GEMM the new Gem: The Inevitable Kernel and its Sensitivity to Compiler Optimizations and Libraries

Siyuan Ma, Bagus Hanindhito, Anushka Subramanian and Lizy K. John, *Fellow, IEEE*

***Abstract*—When the SPEC benchmark suite was first assembled in 1989, matrix multiplication code *matrix300* was one of the 10 programs in the suite, but it was discarded within 2-3 years due to the high sensitivity of matrix multiplication to compiler optimizations. However, with the advent of machine learning (ML), neural networks and generative AI (GenAI), matrix multiplication is an integral part of the modern computing workload. While sensitive, general matrix multiplication (GEMM) cannot be ignored anymore, especially if hardware that runs ML workloads is being evaluated. In this paper the sensitivity of GEMM workloads to libraries and compiler optimizations is studied. While it may be inevitable to use matmul kernels as a benchmark to understand the performance of accelerators for machine learning, understanding the sensitivity to compiler optimizations and software libraries can help to optimize and interpret the results appropriately. We observe more than 9000× variation in CPU runtimes and around 84× variation in GPU run times depending on the optimizations used.**

***Index Terms*—Matrix Multiplication, Performance, Energy, Tensor Cores.**

I. Introduction

When SPEC benchmark suite was first created in 1989, a matrix multiplication program `matrix300` multiplying two 300 × 300 matrices was one of the ten programs in the suite; however it was discarded very soon and excluded from the SPEC CPU 92 suite due to its high sensitivity to compiler optimization. Figure 1, taken from the "In More Depth: The Difficulty with Kernel Benchmarks," section of the 2006 edition of Patterson et al. [1] presents a piece of history around the challenges surrounding the `matrix300` benchmark. This history is also described in Kounev's book [2].

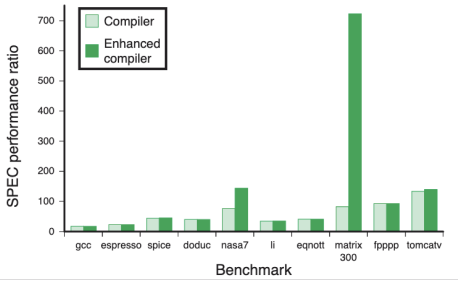

Fig. 1. SPEC89 performance ratios for the IBM Powerstation 550 using two different compilers; adapted from [1] Only nasa7 and matrix300 shows improvement.

Siyuan Ma and and Lizy K. John are with UT Austin, Austin, TX 78712, USA. Bagus Hanindhito was with UT Austin when this work was performed. (e-mail: siyuanma@utexas.edu, bagus@utexas.edu, ljohn@ece.utexas.edu).

TABLE I
Matrix multiplication operations in ChatGPT 175B-parameter model with batch size of 1

Stages	Matmul Size			No. of MACs
	M	K	N	M × K × N
Tokenization & Word Embed.				
One-hot × Embedded Weight	2,048	51,200	12,288	1.288e12
Positional Encoding				
Word Embed. + Pos. Encoded	2,048	12,288		
Multi Head Attn. (96 blocks)				
X × WQ=Q	2,048	12,288	12,288	3.09e11
X × WK=K	2,048	12,288	12,288	3.09e11
X × WV=V	2,048	12,288	12,288	3.09e11
Q × KT=QK (96 heads)	2,048	128	2,048	5.2e10
QK × V (96 heads)	2,048	2,048	128	5.2e10
Linear Transformation	2,048	12,288	12,288	3.09e11
Feed-Forward Network				
Linear Transformation + Bias	2,048	12,288	49,152	1.237e12
Linear Transformation + Bias	2,048	49,152	12,288	1.237e12
Decoding				
Decoder	2,048	12,288	51,200	1.288e12
Output	2,048	51,200		

The dropping of `matrix300` was well justified in 1992, however, with the advent of machine learning (ML), neural networks, and generative AI (GenAI), the matrix multiplication (GEMM) kernel has assumed a new status. One may look at the various layers of the popular ChatGPT and can see the various matrix multiplications in it, as illustrated in Table I. GEMM is an integral part of the modern AI computing workload and, in a way, the new "gem" that ML hardware accelerator designers are seeking to optimize. For instance, NVIDIA GPUs feature Tensor Cores, dedicated matrix multiplication units, and the Google TPU employs a systolic array to accelerate multiplication of matrices.

Although sensitive, general matrix multiplication (GEMM) cannot be ignored anymore, especially if hardware running ML workloads is being evaluated. However, it is also crucial to understand sensitivity, as this allows optimization for high performance and the proper interpretation of results. It might be worthwhile to evaluate the performance of emerging hardware across multiple versions of a GEMM kernel with various libraries and compiler optimizations, with different data dimensions and data types, rather than discard GEMMs altogether. A series of matrix multiplications with some producer-consumer dependencies also form interesting benchmarks in this new light.

In this paper, the sensitivity of GEMM workloads to libraries and compiler optimizations is studied systematically and quantitatively across (i) multiple vendor libraries (PyTorch, cuBLAS, CUTLASS, hipBLASLt, rocBLAS), (ii) multiple architectures (Nvidia A100, Nvidia H200, AMD MI300X, and EPYC CPUs), and (iii) multiple matrix sizes while still arguing for the use of GEMM kernels as a benchmark to understand the performance of hardware accelerators for machine learning.

II. Background and Prior Work

Matrix multiplication can be done with high level languages such as Python or C, and it can be done with various vendor-optimized libraries, for example, Intel MKL libraries on Intel CPUs. Prior work [3], [4] reported on the sensitivity of CPU execution times for multiplying two 4096 × 4096 matrices as shown in Table II. Starting from Python and Java implementations, proceeding through various levels of optimizations applied to C code, and also utilizing the AVX vector instruction set, they illustrate a performance variability of more than 50,000 times. The best performance from their nine different sets of codes is also similar to the highly optimized Intel MKL libraries. The Intel® oneAPI Math Kernel Library (Intel® oneMKL) provides highly optimized math routines, including BLAS (Basic Linear Algebra Subprograms), which contains matrix multiplication functions (e.g., SGEMM and DGEMM[1]) that are tailored for Intel hardware. Intel oneMKL uses instruction sets like AVX. Vendor-specific libraries also exist for AMD. AMD provides Optimizing CPU Libraries (AOCL), specifically the AOCL-BLAS component, which provides a high-performance BLAS implementation tuned for AMD's Zen architecture processors, such as Ryzen and EPYC.

TABLE II
Results from an MIT study on CPUs [3], [4]

Ver	Implementation	Running Time (s)	Relative Speedup	Absolute Speedup	GFLOP	% Peak
1	Python	21041.7	1.00	1	0.006	0.001
2	Java	2387.32	8.81	8	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	Parallel loops	3.04	17.97	6,921	45.211	5.408
7	Parallel divide & conquer	1.30	1.38	16,197	105.722	12.646
8	+ compiler vectorization	0.70	1.87	30,272	196.341	23.486
9	+ AVX intrinsics	0.39	1.76	53,292	352.408	41.677
10	Intel MKL	0.41	0.97	51,497	335.217	40.098

With the popularity of GPUs for handling machine learning workloads, optimized CUDA codes, and heavily optimized GPU libraries, such as CUBLAS and CUTLASS, are available. Also, while CPUs use the AVX intrinsics to make use of the vector extension instructions, GPUs provide Tensor Cores hardware units (Figure 2) along with special matrix multiplication instructions [5]. A few example instructions are shown in Table III. These instructions are available in a variety of data precisions and for a variety of matrix sizes. In light of the newly added Tensor Cores, the original cores in GPUs are called CUDA cores. Both CUDA Cores and Tensor Cores operate side-by-side as shown in the right side of Figure 2.

Most vendors who make AI hardware offer special accelerators for matrix operations. Following NVIDIA's lead, AMD integrated matrix accelerators called Matrix Cores into their CDNA GPU architecture [6] in 2020. Finally, Intel introduced their version called XMX Matrix Engine into their Ponte Vecchio GPU architecture [7] in 2022.

[1]SGEMM stands for single precision GEMM and DGEMM stands for double precision GEMM.

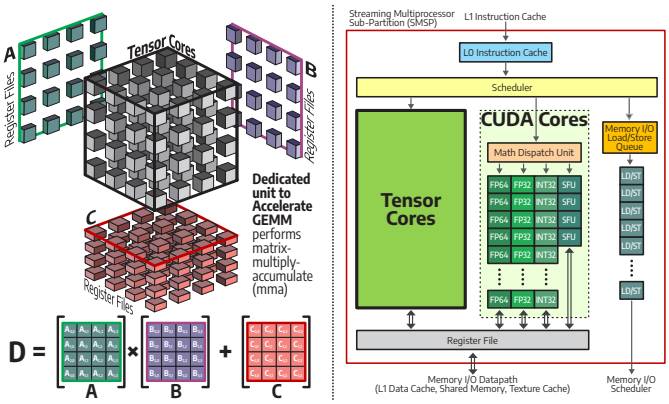

Fig. 2. Tensor Cores in GPU for accelerating matrix multiplication (left) work side-by-side with the regular vector ALUs (CUDA Cores) in GPU (right).

TABLE III
Example Instructions in GPU Tensor Cores for accelerating matrix multiplication

Inst.	Arch.	Data Precision		Instruction Size
		Input	Output	
WMMA 16-bit	Volta	FP16	FP16	16x16x16, 8x32x16, 32x8x16
	Turing		FP32	
	Ampere	FP16	FP32	
	Hopper	BF16	FP16	
WMMA 19-bit	Ampere	TF32	FP32	16x16x8
	Hopper			
WMMA 4-bit	Turing	INT4	INT32	8x8x32
	Ampere			
	Hopper			
WMMA 1-bit	Turing	INT1	INT32	8x8x128
	Ampere			
	Hopper			
HMMA 16-bit	Volta	FP16	FP16	8x8x4
	Turing		FP32	8x8x4, 16x8x8, 16x8x16
	Ampere	FP16	FP16	16x8x8, 16x8x16, 16x8x32
	Hopper	BF16	FP32	
HMMA 19-bit	Ampere	TF32	FP32	16x8x4, 16x8x8, 16x8x16
	Hopper			
IMMA 8-bit	Turing	INT8	INT32	8x8x16
	Ampere			8x8x16, 16x8x16, 16x8x32, 16x8x64
	Hopper			
BMMA 1-bit	Turing	INT1	INT32	8x8x128
	Ampere			8x8x128, 16x8x128, 16x8x256
	Hopper			
DMMA 64-bit	Ampere	FP64	FP64	8x8x4
	Hopper			8x8x4, 16x8x4, 16x8x8, 16x8x16

Following the approach of prior work [3], [4] that examined GEMM performance sensitivity on earlier CPUs, we conduct a contemporary analysis on modern AMD CPUs and NVIDIA/AMD GPUs using measurements collected in early 2025.

III. GEMM Performance on CPUs

In this section, we present the performance of GEMM kernels on CPUs. The study uses AMD EPYC 7742 64-Core Processor with the `gcc` 11.4.0 and `clang` 14.0.0 compilers. The `C` code shown in Listing 1 is compiled with `-O1` and `-O3` flags with `gcc` and `clang`, and with LLVM polyhedral optimization, leading to a performance variation of 53.07× (Table IV).

```c
void multiply_matrices(float A[N][N], float B[N][N], float C[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = 0;
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            } } } }
```

Listing 1. Matrix multiplication implementation in C language

```python
A = np.random.rand(M, K)
B = np.random.rand(K, N)
# Perform GEMM
for i in range(iter):
    C = np.matmul(A, B)
```

Listing 2. GEMM with Numpy in Python

```python
A = torch.randn(M, K).to(device)
B = torch.randn(K, N).to(device)
# Perform GEMM
for i in range(iter):
    C = torch.matmul(A, B)
```

Listing 3. GEMM with Torch in Python

TABLE IV
PERFORMANCE OF GEMM ON AMD EPYC 7742 CPU

Configuration	Language / Library	Latency (ms)	Speed-up
Single Thread	GCC -O1	478890	1.00×
	GCC -O3	262830	1.82×
	Clang -O1	582416	0.82×
	Clang -O3	541108	0.89×
	LLVM Poly	9023	53.07×
Multi Thread	Py (NumPy)	276.97	1,729×
	Py (Torch)	49.79	9,618×

Since the GEMM kernel written in C code can only utilize one out of 64 cores available, we run two other versions of the GEMM kernel provided by NumPy and Torch libraries, as shown in Listings 2 and 3, respectively. Both libraries are multi-threaded, providing significant performance due to the use of 64 cores and other optimizations. One may note from Table IV that NumPy is 950× faster than the `gcc -O3`, and Torch is 9600× faster than the baseline `C`.

IV. GEMM Performance on GPUs

We compared four different GEMM kernel implementations[2] on GPU (CUDA, Torch, CUBLAS, CUTLASS) using both `fp32` and `fp16` data types. In addition, experiments were performed with and without Tensor Cores. The GEMM kernel in CUDA is illustrated in (Listing 4).

In the CUDA code version, the kernel `MatrixMulCUDA` implements a blocked GEMM, where each thread block computes a sub-matrix (block) of the result, and each thread computes one element of the output block. Shared memory is used to load tiles of A and B, reducing global memory accesses. On the other hand, the CUBLAS GEMM kernel is called using the API `cublasGemmEx()`, which takes various arguments, including matrix transposes, dimensions, data type, compute type, and compute algorithm. Finally, the CUTLASS API, `gemm_op()`, also takes arguments, including matrix dimensions, matrix references, and scaling factors.

The NVIDIA A100, H200 and AMD MI300X GPUs are used for the experiment. The CUDA GEMM kernel is implemented in `fp32`, while the GEMM kernel from PyTorch uses either `fp32` or `fp16`. However, when using `fp32`, it utilizes the CUDA cores since `fp32` is not directly supported by the Tensor Cores. When using `fp16`, PyTorch automatically utilizes Tensor Cores. The kernel from CUBLAS and CUTLASS in `fp16` can use either CUDA or Tensor Cores. The A100 and H200 execution time was measured using the CUDA runtime API[8] (`cudaEventElapsedTime()`). Energy was measured using NVML [9] by sampling power during execution. The MI300X execution time was measured using AMD HIP Runtime API and energy was measured using `amd-smi`.

Table V presents the execution time and energy consumption of the A100 GPU across different libraries for a 4096×4096×4096 GEMM. The CUDA version gives the highest latency and highest energy consumption. CUBLAS libraries utilizing Tensor Cores provide the best latency, with 52.51× speedup compared to the CUDA version, followed by CUTLASS (45.14×) and PyTorch(42.88×). The CUBLAS, in conjunction with Tensor Cores, gives the lowest energy consumption (45.85× better than CUDA), followed by CUTLASS(33.11×) and PyTorch(28.38×).

Performance and energy characteristics vary significantly with GEMM dimensions. Tables VI and VII present latency and energy across multiple GEMM sizes and libraries measured on A100. The GEMM-1024×4096×16384 and GEMM-16384×16384×16384 cases follow trends consistent with Table V. However, for highly imbalanced shapes such as GEMM-32×16384×4096 and GEMM-16384×32×4096, CUTLASS on CUDA cores performs the worst. Notably, for GEMM-16384×32×4096, PyTorch outperforms both cuBLAS and CUTLASS on tensor cores.

Substantial performance and energy variability across libraries is also evident on different GPU architectures. As shown in Table VIII, the Nvidia H200 exhibits up to 84.39× latency variation and 70.29× energy variation, while the AMD MI300X shows up to 54.66× latency variation and 101× energy variation across libraries.

```c
template <int BLOCK_SIZE, typename T>
__global__ void MatrixMulCUDA(T *C, T *A, T *B, int wA, int wB)
{
  // ... initialization code ...
  T Csub = 0;
  for (int a=aBegin, b=bBegin; a<=aEnd; a+=aStep, b+=bStep) {
    __shared__ T As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ T Bs[BLOCK_SIZE][BLOCK_SIZE];
    As[ty][tx] = A[a + wA * ty + tx];
    Bs[ty][tx] = B[b + wB * ty + tx];
    __syncthreads();

#pragma unroll
    for (int k = 0; k < BLOCK_SIZE; ++k) {
      Csub += As[ty][k] * Bs[k][tx];
    }
    __syncthreads();
  }
  int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
  C[c + wB * ty + tx] = Csub;
}
```

Listing 4. GEMM kernel in GPU, written using CUDA

While Python codes were heavily underperforming in Table II, the modern day GPU software libraries convert the Python to highly optimized codes, resulting in good performance for Python and PyTorch codes as of 2025.

[2]https://github.com/hibagus/CUDA_Bench

TABLE V
LATENCY AND ENERGY OF GEMM ON NVIDIA A100 GPU

Cores	Language / Library	Latency	Speedup	Energy	Energy Efficiency
CUDA Cores	CUDA fp32	25.73 ms	1.00×	5.96 J	1.00×
	PyTorch fp32	7.43 ms	3.46×	2.53 J	2.36×
	CUBLAS fp32	7.22 ms	3.56×	1.63 J	3.66×
	CUTLASS fp32	7.83 ms	3.29×	1.72 J	3.47×
	CUBLAS fp16	3.84 ms	6.70×	0.88 J	6.77×
	CUTLASS fp16	4.26 ms	6.04×	0.97 J	6.14×
Tensor Cores	CUBLAS tf32	0.81 ms	33.98×	0.23 J	25.91×
	PyTorch fp16	0.60 ms	42.88×	0.21 J	28.38×
	CUBLAS fp16	0.49 ms	52.51×	0.13 J	45.85×
	CUTLASS fp16	0.57 ms	45.14×	0.18 J	33.11×

TABLE VI
AVERAGE GEMM LATENCY (MILLISECONDS)

Config.	Library	32 x16384 x4096	1024 x4096 x16384	4096 x4096 x4096	16384 x16384 x4096	16384 x32 x4096
CUDA Cores	CUDA fp32	0.91	25.83	25.73	1627.46	0.91
	PyTorch fp32	0.30	7.40	7.43	466.32	0.30
	CUBLAS fp32	0.40	7.53	7.22	460.75	0.28
	CUTLASS fp32	1.61	9.34	7.83	472.38	1.61
	CUBLAS fp16	0.51	3.76	3.84	228.84	0.28
	CUTLASS fp16	0.91	5.28	4.26	250.89	0.91
Tensor Cores	CUBLAS tf32	0.20	0.89	0.81	50.79	0.29
	PyTorch fp16	0.10	0.60	0.60	34.20	0.09
	CUBLAS fp16	0.09	0.57	0.49	29.08	0.19
	CUTLASS fp16	0.09	0.60	0.57	55.24	0.19
Max/Min		17.88x	45.32x	52.51x	55.97x	17.88x

Pytorch as well as CUBLAS and CUTLASS GPU libraries, give excellent GEMM performance.

Profiling on NVIDIA A100 GPUs (Table IX) using Nsight Compute [10] explains the speedup obtained through optimized software libraries (CUDA, CUBLAS with fp32), as well as specialized tensor hardware (CUBLAS with tf32).

Using the CUDA Core datapath, CUBLAS fp32 achieves 3.56× speedup over the CUDA kernel due to highly tuned implementation with improved L2 locality. While both kernels execute a similar number of FMAs, CUBLAS has 15.7× reduction in loads leading to fewer cycles on memory accesses, and higher throughput.

CUBLAS execution on Tensor Cores leads to a further 8.9× speedup due to computational density of the tensor pipeline – each tensor MMA replaces multiple scalar FMAs on CUDA Cores. We observe similar cache performances at the L1. [3]

V. CONCLUSION

This study reaffirms that matrix multiplication remains highly sensitive to software-level optimizations. We observe over 9000× CPU and 84.39× GPU runtime variation across compilers, optimizations and libraries, underscoring the need of attention to those settings. Nevertheless, despite this sensitivity, GEMM kernels remain indispensable as benchmarks due to their central role in modern computing and machine learning workloads.

[3]The observed L1 hit rate is near zero for CUDA Cores, and exactly zero in Tensor Cores due to the asynchronous copy instruction[11], which stages data from global memory into shared memory, bypassing the L1 cache.

TABLE VII
AVERAGE GEMM ENERGY (JOULES)

Config.	Library	32 x16384 x4096	1024 x4096 x16384	4096 x4096 x4096	16384 x16384 x16384	16384 x32 x4096
CUDA Cores	CUDA fp32	0.20	6.06	5.96	393.47	0.20
	PyTorch fp32	0.08	2.77	2.53	173.91	0.09
	CUBLAS fp32	0.10	1.91	1.63	106.11	0.07
	CUTLASS fp32	0.27	1.85	1.72	114.27	0.27
	CUBLAS fp16	0.11	0.76	0.88	56.83	0.07
	CUTLASS fp16	0.16	1.05	0.97	65.07	0.15
Tensor Cores	CUBLAS tf32	0.05	0.25	0.23	17.43	0.05
	PyTorch fp16	0.02	0.23	0.21	13.57	0.02
	CUBLAS fp16	0.02	0.16	0.13	10.22	0.03
	CUTLASS fp16	0.02	0.17	0.18	16.28	0.03
Max/Min		13.5x	37.88x	45.84x	38.5x	13.5x

TABLE VIII
VARIABILITY RANGE IN LATENCY AND ENERGY FOR A100, H200 AND MI300 GPUS

Max/Min	32 x16384 x4096	1024 x4096 x16384	4096 x4096 x4096	16384 x16384 x16384	16384 x32 x4096
A100 Latency	17.88x	45.32x	52.51x	55.97x	17.88x
A100 Energy	13.5x	37.88x	45.84x	38.5x	13.5x
H200 Latency	10.16x	20.94x	18.85x	84.39x	7.06x
H200 Energy	6.63x	47.25x	37.13x	70.29x	4.4x
MI300X Latency	8.00x	35.83x	45.06x	54.66x	6.6x
MI300X Energy	19.00x	101.5x	84.71x	55.47x	20.00x

TABLE IX
MICROARCHITECTURAL STATISTICS FOR 4096X4096X4096 GEMM

Library	CUDA	CUBLAS	CUBLAS
Data Path	CUDA Cores		Tensor Cores
Throughput (TFLOPS/s)	5.34	19.03	169.67
Total I-count	49.9M	22.2M	1.6M
Total load I-count	26M	1.65M	381k
FMA/MMA I-count	20.5M	19.1M	606.8k
L1 hit rate (%)	∼0	∼0	0
L2 hit rate (%)	50.05	84.08	74.25

REFERENCES

[1] D. A. Patterson *et al.*, "In more depth: The difficulty with kernel benchmarks," *Computer Organization and Design: The Hardware/Software Interface*, 2006.

[2] S. Kounev *et al.*, *The SPEC CPU Benchmark Suite*. Cham: Springer International Publishing, 2020, pp. 231–250. [Online]. Available: https://doi.org/10.1007/978-3-030-41705-5_10

[3] S. Amarasinge *et al.*, "6.106 Software Performance Engineering," *MIT Course*, 2024.

[4] C. Leiserson, "Setting a course for post-moore software performance," *Keynote Speech at IEEE High Performance Computer Architecture Symposium*, 2025.

[5] B. Hanindhito *et al.*, "Accelerating ml workloads using gpu tensor cores: The good, the bad, and the ugly," in *Proc. of Intl. Conf on Performance Engineering*. ACM, 2024, p. 178–189. [Online]. Available: https://doi.org/10.1145/3629526.3653835

[6] Advanced Micro Devices, "AMD CDNA Architecture," Advanced Micro Devices, California, US, Whitepaper, Nov. 2020. [Online]. Available: https://www.amd.com/system/files/documents/amd-cdna-whitepaper.pdf

[7] H. Jiang, "Intel's Ponte Vecchio GPU : Architecture, Systems & Software," in *2022 IEEE Hot Chips 34 Symposium (HCS)*. Los Alamitos, CA, USA: IEEE Computer Society, aug 2022, pp. 1–29. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/HCS55958.2022.9895631

[8] NVIDIA Corp., "CUDA Runtime API," https://docs.nvidia.com/cuda/cuda-runtime-api/index.html.

[9] ——, "NVIDIA Management Library (NVML)," https://developer.nvidia.com/management-library-nvml, 2025.

[10] ——, "NVIDIA Nsight Compute (NCU)," https://docs.nvidia.com/nsight-compute/index.html, 2025.

[11] ——, "NVIDIA A100 Tensor Core GPU Architecture," https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, 2020.