Matched Filtering Accelerated by Tensor Cores on Volta GPUs With Improved Accuracy Using Half-Precision Variables

Takuma Yamaguchi, Tsuyoshi Ichimura, Kohei Fujita, Aitaro Kato, and Shigeki Nakagawa

***Abstract*—Matched Filtering can be applied to various fields owing to its ability to compute a correlation coefficient of two vectors and detect many template events. With an improvement in observation techniques, massive observation data and templates have been accumulated, in which a reduction of computation cost of Matched Filtering has become an important issue. This computation is mainly matrix-matrix product and Tensor Core on NVIDIA Volta GPU is expected to compute it rapidly. However, actual performance of Tensor Core is usually limited by the bandwidth of shared memory or global memory. In addition, only lower-precision data types are supported in the current API for Tensor Core. Therefore, we have to prevent a decline in accuracy in the computation. In this letter, we designed a Matched Filtering algorithm to solve these problems mentioned above and utilized high arithmetic capacity on Tensor Core. Specifically, we reduced the number of memory access to global memory and shared memory by using low-level description. In addition, we introduced local normalization to reduce the numerical error. We applied our developed kernel to template matching of seismic observation data and compared the performance and the accuracy with cuBLAS, a common library in GPU computation. When we compared the performance with the function in cuBLAS that offered almost the same accuracy as our kernel, we reduced the elapsed time by a factor of 4.74.**

***Index Terms*—Matched Filtering, GPU computation, Tensor Core, half precision arithmetic.**

I. Introduction

MATCHED Filtering [1] is a process of detecting specific pattern in a wave with noise, and it has been applied to various fields, which include signal detection of radar [2], detection of gravitational waves [3], and detection of earthquake events [4]. With the improvement of measurement technology, massive observation data have been accumulated; thus, reduction of the computation cost in Matched Filtering becomes an important issue. Methods using GPUs are proposed by [5]; however, knowledge based on latest computer architectures can achieve further speeding-up.

Recently, NVIDIA Volta GPU [6] has Tensor Core [7] for acceleration of dense matrix-matrix multiplication, which is one of the biggest features of the architecture. Two problems to accelerate computation with Tensor Core are identified. The first point is that the performance is often memory bandwidth bound as Tensor Core has extremely high peak theoretical performance. The second point is that current Tensor Core supports only lower precision data types, i.e., a 16-bit floating point number, an 8-bit integer, and an 1-bit integer. For example, the 16-bit floating point number is unable to guarantee the accuracy of more than 4 digits. In some specific fields where very high accuracy are not required, it is easy to apply these data types; however, it is challenging to apply them in general numerical simulations. Numerical error in Matched Filtering can lead to detection of unnecessary events or overlook of events; thus, the effect of numerical error should be minimized. If we design the algorithm which satisfies conditions mentioned above, we can achieve benefits of very high performance by Tensor Core operations.

This letter proposes an algorithm to accelerate the core computation in Matched Filtering using Tensor Core with 16-bit floating point number. We issue Tensor Core operations with lower memory access cost. Besides, we locally normalize the components of matrices to reduce the effect of using lower precision data types. We demonstrate that our algorithm attains a reasonable speeding up and improvement in accuracy when compared to cuBLAS [11], the common library for Tensor Core. Matched filtering is mathematically a normalized 1D convolution, so our approach can be beneficial for other implementations targeting convolutional neural networks [8].

The rest of this letter is organized as follows: Section II describes our proposed algorithm. Section III describes the performance measurement using seismic observation data.

II. Methodology

Matched Filtering detects waves similar to templates from the observation data by calculating correlation coefficient as follows:

$$CC(i,j) = \frac{\sum_{k=1}^{K} T_j(k) S(k+i)}{\sqrt{\sum_{k=1}^{K} T_j^2(k) \sum_{k=1}^{K} S^2(k+i)}}, \qquad (1)$$

where T_j is the j-th template wave, S is the observation wave, i is the initial time step of clipped observation wave, and K is the length of template wave. We focused on the computation in the time domain while the computation in the frequency domain is also available. When computing $CC(i,j)$ for many templates and for many time steps, calculation of the dot product in the

Manuscript received July 12, 2019; revised September 20, 2019; accepted October 20, 2019. Date of publication November 4, 2019; date of current version December 5, 2019. This work was supported in part by the Japan Society for the Promotion of Science under Grant 18H05239. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Shiwen He. *(Corresponding author: Takuma Yamaguchi.)*

The authors are with the Earthquake Research Institute, The University of Tokyo, Tokyo 113-0032, Japan (e-mail: yamaguchi@eri.u-tokyo.ac.jp; ichimura@eri.u-tokyo.ac.jp; fujita@eri.u-tokyo.ac.jp; akato@eri.u-tokyo.ac.jp; nakagawa@eri.u-tokyo.ac.jp).

Digital Object Identifier 10.1109/LSP.2019.2951305

```
__device__ void tensor_core_matmul(
   half *amat, half *bmat, float *cmat) {
  wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::col_major> A;
  wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::col_major> B;
  wmma::fragment<wmma::accumulator, 16, 16, 16, float> C;
  wmma::load_matrix_sync(A, amat, 16);
  wmma::load_matrix_sync(B, bmat, 16);
  wmma::mma_sync(C, A, B, C);  // C+=A*B
  wmma::store_matrix_sync(cmat, C, 16, wmma::row_major); }
```

Fig. 1. A common usage of Tensor Core operations by calling wmma API in CUDA C. Each load_matrix_sync distributes a matrix on shared memory to registers for Tensor Core multiplication.

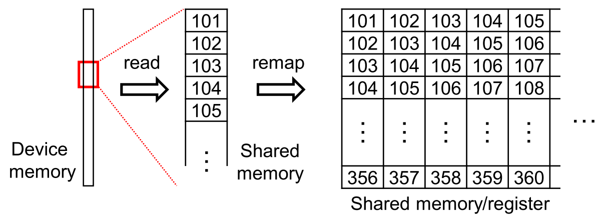

Fig. 2. Memory transaction for an observation matrix. The matrix has many duplicated components. Shared memory is used as a buffer, and components of the actual matrix are read from shared memory.

numerator of Eq. (1) accounts for the largest computation cost. These dot products are a matrix-matrix multiplication when we calculate them with many i and j at the same time. Therefore, we can introduce Tensor Core operations for this computation.

In this letter, we use warp matrix multiply-accumulate (wmma) API [7] for Tensor Core operations to optimize the performance and improve the accuracy using low-level descriptions. This API facilitates the computation of the two 16×16 matrices multiplication using 32 threads as shown in Fig. 1.

As GPU has high peak theoretical performance, we must provide data to cores rapidly to prevent memory bandwidth from binding the performance.

In matrix-matrix multiplications, it is efficient to use shared memory as a buffer and to reduce the amount of memory access to global memory. We assume that K is at most 256; therefore, we calculate a correlation coefficient by conducting 16 multiplications of 16×16 matrices. As the number of templates depends on problems, we construct our algorithm to compute correlation coefficients for 16 templates at a time, which is the smallest configuration for Tensor Core operations. We assign a 16×256 template matrix and a $256 \times N$ matrix to each thread block in computing on GPU. While we have to read all components for template matrices, we can reduce the memory access cost for reading observation matrix by applying the same method as [5]. Observation matrix has duplicated components as the matrix consists of multiple observation vectors which slide initial time steps. Given the specific characteristic in this problem, we store observation data required for each thread block in shared memory as described in Fig. 2. Then, we reduce the number of memory access to global memory. The number of components of the observation matrix per thread block is $N \times 256$, proportional to the floating operation counts in the multiplication. To generate this observation matrix, we require $N + 255$-time steps of observation data. Since the size of the observation matrix for each thread block N increases, higher performance is expected as the amount of memory access per

```
asm("{¥n¥t"
  ".reg .f32 f<1>;¥n¥t"
  "mov.f32 f0, 0f00000000;¥n¥t"
  "wmma.mma.sync.aligned.col.col.m8n32k16.f32.f32
   {%0,%1,%2,%3,%4,%5,%6,%7}, {%8,%9,%10,%11,%12,%13,%14,%15},
   {%16,%17,%18,%19,%20,%21,%22,%23}, {f0,f0,f0,f0,f0,f0,f0,f0};¥n¥t" "}"
  :"=f"(c[0]),"=f"(c[1]),"=f"(c[2]),"=f"(c[3]),"=f"(c[4]),"=f"(c[5]),"=f"(c[6]),"=f"(c[7])
  :"r"(a[0]),"r"(a[1]),"r"(a[2]),"r"(a[3]),"r"(a[4]),"r"(a[5]),"r"(a[6]),"r"(a[7]),
   "r"(b[0]),"r"(b[1]),"r"(b[2]),"r"(b[3]),"r"(b[4]),"r"(b[5]),"r"(b[6]),"r"(b[7]));
```

Fig. 3. A simple example to call wmma API from PTX assembly in CUDA C. 32 components of input 16×16 matrices in half precision and 8 components of output 16×16 matrices in single precision are assigned per thread as input arrays a and b and an output array c, respectively, as explained in [9].

the computation is reduced. However, the memory resources per thread increase, and it is harder to overlap latencies in the computation owing to a decline in the number of available threads. We decide an appropriate matrix size through a performance comparison of different matrix size. Our algorithm can lead to a reduction in access cost of global memory.

However, computation using Tensor Core tends to be a shared memory bandwidth bound for the following reasons. Tensor Core requires cooperation of 32 threads for one matrix-matrix multiplication. Here components of the matrices must be stored in registers of the corresponding thread. Then, the data mapping between threads is required before and after Tensor Core operations. This mapping is so complex that wmma API provides functions to map values of registers. These functions are using shared memory for the data distribution, that is, transferring data between shared memory and registers. If we issue these functions too frequently, the amount of memory access to shared memory increases. To exhibit high peak performance on Tensor Core, we must reduce the memory access mentioned above.

We construct observation matrix reading values in shared memory. The components of the matrix in shared memory are transferred to registers using load_matrix_sync function of wmma API as shown in Fig. 1; however, we can map values to registers based on the distribution of thread mapping as analyzed by [9]. Thus, we use PTX assembly to pass the variables, required for each thread, for the Tensor Core operations, which is difficult to implement by CUDA C. Fig. 3 shows an example of PTX assembly in CUDA C. This implementation has skipped the mapping of matrices from shared memory to registers that would occurred when we used the function load_matrix_sync in wmma API as described in Fig. 1, so very high peak performance on Tensor Core can be utilized since the memory access cost in shared memory as well as the memory access cost in global memory is reduced.

In the Tensor Core multiplication, 16×16 input matrices are in half precision whereas the output matrix is stored in single precision to avoid numerical error in the summation of the results. Furthermore, we introduce localized normalization to reduce numerical error when input matrices are converted into half precision. Template matrices are normalized per 16 components, and the observation matrix is normalized for the components stored in shared memory, as shown in Fig. 4. When we call the kernel, template waves are stored in half precision and the observation wave is stored in single precision. The observation data are converted into half precision with local normalization in each thread block. In our computations, thread block includes only 32 threads; thus normalizations including searches of the maximum value can be computed only by using

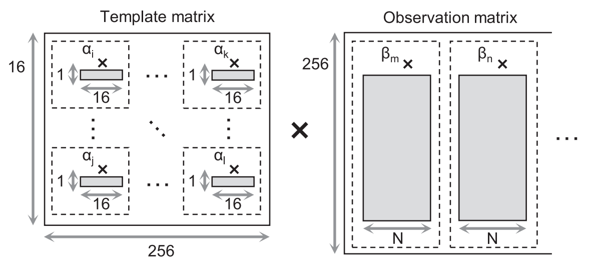

Fig. 4. Rough scheme of normalization. α and β are single-precision scaling factors for matrices surrounded by dash lines.

warp shuffle functions. We must add results of 16×16 matrix multiplication in single precision after reflecting the values of scaling factors, involved in the template matrices. Components of 16×16 matrix are distributed among registers in 32 threads and usage of shared memory via the function load_matrix_sync makes it easy to identify rescaling factors; however, data transfer between shared memory and registers for rescaling also decreases the performance. Therefore, we have to rescale the results on registers. We specify which scaling factor must be multiplied with registers, considering distribution of matrix and using PTX assembly. It is not until we introduce low-level description considering register allocation between threads that we can carry out our fine normalization without decreasing the performance greatly.

III. Application Example

We apply our proposed kernel to the seismic waveforms and evaluate the performance and the accuracy via comparison with a common library. In recent years, nation-wide seismic observation networks have been operated (e.g., MOWLAS [10] in Japan) with a lot of continuously recorded data; besides, the amount of data is expected to increase. For instance, MOWLAS is currently providing about million template waves and observation data of around 2,100 channels consisting of 4.32×10^6 time steps per day for approximately 10 years. Using Matched Filtering for these massive data requires much computation cost; thus, a faster algorithm is necessary to lower the cost. We use a subset of observation data provided by MOWLAS. We target calculation using 16 template waves and observation wave with 4.32×10^6 time steps; we also target matrices with the sizes of 16×256 and $256 \times (4.32 \times 10^6)$.

NVIDIA Tesla V100 GPU is used as our computing environment. Its peak FLOPS are 7.8 TFLOPS in double precision, 15.7 TFLOPS in single precision, 31.4 TFLOPS in half precision, and 125 TFLOPS in half precision with Tensor Core. The peak memory bandwidth is 900 GB/s. Our code is written with CUDA Fortran/C and complied with PGI 18.10 and nvcc 10.0.130. Elapsed time and actual memory bandwidth of kernels are measured by nvprof, whereas FLOPS are counted manually. cuBLAS is provided by cuda 10.0.130. With multiplication using cuBLAS, we generate the entire matrices explicitly. A function cublasGemmEx is provided by cuBLAS for dense matrix-matrix multiplication [11], and we prepare four types of kernels in which data types of matrices and precision in each operation are different. We must note that the elapsed time required to construct the input matrices including normalization prior to cuBLAS functions is not included. We entirely normalize input matrices when we use cuBLAS with half precision variables, by searching the maximum value in matrices as a scaling factor. Without normalization, multiplication caused an overflow. We prepare two versions of our proposed kernel: the first computes without PTX assembly, and the second skips the data transfer between shared memory and registers using PTX assembly. Targeting kernels are shown in Table I. In the next section, we compare the performance and accuracy of each kernel.

A. Evaluation of Performance

The middle part of Table I summarizes the elapsed time of the kernel and actual bandwidth of shared memory and global memory. When comparing cuBLAS ver. 1, ver. 2, and ver. 3 kernels, the computation time decreased as the precision of input data reduced. General dense matrix-matrix multiplication is known as dense computation; however, the performance of our targeting multiplication was not arithmetic bound and limited by the memory access cost. While cuBLAS ver. 2 and ver. 3 partly used half precision variables and required data conversion cost, cuBLAS ver. 4 kernel entirely used half precision variables and improved the performance as the data conversion was unnecessary and Tensor Core operations were enabled. In our case, Tensor Core operations were disabled even when we specified options to use Tensor Cores except for cuBLAS ver. 4. The actual bandwidth of global memory in the kernel cuBLAS ver. 4 reached 768 GB/s. This was close to the result of the benchmark by [12], which was 900 GB/s $\times$ 83.3% $=$ 750 GB/s; thus, the performance of this kernel was limited by the global memory bandwidth. On the other hand, the proposed kernels increased the bandwidth of shared memory and reduced the bandwidth of global memory. This was because we took components of the observation matrix from shared memory instead of global memory, reducing the memory access cost of global memory and increasing the cost of shared memory. Our proposed kernel used Tensor Core operations; thus, data transfer between shared memory and registers was issued to distribute components of matrices and additional memory access cost was required if we used only wmma API. Accordingly, the performance was bound by the bandwidth of shared memory, increasing the elapsed time. Contrarily, when we used assembly shown in Section II, the performance significantly improved as the memory access to shared memory reduced.

We chose the optimal size of matrix per thread block in the kernel. Elapsed time and the register usage for different sizes of matrices are described in Fig. 5. By increasing the size of the matrix per thread block, memory access cost for computation cost was reduced as we reused template matrices many times. However, the register usage per thread increased as the size of the matrix increased because the results of multiplication must be stored in registers. This made it difficult to overlap latencies involved in memory accesses because the number of available threads is declined. For our developed kernel, $N = 96$ was the equilibrium point of these factors.

Our kernel attained 28.4 TFLOPS that was higher than the peak FP32 FLOPS. This performance was 22.7% of 125 TFLOPS, which was theoretical peak performance when using Tensor Core on V100 GPU. This was because our kernel included operations without Tensor Core required for

TABLE I
PERFORMANCE USING CUBLAS AND OUR DEVELOPED KERNEL. MATRICES ARE INPUT OR OUTPUT IN THE PRECISION NOTED IN THE ROW OF "INPUT" OR "OUTPUT" AND MULTIPLICATION IS DONE IN THE PRECISION NOTED IN THE ROW OF "COMPUTATION"

	cuBLAS				proposed	
	ver.1	ver.2	ver.3	ver.4	w/o PTX	with PTX
Input (Template wave)	FP32	FP16	FP16	FP16	FP16	FP16
Input (Observation wave)	FP32	FP16	FP16	FP16	FP32	FP32
Output	FP32	FP32	FP16	FP16	FP32	FP32
Computation	FP32	FP32	FP32	FP16	FP16	FP16
Tensor Core operation	Disabled	Disabled	Disabled	Enabled	Enabled	Enabled
Elapsed Time	12.775 ms	5.894 ms	5.540 ms	3.032 ms	3.365 ms	1.243 ms
Device Memory Bandwidth	447 GB/s	619 GB/s	594 GB/s	768 GB/s	89 GB/s	232 GB/s
Shared Memory Bandwidth	3296 GB/s	8159 GB/s	8468 GB/s	6384 GB/s	10978 GB/s	9882 GB/s
$Error$ in actual data	7.5×10^{-7}	3.0×10^{-4}	5.4×10^{-4}	4.3×10^{-3}	3.0×10^{-4}	1.8×10^{-4}
$Error$ in dummy data	2.9×10^{-7}	9.2×10^{-5}	1.6×10^{-4}	1.3×10^{-3}	9.2×10^{-5}	9.2×10^{-5}

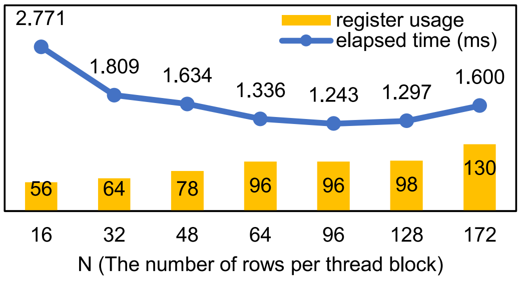

Fig. 5. Performance of our proposed kernel when the size of matrix per thread changes.

normalization and data conversions required for PTX assembly. Performance measurement by [7] showed that Tensor Core on V100 GPU targeting 512 × 512 matrices attained no more than 20 TFLOPS in any implementations. In that multiplication, the number of components of matrices was $3 \times 512 \times 512$ and the number of floating point operations was $2 \times 512 \times 512 \times 512$. On the other hand, for our targeting matrices, the number of components of matrices was $16 \times 256 + 256 \times (4.32 \times 10^6) + (4.32 \times 10^6) \times 16$, and the number of floating point operations was $2 \times 16 \times 256 \times (4.32 \times 10^6)$. As the number of matrices components was 1,500 times larger and computation cost was only 132 times larger, it was more difficult to attain higher performance with our targeting matrices. Considering these conditions, we demonstrated that our developed kernel attained reasonable performance. We computed each template that had 256-time steps in 1.243 ms/16 = 77.7 us with observation data that had 4.32×10^6-times steps.

B. Evaluation of Accuracy

We evaluated the numerical error in the result $CC_{i,j}$ obtained by each kernel based on the result $CC_{i,j}^{FP64}$ computed in double precision variables. We used an $Error$ defined below:

$$Error = \max_{i,j} |CC_{i,j} - CC_{i,j}^{FP64}|, \tag{2}$$

which is the absolute maximum value of each error. We used the same data as the previous subsection. We refer to this data as actual data. In addition, we generated data from a uniform pseudorandom number with the interval $[-50, 50]$. We refer to this data as dummy data. The numerical error of cuBLAS functions and proposed methods using actual data and dummy data, respectively are shown in the lower part of Table I. Error in actual data was larger than in dummy data for all kernels. Targeting waves in MOWLAS had a wide dynamic range and their values increased locally; thus, results of actual data were more affected by numerical errors. The proposed method reduced the numerical error by introducing local normalization with low-level description. This approach can work for improving the accuracy unless values change too rapidly even within tens of time steps. By contrast, numerical error in cuBLAS increased as the precision of variables decreased. To attain the same degrees of accuracy as our proposed method, internal single-precision computation such as cuBLAS ver. 2 was required. This kernel cuBLAS ver. 2 took 5.894 ms; therefore, we evaluated that our kernel attained 4.74-fold speeding up (5.894 ms/1.243 ms) compared to the common library with returned equally accurate results. When we detected patterns that had $CC_{i,j} > 0.7$, there was no erroneous detection in all kernels. This result can change depending on the problem settings; therefore more verification for accuracy is required as a future task.

IV. CONCLUSION

We focused on Matched Filtering in the time domain. The largest proportion of the computation cost of Matched Filtering is in matrix-matrix product. Considering massive observation data, the reduction of computation cost was a critical issue. Using Tensor Core on NVIDIA Volta GPUs, we designed an algorithm to use fast matrix-matrix product. When we computed using Tensor Core, memory access to global memory or shared memory became the bottleneck for the performance. Thus, we reduced the memory access cost reusing the data and skipping unnecessary data movement. In addition, current Tensor Core only supported lower precision data types; thus, we had to reduce the effect of numerical errors in the computation. We introduced localized normalization for the target matrices to improve the accuracy of the computation. This normalization was issued by low-level description to minimize the data transfer cost. With the appropriate algorithm design, we achieved 28.4 TFLOPS in the kernel, which was reasonable performance for the sizes of our targeting matrices. We confirmed that our proposed kernel had a smaller numerical error than matrix-matrix multiplication on Tensor Core in cuBLAS, which was a common linear algebra library on NVIDIA GPUs. When we compared our kernel and a function of cuBLAS that exhibited the same degree of accuracy, our kernel was 4.47 times faster.

References

[1] G. Turin, "An introduction to matched filters," *IRE Trans. Inf. Theory*, vol. IT-6, no. 3, pp. 311–329, Jun. 1960.

[2] P. M. Woodward, *Probability and Information Theory, With Applications to Radar: International Series of Monographs on Electronics and Instrumentation*. Elsevier, vol. 3, 2014. [Online]. Available: https://www.elsevier.com/books/probability-and-information-theory-with-applications-to-radar/woodward/978-0-08-011006-6

[3] B. F. Schutz, "Gravitational wave astronomy," *Classical Quantum Gravity*, vol. 16, no. 12A, p. A131, 1999.

[4] S. J. Gibbons and F. Ringdal, "The detection of low magnitude seismic events using array-based waveform correlation," *Geophys. J. Int.*, vol. 165, no. 1, pp. 149–166, 2006.

[5] E. Beaucé, W. B. Frank, and A. Romanenko, "Fast matched filter (FMF): An efficient seismic matched-filter search for both CPU and GPU architectures," *Seismological Res. Lett.*, vol. 89, no. 1, pp. 165–172, 2017.

[6] NVIDIA, "NVIDIA Tesla V100 GPU architecture." Accessed: Jul. 3, 2019. [Online]. Available: http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[7] S. Markidis, S. W. Der Chien, E. Laure, I. B. Peng, and J. S. Vetter, "NVIDIA tensor core programmability, performance & precision," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2018, pp. 522–531.

[8] Y. LeCun *et al.*, "Convolutional networks for images, speech, and time series," *Handbook Brain Theory Neural Netw.*, vol. 3361, no. 10, p. 1995, 1995.

[9] M. A. Raihan, N. Goli, and T. M. Aamodt, "Modeling deep learning accelerator enabled GPUs," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2019, pp. 79–92.

[10] National Research Institute for Earth Science and Disaster Resilience. *NIED MOWLAS*. (2019). Accessed: Jul. 3, 2019. [Online]. Available: https://doi.org/10.17598/NIED.0009

[11] NVIDIA. *cuBLAS*. (2019). Accessed: Jul. 3, 2019. [Online]. Available: https://docs.nvidia.com/cuda/cublas/index.html

[12] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA Volta GPU architecture via microbenchmarking," *CoRR*, 2018. [Online]. Available: http://arxiv.org/abs/1804.06826