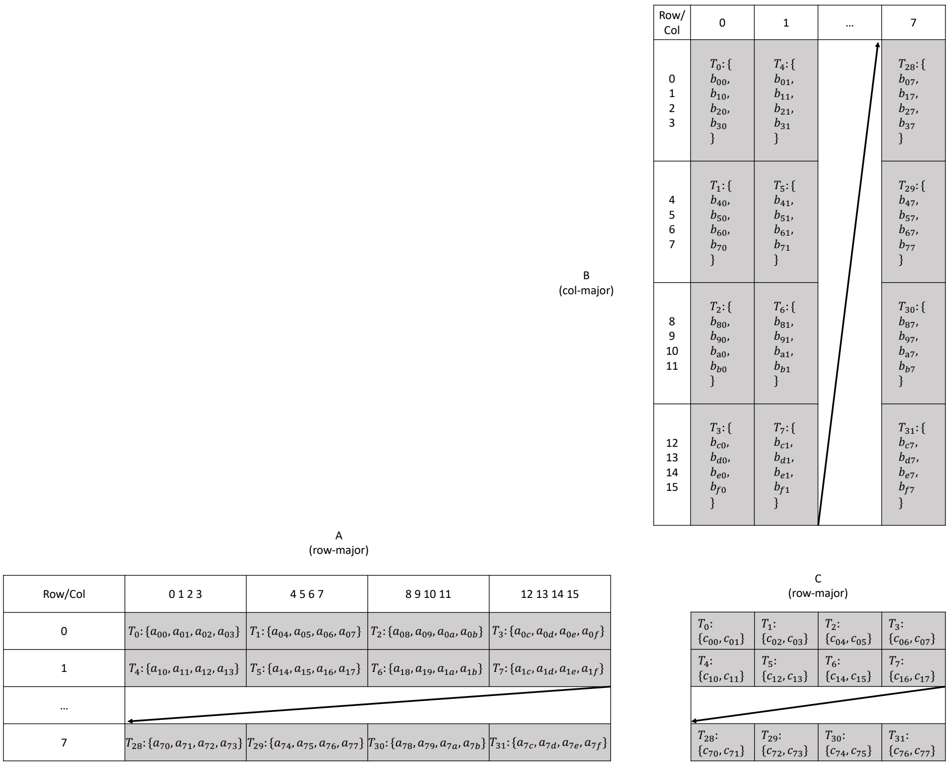
Listing 1. C++ statement using WMMA APIs.

```
// define the register fragment
wmma::fragment<wmma::matrix_a, M, N, K, half, wmma::col_major> a_frag
    ;
// load a tile of matrix A to regiter fragment
wmma::load_matrix_sync(a_frag, A, M);
// warp matrix multiply operation
wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);
// store the tile of output matrix C
wmma::store_matrix_sync(C, c_frag, N, wmma::mem_row_major);
```

The `wmma::fragment` is a templated type defined in C++ to describe the shape, layout and element type of the tile to be loaded into GPU registers. It is used to determine the number of registers required to store the loaded elements. Then, for the `wmma::load_matrix_sync` and `wmma::store_matrix_sync` APIs, they are defined to load and store the matrix tiles between global memory and registers. The `sync` suffix indicates that the substantial calculation instructions will not be issued until the dependent data are stored in the registers, compared with some asynchronous version of memory access instructions added to CUDA recently. `wmma::mma_sync` is used to conduct the $\mathbf{C} = \alpha\mathbf{AB} + \beta\mathbf{C}$ computation on the registers given by the input fragments, which is also a synchronous version.

It should be emphasized that CUDA only supports WMMA operations on some fixed shapes of matrix tiles for a specific data type. For instance, current WMMA implementation requires a 8×16 tile for matrix **A**, and 16×8 tile for matrix **B**, thus producing a 8×8 output tile for matrix **C**. Note that the tile shape constraints may vary from different generations of GPUs and for one specific data type (like `half`, i.e., `fp16`); there are more than one shape option for the register fragment.

Figure 4 illustrates the data layout and orchestration of corresponding threads. Elements from input tiles are scattered among different thread private registers; during the `mma_sync` operation, inter-thread register sharing is necessary for the Tensor Core to fetch data from different threads to produce the accumulated tile.



B
(col-major)

Row/ Col	0	1	…	7

0
1
2
3
	T_0:{
	b_{00},
	b_{10},
	b_{20},
	b_{30}
	}
		T_4:{
		b_{01},
		b_{11},
		b_{21},
		b_{31}
		}
				T_{28}:{
				b_{07},
				b_{17},
				b_{27},
				b_{37}
				}

4
5
6
7
	T_1:{
	b_{40},
	b_{50},
	b_{60},
	b_{70}
	}
		T_5:{
		b_{41},
		b_{51},
		b_{61},
		b_{71}
		}
				T_{29}:{
				b_{47},
				b_{57},
				b_{67},
				b_{77}
				}

8
9
10
11
	T_2:{
	b_{80},
	b_{90},
	b_{a0},
	b_{b0}
	}
		T_6:{
		b_{81},
		b_{91},
		b_{a1},
		b_{b1}
		}
				T_{30}:{
				b_{87},
				b_{97},
				b_{a7},
				b_{b7}
				}

12
13
14
15
	T_3:{
	b_{c0},
	b_{d0},
	b_{e0},
	b_{f0}
	}
		T_7:{
		b_{c1},
		b_{d1},
		b_{e1},
		b_{f1}
		}
				T_{31}:{
				b_{c7},
				b_{d7},
				b_{e7},
				b_{f7}
				}

A
(row-major)

Row/Col	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15
0	T_0: $\{a_{00}, a_{01}, a_{02}, a_{03}\}$	T_1: $\{a_{04}, a_{05}, a_{06}, a_{07}\}$	T_2: $\{a_{08}, a_{09}, a_{0a}, a_{0b}\}$	T_3: $\{a_{0c}, a_{0d}, a_{0e}, a_{0f}\}$
1	T_4: $\{a_{10}, a_{11}, a_{12}, a_{13}\}$	T_5: $\{a_{14}, a_{15}, a_{16}, a_{17}\}$	T_6: $\{a_{18}, a_{19}, a_{1a}, a_{1b}\}$	T_7: $\{a_{1c}, a_{1d}, a_{1e}, a_{1f}\}$
…				
7	T_{28}: $\{a_{70}, a_{71}, a_{72}, a_{73}\}$	T_{29}: $\{a_{74}, a_{75}, a_{76}, a_{77}\}$	T_{30}: $\{a_{78}, a_{79}, a_{7a}, a_{7b}\}$	T_{31}: $\{a_{7c}, a_{7d}, a_{7e}, a_{7f}\}$

C
(row-major)

T_0: $\{c_{00}, c_{01}\}$	T_1: $\{c_{02}, c_{03}\}$	T_2: $\{c_{04}, c_{05}\}$	T_3: $\{c_{06}, c_{07}\}$
T_4: $\{c_{10}, c_{11}\}$	T_5: $\{c_{12}, c_{13}\}$	T_6: $\{c_{14}, c_{15}\}$	T_7: $\{c_{16}, c_{17}\}$
T_{28}: $\{c_{70}, c_{71}\}$	T_{29}: $\{c_{72}, c_{73}\}$	T_{30}: $\{c_{74}, c_{75}\}$	T_{31}: $\{c_{76}, c_{77}\}$

Figure 4. Memory and thread layouts of matrix tiles of `m8n8k16` (i.e., 8 × 16 for tile from matrix **A**, 16 × 8 for tile from matrix **B**, and 8 × 8 for tile from the row-major output matrix **C**) for one warp. The element type of input tiles is `int8`, and the accumulated output type is `int32`. The notation $T_x : \{\cdots\}$ in the table cells represents that thread x in the warp is responsible for loading the matrix elements indicated between the brackets.

2.2. Accelerating GEMM on GPU

GEMM is a widely used operator in data intensive applications. The computing pattern of the fully connected layer in deep neural networks is GEMM in fact, and countless engineering problems depend on GEMM in their computation. GEMM is defined as follows:

$$\mathbf{D} = \alpha\mathbf{AB} + \beta\mathbf{C}, \tag{1}$$

where α and β are given scalars. In reality, the output matrix **D** is usually the same as the input matrix **C** (like the bias tensor in the fully connected layer).

There are multiple GEMM implementations designed to accelerate it in many-core architectures. One of the most common implementations in GPUs is tiled GEMM.

Figure 5 demonstrates the process of tiled GEMM implementation. As discussed in Section 1, the output matrix is firstly partitioned into multiple tiles, whose shape is empirically determined by the on-chip cache size ($Mtile \times Ntile$ in the green-colored tile in Figure 5). To obtain the results of this target tile, one has to use the corresponding tiles in both input matrices, colored as light blue and yellow in Figure 5. However, in most cases, the required memory space is far more than that of the on-chip cache capacity if we load all elements located in both input tiles onto the chip. Therefore, a common approach is to iterate both tiles at $Ktile$ step to obtain partially temporary results at each step and, finally, accumulate the overall results.

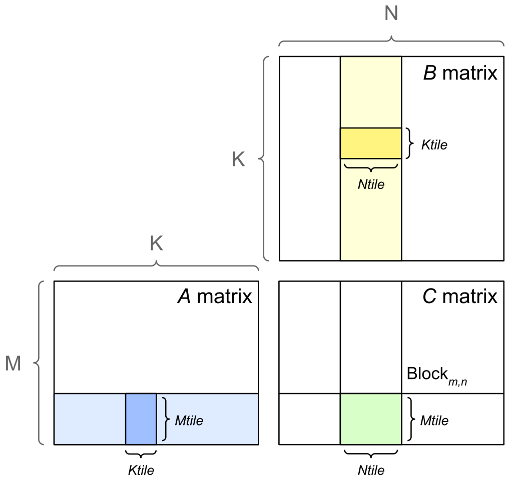


Figure 5. Demonstration of tiled GEMM implementation on GPU, from [13]. The input matrices **A** and **B** are multiplied to form the output matrix **C**, where the calculation is conducted in the tile-wise way to produce the final results. Each CTA is responsible for one output tile in matrix **C**.

Inside each tile step, the calculation as well as the matrix tile are partitioned into several iterations. For each iteration, the threads in the CTA will load the required data elements in the current iteration into shared memory to minimize off-chip memory accesses later, since there will be certain data reuse across warps within a CTA. A synchronization barrier must be inserted right after the memory access instructions to ensure all data elements required for the current iteration are residing on shared memory already to be loaded into registers later. For each iteration tile, warps are responsible for temporarily loading data elements into registers and conducting WMMA calculations to save the intermediate results in registers, then storing them back to shared memory.

After all iterations finish, the CTA should write the final results from shared memory back to global memory. Note that if multiple CTAs are responsible for the same target tile from matrix **C** (called the *split_k* mode), some extra constraints must be added to avoid data race and ensure memory consistency.

2.3. CUTLASS Design Principles

As discussed earlier, CUTLASS is a hierarchical and flexible library providing APIs abstracting at different level of granularities.

It allows programmers to make use of these APIs to assemble the high-performance kernel and express their desires for various problem sizes. There are five levels of APIs that CUTLASS provides, as shown in Table 1. In CUTLASS, users are responsible for exploiting different levels of abstractions through the APIs to assemble the desired kernels meeting both performance and customization (Listing 2). The mainloop and epilogue, as the main body of GEMM kernels, are described via the collective level APIs, which orchestrate the copy/math micro-kernels with architecture-specific synchronizations. Meanwhile, the tile APIs are compiled into core GPU micro-kernels performing substantial math/copy operations spanning cooperative threads, and the *Atom* APIs are architecture instructions associated with meta-information. We may conclude that the hierarchical CUTLASS decouples the API design: the *Collective* and *Tile* APIs aim at describing algorithm-specific CTA/warps orchestration, while the *Atom* APIs target architecture-specific instruction generation. Note that in Table 1, the *Tile* and *Atom* APIs are decorated with the `cute` namespace prefix. CuTE is a layout library introduced in the latest CUTLASS 3.0 release [14]. The purpose of this library is to define and manipulate hierarchical, multidimensional layouts of threads and data. It is worth noting that almost all of these APIs or classes are templated, desiring some compile-time known information to specialize corresponding implementations; then, they are compiled into dedicated instructions to copy or calculate the specific type.

Table 1. CUTLASS API overview.

API Level	API Class/Function
Device	`device::GemmUniversalAdapter`
Kernel	`kernel::GemmUniversal`
Collective	`collective::CollectiveMma`
Tile (MMA and copy)	`cute::TiledMma`, `cute::TiledCopy`
Atom	`cute::Mma_Atom`, `cute::Copy_Atom`

Like other numeric or operator libraries, CUTLASS allows users to call GEMM or any other kernels from the host. A typical procedure of producing the GEMM results of `cutlass::half_t` (i.e., `fp16`) as input type is presented below:

Listing 2. A typical procedure producing the GEMM results.

```
using ElementA = cutlass::half_t;
using LayoutA = cutlass::layout::RowMajor;
constexpr int AlignmentA = 128 / cutlass::sizeof_bits<ElementA>::
    value;
// Same as matrices B and C
using ElementAccumulator = cutlass::half_t;
using ArchTag = cutlass::arch::Sm80;
using OperatorClass = cutlass::arch::OpClassTensorOp;
using ThreadblockShape = cutlass::gemm::GemmShape<128, 128, 32>;
using WarpShape = cutlass::gemm::GemmShape<64, 64, 32>;
using InstructionShape = cutlass::gemm::GemmShape<16, 8, 16>;
constexpr int NumStages = 4;
using EpilogueOp =
    cutlass::epilogue::thread::LinearCombination<
    ElementC, AlignmentC, ElementAccumulator, ElementAccumulator>;
// Classic data-parallel device GEMM implementation type
using DeviceGemmBasic =
    cutlass::gemm::device::GemmUniversal<ElementA, LayoutA, ElementB,
         LayoutB, ElementC, LayoutC, ElementAccumulator, OperatorClass
        , ArchTag, ThreadblockShape, WarpShape, InstructionShape,
        EpilogueOp,
cutlass::gemm::threadblock::
GemmIdentityThreadblockSwizzle<>, NumStages, AlignmentA, AlignmentB>;
```

As we claimed before, CUTLASS is a highly templated library; hence, nearly all of its data types and interfaces are specified with templates. The matrix is defined by the tuple `<Element, Layout>` and `AlignmentA`, and it is used to determine the memory access granularity to the corresponding memory region. Since CUDA now supports 128-bit coalesced memory access (using `LDG.128` SASS instruction), the number of elements per instruction loads is thus determined by `128/sizeof_bits(Element)`. Next, some required information is defined such as the MMA operation type (whether CUDA Core or Tensor Core will be used) and the underlying GPU hardware architecture tag. Lastly, the hierarchical tile shapes assigned to different levels of the collaboration group (thread block, warp or per MMA) are defined with the `GemmShape` template type. The epilogue operation is the auxiliary phase after the main loop of kernel for the scalar calculation or type conversion. Note that asynchronous global memory to shared memory instruction (`LDG.STS` SASS instruction) is proposed since Ampere generation; therefore, this producer–consumer pattern is pipelined and several memory access stages are exploited to overlap the MMA operation and global memory access. The number of pipelined stages is defined by `NumStages`. All the above information is then passed to GEMM kernel templates to generate the target code.

There are several advantages of encoding key information (such as element types, tile shapes and memory access granularities) as template parameters. For example, if the tile shape is fixed at the compile-time, register spills could be avoided when indexing tile elements. Besides, loop unrolling could be performed accordingly to saturate the instruction cache. Further, with this compiler–assistant information, register usage could be reduced since some variables are encoded as constants and part of input data could be fetched from CUDA constant memory to mitigate the DRAM bandwidth.

3. Results and Discussion

In this section, we demonstrate the key insights discovered from the benchmarks.

3.1. Testbed

Our experiments are conducted with NVIDIA A100 40 GB PCIe version. There are 108 Tensor Core SMs and 40 MB L2 cache total, with 1555 GB/s off-chip memory bandwidth. On the software side, we built CUTLASS 3.0 with host compiler `gcc@10.4.0` and CUDA version ‘11.4.0’ under `Debian` 11 as the host OS. All measured kernels were warmed up with 10 repeated runs and invoked 20 times to obtain the average execution elapsed times. The standard deviations of all kernel executions are less than 5% and the results are stable. The theoretical peak performance for various kinds of data formats on different hardware units is listed in Table 2.

Table 2. Theoretical peak performance of NVIDIA Tesla A100 PCIe version for different data formats and the underlying hardware units.

Data Format	Peak Performance (TFLOPS)
`fp64`	9.7
`fp64` Tensor Core	19.5
`fp32`	19.5
`tf32`	156
`fp16` Tensor Core	312
`int8` Tensor Core	624

Theoretically, we can conclude that exploiting ASIC hardware units in the GPU could dramatically boost the performance for low-precision data formats. However, the substantial speedup may be restricted by some other factors, like limitations of CTA slots per SM, shared memory usage and memory access latency. Thus, benchmarks of GEMM on off-the-shelf commercial GPUs are necessary for engineers to assess the Tensor Core benefits.

Before we continue to dig out the performance insight brought by the Tensor Core benchmarks, it should be pointed out that the CUTLASS naming conventions indicate the implementation details of kernels. Here is a GEMM kernel example:

```
cutlass_tensorop_s1688gemm_128x128_16x4_tt_align4
```

Fields in the kernel name are separated by underlines. `tensorop` means the underlying hardware component used for the primary MMA operations is Tensor Core (as for CUDA Core, the corresponding name is `simt`). The `s` in the next field means the Tensor Core MMA instructions are targets for *single* floating point numbers (`d` for double floating point numbers and `h` for half types). Then, `1688` indicates that the warp-level MMA tile shape is $16 \times 8 \times 8$—i.e., $m = 16, n = 8, k = 8$—within each CTA tile. The `128x128_16` represents the CTA tile shape ($M \times N \times K$) is $128 \times 128 \times 16$, and the trailing `4` is the number of pipeline stages for asynchronous global memory to shared memory copies (`LDG.STS`). The last field, `align4`, indicates that the maximum alignment between operands **A** and **B** is 4.

3.2. Tensor Core vs. CUDA Core

We first compare GEMM kernels in the cases of with and without Tensor Core involved. The runtime comparisons are demonstrated in Figure 6. As we can see, the GFLOPS of GEMM kernels with Tensor Core are much higher than those without Tensor Core.

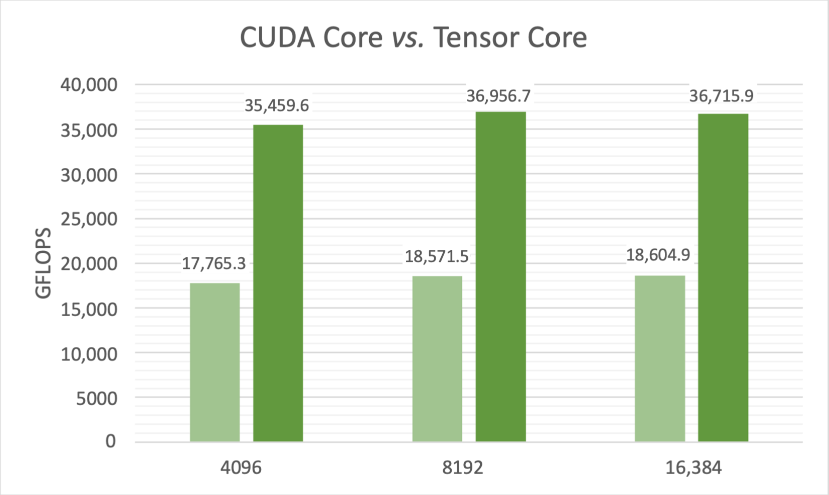


Figure 6. Comparison among kernels with and without Tensor Core, where both input and output types are `fp32`. Performance is measured by GFLOPS, which is a division of the total number of floating operations and the kernel duration.

The average speedup between kernels with and without Tensor Core is 1.98. It should be pointed out that some implementation details vary between these two kernels, including the CTA tile shape and the asynchronous shared memory fetching pipeline stages.

Putting together Table 2 and Figure 6, we can figure out that the achieved `fp32` GEMM performance via CUTLASS is about 95% of the peak `fp32` performance, which is often regarded as a satisfying implementation.

3.3. Data Formats

In this section, we mainly focus on the varied performance when applying CUTLASS GEMM kernels to different data types.

As Figure 7 illustrates, there is a obvious trend that as the bitwidth of element type decreases, the achieved GEMM performance increases correspondingly. This is reasonable since the high-precision element type requires more bandwidth as well as cycles to execute the wider MMA instructions on Tensor Core. Another observation is that the achieved performance of extreme low-precision data types like `u8` (unsigned 8-bit integer), and even `s4` and `u4`, is much higher than that of conventional data types, which leads to the wide use of such types in deep learning model inferences.

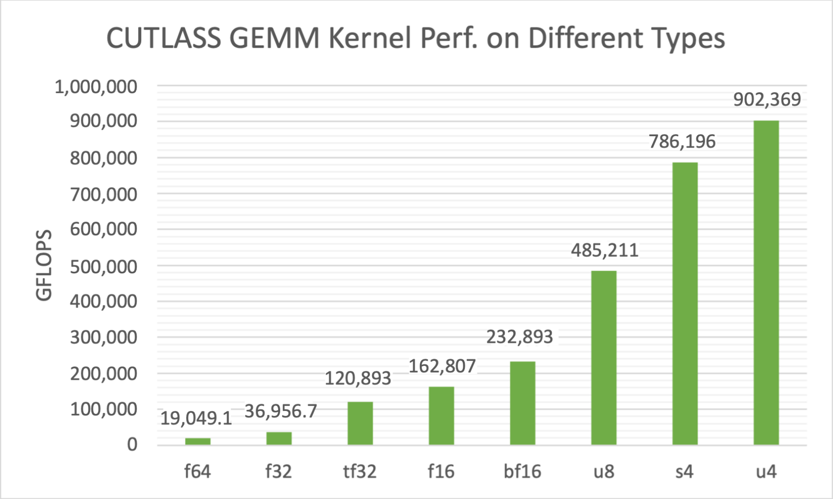


Figure 7. CUTLASS GEMM kernel performance comparisons on different input element types. The GEMM problem size is fixed as $M = N = K = 8192$.

3.4. Tile Shapes and Split_k

In this section, we explore the relationship between the achieved performance, the CTA tile shape and orchestration. We first compare two GEMM kernels under problem size $M = N = K = 8192$ and `fp32` inputs. Configurations of the inspected kernels remain the same, except for the CTA shape: one is `128x128_16` and the other is `256x128_16`. The performance metrics, measured in GFLOPS, obtained for both kernels are 36,956.7 and 115,962, respectively. There is a significant performance gap between these two kernels, even though almost all kernel parameters are identical except the CTA tile shape. As the thread block size (i.e., the number of threads in one CTA) and the orchestration of warps along each tile dimension are the same as well, we can conclude that configuring the CTA tile shape correctly has a huge impact on overall kernel performance.

Another performance-impacting factor that deserves inspection is the *split_k* setting, which controls the number of CTAs working on the same tile of the output matrix. *split_k* is introduced to tackle the problem that for some small GEMM workloads, the one-CTA-one-tile strategy will not saturate the entire GPU SM resources, extremely for cases with small M and N but large K problem sizes. For example, for a 1024 × 1024 output matrix, the abovementioned CTA orchestration with 256 × 128 tile shape strategy ends up with only $(1024/256) \times (1024/128)$ CTAs for this kernel, while there are 108 SMs in the A100 GPU. The number of fetch–calculate–store iterations required for each CTA to loop over alongside the K dimension is K/k, where K is the reduced dimension of input matrices and k is the corresponding CTA tile dimension. To fully utilize the GPU resource and reduce the workload assigned to each CTA, one can parallelize the execution at the dimension K by orchestrating more than one CTA on the same tile of the output matrix. Thus, loop iterations required for that tile could be evenly distributed among multiple CTAs and, for each CTA, the number of iterations would then be reduced by *split_k* times.

Table 3 shows the achieved performance for `fp32` GEMM kernels with a varying *split_k* parameter and with $M = N = 512$ and $K = 8192$ problem size. As we can see, as the value of *split_k* factor increases, the performance of the obtained kernel boosts as well. However, there is a kneepoint where increasing the *split_k* value will harm the overall performance. It should be emphasized that enabling parallel *split_k* mode with its value larger than one needs one extra reduction kernel to aggregate temporary results from different CTAs and the corresponding predefined global buffers into the final output tile destination. Hence, the larger *split_k* is, the longer the reduction latency will be.

Table 3. Achieved performance for `fp32` GEMM kernel with the same compile-time configurations, except for the runtime-determined *split_k* parameter.

Split_k	Achieved Performance (TFLOPS)
1	5641.83
2	10,641.9
4	17,577.3
8	20,913.4
10	19,182.4

3.5. Asynchronous Copy

To implement a performant GEMM kernel, software pipelining is required to hide the global-to-shared memory access latency as much as possible. Asynchronous copy is a new hardware feature proposed in Ampere that allows programmers to explicitly invoke non-blocking global-to-shared memory access with no extra registers required for storing temporarily accessed data.

Table 4 lists the relative speedup for kernel `s1688gemm_f16_256x128_32x2` gained via applying asynchronous global-to-shared memory copies. Observation shows that enabling this new hardware feature could bring averaged 1.195 speedups over different matrix sizes with `fp16` input and `fp32` output. Enabling async. copy brings more flexibility for the SM scheduler since Tensor Cores only accelerate the computation while the off-chip bandwidth remains the same. It offers the opportunities to pipeline and overlap the computation and memory access.

Table 4. Speedups obtained via asynchronous global-to-shared memory copies.

M, N, K	w/o Async. Copy	w/ Async. Copy	Speedup
4096	103,483	127,160	1.228
8192	110,038	136,912	1.244
16,384	122,674	139,075	1.113

4. Related Works

Tensor Core has been a hot topic since the release of its first generation. Works [15–18] conducted micro-benchmarks to demonstrate the underlying mechanisms of Tensor Core, including the SASS instruction of `mma_sync`, the cost cycles for each MMA instruction as well as the tile orchestrations for each thread within one warp. Zhao et al. [19] pointed out the phenomenon that the utilization of CUDA Core and Tensor Core could not be high simultaneously; therefore, they proposed a source-to-source compiler that transforms a common kernel into a persistent SM-centric kernel, allowing workloads using Tensor Core or CUDA Core to be scheduled into the same SM. SIMD2 [20] used off-the-shelf commercial GPUs to emulate general purpose computation on the dedicated Tensor Core, revealing the potential of extending the calculation capacity for a wider range of workloads. Likewise, PET [21] proposed a transformation that allows partial equivalent operations and an automatic correction mechanism to ensure that the results are acceptable for tensor programs. Another direction is to fuse isolated kernels utilizing the CUDA Core or Tensor Core separately into monolithic macro-kernels and integrating architecture supports in hardware for this change [22]. There are also some workload-specific optimizations exploiting Tensor Core to accelerate the computation on GNN or sparse matrix operations [23,24].

GEMM is a well-studied workload accelerated by GPUs, which has many adoptions in deep learning and scientific computing applications. There are some works focusing on exploiting Tensor Core to accelerate GEMM while maintaining the precision requirements [25]. Also, Mix-GEMM [26] explores the usage of GEMM on deep learning model inference running on edge devices. The design of workload distribution on GEMM is also studied to better utilize the GPU on-chip resources [27]. Besides, other specialized linear algebra operations like SYMM and TRMM [28] can be implemented and optimized with CUTLASS.

GPU operator tuning is the process of finding the most suitable tiling size and other kernel configurations like the number of pre-fetching stages for asynchronous copy. Nowadays, machine learning compilers like TVM [29] have enabled us to seek out the optimal solution with automatic search, but the search space is still to large for an exhaustive traverse. To bridge the gap between performant but less flexible vendor-provided libraries (like cuDNN) and highly customizable but less optimal libraries like CUTLASS, Bolt [30] was proposed to integrate CUTLASS into TVM as the backend of the search process for high-quality code generation. Besides, Graphene [31] was proposed to serve as an IR to minimize the effort for kernel experts to express tensor operations at a low level without losing general expression ability.

5. Conclusions

In this paper, we first illustrated the recent trend of emerging quantized calculation workload and architectural ASIC as the corresponding response from the GPU hardware vendors. Then, we briefly introduced the CUTLASS open-source templated library and its design principles for GEMM kernels. With CUTLASS, we built and conducted a series of benchmarks, revealing some insights and factors that will greatly impact the achieved kernel performance. To obtain a satisfying portion of theoretical peak performance, manual tuning is still necessary for the dedicate kernels.

Author Contributions: Conceptualization, X.H. and X.Z.; methodology, X.H.; software, X.H.; validation, X.H. and X.Z.; formal analysis, X.H.; investigation, X.H.; resources, X.Z. and N.X.; data curation, X.H.; writing—original draft preparation, X.H.; writing—review and editing, X.Z.; visualization, X.H.; supervision, X.Z. and N.X.; project administration, X.Z. and P.Y.; funding acquisition, X.Z., P.Y. and N.X. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the the National Natural Science Foundation of China—#62102465, the Major Program of Guangdong Basic and Applied Research NO. 2019B030302002, the Funding by Science and Technology Projects in Guangzhou—#202201011241, and the Open Project of China Electronic Product Reliability and Environmental Testing Research Institute (CEPREI)-#HK202201334.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: This research does not rely on extra datasets as the inputs to the CUTLASS kernels are randomly formed. The obtained performance data are demonstrated in Section 3.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Zhao, G.; Sun, N.; Shen, S.; Wu, X.; Wang, L. GPU-Accelerated Target Strength Prediction Based on Multiresolution Shooting and Bouncing Ray Method. *Appl. Sci.* **2022**, *12*, 6119. [CrossRef]
2. Liu, D.; Li, B.; Liu, G. Calculation of Surface Offset Gathers Based on Reverse Time Migration and Its Parallel Computation with Multi-GPUs. *Appl. Sci.* **2021**, *11*, 10687. [CrossRef]
3. Golosio, B.; Villamar, J.; Tiddia, G.; Pastorelli, E.; Stapmanns, J.; Fanti, V.; Paolucci, P.S.; Morrison, A.; Senk, J. Runtime Construction of Large-Scale Spiking Neuronal Network Models on GPU Devices. *Appl. Sci.* **2023**, *13*, 9598. [CrossRef]
4. Kim, S.; Cho, J.; Park, D. Moving-Target Position Estimation Using GPU-Based Particle Filter for IoT Sensing Applications. *Appl. Sci.* **2017**, *7*, 1152. [CrossRef]
5. Nguyen, D.V.; Choi, J. Toward Scalable Video Analytics Using Compressed-Domain Features at the Edge. *Appl. Sci.* **2020**, *10*, 6391. [CrossRef]
6. Transformer Engine Documentation—Transformer Engine 0.6.0 Documentation. Available online: https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/index.html (accessed on 15 November 2023).

7. Kharya, P. NVIDIA Blogs: TensorFloat-32 Accelerates AI Training HPC Upto 20x. 2020. Available online: https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/ (accessed on 15 November 2023).
8. Khan, J.; Fultz, P.; Tamazov, A.; Lowell, D.; Liu, C.; Melesse, M.; Nandhimandalam, M.; Nasyrov, K.; Perminov, I.; Shah, T.; et al. MIOpen: An Open Source Library For Deep Learning Primitives. *arXiv* **2019**, arXiv:1910.00078.
9. Jouppi, N.P.; Kurian, G.; Li, S.; Ma, P.; Nagarajan, R.; Nai, L.; Patil, N.; Subramanian, S.; Swing, A.; Towles, B.; et al. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. *arXiv* **2023**, arXiv:2304.01433.
10. Lambert, F. Tesla Unveils New Dojo Supercomputer so Powerful It Tripped the Power Grid. 2022. Available online: https://electrek.co/2022/10/01/tesla-dojo-supercomputer-tripped-power-grid/ (accessed on 15 November 2023).
11. rocWMMA. 2023. Available online: https://github.com/ROCmSoftwarePlatform/rocWMMA (accessed on 15 November 2023).
12. NVIDIA Ampere Architecture. 2020. Available online: https://images.nvidia.cn/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf (accessed on 15 November 2023).
13. Matrix Multiplication Background User's Guide. Available online: https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html (accessed on 15 November 2023).
14. CUTLASS 3.0 Is Now Available! · NVIDIA/Cutlass · Discussion #787. Available online: https://github.com/NVIDIA/cutlass/discussions/787 (accessed on 15 November 2023).
15. Jia, Z.; Maggioni, M.; Staiger, B.; Scarpazza, D.P. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *arXiv* **2018**, arXiv:cs/1804.06826.
16. Jia, Z.; Maggioni, M.; Smith, J.; Scarpazza, D.P. Dissecting the NVidia Turing T4 GPU via Microbenchmarking. *arXiv* **2019**, arXiv:1903.07486.
17. Yan, D.; Wang, W.; Chu, X. Demystifying Tensor Cores to Optimize Half-Precision Matrix Multiply. In Proceedings of the 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), New Orleans, LA, USA, 18–22 May 2020; pp. 634–643. [CrossRef]
18. Sun, W.; Li, A.; Geng, T.; Stuijk, S.; Corporaal, H. Dissecting Tensor Cores via Microbenchmarks: Latency, Throughput and Numeric Behaviors. *IEEE Trans. Parallel Distrib. Syst.* **2023**, *34*, 246–261. [CrossRef]
19. Zhao, H.; Cui, W.; Chen, Q.; Zhao, J.; Leng, J.; Guo, M. Exploiting Intra-SM Parallelism in GPUs via Persistent and Elastic Blocks. In Proceedings of the 2021 IEEE 39th International Conference on Computer Design (ICCD), Storrs, CT, USA, 24–27 October 2021; pp. 290–298. [CrossRef]
20. Zhang, Y.; Tsai, P.A.; Tseng, H.W. SIMD^2: A Generalized Matrix Instruction Set for Accelerating Tensor Computation beyond GEMM. In Proceedings of the 49th Annual International Symposium on Computer Architecture, ISCA '22, New York, NY, USA, 18–22 June 2022; pp. 552–566. [CrossRef]
21. Wang, H.; Zhai, J.; Gao, M.; Ma, Z.; Tang, S.; Zheng, L.; Li, Y.; Rong, K.; Chen, Y.; Jia, Z. PET: Optimizing Tensor Programs with Partially Equivalent Transformations and Automated Corrections. In Proceedings of the USENIX Symposium on Operating Systems Design and Implementation, Virtual, Online, 14–16 July 2021.
22. Zhao, H.; Cui, W.; Chen, Q.; Zhang, Y.; Lu, Y.; Li, C.; Leng, J.; Guo, M. Tacker: Tensor-CUDA Core Kernel Fusion for Improving the GPU Utilization While Ensuring QoS. In Proceedings of the 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA), Seoul, Republic of Korea, 2–6 April 2022; pp. 800–813. [CrossRef]
23. Wang, Y.; Feng, B.; Ding, Y. QGTC: Accelerating Quantized Graph Neural Networks via GPU Tensor Core. In Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '22, New York, NY, USA, 2–6 April 2022; pp. 107–119. [CrossRef]
24. Li, S.; Osawa, K.; Hoefler, T. Efficient Quantized Sparse Matrix Operations on Tensor Cores. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '22, Dallas, TX, USA, 13–18 November 2022; pp. 1–15.
25. Feng, B.; Wang, Y.; Chen, G.; Zhang, W.; Xie, Y.; Ding, Y. EGEMM-TC: Accelerating Scientific Computing on Tensor Cores with Extended Precision. In Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '21, New York, NY, USA, 27 February 2021; pp. 278–291. [CrossRef]
26. Reggiani, E.; Pappalardo, A.; Doblas, M.; Moreto, M.; Olivieri, M.; Unsal, O.S.; Cristal, A. Mix-GEMM: An Efficient HW-SW Architecture for Mixed-Precision Quantized Deep Neural Networks Inference on Edge Devices. In Proceedings of the 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA), Montreal, QC, Canada, 25 February–1 March 2023 ; pp. 1085–1098. [CrossRef]
27. Osama, M.; Merrill, D.; Cecka, C.; Garland, M.; Owens, J.D. Stream-K: Work-Centric Parallel Decomposition for Dense Matrix-Matrix Multiplication on the GPU. In Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP '23, New York, NY, USA, 25 February–1 March 2023; pp. 429–431. [CrossRef]
28. PolyBench. Available online: https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/ (accessed on 15 November 2023).
29. Chen, T.; Moreau, T.; Jiang, Z.; Zheng, L.; Yan, E.; Shen, H.; Cowan, M.; Wang, L.; Hu, Y.; Ceze, L.; et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), Carlsbad, CA, USA, 8–9 October 2018; pp. 578–594.

30. Xing, J.; Wang, L.; Zhang, S.; Chen, J.; Chen, A.; Zhu, Y. Bolt: Bridging the Gap between Auto-tuners and Hardware-native Performance. *Proc. Mach. Learn. Syst.* **2021**, *4*, 204–216.
31. Hagedorn, B.; Fan, B.; Chen, H.; Cecka, C.; Garland, M.; Grover, V. Graphene: An IR for Optimized Tensor Computations on GPUs. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2023, Vancouver, BC, Canada, 25–29 March 2023; Aamodt, T.M., Jerger, N.D.E., Swift, M.M., Eds.; ACM: New York, NY, USA, 2023; Volume 3, pp. 302–313. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.