Accelerating ML Workloads using GPU Tensor Cores: The Good, the Bad, and the Ugly

Bagus Hanindhito
hanindhito@bagus.my.id
The University of Texas at Austin
Austin, Texas, USA

Lizy K. John
ljohn@ece.utexas.edu
The University of Texas at Austin
Austin, Texas, USA

ABSTRACT

Machine Learning (ML) workloads generally contain a significant amount of matrix computations; hence, hardware accelerators for ML have been incorporating support for matrix accelerators. With the popularity of GPUs as hardware accelerators for ML, specialized matrix accelerators are embedded into GPUs (e.g., Tensor Cores on NVIDIA GPUs) to significantly improve the performance and energy efficiency of ML workloads. NVIDIA Tensor Cores and other matrix accelerators have been designed to support General Matrix-Matrix Multiplication (GEMM) for many data types. While previous research has demonstrated impressive performance gains with Tensor Cores, they primarily focused on Convolutional Neural Networks (CNNs).

This paper explores Tensor Cores' performance on various workloads, including Graph Convolutional Networks (GCNs), on NVIDIA H100 and A100 GPUs. In our experiments with NVIDIA GPUs, CNNs can achieve 1.91× (TF32) and 2.42× (FP16) end-to-end performance improvements with the use of Tensor Cores, whereas GCNs struggle to surpass a 1.03× (FP16) boost. Some implementations even experience slowdowns despite software transformation. Additionally, we explore the potential of Tensor Cores in non-GEMM-like kernels, providing insights into how software techniques can map diverse computation patterns onto Tensor Cores. Our investigation encompasses several kernels and end-to-end applications, aiming to comprehend the nuanced performance impact of Tensor Cores. Furthermore, we are among the first to present third-party evaluations of H100 GPU performance over the prior A100 GPU.

CCS CONCEPTS

• **General and reference** → **Performance**; **Measurement**; **Evaluation**; **Experimentation**; • **Computing methodologies** → *Machine learning*.

KEYWORDS

Machine Learning; Matrix Accelerators; Performance Evaluation; Workload Characterization; Measurement

ICPE '24, May 7–11, 2024, London, United Kingdom

ACM ISBN 979-8-4007-0444-4/24/05.
https://doi.org/10.1145/3629526.3653835

ACM Reference Format:
Bagus Hanindhito and Lizy K. John. 2024. Accelerating ML Workloads using GPU Tensor Cores: The Good, the Bad, and the Ugly. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), May 7–11, 2024, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3629526.3653835

1 INTRODUCTION

At the heart of Artificial Intelligence (AI) and Machine Learning (ML), General Matrix-Matrix Multiplications (GEMMs) are the most important building blocks for many applications [4, 26, 81]. In 2017, with the launch of Volta architecture [45], NVIDIA introduced Tensor Cores in their GPUs to accelerate GEMM. Tensor Core provides significant performance boost and energy efficiency when performing GEMM operations, and is accessible either through low-level assembly or various CUDA libraries [38]. Other manufacturers followed by integrating matrix accelerators into their GPUs years later [1, 24]. Recently developed hardware that targets AI and ML, including FPGA and ASIC, also has matrix accelerators, such as in Xilinx Versal FPGA [16] and Google TPU ASIC [25].

In this paper, the performance benefits of Tensor Cores are investigated across multiple workloads. Prior works on Tensor Cores evaluate Convolutional Neural Networks (CNN) [57, 76] and GEMM [14, 17]. However, the benefits of Tensor Cores in Graph Convolutional Networks (GCN) [29], which is an important emerging ML workload, have not been explored. We analyze the performance of four configurations of the GCN model and several kernels including element-wise operations. Another contribution of this paper is the measurement-based evaluation of ML acceleration using the NVIDIA H100 GPU. Apart from NVIDIA publications, there have been very few third-party works evaluating H100 GPUs. This is also one of the earliest third-party papers to measure and analyze the performance of H100 compared to its predecessor, A100. While performance evaluation of H100 appears in prior work [7], they do not present Tensor Core performance.

The objectives of this study are the following:

- Investigate the performance of the CNN and GCN, both with and without Tensor Cores, across two generations of NVIDIA GPUs, A100 [48] and H100 [49], based on hardware measurement.
- Provide third-party performance evaluation of NVIDIA H100 GPU compared to the previous generation GPU, NVIDIA A100.
- Conduct roofline analysis of the workloads to understand their characteristics and correlation with Tensor Cores performance.
- Develop GEMM-like and non-GEMM-like microbenchmark kernels to understand the performance patterns of Tensor Cores.
- Analyze the floating-point instruction mix of workloads and shed light on the types of lower precision instructions utilized, the functional units where they are being executed (e.g., CUDA

Cores, Tensor Cores), etc. across different networks and training configurations (e.g., full-precision, mixed-precision).
- Investigate the impact of new data types, such as TF32 [8].
- Investigate whether code optimizations like reshaping and padding can make non-GEMM kernels utilize Tensor Cores (eg: Implicit GEMV vs. Reshaped GEMV for FIR)

The major insights from this study are the following:
- Tensor Cores provides 1.3× to 2.9× improvements in CNN whereas only 1.03× in GCN. Among kernels, GEMM, GEMV, and Conv2D get the benefits while Element-wise and FIR fail to get any improvements in spite of transformations.
- Four different CNNs yield an average of 1.93× improvement on H100 versus the previous A100 GPU. Among the four GCN configurations experimented, two yield an impressive 8× improvement on H100 compared to A100, whereas two of the GCN configurations provide nearly no improvements.
- GCNs have 10× lower arithmetic intensity compared to CNNs, and benefits from Tensor Cores are difficult to obtain.
- There are performance anomalies while using different CUDA versions. For instance, the newest CUDA libraries gave improved performance for many workloads, however, for some of the GCNs, they yielded poorer performance than the older CUDA version.
- Non-GEMM-like kernels struggle to get any performance improvements from Tensor Cores, even with data transformations. Reshaped FIR can use batching in order to reduce performance overheads, whereas naive FIR is not even supported and cannot run on Tensor Cores.

2 BACKGROUND AND PRIOR WORK

2.1 Tensor Cores

Starting from Volta architecture (2017), NVIDIA GPUs contain CUDA Cores and Tensor Cores as illustrated in Figure 1. CUDA Cores are the default (traditional) compute units in GPUs, while Tensor Cores were later added specifically for accelerating matrix multiplications, which are abundant in many machine learning (ML) workloads [4, 26, 81]. With libraries provided by NVIDIA, Tensor Cores quickly became the workhorse for accelerating ML workloads as popular machine learning frameworks, such as PyTorch and TensorFlow, support Tensor Cores.

2.1.1 Architectural Overview. Figure 2 gives a high-level illustration of the **matrix-multiply-accumulate (MMA)** operations performed by Tensor Cores on two 4×4 matrices to produce a 4×4 matrix. Essentially, Tensor Cores perform the arithmetic expression $D = A \times B + C$ where A, B, C, D are matrices. Larger dimension matrices are possible using larger Tensor Cores instruction size and hierarchical matrix multiplication [28].

The NVIDIA Tesla V100 with Volta architecture [45] contains 640 first-generation Tensor Cores across 80 SMs[1]. The Tensor Cores in each SM can deliver 1024 FLOPs per cycle, resulting in up to 120 TFLOPs/s FP16 performance [9]. Only half-precision matrix multiplication is supported in this generation. Thus, the A and B matrices in Figure 2 are in FP16, while the resulting product matrix can be in either FP16 or FP32. The subsequent version of Tensor Cores supports more data types as given in Table 1.

The second generation of Tensor Cores was introduced in 2019 with Turing architecture [46] focusing on accelerating the quantized ML inference workload. It supports new data types INT8,

[1]SM stands for Streaming Multiprocessor, which contains a collection of SIMD Units referred to as CUDA Cores (e.g., FP64, FP32, INT32), instruction schedulers, registers, shared memory, L1 cache, and texture cache (Figure 1). GPUs usually have multiple numbers of SM to achieve even higher parallelism.

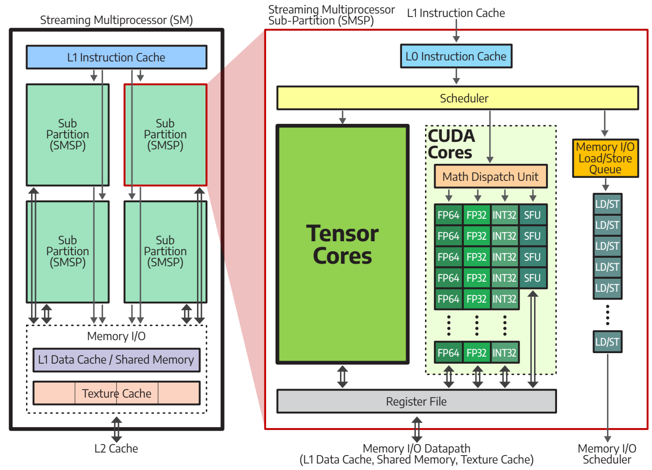

Figure 1: CUDA Cores are the default compute units while Tensor Cores are additions to accelerate matrix multiplications in GPUs

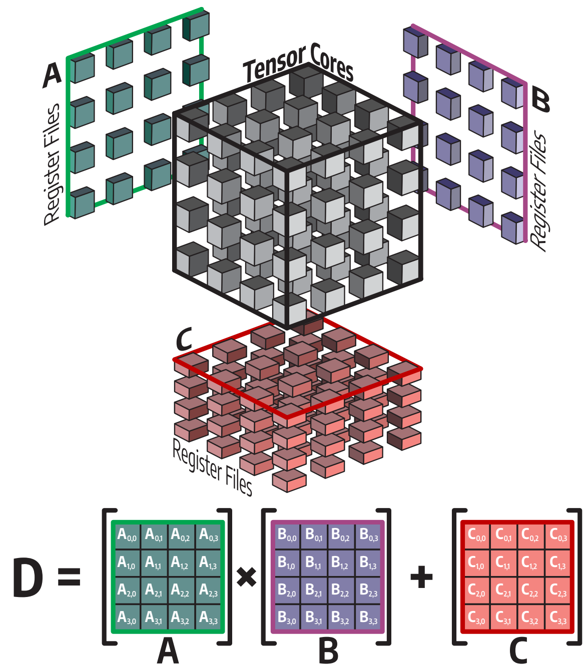

Figure 2: Tensor Cores compute $D = A \times B + C$

Table 1: Tensor Cores Evolution and Supported Precision

Generation	Archi-tecture	Product Name	Specification			Precision Support													
						Tensor Cores								CUDA Cores					
			#SM	#CC (FP32)	#TC	FP64	TF32	FP16	BF16	FP8	INT8	INT4	INT1	FP64	FP32	FP16	BF16	INT32	INT8
1	Volta [45]	V100S	80	5120	640	-	-	✔	-	-	-	-	-	✔	✔	✔	-	✔	✔
2	Turing [46]	RTX 6000	72	4608	576	-	-	✔	-	-	✔	✔	✔	●	✔	✔	-	✔	✔
3	Ampere [48]	A100	108	6912	432	✔	✔	✔	✔	-	✔	✔	✔	✔	✔	✔	✔	✔	✔
4	Hopper [49]	H100	132	16896	528	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
4	Ada [54]	L40S	142	18176	568	-	✔	✔	✔	✔	✔	✔	✔	●	✔	✔	✔	✔	✔

✔: full-support; ●: support with reduced performance; -: not supported.

INT4, and INT1, which are specifically useful for ML inference workloads that can tolerate lower precision with minimum impact on model accuracy [34] as well as binary neural networks [32]. Third-generation Tensor Cores, launched with Ampere architecture [48], support new data types such as BFloat16 (BF16) [74] and TensorFloat32 (TF32) [8] with additional support for accelerating sparse matrix operations [68]. Moreover, new FP64 support opens new possibilities for Tensor Cores to be used in HPC and scientific applications [17]. The fourth-generation Tensor Cores, introduced in the Hopper architecture [49] in 2022, double the throughput per SM per cycle compared to its predecessor for all data formats [7]. A new quarter precision data type (FP8), which supports both `e4m3` (4 exponent bits, 3 mantissa bits) for more accuracy and `e5m2` (5 exponent bits, 2 mantissa bits) for more dynamic range [42], is useful for large language models. More FP64 shapes and new warp-group level Tensor Cores instructions are introduced, supporting larger instruction sizes. Fifth-generation Tensor Cores, introduced in the Blackwell architecture in 2024, support FP6 and FP4 data types.

2.2 Mixed Precision Training

Mixed precision training [41] can help reduce the amount of memory required to train the model, ease the bandwidth requirement (e.g., off-chip memory and inter-node network bandwidth), and lower the computational power needed. It uses multiple precision formats; lower precision (e.g., FP16) is used in most of the network during the training while single precision (e.g., FP32) is used in the critical parts of the network (e.g., accumulation of gradients after each optimizer step) to maintain numerical stability and accuracy. Some of the hardware has FP32 units that can execute FP16 twice the rate of FP32, such as NVIDIA Pascal architecture [44], which improves training performance. With many advantages offered by mixed precision training, vendors try to find even more efficient data formats to train AI and ML models without sacrificing the performance of the models. Google introduced BF16, which retains the dynamic range of FP32 in 16-bit format [74], while NVIDIA introduced TF32, which retains the dynamic range of FP32 while keeping the accuracy of FP16 in 19-bit format [8].

2.3 Prior Evaluation of Tensor Cores

Since its introduction in 2017, Tensor Cores have been investigated in academic and industry research. Tensor Cores improve the performance of ML workloads by using mixed precision while maintaining model accuracy [39]. Memory-bound operations often see around two times speed-up thanks to the reduced data size (e.g., FP16). In contrast, compute-bound operations benefit from Tensor Cores depending on their arithmetic intensity [40]. Prior studies show the use of Tensor Cores on NVIDIA V100 GPU gives more than 2× speed-up in training for ResNet50 [57], GNMT [51], Inception v3, and Vgg16 models [76]. In addition, quantized inference gets up to 5× higher throughput and lower latency by using Tensor Cores inside NVIDIA Turing GPU [46] across many models, including ResNet50 v2, MobileNet v2, and SSD MobileNet v2 [18, 73]. Other models, including UNet Industrial Defect Segmentation, show a slight performance drop [58]. Prior works also include arithmetic accuracy studies for GEMM [2, 34, 59], scientific computation using double precision on third and fourth-generation Tensor Cores [14, 17], and mapping GEMM-like application into Tensor Cores [10], which include Fast Fourier Transform [13], reduction [43], scientific simulations [11], and linear system solver [19]. However, prior works mostly focused on convolution and GEMM-like workloads. Workloads such as GCNs and non-GEMM-like applications have not been studied. Finally, a study is done to characterize Tensor Cores latency, throughput, and numerical behavior to get the low-level detail of Tensor Cores [67]. However, it does not show how applications behave in different generations of Tensor Cores.

2.4 Programming Tensor Cores

With CUDA, programmers can develop applications that target NVIDIA GPUs using high-level languages, such as C, C++, Fortran, and Python. The high-level code is then compiled by a compiler (e.g., `nvcc`) to an intermediate assembly language called PTX (Parallel Thread eXecution) [27], whose ISA is openly documented [56]. The PTX instructions are then compiled to device-specific Streaming Assembly (SASS) either through ahead-of-time compilation using PTX assembler (e.g., `ptxas`) or just-in-time compilation by the display driver [66].

While developing applications that only utilize CUDA Cores can be done more easily using the high-level language of choice, developing applications that specifically target Tensor Cores to achieve higher performance is a different story. The instructions that run on Tensor Cores perform **matrix multiplication and accumulation (MMA)** [38] on a predefined dimension called instruction size. The programming model of Tensor Cores constructs this operation at the warp[2] level, which is different than the regular CUDA model which constructs the operation at the thread level [39]. Multiple sizes and operands are supported using `MMA`, including half-precision (`hmma`), integer (`imma`), binary (`bmma`), and double-precision (`dmma`). These instructions is accessed via PTX through inline assembly.

Since there is a limited number of instruction sizes for Tensor Cores, tiling must be done for operations that involve arbitrary dimensions of matrices. This consists of dividing the large matrices hierarchically at the grid[3] level into multiple thread block tiles, and further decomposing them

into warp tiles with multiple thread tiles utilizing all memory types in the hierarchy (e.g., global memory, shared memory, register files) [28]. Moreover, fulfilling the data layout and memory alignment requirements of Tensor Cores may not always be straightforward, especially for applications that have irregular data structures and computation patterns [15, 61, 75]. It is also challenging to handle sub-byte operations, such as INT4 and INT1 [6]. Therefore, programming Tensor Cores is an uphill task.

To overcome this issue, NVIDIA provides libraries that implement various functions that target Tensor Cores [3], and thus instead of having to write in-line assembly for PTX, developers can call the provided functions from their applications. Among the libraries include cuBLAS, cuSPARSE, cuTENSOR, cuDNN, and CUTLASS. CUTLASS is the only open-source library from the previous list that provides C++ template for developing high-performance

[2]Warp is a group of 32 threads concurrently executing the same instructions in a lock-step fashion. A collection of warp constitutes a thread block, which runs on an SM. The scheduler inside the SM will choose which warp runs based on the readiness of operands and perform context-switching across warps to hide memory access latency.
[3]Grid is a collection of thread blocks executing a GPU kernel.

Table 2: Hardware Configuration

Platform	DGX-A100	XE9680
GPU		
Model	NVIDIA A100	NVIDIA H100
Form Factor	SXM4	SXM5
Memory Size & Type	40 GB HBM2	80 GB HBM3
Memory Bandwidth	1,555 GBps	3,350 GBps
# CUDA/Tensor Cores	6912/432	16896/528
FP32 on CUDA Cores	19.5 TFLOP/s	67 TFLOP/s
FP16 on CUDA Cores	39 TFLOP/s	133.8 TFLOP/s
TF32 on Tensor Cores[1]	156/312 TFLOP/s	494.7/989.4 TFLOP/s
FP16 on Tensor Cores[1]	312/624 TFLOP/s	989.4/1978.9 TFLOP/s
CPU		
Model (# Sockets)	EPYC 7742 (2)	Xeon 8470 (2)
Base/Turbo Clock	2.25/3.4 GHz	2.00/ 3.80 GHz
Total Cores / Threads	128 / 256	104 / 208

[1] Dense/Sparse GEMM performance

matrix multiplications with support for Tensor Cores [70]. Other libraries such as cuBLAS and cuDNN are closed-source and contain multiple algorithms and implementations, including kernels from CUTLASS, to perform linear algebra and neural network operations, respectively. They use heuristics to choose the most optimized algorithms for a given problem and target devices [33, 72]. Even though libraries make developing applications that target Tensor Cores easier, the developer must still take care of data layout and memory alignment in order to correctly use Tensor Cores.

3 EXPERIMENTAL METHODOLOGY

3.1 Hardware and Software Setup

The experiments in this paper are conducted on two different platforms, each with different generations of NVIDIA GPU, as shown in Table 2. The NVIDIA A100 (Ampere) GPU is housed in the NVIDIA DGX-A100 chassis and features third-generation Tensor Cores while the NVIDIA H100 (Hopper) GPU is housed in Dell PowerEdge XE9680 chassis and features fourth-generation Tensor Cores with double the throughput of its predecessor. Both GPUs have sparsity support in their Tensor Cores which is expected to be useful for GCN that has some sparse matrix multiplications (spMM) [23, 85]. For simplification, NVIDIA A100 and NVIDIA H100 GPUs will be referred to as A100 and H100, respectively.

On the software side, the DGX-A100 is equipped with CUDA Toolkit 11.8, along with Python 3.11.4 and PyTorch 2.0.1. Meanwhile, the Dell PowerEdge XE9680 uses CUDA Toolkit 12.0, along with Python 3.11.4 and PyTorch 2.0.1 built from the source.

3.2 Performance Measurement

The **Nsight Compute** (`ncu`) is used to characterize kernels of each workload to gain access to their low-level detail. Kernel runtime is measured by collecting `gpu__time_duration` metric with cache and clock control disabled. For measuring kernel runtime in microbenchmark, the kernel is run 100 times and the average is taken. In addition, instruction count and DRAM transactions are collected. The FLOPs number is derived from the instruction count after multiplying with the weight (e.g., `fma`, `fadd`, and `fmul` have weight of 2, 1, and 1, respectively). The weight of Tensor Cores instruction is obtained based on instruction size.

The training performance for ML workloads is measured using a wall clock. For CNN, the model is trained using their respective dataset in 10 epochs with a default batch size of 128 for Image Classification and 4 for Object Detection. On the other hand, the GCN is trained in 1000 epochs because the model and dataset are small. Wall clock time measurements for determining speed-up use a large number of epochs while `ncu` profiling for roofline and instruction-mix analysis use 2 and 5 epochs for CNNs and GCN, respectively, to ensure acceptable running times with the profiler.

3.3 Profiling Tensor Cores

The legacy NVIDIA profiling tool `nvprof` [47] only provides single metric that indicates whether tensor cores are being used by a particular GPU kernel, which is accessible through `tensor_precision_fu_utilization` metric. This legacy profiling tool is no longer supported since Ampere. Meanwhile, its successor, the NVIDIA NSight Compute (`ncu`) [53] provides access to more valuable metrics on Tensor Cores with support starting from Volta.

Prior to CUDA 12.2, `ncu` provides access to `sm__inst_executed_pipe_tensor_op_xmma` to count the number of instructions being executed by Tensor Cores. It also provides access to measure Tensor Cores utilization through `sm__pipe_tensor_cycles_active` and `sm__pipe_tensor_op_xmma_cycles_active`. Note that `xmma` can be `dmma`, `hmma`, and `imma`. While the metrics are useful to indicate the interaction of the applications with the Tensor Cores, more efforts are needed to obtain more characterization metrics, such as the total number of FLOPS being executed, which is important for roofline analysis [78]. The instructions being executed on the Tensor Cores may have different shapes, which contain a different number of FLOPs per instruction. Sometimes, the kernel name suggests the instruction size being used [64] (e.g., `ampere_h16816gemm_...` means it uses `hmma.16x8x16`, which contains 4096 FLOPs per instruction), but it is difficult and is not a universal solution. Some instructions are difficult to infer the number of FLOPs without documentation, such as `hfma2.mma` that contain 4 FLOPs [31].

Finally, `ncu` shipped with CUDA 12.2 in June 2023 provides more detailed information on how many FLOPs (or IOPs) are executed on Tensor Cores. It provides access to the new metric `sm__ops_path_tensor_src_(in)_dst_(out)` where `(in)` and `(out)` are input data type and output data type. respectively. It can also be used to identify sparse FLOPs and dense FLOPs. These metrics make profiling applications that target Tensor Cores easier, especially those that use `wgmma.mma_async` in Hopper.

3.4 Workload Configuration

To evaluate Tensor Cores' performance, two groups of workloads are prepared, consisting of end-to-end ML training and microbenchmark as shown in Table 3 and 4, respectively.

3.4.1 Machine Learning Workloads. The CNN workloads consist of four models with two different tasks, as shown in Table 3. The ResNet50 [21] and EfficientNet [69] are CNN models for image classification, which are trained using the ImageNet dataset [12]. In addition, Faster-RCNN [62] and RetinaNet [35] are CNN models for Object Detection, which are trained using COCO dataset [36].

In addition to CNNs, which have been widely evaluated on GPUs, we also use GCNs as an emerging ML workload for this study. The GCN consists of only one model [29] with two different tasks: semi-supervised node classification tasks, either transductive or inductive [77]. For the transductive approach, PubMed [65] and Chameleon [63] datasets are used, while for the inductive approach, Yelp [82] and Reddit [20] datasets are used. Interested readers can obtain

Table 3: Machine Learning Workload Configuration

Type	Model	Task	Dataset
CNN	ResNet50	Image Classification	ImageNet
CNN	EfficientNet		
CNN	FasterRCNN	Object Detection	COCO
CNN	RetinaNet		
GNN	GCN	Node Classification Transductive	PubMed
GNN	GCN		Chameleon
GNN	GCN	Node Classification Inductive	Reddit
GNN	GCN		Yelp

Table 4: Microbenchmark Kernels Configuration

Type	ID	Dimension	Config
GEMM {m,n,k}	512	{512,512,512}	fp32 fp16.1688 fp16.16816
	2K	{2048,2048,2048}	
	8K	{8192,8192,8192}	
	32K	{32768,32768,32768}	
GEMV {m,1,k}	512	{512,1,512}	fp32 fp16.1688 fp16.16816
	2K	{2048,1,2048}	
	8K	{8192,1,8192}	
	32K	{32768,1,32768}	
Conv2D {N,H,W,C}; {K,R,S}; {U,V}	A	{64,1024,1024,32}; {16,32,32}; {1,1}	fp32 fp16
	B	{64,1024,1024,32}; {16,32,32}; {1,1}	
	C	{256,512,512,32}; {16,32,32}; {1,1}	
	D	{64,1024,1024,32}; {16,4,4}; {8,8}	
	E	{32,512,512,256}; {16,32,32}; {1,1}	
	F	{32,512,512,32}; {256,32,32}; {1,1}	
FIR {s,f}	8M4	{8388608, 4}	fp32.af fp16.ig fp16.rg
	8M8	{8388608, 8}	
	32M8	{33554432, 8}	
	32M16	{33554432, 16}	
	128M16	{134217728, 16}	
	128M32	{134217728, 32}	
ElWiseAdd {v}	256K	{262144}	fp32 fp16
	4M	{4194304}	
	64M	{67108864}	
	1B	{1073741824}	

more details from the original paper [29], a review by Zhang et al. [83, 84], and a summary by Heidar et al. [22],

PyTorch [60] is used as the framework to perform ML training in this experiment. All of the CNN models are taken from TorchVision [37] while the GCN model is taken from CogDL [5], a research toolkit for deep learning graphs. This toolkit integrates the original code from Kipf et al. [29] with built-in methods to load various datasets, making it easier to do experiments. For the FP32 (full-precision) training flow, PyTorch Automatic Mixed Precision (AMP) is disabled to avoid Tensor Cores usage, while for the FP16 (mixed-precision), AMP is enabled, allowing Tensor Cores usage.

3.4.2 Microbenchmark Kernels. The microbenchmark consists of five kernels with configurations given in Table 4. The kernels are developed using C++ and CUDA which target CUDA Cores or Tensor Cores. The kernels have customizable precision, input dimensions, target execution units, and libraries. Except otherwise noted, CUTLASS [70] is the library used for two reasons: 1) CUTLASS is open-source, which allows modification of template header or low-level assembly; and 2) CUTLASS is deterministic in terms of overall execution, which allows using application replay in `ncu` for profiling while cuBLAS use heuristics to choose the best kernel depending on problem size and device. The FP32 and F16 implementations target CUDA Cores and Tensor Cores, respectively.

- **GEMM**: General matrix-matrix multiply with dimensions {m,n,k} denoting $A_{m\times n} \times B_{n\times k} = C_{m\times k}$ where A, B, C are matrices. GEMM is well-supported by CUTLASS, which has one of the most efficient hierarchical GEMMs supporting CUDA Cores or Tensor Cores. However, data layout in memory must be taken care of carefully [28]. FP16 implementation uses two Tensor Cores instructions: `hmma.16816` (fp16.16816) and `hmma.1688` (fp16.1688).
- **GEMV**: General matrix-vector multiply with dimensions {m,1,k} can be viewed as a special case of GEMM. It follows the same implementation as GEMM.
- **Conv2D**: Two-dimension convolution is decomposed into implicit GEMM [86] by CUTLASS on CUDA Cores or Tensor Cores. The Conv2D kernel has multiple configurations with {N,H,W,C} denotes batch size, height, width, and number of input channels, respectively. In contrast, {K,R,S} denotes the number of channels, height, and width of the filter, respectively. The {U,V} are horizontal and vertical stride, respectively.
- **FIR**: 1D Finite Impulse Response filtering which operates on 1D signal s and 1D filter f. The FP32 uses the ArrayFire library (fp32.af) [79] while the FP16 implementation is not supported by Tensor Cores by default. Although earlier studies have tried to map FIR into Tensor Cores, they use 2D signals and filters [30]. Therefore, for the purpose of this experiment, two approaches to map 1D FIR into Tensor Cores are proposed as follows:
  – *Implicit GEMV (fp16.ig)*: This approach is done by modifying CUTLASS implicit GEMM into implicit GEMV. Due to memory alignment requirements, many zero-padding needs to be added, resulting in 64× more operations than is necessary.
  – *Reshaped GEMV (fp16.rg)*: Another approach is to construct a matrix from a 1D signal, which will be multiplied by the vector containing the filter. Suppose a filter $f = \{f_0\ f_1\ f_2\}$ is applied into input signal $s = \{s_0\ s_1\ s_2\ s_3\}$. Then, a 4×3 matrix is constructed with first row $\{0\ 0\ s_0\}$, second row $\{0\ s_0\ s_1\}$, third row $\{s_0\ s_1\ s_2\}$, and fourth row $\{s_1\ s_2\ s_3\}$. Then, a GEMV can be performed between the signal matrix and the filter vector. This approach has one drawback regarding data reuse and memory usage where the same data appears multiple times (e.g., s_0 in the first row is the same data as s_0 in the second row but stored twice in the memory).
- **ElWiseAdd**: Element-wise vector addition operates on two vectors of the same configurable lengths $\{v\}$. The FP32 implementation (fp32) uses only C++, while the FP16 implementation (fp16) is not supported in Tensor Cores. While cuBLAS supports vector addition operation, which can be represented by $ax + y$ with scaling factor $a = 1$, at the time of writing, cuBLAS only supports this operation in CUDA Cores for single precision and double precision with no Tensor Cores support [50]. Therefore, to be able to run vector addition in Tensor Cores, both vectors must be transformed into matrices to follow the Tensor Cores operation shown in Figure 2 with matrix B being an identity matrix and matrix A and C are the two input vectors. The multiplication cannot be skipped as it is the basic operation of Tensor Cores (i.e., `mma`), resulting in expensive computation and memory access.

4 EVALUATION & DISCUSSION

4.1 What do Tensor Cores bring to the table over CUDA Cores?

Tensor Cores provide a significant jump in compute throughput for GEMM and GEMM-like kernels if specific precisions are used. Figure 3 presents the speed-up achieved for CNN and GCN workloads (Table 3), and microbenchmark kernels (Table 4) on H100.

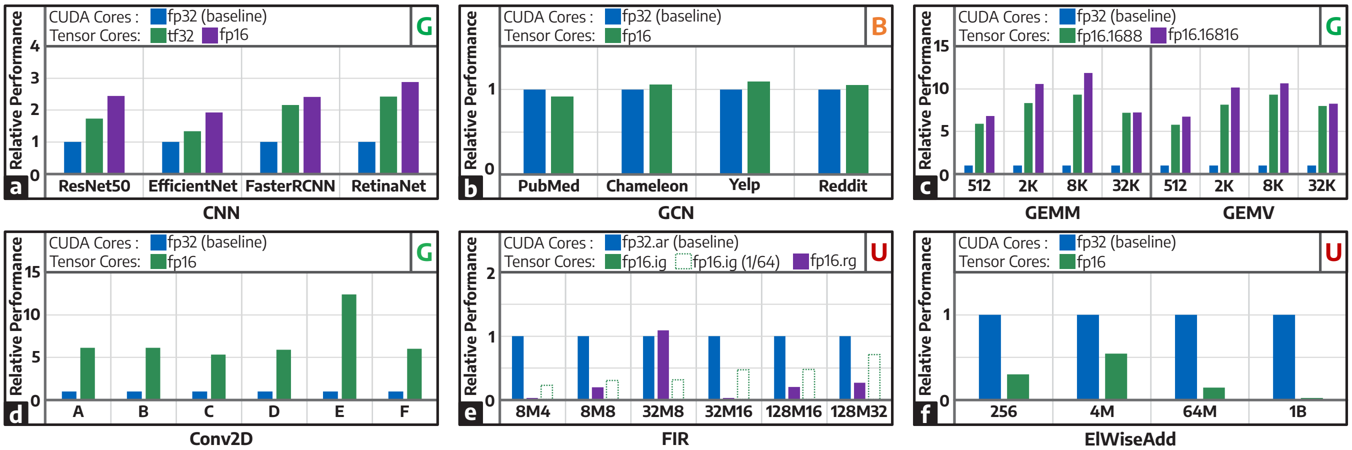

Figure 3: The speed-up obtained by using Tensor Cores over CUDA Cores on H100 across CNN/GCN workloads, and microbenchmark kernels. CNNs, Conv2D, and GEMM/GEMV are high performers with G, B, and U indicate Good, Bad, and Ugly, respectively.

4.1.1 CNN Workloads. Figure 3-a illustrates that going from FP32 on CUDA Cores to TF32 on Tensor Cores gives an average 1.91× speed-up while going from FP32 on CUDA Cores to FP16 on Tensor Cores gives an average 2.42× speed-up. To run FP32 full precision training on CUDA Cores as the baseline, PyTorch Automatic Mixed Precision (AMP) is explicitly disabled. However, the underlying CUDA libraries (e.g., cuBLAS, cuDNN) automatically demote FP32 to TF32 [8] to take advantage of Tensor Cores. Hence, an environment variable `NVIDIA_TF32_OVERRIDE=0` is set to tell CUDA libraries not to use TF32 explicitly. As a result, there are three configurations shown in Figure 3-a: full precision FP32 (blue), full precision TF32 (green), and mixed precision FP16 (purple).

4.1.2 GCN Workloads. Unlike CNN, GCN uses FP32 by default for full precision training, most likely due to the CogDL [5] that does not take advantage of TF32 on the underlying CUDA libraries. Furthermore, as shown in Figure 3-b, it only sees an average speed-up of 1.03× when going from FP32 on CUDA Cores to FP16 on Tensor Cores. Further explanation using rooflines and matrix instruction usage is provided in Section 4.2.4 and Section 4.3.2.

4.1.3 Microbenchmark Kernels. In summary, GEMM, GEMV, and Conv2D kernels get the performance benefit while FIR and ElWiseAdd experience performance degradation, as discussed below.

- **GEMM:** GEMM gets an average speed-up of 7.69× and 9.14× for fp16.1688 and fp16.16816, respectively, as shown in Figure 3-c. The highest speed-up is observed with GEMM_8K at 9.32× for fp16.1688 and 11.89× for fp16.16816, before dropping to 7.25× and 7.20× , respectively, for GEMM_32K. The GEMM_32K has vastly more elements (200M for GEMM_8K vs. 3.2B for GEMM_32K) and more intermediate results, exacerbating the data movement between on-chip and off-chip memory, which will become clear when we perform roofline analysis in Section 4.2.5.
- **GEMV:** GEMV gets performance benefits from Tensor Cores, although its average speed-up is lower than GEMM due to its lower arithmetic intensity. The achieved average speed-up is 7.82× for fp16.1688 and 8.96× for fp16.16816 (Figure 3-c).
- **Conv2D:** Since Conv2D is decomposed into implicit GEMM, it can take advantage of Tensor Cores; it achieves an average speed-up of 6.99× (Figure 3-d). The highest speed-up of 12.42× comes from Conv2D_E, whose reason will become clear in Section 4.2.5.
- **FIR:** Both FP16 implementations that target Tensor Cores show significant performance degradation as shown in Figure 3-e; the fp16.rg and fp16.ig only achieve an average of 0.30× and 0.01× performance achieved by fp32.ar that runs on CUDA Cores, respectively. The fp16.rg has redundant operations (Section 3.4.2), causing the performance drop for larger signal and filter dimensions. The fp16.ig is even more slower than the fp16.rg because of the 64 times more operations it needs to perform due to the zero-padding (Section 3.4.2). Even if there is a way to make these additional operations useful (e.g., having batched inputs with the same FIR filter or multiple independent FIR filters), it still cannot compete with the fp32.ar for large signal size (dashed green bars).
- **ElWiseAdd:** Like the FIR, ElWiseAdd also sees performance degradation, especially for larger dimensions, where it achieves an average of 0.25× performance offered by CUDA Cores as shown in Figure 3-f. While the matrix addition is fast, the multiplication with the identity matrix that cannot be skipped is expensive, especially in larger dimensions (Section 3.4.2).

4.2 Is Compute the Bottleneck or Memory?

4.2.1 Overview of Roofline Model. We use roofline charts [78] to visualize the achieved performance of applications or kernels compared to the hardware's compute capabilities and draw insights on the arithmetic intensity of applications. Both axes of the model are plotted in logarithmic scale: the y-axis represents the compute throughput (e.g., floating-point operations per second) while the x-axis represents the arithmetic intensity, which is the amount of computing that can be done per byte of data (e.g., floating-point operations per byte). The hardware roofline model, which can be obtained theoretically (e.g., from manufacturer datasheet, such as the data provided in Table 2) or empirically (e.g., using Empirical Roofline Toolkit [80]), consists of peak compute throughput, drawn as the roof, and the peak memory bandwidth (e.g., off-chip memory, cache bandwidth), drawn as the slope. Using data obtained from profiling tools (e.g., execution duration, the number of operations, and the number of memory read and write), the position of each application or kernel in the roofline chart can be determined, which gives insight whether the application or kernel is compute- (i.e., closer to the roof) or memory-bound (i.e., closer to the slope) and what optimization techniques should be performed.

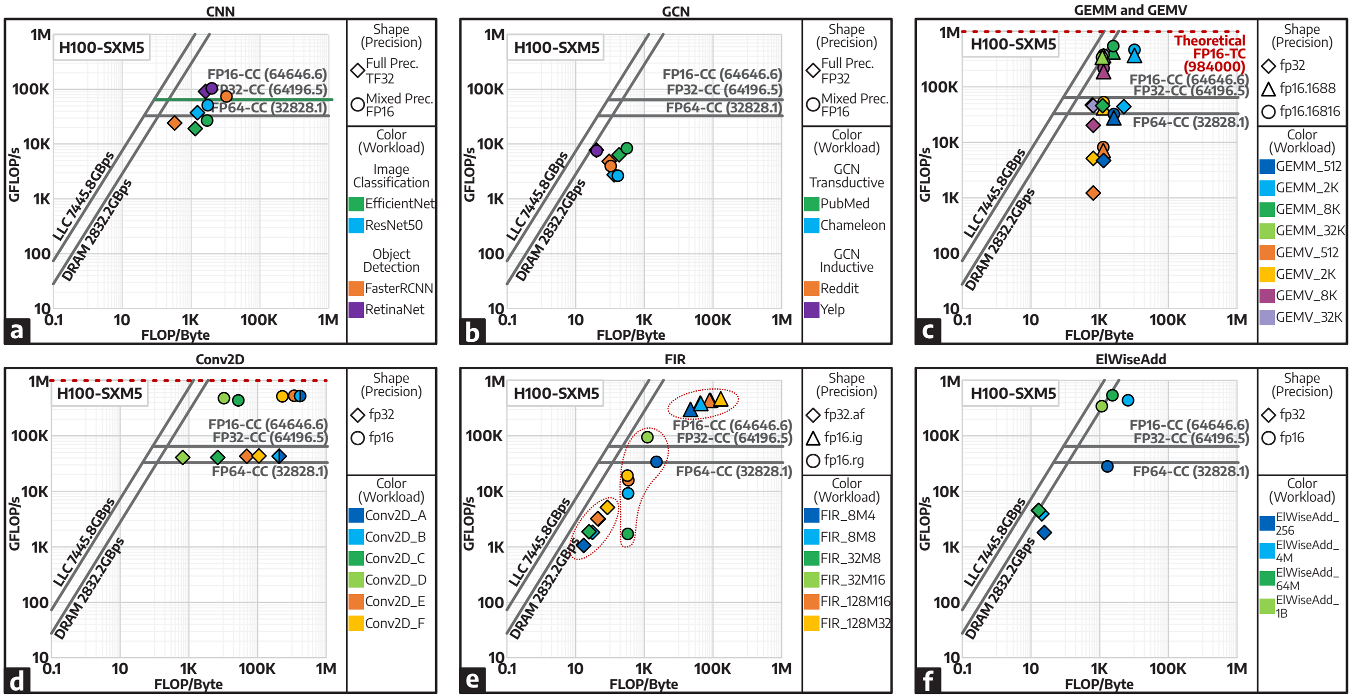

Figure 4: The roofline model for H100 (obtained using ERT), and the characterization of CNN/GCN workloads, and microbenchmark kernels. GCN has less than 1K flops/byte while Conv2D goes above 100K flops/byte. Diamond shape indicates baseline without Tensor Cores and triangles/circles indicate Tensor Cores versions.

4.2.2 H100 Roofline Model. Figure 4 shows the hardware roofline model for H100, obtained using ERT [80]. There are two points to highlight: 1) The roofs represent the peak compute throughput of the CUDA Cores: 64.64 TFLOP/s for FP16 (FP16-CC), 64.12 TFLOP/s for FP32 (FP32-CC), and 32.82 TFLOP/s for FP64 (FP64-CC) since, at the time of writing, ERT does not support `hmma` nor `hgmma` to measure the peak compute throughput of Tensor Cores (FP16-TC); and 2) the ERT is only able to achieve 50% of the theoretical compute throughput of FP16 on CUDA Cores (Table 2). The latter may be caused by two reasons: 1) ERT may need to be updated to account for new architecture, or 2) The CUDA Cores of Hopper may have the same FP16 compute throughput as the FP32. This happens with Ada Lovelace [54] (e.g., NVIDIA L40S [55]), which shares some of the architecture with Hopper, although Hopper datasheet mentions FP16 to be twice the rate of FP32 on CUDA Cores [49]. For the bandwidth, ERT is able to achieve 2,832 GBps on the HBM3 DRAM (84.5% of 3,350 GBps theoretical bandwidth for H100).

4.2.3 CNN Workloads. The use of TF32 during full precision training (Section 4.1.1) allows all models to achieve significantly higher GFLOP/s, with some exceeding the FP32-CC roof, by leveraging Tensor Cores (Figure 4-a). The performance improvements in using TF32 compared to FP32 for full-precision training are two folds: 1) Convolution operations, which are abundant in CNN, can be done on Tensor Cores, which have significantly higher compute throughput than CUDA Cores; and 2) TF32 has lower 19-bit data size compared to FP32 32-bit data size, which reduces the pressure on the memory bandwidth. Furthermore, the use of FP16 on mixed precision training by enabling PyTorch Automatic Mixed Precision improves performance even further, which comes from the ability of Tensor Cores to compute FP16 at twice the rate of TF32 and slightly lower data size (16-bit FP16 vs. 19-bit TF32). Special mention goes to FasterRCNN, shown in orange color, which gets the most benefits (i.e., biggest change in FLOPs/byte) from reduced memory bandwidth by switching from TF32 to FP16.

4.2.4 GCN Workloads. In general, all of the GCN workloads are memory-bound, even after switching from full-precision training (FP32) to mixed-precision training (FP16) as shown in Figure 4-b. The use of Tensor Cores for mixed-precision training has very few improvements in performance as discussed in Section 4.1.2; only PubMed, shown in green, enjoys some improvements compared to other GCN configurations in terms of arithmetic intensity and achieved compute throughput. However, it does not translate to positive speed-up (Figure 3) due to extra operations needed when using Tensor Cores (e.g., COO to CSR sparse matrix format conversion).

4.2.5 Microbenchmark Kernels. The roofline analysis for each kernel of the microbenchmark is given as follows.

- **GEMM:** The GEMM kernels are shown in dark blue, light blue, dark green, and light green colors in Figure 4-c. The dimension of GEMM_512 (dark blue) is too small to take advantage of the compute throughput offered by either CUDA Cores (diamond) or Tensor Cores (triangle and circle). Meanwhile, the other GEMM configurations (GEMM_2K, GEMM_8K, GEMM_32K) in FP32 (diamond) can almost saturate the FP32 compute throughput offered by CUDA Cores (i.e., almost hitting the roof of FP32-CC). The FP16 version of GEMM_8K and GEMM_32K (dark green triangle, green triangle, dark green circle, and green circle) can push through the roof of FP16-CC thanks to the use of Tensor Cores until the memory bandwidth of HBM3 DRAM becomes their limit. The theoretical FP16 performance of the Tensor Cores in H100 is 989 TFLOP/s (Table 2), which most likely won't be achieved by

GEMM due to memory bandwidth limitation. The use of larger `hmma.16816` (fp16.16816), denoted by circle, gives higher compute throughput compared to the `hmma.1688` (fp16.1688), denoted by triangle, while giving the same arithmetic intensity. Finally, it is worth mentioning that the GEMM_2K has a significantly higher FLOP/byte compared to other configurations. The dimension of the matrices is small enough to fit into on-chip memory. The 12 Million (2048×2048×3) FP16 elements have a total size of around 24 MB while H100 has 33 MB of registers, 33 MB of combined L1 Cache and Shared Memory, and 50 MB of L2 cache.

- **GEMV:** The GEMV kernels are shown in orange, yellow, purple, and violet colors in Figure 4-c. GEMV has lower data reuse compared to GEMM, and hence lower arithmetic intensity (i.e., located to the left of GEMM counterparts) and lower number of operations, especially for lower dimensions GEMV_512 and GEMV_2K (orange and yellow) whose FP32 versions (diamond) cannot fully utilize the available CUDA Cores on H100. On the other hand, the largest dimension (GEMV_32K) can almost hit the FP32-CC roof. Moving to FP16 versions (triangle and circle), only GEMV_8K and GEMV_32K can push through the roof of FP16-CC until they hit the memory bandwidth slope. Like the GEMM, the use of `hmma.16816` (fp16.16816) gives higher compute throughput compared to the `hmma.1688` (fp16.16816) on GEMV.
- **Conv2D:** As mentioned earlier in Section 3.4.2, the 2D Convolution is decomposed into implicit GEMM. The 2D convolution has more data reuse compared to GEMM, where the data reuse mostly comes from the use of 2D filters, which are applied to many 2D input signals. As shown in Figure 4-d, in general, both FP32 (diamond) and FP16 (circle) of Conv2D almost reach the roof of FP32-CC and the theoretical roof of FP16-TC (drawn as a dashed red line), respectively. The Conv2D_A (dark blue) and Conv2D_B (light blue), which have sixteen 32×32 filters (Table 4), have the most data reuse, leading to the highest arithmetic intensity (i.e., located to the right side of the roofline chart). The amount of memory needed to store all of these filters in both FP32 and FP16 are 64 KB and 32 KB, respectively, which can be stored sufficiently inside the shared memory of H100 (256 KB of combined L1+Shared memory per SM). On the other hand, Conv2D_C (dark green) and Conv2D_D (light green) have the least data reuse due to the smaller size of filters being used (Conv2D_C) and the larger convolution stride (Conv2D_D). Moving to FP16 with Tensor Cores (circle), all Conv2D configurations push through the FP16-CC roof. Special mention goes to Conv2D_E (orange) with its 256 input channels and smaller 512×512 input signals that allow for more data reuse. It almost achieves the theoretical FP16 peak performance of Tensor Cores, followed by Conv2D_F (yellow).
- **FIR:** Figure 4-e shows three clusters of workloads, which correspond to three implementations of FIR as discussed in (Section 3.4.2): fp32.ar (diamond), fp16.rg (circle), and fp16.ig (triangle). The FP32 version (fp32.ar) is already bandwidth-limited, with all of them positioned near each other at the slope of HBM3 DRAM. This also indicates that Tensor Cores cannot accelerate FIR as it is already bandwidth limited, unlike GEMM, GEMV, and Conv2D. The fp16.rg implementation has higher arithmetic intensity due to the redundant operations as a result of how the signal's data is laid out to form a matrix as discussed in Section 3.4.2. On the other hand, the fp16.ig tries to mimic the implicit GEMM that Conv2D has, except it uses implicit GEMV. Nevertheless, both approaches to map FIR to Tensor Cores (fp16.ig and fp16.rg) show unfavorable results compared to the fp32.ar on CUDA Cores.
- **ElWiseAdd:** Figure 4-f shows the FP32 version (diamond) of element-wise addition is already memory-bound with very low arithmetic intensity, hitting the slope of HBM3 DRAM bandwidth. On the other hand, the FP16 version (circle) has higher compute throughput and arithmetic intensity, which solely comes from the fact that the element-wise addition must be transformed to matrix-multiply-accumulate operations to be able to use Tensor Cores. Sadly, this does not improve performance since the multiplication is expensive, especially for large matrix sizes.

4.3 What Percentage of Floating-Point Instructions Offloaded to Tensor Cores?

Figure 5 shows the floating-point instruction/operation mix for CNN, GCN, and microbenchmark kernels. Since Tensor Cores instruction performs multiple floating-point operations, the weighted numbers are used (Section 3.2). The instruction/operation mix gives insight into what instructions could be offloaded to Tensor Cores.

4.3.1 CNN Workloads. As previously discussed in Section 4.2.3, the underlying CUDA libraries demote the FP32 to TF32 for full precision training in order to use Tensor Cores. This is further confirmed by the instruction mix shown in Figure 5 (top four sets) where most floating-point instructions are TF32 running on Tensor Cores with `hmma.1688` instructions shown as yellow bar (e.g., GEMM kernel `sm80_xmma..._tf32f32...`) and newer `hgmma` shown as olive-green bar (e.g., GEMM kernel `sm90_xmma..._tf32f32...`). Small percentage of operations are still executed by CUDA Cores as shown by the green (FP16) and blue (FP32) bar, which come from kernels that cannot be mapped into Tensor Cores (e.g., element-wise). Moving to mixed precision training with FP16, the composition is largely the same with FP16 running on Tensor Cores with `hmma.1688` (light orange bar), `hmma.16816` (dark orange bar), and newer `hgmma` (dark brown bar) instructions. It is worth mentioning that `ncu` shipped with CUDA 12.2 is used to calculate the number of floating-point operations that `hgmma` instructions do as it is difficult to infer this information from kernel name alone (Section 3.3).

4.3.2 GCN Workloads. Unlike CNN, the full precision training on GCN uses FP32 on CUDA Cores as shown in Figure 5 (middle four sets of bars) where majority of the instructions are `ffma`. Moving to mixed precision training with FP16, none of them use the newer `hgmma` instructions on Tensor Cores; the majority use `hmma.1688` and `hmma.16816` with Chameleon is observed to use older `wmma.161616` instructions. In addition, a small number of FP32 and FP16 instructions are executed on CUDA Cores, particularly for element-wise kernels, which are many in GCN workloads, outweighing the speed-up provided by Tensor Cores.

4.3.3 Microbenchmark Kernels. Unlike CNN and GCN workloads, the data type and instruction size used in the microbenchmark kernels can be specified explicitly. The lowest four sets of bars illustrate the microbenchmarks in Figure 5.

- **GEMM, GEMV, and Conv2D:** Both GEMM and GEMV have the instruction mix corresponding to the data type and the instruction size used: FP32 mostly uses `ffma` on CUDA Cores (blue bar) while FP16 mostly uses either `hmma.1688` (light orange bar) or

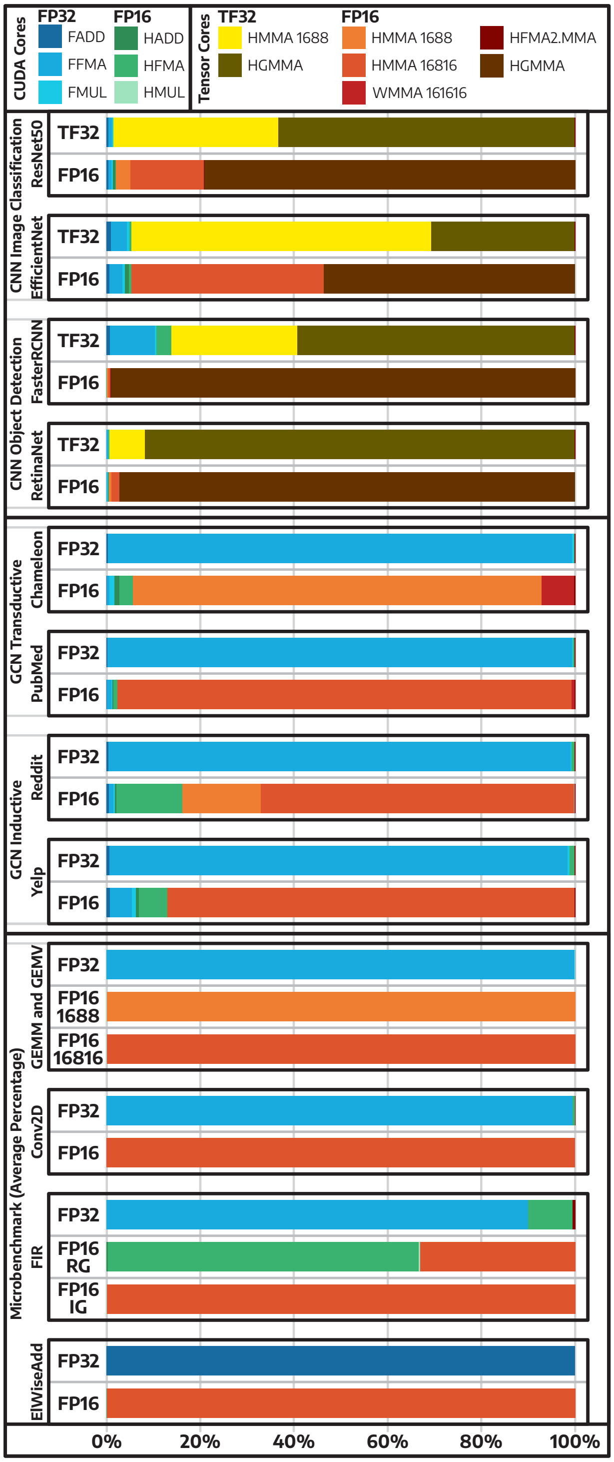

Figure 5: The floating-point instruction mix of CNNs, GCNs, and microbenchmark kernels on H100 utilizing CUDA libraries (e.g., cuDNN, cuBLAS, CUTLASS). Note that full precision training in CNN will, by default, use TF32 instead of FP32. Yellow/brown/orange/red run in Tensor Cores and blue/green run in CUDA Cores.

`hmma.16816` (dark orange bar) on Tensor Cores. Since Conv2D is decomposed to implicit GEMM, it follows the behavior of GEMM.

- **FIR:** The fp32.ar implementation uses `ffma` and `hfma` that runs on CUDA Cores. On the other hand, the fp16.rg implementation still has the majority of the FP16 instructions executed in CUDA Cores as `hfma` (green bar) while some of the instructions are executed in Tensor Cores with `hmma.16816` instructions. Finally, the fp16.ig implementation spends the majority of the instructions on Tensor Cores as `hmma.16816`. Only a small percentage of Tensor-Core-bound instructions are useful since most of them are due to padding and memory alignment.
- **ElWiseAdd:** The FP32 implementation uses `fadd` on CUDA Cores while the FP16 implementation uses `hmma.16816` on Tensor Cores. Unfortunately, for FP16, most of the instructions are spent on the expensive matrix-multiply operations, which are not useful since the only useful operation is addition.

4.4 How much more performance does H100 provide over A100?

Table 2 shows the theoretical peak performance of H100 is 3.4× in FP32 and FP16 on CUDA Cores and 3.2× in TF32 and FP16 on Tensor Cores compared to A100. The H100 achieves these theoretical performance improvements by having 2.5× higher number of CUDA Cores (16896 vs. 6912), doubling the Tensor Cores throughput per SM per cycle, doubling the memory bandwidth (3.3 TB/s vs. 1.5 TB/s), pushing the TDP higher (700 W vs. 400 W), running at higher sustained clock frequency (1980 MHz vs. 1410 MHz), and having other new features that help with execution efficiency. This section compares the achieved performance improvements of H100 over its predecessor, the A100, for the experimented CNNs, GCNs, and microbenchmark kernels as shown in Figure 6.

4.4.1 CNN Workloads. The H100 achieves an average of 1.96×, 1.96×, and 1.88× speed-up for FP32 on CUDA Cores, TF32 on Tensor Cores, and FP16 on Tensor Cores, respectively, across four CNN workloads over A100 as shown in Figure 6-a.

4.4.2 GCN Workloads. Figure 6-b shows the speed-up achieved by H100 over A100 on GCN. We observed a significantly high speedup on GCN with Yelp and Reddit datasets. For GCN with PubMed and Chameleon datasets, performance improvements on H100 over A100 are insignificant, with an average speed-up of 1.12×. When running GCN training on H100 with CUDA 12.0, the Chameleon mixed precision training flow is broken while its full precision shows double the time needed compared to A100. Reverting back to CUDA 11.8 solves the issue. Interestingly, it is the other way around for both Yelp and Reddit which enjoy significant improvements when using CUDA 12.0 on H100 for two reasons: 1) sparse-matrix multiplication (`spmm` [23]) kernel is being used, which is not found when running on A100; 2) the use of newer `hgmma` instruction.

4.4.3 Microbenchmark Kernels. The microbenchmark kernels that target Tensor Cores use either `hmma.1688` and `hmma.16816` instructions; none of them use the newer `hgmma` instructions supported by H100, which may affect the attainable performance.

- **GEMM, GEMV, and Conv2D:** H100 achieves average speed-up of 3.01×, and 2.36×, and 1.98× for GEMM with FP32 on CUDA Cores, FP16 using `hmma.16816` on Tensor Cores, and FP16 using `hmma.1688` on Tensor Cores, respectively over A100 (Figure 6-c). The speed-up is lower for GEMV with an average of 2.74×, 2.00×,

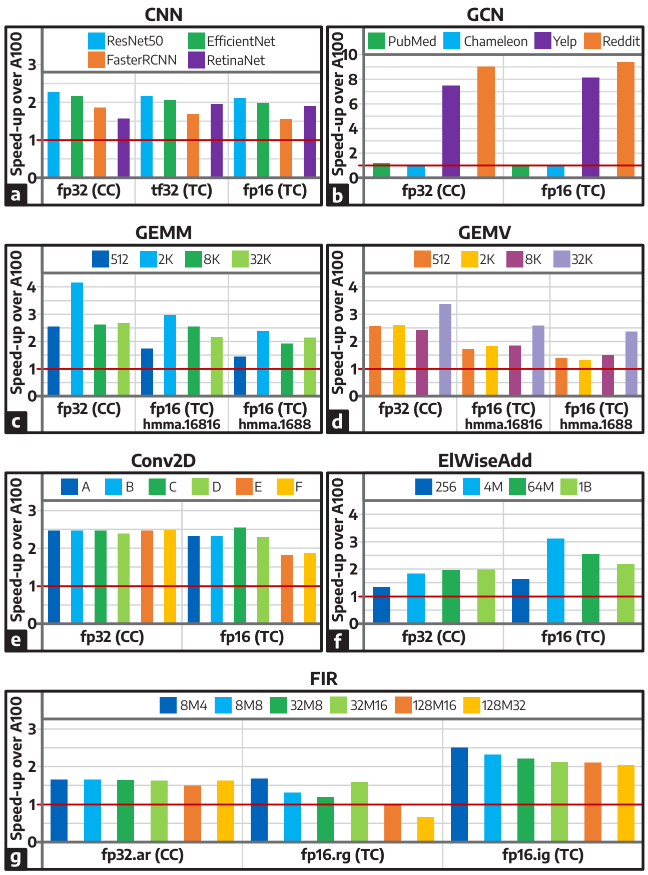

Figure 6: Significant speed-up is achieved by H100 over A100 for most benchmarks. Red line indicates A100 performance (baseline).

and 1.65×, respectively, due to lower arithmetic intensity (Figure 6-d). Finally, H100 reaches an average speed-up of 2.45× and 2.20× on Conv2D for FP32 (CUDA Cores) and FP16 (Tensor Cores) over A100, respectively (Figure 6-e).

- **FIR and ElWiseAdd:** While FIR (Figure 6-f) and ElWiseAdd (Figure 6-e) do not benefit from Tensor Cores, H100 achieved an average speed-up of 1.62×, 1.23×, 2.21×, 1.78×, and 2.37× for FIR fp32.ar, FIR fp16.rg, FIR fp32.ig, ElWiseAdd FP32, and ElWiseAdd FP16, respectively.

4.5 Discussion

4.5.1 Empirical Roofline Toolkit. The ERT [80] is a useful tool for creating a roofline model of the hardware. However, it does not have support to find the roof for Tensor Cores using either `mma` or `wgmma.mma_async`. From the roofline analysis (Figure 4), Conv2D is one of the likely kernels that can be used to measure the roof of Tensor Cores performance, since it can almost reach the theoretical peak throughput of Tensor Cores.

4.5.2 Profiling non-deterministic application. While it is recommended to use application replay when profiling using `ncu` [52] to avoid the overhead of kernel replay, profiling non-deterministic workloads such as ML training flows [87] may need to use kernel replay instead. Although we have followed steps to maintain reproducibility and control randomness in PyTorch [71], `ncu` with application replay is unable to consolidate profiling results due to the mismatch in kernel names and kernel launch parameters, which is an indication that the applications do not take the same execution path every time it runs during the replay.

4.5.3 Reshaping Optimizations. Both FIR and ElWiseAdd will not run on Tensor Cores without reshaping optimization to map them into GEMM-like operations (Section 3.4.2). Unfortunately, reshaping comes with costs due to memory alignment and padding, making the performance benefit of Tensor Cores difficult to come by. Finer control of Tensor Cores (e.g., the ability to skip the multiplication on MMA operations) may be beneficial for element-wise operations that often follow GEMM/GEMV operations by fusing both GEMM kernels and element-wise kernels to significantly reduce data movement and kernel switching overhead.

4.5.4 TensorFloat32. The TensorFloat32 (TF32) was introduced by NVIDIA along with third-generation Tensor Cores (Section 2.1) [8]. TF32 is a 19-bit data type with 8-bit exponent to retain the dynamic range of FP32 and 10-bit mantissa to achieve the same accuracy as FP16, which has been proven to be sufficient for ML workloads. Since TF32 can run on Tensor Cores and gives significant speed-up over FP32 on CUDA Cores, many frameworks that rely on NVIDIA libraries allow the demotion of FP32 to TF32 (e.g., through option `CUBLAS_TF32_TENSOR_OP_MATH` on cuBLAS) if the GPU supports TF32. While this may work fine for many ML workloads, it may cause numerical instability for applications where accuracy is important, such as in HPC applications. Therefore, making sure of precision to use is important (e.g., explicitly configure CUDA libraries to keep using FP32 when needed).

5 CONCLUSION

Tensor Cores provide significant speed-up for applications that have abundant GEMM operations. CNNs yield "Good" improvements with Tensor Cores, exemplified by the average speedups of 1.91× and 2.42× with TF32 and FP16 training, respectively, compared to FP32 training running on the CUDA Cores. Kernels like GEMM, GEMV, and Conv2D also show "Good" advantage of Tensor Cores with an impressive 8.4×, 8.39×, and 6.99× average speed-up, respectively. The Conv2D kernel almost saturates the FP16 theoretical performance of Tensor Cores on H100. On the other hand, FIR and ElWiseAdd kernels show performance degradation when running on Tensor Cores despite code transformations, making them "Ugly" kernels for Tensor Cores. Furthermore, GCN improvement with Tensor Cores can be classified as "Bad" since they only achieved 1.03× average speed-up and are sensitive to the changes in library versions. Finally, H100 gives an impressive 2.33× average speed-up across CNNs, GCNs, and microbenchmark kernels over A100. These speed-ups are mostly due to the H100 having 2.5× more CUDA Cores, double the throughput of Tensor Cores, and double the memory bandwidth compared to the A100.

ACKNOWLEDGMENTS

This research was supported in part by Semiconductor Research Corporation (SRC) Tasks 3015.001 and 3148.001, National Science Foundation (NSF) Grant #2326894, and NVIDIA Applied Research Accelerator Program Grant. Any opinions, findings, conclusions, or recommendations are those of the authors and not of the funding agencies. The authors would also like to thank the reviewers for their constructive feedback and suggestions.

REFERENCES

[1] Advanced Micro Devices. 2020. *AMD CDNA Architecture*. Whitepaper. Advanced Micro Devices, US.

[2] Pedro Martins Basso, Fernando Fernandes dos Santos, and Paolo Rech. 2020. Impact of Tensor Cores and Mixed Precision on the Reliability of Matrix Multiplication in GPUs. *IEEE Transactions on Nuclear Science* 67, 7 (2020), 1560–1565. https://doi.org/10.1109/TNS.2020.2977583

[3] Harun Bayraktar. 2020. How CUDA Math Libraries Can Help You Unleash The Power of The New NVIDIA A100 GPU. *NVIDIA GPU Technology Conference (GTC)* s21681 (May 2020).

[4] Davis Blalock and John Guttag. 2021. Multiplying Matrices Without Multiplying. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, Virtual, 992–1004. https://proceedings.mlr.press/v139/blalock21a.html

[5] Yukuo Cen, Zhenyu Hou, Yan Wang, Qibin Chen, Yizhen Luo, Zhongming Yu, Hengrui Zhang, Xingcheng Yao, Aohan Zeng, Shiguang Guo, Yuxiao Dong, Yang Yang, Peng Zhang, Guohao Dai, Yu Wang, Chang Zhou, Hongxia Yang, and Jie Tang. 2023. CogDL: A Comprehensive Library for Graph Deep Learning. In *Proceedings of the ACM Web Conference 2023* (Austin, TX, USA) *(WWW '23)*. ACM, New York, NY, USA, 747–758. https://doi.org/10.1145/3543507.3583472

[6] Junkyeong Choi, Hyucksung Kwon, Woongkyu Lee, Jieun Lim, and Jungwook Choi. 2022. Understanding and Optimizing INT4 Convolution for Accelerated DNN Inference on Tensor Cores. In *2022 IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE, Rennes, France, 1–6. https://doi.org/mdrz

[7] Jack Choquette. 2023. NVIDIA Hopper H100 GPU: Scaling Performance. *IEEE Micro* 43, 3 (2023), 9–17. https://doi.org/10.1109/MM.2023.3256796

[8] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. NVIDIA A100 Tensor Core GPU: Performance and Innovation. *IEEE Micro* 41, 2 (2021), 29–35. https://doi.org/10.1109/MM.2021.3061394

[9] Jack Choquette, Olivier Giroux, and Denis Foley. 2018. Volta: Performance and Programmability. *IEEE Micro* 38, 2 (2018), 42–52. https://doi.org/mdrs

[10] Rezaul Chowdhury, Francesco Silvestri, and Flavio Vella. 2020. A Computational Model for Tensor Core Units. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures* (Virtual Event, USA) *(SPAA '20)*. ACM, New York, NY, USA, 519–521. https://doi.org/10.1145/3350755.3400252

[11] Yi-Hua Chung, Cheng-Jhih Shih, and Shih-Hao Hung. 2022. Accelerating Simulated Quantum Annealing with GPU and Tensor Cores. In *High Performance Computing*, Ana-Lucia Varbanescu, Abhinav Bhatele, Piotr Luszczek, and Baboulin Marc (Eds.). Springer International Publishing, Cham, 174–191.

[12] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, Miami, Florida, USA, 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[13] Sultan Durrani, Muhammad Saad Chughtai, Abdul Dakkak, Wen-mei Hwu, and Lawrence Rauchwerger. 2021. FFT Blitz: The Tensor Cores Strike Back. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. ACM, New York, NY, USA, 488–489. https://doi.org/10.1145/3437801.3441623

[14] Boyuan Feng, Yuke Wang, Guoyang Chen, Weifeng Zhang, Yuan Xie, and Yufei Ding. 2021. 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* (Virtual Event, Republic of Korea) *(PPoPP '21)*. ACM, New York, NY, USA, 278–291. https://doi.org/mdr8

[15] Jesun Sahariar Firoz, Ang Li, Jiajia Li, and Kevin Barker. 2020. On the Feasibility of Using Reduced-Precision Tensor Core Operations for Graph Analytics. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Waltham, MA, USA, 1–7. https://doi.org/10.1109/HPEC43674.2020.9286152

[16] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. 2019. Xilinx Adaptive Compute Acceleration Platform: VersalTM Architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '19)*. ACM, New York, NY, USA, 84–93. https://doi.org/10.1145/3289602.3293906

[17] B. Gallet and M. Gowanlock. 2022. Leveraging GPU Tensor Cores for Double Precision Euclidean Distance Calculations. In *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE Computer Society, Los Alamitos, CA, USA, 135–144. https://doi.org/mdrr

[18] Chris Gottbrath. 2018. Using TensorRT to Unlock Tensor Core Performance for Inference. *NVIDIA GPU Technology Conference (GTC)* dc8169 (Oct 2018).

[19] Azzam Haidar, Harun Bayraktar, Stanimire Tomov, Jack Dongarra, and Nicholas J. Higham. 2020. Mixed-precision iterative refinement using tensor cores on GPUs to accelerate solution of linear systems. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 476, 2243 (2020), 20200110. https://doi.org/10.1098/rspa.2020.0110

[20] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 1025–1035.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385 [cs.CV]

[22] Negar Heidari, Lukas Hedegaard, and Alexandros Iosifidis. 2022. Chapter 4 - Graph convolutional networks. In *Deep Learning for Robot Perception and Cognition*, Alexandros Iosifidis and Anastasios Tefas (Eds.). Academic Press, Cambridge, Massachusetts, United States, 71–99. https://doi.org/mdrq

[23] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. GE-SpMM: General-Purpose Sparse Matrix-Matrix Multiplication on GPUs for Graph Neural Networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Atlanta, GA, USA, 1–12. https://doi.org/10.1109/SC41405.2020.00076

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

[25] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. 2021. Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Valencia, Spain, 1–14. https://doi.org/gnqdc9

[26] Pau San Juan, Pedro Alonso-Jordá, and Enrique S. Quintana-Ortí. 2021. High Performance and Energy Efficient Integer Matrix Multiplication for Deep Learning. In *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, Valladolid, Spain, 122–125. https://doi.org/10.1109/PDP52278.2021.00027

[27] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. 2009. A characterization and analysis of PTX kernels. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Austin, TX, USA, 3–12. https://doi.org/10.1109/IISWC.2009.5306801

[28] Andrew Kerr, Duane Merrill, Julien Demouth, John Tran, Naila Farooqui, Markus Tavenrath, Vince Schuster, Eddie Gornish, Jerry Zheng, and Bageshri Sathe. 2018. CUTLASS: CUDA Template Library for Dense Linear Algebra at All Levels and Scales. *NVIDIA GPU Technology Conference (GTC)* s8854 (Mar 2018).

[29] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. arXiv:1609.02907 [cs.LG]

[30] Takumi Kondo, Yoshihiro Maeda, and Norishige Fukushima. 2021. Accelerating Finite Impulse Response Filtering Using Tensor Cores. In *2021 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*. IEEE, Tokyo, Japan, 74–79.

[31] Thorsten Kurth, Shashank Subramanian, Peter Harrington, Jaideep Pathak, Morteza Mardani, David Hall, Andrea Miele, Karthik Kashinath, and Animashree Anandkumar. 2022. FourCastNet: Accelerating Global High-Resolution Weather Forecasting using Adaptive Fourier Neural Operators. arXiv:2208.05419 [physics.ao-ph]

[32] Ang Li and Simon Su. 2021. Accelerating Binarized Neural Networks via Bit-Tensor-Cores in Turing GPUs. *IEEE Transactions on Parallel and Distributed Systems* 32, 7 (2021), 1878–1891. https://doi.org/10.1109/TPDS.2020.3045828

[33] Cheng Li, Abdul Dakkak, Jinjun Xiong, and Wen-mei Hwu. 2020. Benanza: Automatic μBenchmark Generation to Compute "Lower-bound" Latency and Inform Optimizations of Deep Learning Models on GPUs. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, New Orleans, LA, USA, 440–450. https://doi.org/10.1109/IPDPS47924.2020.00053

[34] Guangli Li, Jingling Xue, Lei Liu, Xueying Wang, Xiu Ma, Xiao Dong, Jiansong Li, and Xiaobing Feng. 2021. Unleashing the Low-Precision Computation Potential of Tensor Cores on GPUs. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, Seoul, South Korea, 90–102. https://doi.org/10.1109/CGO51591.2021.9370335

[35] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. 2018. Focal Loss for Dense Object Detection. arXiv:1708.02002 [cs.CV]

[36] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. 2015. Microsoft COCO: Common Objects in Context. arXiv:1405.0312 [cs.CV]

[37] Sébastien Marcel and Yann Rodriguez. 2010. Torchvision the Machine-Vision Package of Torch. In *Proceedings of the 18th ACM International Conference on Multimedia* (Firenze, Italy) *(MM '10)*. ACM, New York, NY, USA, 1485–1488. https://doi.org/10.1145/1873951.1874254

[38] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. 2018. NVIDIA Tensor Core Programmability, Performance & Precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, Vancouver, BC, Canada, 522–531. https://doi.org/10.1109/IPDPSW.2018.00091

[39] Matt Martineau, Patrick Atkinson, and Simon McIntosh-Smith. 2019. Benchmarking the NVIDIA V100 GPU and Tensor Cores. In *Euro-Par 2018: Parallel Processing Workshops*, Gabriele Mencagli, Dora B. Heras, Valeria Cardellini, Emiliano Casalicchio, Emmanuel Jeannot, Felix Wolf, Antonio Salis, Claudio Schifanella, Ravi Reddy Manumachu, Laura Ricci, Marco Beccuti, Laura Antonelli, José Daniel Garcia Sanchez, and Stephen L. Scott (Eds.). Springer International Publishing, Cham, 444–455.

[40] Paulius Micikevicius. 2018. Training Neural Networks with Mixed Precision: Theory and Practice. *NVIDIA GPU Technology Conference* s8923 (Mar 2018).

[41] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018. Mixed Precision Training. In *International Conference on Learning Representations*. Open Review, Vancouver, BC, Canada, 12 pages. https://openreview.net/forum?id=r1gs9JgRZ

[42] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart Oberman, Mohammad Shoeybi, Michael Siu, and Hao Wu. 2022. FP8 Formats for Deep Learning. arXiv:2209.05433 [cs.LG]

[43] Cristóbal A. Navarro, Roberto Carrasco, Ricardo J. Barrientos, Javier A. Riquelme, and Raimundo Vega. 2021. GPU Tensor Cores for Fast Arithmetic Reductions. *IEEE Transactions on Parallel and Distributed Systems* 32, 1 (2021), 72–84. https://doi.org/10.1109/TPDS.2020.3011893

[44] NVIDIA Corporation. 2016. *NVIDIA Tesla P100: The Most Advanced Datacenter Accelerator Ever Built Featuring Pascal GP100, the World's Fastest GPU.* Whitepaper. NVIDIA Corporation, US.

[45] NVIDIA Corporation. 2017. *NVIDIA Tesla V100 GPU Architecture: The World's Most Advanced Data Center GPU.* Whitepaper. NVIDIA Corporation, US.

[46] NVIDIA Corporation. 2018. *NVIDIA Turing GPU Architecture: Graphics Reinvented.* Whitepaper. NVIDIA Corporation, US.

[47] NVIDIA Corporation. 2019. NVIDIA CUDA Toolkit Profiler User's Guide. https://docs.nvidia.com/cuda/profiler-users-guide/#nvprof.

[48] NVIDIA Corporation. 2020. *NVIDIA A100 Tensor Core GPU Architecture: Unprecedented Acceleration at Every Scale.* Whitepaper. NVIDIA Corporation, US.

[49] NVIDIA Corporation. 2022. *NVIDIA H100 Tensor Core GPU Architecture: Exceptional Performance, Scalability, and Security for The Data Center.* Whitepaper. NVIDIA Corporation, US.

[50] NVIDIA Corporation. 2023. cublasAxpyEx(). https://docs.nvidia.com/cuda/cublas/#cublasaxpyex.

[51] NVIDIA Corporation. 2023. GNMT v2 For PyTorch. https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/Translation/GNMT

[52] NVIDIA Corporation. 2023. Kernel Profiling Guide. https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#kernel-replay.

[53] NVIDIA Corporation. 2023. Nsight Compute CLI. https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html.

[54] NVIDIA Corporation. 2023. *NVIDIA Ada GPU Architecture: Designed to deliver outstanding gaming and creating, professional graphics, AI, and compute performance.* Whitepaper. NVIDIA Corporation, US.

[55] NVIDIA Corporation. 2023. *NVIDIA L40S Unparalleled AI and graphics performance for the data center.* Datasheet. NVIDIA Corporation, US.

[56] NVIDIA Corporation. 2023. Parallel Thread Execution ISA Version 8.2. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[57] NVIDIA Corporation. 2023. ResNet-50 v1.5 for MXNet. https://github.com/NVIDIA/DeepLearningExamples/tree/master/MxNet/Classification/RN50v1.5

[58] NVIDIA Corporation. 2023. UNet Industrial Defect Segmentation for TensorFlow. https://github.com/NVIDIA/DeepLearningExamples/tree/master/TensorFlow/Segmentation/UNet_Industrial

[59] Hiroyuki Ootomo and Rio Yokota. 2022. Recovering single precision accuracy from Tensor Cores while surpassing the FP32 theoretical peak performance. *The International Journal of High Performance Computing Applications* 36, 4 (2022), 475–491. https://doi.org/10.1177/10943420221090256

[60] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. arXiv:1912.01703 [cs.LG]

[61] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, San Diego, CA, USA, 58–70. https://doi.org/ggtwps

[62] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2016. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. arXiv:1506.01497 [cs.CV]

[63] Benedek Rozemberczki, Carl Allen, and Rik Sarkar. 2021. Multi-Scale Attributed Node Embedding. *Journal of Complex Networks* 9, 2 (2021), 22 pages.

[64] Valerie Sarge. 2020. Tensor Core Performance: The Ultimate Guide. *NVIDIA GPU Technology Conference (GTC)* s21929 (May 2020).

[65] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. 2008. Collective Classification in Network Data. *AI Magazine* 29, 3 (Sep. 2008), 93. https://doi.org/10.1609/aimag.v29i3.2157

[66] Mark Stephenson, Siva Kumar Sastry Hari, Yunsup Lee, Eiman Ebrahimi, Daniel R. Johnson, David Nellans, Mike O'Connor, and Stephen W. Keckler. 2015. Flexible Software Profiling of GPU Architectures. *SIGARCH Comput. Archit. News* 43, 3S (jun 2015), 185–197. https://doi.org/10.1145/2872887.2750375

[67] Wei Sun, Ang Li, Tong Geng, Sander Stuijk, and Henk Corporaal. 2023. Dissecting Tensor Cores via Microbenchmarks: Latency, Throughput and Numeric Behaviors. *IEEE Transactions on Parallel and Distributed Systems* 34, 1 (2023), 246–261. https://doi.org/10.1109/TPDS.2022.3217824

[68] Yufei Sun, Long Zheng, Qinggang Wang, Xiangyu Ye, Yu Huang, Pengcheng Yao, Xiaofei Liao, and Hai Jin. 2022. Accelerating Sparse Deep Neural Network Inference Using GPU Tensor Cores. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Waltham, MA, USA, 1–7. https://doi.org/10.1109/HPEC55821.2022.9926300

[69] Mingxing Tan and Quoc V. Le. 2020. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. arXiv:1905.11946 [cs.LG]

[70] Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. 2023. CUTLASS.

[71] The Linux Foundation. 2023. Reproducibility. https://pytorch.org/docs/stable/notes/randomness.html.

[72] Philippe Tillet and David Cox. 2017. Input-Aware Auto-Tuning of Compute-Bound HPC Kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '17)*. ACM, New York, NY, USA, Article 43, 12 pages. https://doi.org/mdr3

[73] Gaurav Verma, Yashi Gupta, Abid M. Malik, and Barbara Chapman. 2021. Performance Evaluation of Deep Learning Compilers for Edge Inference. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, Portland, OR, USA, 858–865. https://doi.org/mdr2

[74] Shibo Wang and Pankaj Kanwar. 2019. BFloat16: The secret to high performance on Cloud TPUs. *Google Cloud Blog* 4 (2019), 1 pages.

[75] Yuke Wang, Boyuan Feng, and Yufei Ding. 2022. 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* (Seoul, Republic of Korea) *(PPoPP '22)*. ACM, New York, NY, USA, 107–119. https://doi.org/10.1145/3503221.3508408

[76] Yuxin Wang, Qiang Wang, Shaohuai Shi, Xin He, Zhenheng Tang, Kaiyong Zhao, and Xiaowen Chu. 2020. Benchmarking the Performance and Energy Efficiency of AI Accelerators for AI Training. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, Melbourne, Australia, 744–751. https://doi.org/10.1109/CCGrid49817.2020.00-15

[77] Zhihao Wen, Yuan Fang, and Zemin Liu. 2021. Meta-Inductive Node Classification across Graphs. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval* (Virtual Event, Canada) *(SIGIR '21)*. ACM, New York, NY, USA, 1219–1228. https://doi.org/mdr4

[78] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (apr 2009), 65–76. https://doi.org/10.1145/1498765.1498785

[79] Pavan Yalamanchili, Umar Arshad, Zakiuddin Mohammed, Pradeep Garigipati, Peter Entschev, Brian Kloppenborg, James Malcolm, and John Melonakos. 2015. ArrayFire - A high performance software library for parallel computing with an easy-to-use API. https://github.com/arrayfire/arrayfire

[80] Charlene Yang. 2015. Berkeley CS Roofline Toolkit. https://bitbucket.org/berkeleylab/cs-roofline-toolkit.

[81] Zhiwei Yang, Lu Lu, and Ruimin Wang. 2022. A batched GEMM optimization framework for deep learning. *The Journal of Supercomputing* 78, 11 (March 2022), 13393–13408. https://doi.org/10.1007/s11227-022-04336-3

[82] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2020. GraphSAINT: Graph Sampling Based Inductive Learning Method. arXiv:1907.04931 [cs.LG]

[83] Si Zhang, Hanghang Tong, Jiejun Xu, and Ross Maciejewski. 2018. Graph Convolutional Networks: Algorithms, Applications and Open Challenges. In *Computational Data and Social Networks*, Xuemin Chen, Arunabha Sen, Wei Wayne Li, and My T. Thai (Eds.). Springer International Publishing, Cham, 79–91.

[84] Si Zhang, Hanghang Tong, Jiejun Xu, and Ross Maciejewski. 2019. Graph convolutional networks: a comprehensive review. *Computational Social Networks* 6, 1 (Nov. 2019), 23 pages. https://doi.org/10.1186/s40649-019-0069-y

[85] Zhihui Zhang, Jingwen Leng, Lingxiao Ma, Youshan Miao, Chao Li, and Minyi Guo. 2020. Architectural Implications of Graph Neural Networks. *IEEE Computer Architecture Letters* 19, 1 (2020), 59–62. https://doi.org/10.1109/LCA.2020.2988991

[86] Yangjie Zhou, Mengtian Yang, Cong Guo, Jingwen Leng, Yun Liang, Quan Chen, Minyi Guo, and Yuhao Zhu. 2021. Characterizing and Demystifying the Implicit Convolution Algorithm on Commercial Matrix-Multiplication Accelerators. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Storrs, CT, USA, 214–225. https://doi.org/10.1109/IISWC53511.2021.00029

[87] Jinming Zhuang, Jason Lau, Hanchen Ye, Zhuoping Yang, Yubo Du, Jack Lo, Kristof Denolf, Stephen Neuendorffer, Alex Jones, Jingtong Hu, Deming Chen, Jason Cong, and Peipei Zhou. 2023. CHARM: Composing Heterogeneous AcceleRators for Matrix Multiply on Versal ACAP Architecture. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '23)*. ACM, New York, NY, USA, 153–164. https://doi.org/10.1145/3543622.3573210