M^{3}XU: Achieving High-Precision and Complex Matrix Multiplication with Low-Precision MXUs

Dongho Ha[1,2]
Yonsei University
Seoul, South Korea
dongho.ha@yonsei.ac.kr

Yunan Zhang[1,3]
University of California, Riverside
Riverside, USA
yzhan828@ucr.edu

Chen-Chien Kao
University of Michigan
Michigan, USA
cchkao@umich.edu

Christopher J. Hughes
Intel Corporation
Santa Clara, USA
christopher.j.hughes@intel.com

Won Woo Ro
Yonsei University
Seoul, South Korea
wro@yonsei.ac.kr

Hung-Wei Tseng
University of California, Riverside
Riverside, USA
htseng@ucr.edu

***Abstract*—Beyond the high-profile artificial intelligence and machine learning (AI/ML) workloads, the demand for high-performance matrix operations on standard and complex floating-point numbers remains strong but underserved. However, the widely adopted low-precision matrix processing units (MXUs) can only fulfill the need for AI/ML workloads, which are underutilized or idle when running applications outside their target domains.**

This paper presents M^{3}XU, multi-mode matrix processing units that support IEEE 754 single-precision and complex 32-bit floating-point numbers. M^{3}XU does not rely on more precise but costly multipliers. Instead, M^{3}XU proposes a multi-step approach that extends existing MXUs for AI/ML workloads. The resulting M^{3}XU can seamlessly upgrade existing systems without programmers' efforts and maintain the bandwidth demand of existing memory subsystems. This paper evaluates M^{3}XU with full-system emulation and hardware synthesis. M^{3}XU can achieve a 3.64× speedup for 32-bit matrix multiplications and 3.51× speedup for complex number operations on average compared with conventional vector processing units.

I. Introduction

Matrix multiplication units (MXUs) or matrix processing units have become ubiquitous in all computing scenarios due to the criticality of matrix operations in artificial intelligence and machine learning (AI/ML) workloads. MXUs can serve as the core in standalone AI/ML accelerators [25], [36]–[38], present as another compute engine in modern GPU architectures [1], [58], [59], or integrate into CPUs as extensions to existing instruction set architectures (e.g., Intel AMX [35], ARM's SME [5], and Apple's Matrix Extensions [3]). The evolution of MXUs has continuously lifted the roofline of core neural networks (NNs) operations to the memory bandwidth and provided a more scalable processing model through the embarrassingly parallel matrix operations for huge problem sizes [38]. However, as modern adoption of MXUs targets AI/ML applications, most existing MXUs only support low-precision matrix operations (e.g., 16-bit half-precision, INT8) or introduce formats (e.g., BF16, TF16, TF32) for better performance, energy, and area efficiency.

Beyond accelerating workloads dominated by low-precision matrix operations, MXUs can help a broader set of compute-intensive workloads to scale with the advances of modern AI/ML hardware and parallelize through matrix processing models if they support the following two formats.

Single-precision floating point numbers (FP32) Scientific applications [6], [7], [7], [16], [29], [55], data analytics/mining applications [20], [23], statistical learning [44], and graph analytics [15], [72] are sensitive to numerical errors and most existing implementations must rely on IEEE 754 standard single-precision floating-point-numbers (FP32) to function correctly. Many Domain-Specific Accelerators require FP32 inputs [26], [28], [87], and using other formats can lead to unwanted results. Despite the error tolerance in inferencing, training NN models still rely on intensive FP32 operations [58], or require significant re-engineering to accommodate other data types [54].

Single-precision complex floating point numbers (FP32C) Fast Fourier Transforms (FFTs) that rely on matrix multiplications with complex numbers are the core of signal processing [9], [10], [17], [47], [73] and security applications [49], [66]. Also, simulating quantum computing needs complex matrix multiplications to represent qubits and their operations [8], [48], [70], [78], [89], [91]. As multimedia signals become complex numbers after transformations, recent studies also show neural networks using complex number matrix multiplications are advantageous [4], [14], [31], [42], [43], [56], [76], [77], [83].

However, extending MXUs to support higher precision floating point or complex numbers is expensive. The cost of FMA logic is roughly quadratic in the input bitwidth. For example, going from 16-bit to 32-bit floating-point inputs

[1]Both authors contributed equally to this work.

[2]The author is working at MangoBoost (dongho.ha@mangoboost.io) now. This paper is done when the author was a visiting scholar at University of California, Riverside

[3]The author is working at Google now.

SC24, November 17-22, 2024, Atlanta, Georgia, USA

and maintaining the number of operations per cycle roughly quadruples the hardware area. Furthermore, even if we are willing to pay the quadratic hardware cost in MXUs, the doubled data width also requires doubling the memory subsystem bandwidth to match the consumption rate.

By revisiting the mathematical operations of matrix multiplications with higher-precision and complex numbers, we can decompose each computation step as a series of low-precision matrix multiplications between different components of the input matrices. Also, considering the limitations on feeding the MXU with data from memory, we can hit the roofline of the existing memory hierarchy if we use multiple low-precision steps to perform both high-precision and complex matrix multiplications. In other words, the matrix hardware can reuse existing components to perform wider and/or complex multiplications at reasonable performance if we enable operations on different matrix components on the MXU.

Inspired by the insights from mathematical observations, this paper presents M^{3}XU, a multi-mode MXU that extends half-precision MXUs to support matrix operations using FP32 and FP32C inputs, in addition to low-precision floating point numbers at low hardware costs. M^{3}XU simply requires (1) additions of logic to feed different parts of matrix inputs in each step of operations, (2) minor extensions to the arithmetic units to support exact FP32 precisions, and (3) slight extensions to accumulators to accumulate numbers in correct double-precision formats. Moreover, M^{3}XU does not double the bitwidth of arithmetic units, avoiding the considerable area overhead or the increase in memory bandwidth. M^{3}XU still delivers FP32 and FP32C matrix multiplications at the theoretical throughput that the current memory bandwidth can support. The same M^{3}XU remains the support of the original functions. As M^{3}XU supports standard FP32 and FP32C, M^{3}XU does not require any modification to existing programs.

Compared to software alternatives that perform FP32 and FP32C operations with multiple low precision ones, M^{3}XU reduces dynamic instructions, allowing M^{3}XU to execute equivalent computation more efficiently and maximize reuse of register contents. More importantly, as M^{3}XU faithfully supports FP32 operations, M^{3}XU requires zero changes in software to accommodate the loss of precision in existing software solutions [18], [50], [53], [62], [63]. As M^{3}XU enables native FP32C computations, M^{3}XU delivers better performance and more accurate results than software approximations [17], [47], [73].

Our experimental results show an average 3.89× speedup compared to conventional implementations on FP32 precision optimized for CUDA/SIMT(Single instruction, multiple threads) cores. As M^{3}XU brings hardware support for complex numbers, M^{3}XU can directly perform FFT calculations without approximations and achieves up to 1.99× speedup compared with state-of-the-art cuFFT libraries. The synthesized M^{3}XU hardware incurs 47% area-overhead, significantly smaller than the 3.55× overhead from extending arithmetic logic. If we make M^{3}XU an extension to NVIDIA's Ampere architecture, the resulting overhead is 4% of the streaming multiprocessors (SMs).

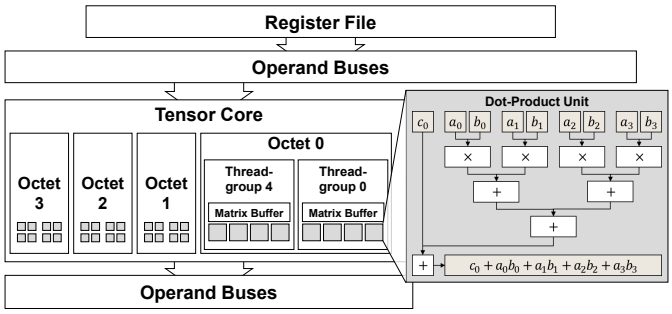

Fig. 1: The baseline Tensor Core architecture

This paper makes the following contributions.
(1) It presents M^{3}XU, the first MXU supporting complex number operations, to the best of our knowledge.
(2) Unlike traditional multi-precision MXUs use higher-precision ALUs to downward support lower-precisions, M^{3}XU is the first study that uses a multi-cycle design to extend the architectural supports in lower-precision MXUs for higher-precisions, to the best of our knowledge.
(3) It justifies and quantifies that M^{3}XU is the most efficient, least expensive MXU design to support higher precision for memory-bandwidth limited systems.
(4) It evaluates the performance benefit and area-overhead of the proposed M^{3}XU on critical matrix kernels/algorithms that a wide range of applications uses.

II. Background and Motivation

This section describes the exemplary MXU architecture that M^{3}XU extends, as well as the challenges of supporting higher precision or complex numbers in MXUs.

A. Tensor Core Architecture

Among commercial matrix accelerators, this paper selected NVIDIA's Tensor Cores as the baseline accelerator as (1) the hardware of Tensor Cores is commercially available to the public, and (2) the low-level programming interface is available for this paper to assess the performance of our proposed extensions. However, the extension that M^{3}XU proposes can apply to any MXU architecture, regardless of whether the underlying implementation is dot-product-unit-based, outer-product-unit-based, or a systolic array.

In NVIDIA's GPU architectures, Tensor Cores are part of the streaming multiprocessors (SMs). They share the register file, schedulers, and caches with other SM components. The only type of operation that a Tensor Core supports is matrix multiplications. Though NVIDIA does not reveal Tensor Cores' microarchitecture, the model that GPGPU-sim uses seamlessly resembles the measured performance characteristics [41], [67], [82]; Figure 1 depicts this. Each Tensor Core consists of multiple four-element dot-product units that can perform all necessary multiplications and accumulations for MMA operations per cycle. According to NVIDIA's datasheet, each Tensor Core unit supports 16-bit floating-point MMA

Data Type	Bit Format*	Peak Throughput
FP32	(1,8,23)	19.5 TFLOPS
FP16	(1,5,10)	78 TFLOPS
BF16	(1,8,7)	39 TFLOPS
TF32 Tensor Core	(1,8,10)	156 TFLOPS
FP16 Tensor Core	(1,5,10)	312 TFLOPS
BF16 Tensor Core	(1,8,7)	312 TFLOPS

* Each bit format of floating-point data type means (the number of sign bits, exponenet bits, mantissa bits)

TABLE I: A100 HMMA peak throughput

operations in $8\times4\times8$ (i.e., multiplying an 8×8 matrix by a 8×4 matrix, resulting in an 8×4 matrix) by default.

Table I excerpts the peak throughput of NVIDIA A100's Tensor Cores on various data types from the datasheet [58]. Based on the datasheet, NVIDIA's programming interface, and reverse engineering from prior work [74], [88], the hardware architecture of Tensor Cores can provide native support of MMA operations using FP16, BF16, and TF32 inputs. By observing the union of these three formats, a reasonable design of a dot-product unit uses a one-bit sign, eight-bit exponent, and 11-bit mantissa (including an implicit bit). Current Tensor Cores provide no hardware support for true FP32 arithmetic or complex numbers. NVIDIA's Tensor Cores support TF32, seamlessly allowing the software to provide FP32 inputs and deliver results at half the BF16/FP16 FLOPS. However, TF32 has 13-bit fewer mantissa bits than FP32; programmers must handle the information loss for usages needing more precision. To get "real" FP32 operations (or FP32C), we must rely on (1) the SIMD hardware, which has 8x less throughput than TF32 Tensor Cores, or (2) software modifications using multiple MMA operations at a lower precision.

B. Challenges of Extending MXUs

Despite the demand for FP32 and FP32C and the shortfalls in using alternative data types, extending MXUs to support either FP32 or FP32C has yet to be done because it is expensive and challenging.

Area overhead FP32 and FP32C use a 23-bit mantissa, so we must double the bitwidth of multipliers and accumulators. Expanding multipliers is especially costly as the area is quadratic to the input bandwidth. We synthesized the area overhead of an FP32-MXU (with no FP32C support) with as many FP32 FLOPS as FP16/BF16 FLOPS using the same process technology and tool that Section V-A will describe later. The FP32-MXU is 3.55× larger than a baseline MXU without FP32, increasing the SM area by 11%.

Memory pressure Suppose an MXU, with p-bit inputs, can multiply an $M\times K$ matrix with an $K\times N$ (we abbreviate these dimensions as $M\times N\times K$ in the rest of the paper) each cycle. Such an MXU will consume $M\times K+K\times N$ p-bit elements, or $(M\times K+K\times N)\times\frac{p}{8}$ bytes per cycle and generate $M\times N$ p-bit elements, at full utilization. If the SM runs at frequency F and contains X MXUs, the total memory bandwidth B to keep the MXUs fed is: $B=(M\times K+K\times N+M\times N)\times\frac{p}{8}\times F\times X$.

In an A100 GPU with 432 Tensor Cores running at 1.41 GHz, at 16-bit precision, B is 156 TB/sec. A100 already uses a 128B-blocked cache and 1024-bit wide interface to feed the Tensor Cores. If we double the bitwidth of MXUs

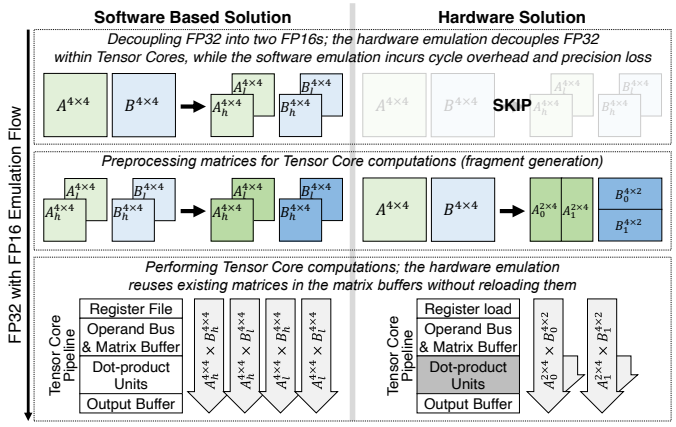

Fig. 2: Comparison between software-based and hardware-based solutions

and maintain the same clock rate, the required bandwidth will become 312 TB/sec. However, building a memory hierarchy supporting the required bandwidth is very expensive: we will need to double the bitwidth of the front-end bus between the cache and Tensor Cores, as well as the bandwidth of the caches and DRAM. As the white paper of H100 documents, the latest high bandwidth memory (HBM) technologies can only deliver 3.35 TB/sec. Modern Tensor Core library implementations have already applied intensive optimizations to extract the reuse of matrix tiles to mitigate the memory gap. Recent studies have shown that even the most optimized cuBLAS still cannot reach the peak throughput with the default 16-bit number format [64], [68].

Trade-offs between memory-MXUs As doubling the bitwidth of MXUs and the memory interface is expensive, we could maintain the same memory bandwidth. However, in this case, the extended MXUs can only deliver 50% of their peak performance. An alternative is to halve the number of MXUs. However, as each FP32-MXU is 3.55× larger, halving the number of MXUs still incurs 1.78× area overhead, increasing the SM area by 6%. This also would halve the low-precision compute throughput and still not provide hardware support for complex numbers in the MXUs.

C. Alternatives

Prior software-based alternatives have tried supporting the demand for matrix multiplications on FP32 [18], [50], [53], [62], [63] and FP32C [17] numbers, but all have limitations. Some MXU architectures also try to accommodate lower-precision matrix multiplications with more-precise hardware [1], [57], [58]. However, no project like M^{3}XU can perform complex number matrix multiplications in hardware or even try to combine complex number and conventional floating point matrix multiplications in a single hardware unit, to the best of our knowledge.

1) Software-based Alternatives: Despite the advantage of zero additional hardware costs, existing software alternatives [18], [50], [53], [53], [62], [63] have limitations on performance in two major aspects. First, software alternatives must explicitly control the data accesses, incurring additional

matrix loads, register accesses, and dynamic instructions on tile matrix operations. Second, software alternatives unavoidably have to decouple values and compensate for potential precision losses.

Figure 2 compares existing software-based FP32 GEMM solutions on FP16 MXUs and on FP32 MXUs. The same philosophy applies to software-based FP32C implementations. Without hardware support, the software solution needs additional instructions to compute, shift, and split the exponent, mantissa parts, and flipping sign bits before feeding data into MXUs. In contrast, appropriate hardware support can implicitly handle the bit assignments, shifts, and splits without incurring instructions.

After decoupling data, software solutions must explicitly control the loads and stores for each tiled matrix operation as separate instruction streams with no guarantees in scheduling but increasing the total number of dynamic instructions. In contrast, hardware solutions can perform the same computation within a single stream, with fewer loads/stores and fewer instructions.

2) Hardware Solutions: Existing multi-precision hardware MXUs support FP32 MMA operations by providing logic that natively supports the highest precisions in the design [1]. Such design philosophy allows the hardware to offer downward-support of lower-precision arithmetics without suffering precision loss. However, the area cost and energy consumption are higher than native supports of lower precision arithmetics.

Similar to the philosophy of M^{3}XU, recent MXUs that originally targeted AI/ML applications have supported data types with higher precisions [57], [58]. However, all existing MXUs slightly extend the exponent or the mantissa fields but implicitly discard bits incompatible with internal low-precision MXUs to create an illusion of higher-precision supports. Despite the performance, area, and energy advantages, this line of MXUs will lead to unprecedented numerical errors and floating-point exceptions that are unacceptable to existing FP32 applications and require significant software rewriting and debugging efforts.

III. Opportunities For M^{3}XU

Through mathematical analysis of general matrix multiplications (GEMMs), we can identify the minimum requirement to extend a lower-precision MXU to support higher-precision operations at the peak throughput without increasing the memory bandwidth. This section describes the insights that inspired the design of M^{3}XU.

A. Higher precision GEMM with lower precision MXUs

Assume that we have three input matrices, A, B, and C, where A is an $M \times K$ matrix, B is an $K \times N$ matrix, and C is an $M \times N$ matrix. Equation 1 shows the calculation of the most frequently used matrix function – general matrix multiplication (GEMM), $D = A \cdot B + C$, with a scaling factor as 1.

$$\begin{aligned} &\forall a_{i,j} \in A, b_{i,j} \in B, c_{i,j} \in C, d_{i,j} \in D, \\ &d_{i,j} = \sum_{k=0}^{K-1} a_{i,k} b_{k,j} + c_{i,j} \end{aligned} \tag{1}$$

If we expand Equation 1 by separating the summation between the cases where k is odd or even, we get Equation 2.

$$d_{i,j} = \sum_{k=0}^{\frac{K}{2}-1} a_{i,2\times k} b_{2\times k,j} + \sum_{k=0}^{\frac{K}{2}-1} a_{i,2\times k+1} b_{2\times k+1,j} + c_{i,j} \tag{2}$$

Now, consider the case where we have three input matrices, A', B', and C', where A' is an $M \times \frac{K}{2}$ matrix and B' is a $\frac{K}{2} \times N$ matrix and C' is an $M \times N$ matrix. In other words, we halve the K of A and B. In addition, each number in A', B', and C' is at $2p$-bit precision, where p is an arbitrary constant value. Then, we split A' into two matrices, A'_H and A'_L, where they store the upper and lower p bits, respectively, of each number in A'. Therefore, $A' = A'_H \cdot 2^p + A'_L$. Similarly, we split B' as $B' = B'_H \cdot 2^p + B'_L$. Equation 3 summarizes the GEMM calculation of $D' = A' \cdot B' + C'$.

$$\begin{aligned} D' &= A' \cdot B' + C' \\ &= (A'_H \cdot 2^p + A'_L) \cdot (B'_H \cdot 2^p + B'_L) + C' \\ &= A'_H \cdot B'_H \cdot 2^{2p} + (A'_H \cdot B'_L + A'_L \cdot B'_H) \cdot 2^p + A'_L \cdot B'_L + C' \end{aligned} \tag{3}$$

Again, let us create a $M \times K$ matrix A" and $K \times N$ matrix B" using the following equation.

$$\forall a''_{i,j} \in A'', a'_{H\,i,j} \in A'_H, a'_{L\,i,j} \in A'_L, \begin{cases} a''_{i,2\times j} & = a'_{H\,i,j} \\ a''_{i,2\times j+1} & = a'_{L\,i,j} \end{cases} \tag{4}$$

$$\forall b''_{i,j} \in B'', b'_{H\,i,j} \in B'_H, b'_{L\,i,j} \in B'_L, \begin{cases} b''_{2\times i,j} & = b'_{H\,i,j} \\ b''_{2\times i+1,j} & = b'_{L\,i,j} \end{cases} \tag{5}$$

If we perform matrix multiplication as $D'_H = A'' \cdot B''$ and apply a similar decomposition as in Equation 2, we can derive Equation 6 as below.

$$\begin{aligned} &\forall a''_{i,j} \in A'', b''_{i,j} \in B'', d'_{H\,i,j} \in D'_H, \\ d'_{H\,i,j} &= \sum_{k=0}^{\frac{K}{2}-1} a''_{i,k} b''_{k,j} \\ &= \sum_{k=0}^{\frac{K}{4}-1} a''_{i,2\times k} b''_{2\times k,j} + \sum_{k=0}^{\frac{K}{4}-1} a''_{i,2\times k+1} b''_{2\times k+1,j} \\ &= A'_H \cdot B'_H + A'_L \cdot B'_L \end{aligned} \tag{6}$$

Equation 6 shows that $A'' \cdot B''$ covers the multiplication results for $A'_H \cdot B'_H$ and $A'_L \cdot B'_L$. If we flip the order of assignment in matrix B" and create another $K \times N$ matrix B"' using the following equation,

$$\forall b'''_{i,j} \in B''', b'_{H\,i,j} \in B'_H, b'_{L\,i,j} \in B'_L, \begin{cases} b'''_{2\times i,j} = b'_{L\,i,j} \\ b'''_{2\times i+1,j} = b'_{H\,i,j} \end{cases} \tag{7}$$

and perform $D'_L = A'' \cdot B'''$, we can derive Equation 8 as:

$$\begin{aligned} &\forall a''_{i,j} \in A'', b'''_{i,j} \in B''', d'_{L\,i,j} \in D'_L, \\ &d'_{L\,i,j} = \sum_{k=0}^{\frac{K}{2}-1} a''_{i,k} b'''_{k,j} \\ &\quad = \sum_{k=0}^{\frac{K}{4}-1} a''_{i,2\times k} b'''_{2\times k,j} + \sum_{k=0}^{\frac{K}{4}-1} a''_{i,2\times k+1} b'''_{2\times k+1,j} \\ &\quad = A'_H \cdot B'_L + A'_L \cdot B'_H \end{aligned} \tag{8}$$

Equation 8 shows that $A'' \cdot B''$ covers the multiplication results for $A'_H \cdot B'_L$ and $A'_L \cdot B'_H$. We can conclude the first observation by summarizing the result in Equation 6 and Equation 8:

Observation 1: An MXU that can perform a $M \times N \times K$ matrix multiplications (or in general, any Matrix Semiring operation) at p-bit precision can perform all multiplications that are necessary in a $M \times N \times \frac{K}{2}$ matrix multiplication at $2p$-bit precision in two steps if the hardware can reassign inputs in these two different steps.

However, directly summing up the result $A'_H \cdot B'_H$ and $A'_L \cdot B'_L$ is not useful to the final result as we need $A'_H \cdot B'_H \times 2^{2p} + A'_L \cdot B'_L$. Therefore, our second observation is:

Observation 2: We need to extend the p-bit MXU to shift the accumulation result of $A'_H \cdot B'_H$ by $2p$ bits and shift D'_L by p bits and accumulate these multiplication results to support $2p$-bit matrix multiplications. These two observations also lead to the following corollaries:

Corollary 1: By reusing existing multipliers, extending accumulators, and adding shifters, an MXU capable of a p-bit $M \times N \times K$ matrix multiplication every c cycles can support a $2p$-bit $M \times N \times \frac{K}{2}$ matrix multiplication every $2c$ cycles.

Corollary 2: The extended MXU of Corollary 1 can support $2p$-bit $M \times N \times K$ matrix multiplications at $\frac{1}{4}$ of the peak TOPS (tensor operations per second) of p-bit $M \times N \times K$ matrix multiplications.

B. Complex number GEMM with existing MXUs

We can also perform a similar analysis on complex number matrix multiplications (CGEMM). Assume that we have a set of three input matrices in complex numbers, A'_C, B'_C, and C'_C, where A' is an $m \times \frac{k}{2}$ matrix and B'_C is an $\frac{k}{2} \times n$ matrix and C'_C is an $m \times n$ matrix. Then, we split each number in A'_C to create two matrices, A'_{CR} and A'_{CI}, where A'_{CR} contains the real part of each number in A'_C and A'_{CI} contains the imaginary part of each number. That is, $A'_C = A'_{CR} + A'_{CI}i$. Similarly, we also split B'_C as $B'_C = B'_{CR} + B'_{CI}i$.

$$\begin{aligned} D'_C &= A'_C \cdot B'_C + C'_C \\ &= (A'_{CR} + A'_{CI}i) \cdot (B'_{CR} + B'_{CI}i) + C'_C \\ &= (A'_{CR} \cdot B'_{CR} - A'_{CI} \cdot B'_{CI}) + \\ &\quad (A'_{CR} \cdot B'_{CI} + A'_{CI} \cdot B'_{CR})i + C'_C \end{aligned} \tag{9}$$

Equation 9 expands $D'_C = A'_C \cdot B'_C + C'_C$ with the split $A_C{}'$ and $B_C{}'$. This is almost identical to Equation 3, except for the subtraction. If we repeat the processing as Equation 4 – Equation 8, and treat A'_{CR} and B'_{CR} as A'_H and B'_H and A'_{CI} and B'_{CI} as A'_L and B'_L, then we again see that the existing MXU can perform all necessary multiplications, but needs to additionally support the subtraction of the product of A'_{CI} and B'_{CI}. With Equation 9, we can conclude the third observation:

Observation 3: A p-bit MXU can support p-bit CGEMM in two steps if it has hardware support to subtract the products of imaginary parts. If we want to support CGEMM with $2p$ bits in each number's real and imaginary parts, we can combine the insights from Observation 1 ($2p$-bit $M \times N \times K$ matrix multiplications takes $4c$ cycles) and Observation 3 ($M \times N \times K$ CGEMM takes $4c$ cycles) and derive the following.

Corollary 3: By reusing existing multipliers and adding shifting and subtraction logic, an MXU capable of a p-bit $M \times N \times K$ matrix multiplication every c cycles can support a $2p$-bit $M \times N \times K$ CGEMM every $16c$ cycles.

C. Performance Expectation on Modern Hardware

This section estimates the performance gain on modern hardware using the observations and corollaries from Sections III-A and III-B to derive this work's advantage. Referencing the white papers from NVIDIA's Tensor Core Architectures (our baseline hardware architecture) [58], [59], the peak FP16 FLOPS on Tensor Cores on existing GPUs are 15×-16× higher than that of the FP32 CUDA/SIMT cores. Therefore, the theoretical throughput of our proposed work, M^{3}XU, still has a 4× performance advantage over FP32 CUDA cores, equivalent to 78 TFLOPS on the Ampere architecture or 248 TFLOPS on the Hopper architecture. For FP32C CGEMM, M^{3}XU maintains a 4× peak performance advantage over using conventional CUDA cores. If we extend AMD's Matrix Cores as the baseline, M^{3}XU still has a performance advantage. The total TOPS of Matrix Cores on AMD's MI100 and MI250 are 8× of the SIMT cores, meaning M^{3}XU would have a 2× advantage over SIMT cores on those GPUs.

IV. M^{3}XU Microarchitecture

This paper leverages the insights from Section III to build M^{3}XU via a small extension to an MXU that originally targets low-precision operations, and which is enhanced to support true FP32 and FP32C computations. This section describes the hardware architecture in detail.

A. Extending MXUs for FP32

Summarizing Observations 1 and 2 in Section III, supporting FP32 in a 16-bit MXU using two steps requires the following extensions. (1) The hardware needs the ability to change the dataflow of the inputs in each step. (2) The bit width of each input to the multiplier must be at least half of the width of the mantissa. In the case of FP32, the bit width must be at least 12 (i.e., $p >= 12$). (3) The exponent adder must be as wide as that of the high-precision type (8 bits for FP32). (4) Some accumulators can selectively shift numbers

by $2p$ and p bits. M^{3}XU fulfills these requirements by adding a data-assignment stage and extending the arithmetic logic units.

The data-assignment stage M^{3}XU controls the dataflow of each step of an operation via multiplexers and buffers that store the inputs of each step. Figure 3 depicts the high-level design of this *data-assignment stage*. Since the arithmetic logic must support half of the width of the mantissa in FP32 and the full exponent bits of FP32, each buffer entry contains space for the 1-bit sign, 8-bit exponent, and 12 bits of mantissa. For each dot-product unit that performs s steps of operations for two m-element input vectors, we need $2 \times m \times s$ buffer entries. In the default FP16 mode, the data-assignment stage directly feeds each input value into the pairs of input buffers. As FP16 contains a hidden, leading 1 in the mantissa field, the circuit will fill the hidden 1 in the input buffer and unused bits in the buffer entry with 0s.

M^{3}XU has native FP32 support without introducing a new data layout. Therefore, as inputs come from registers, the data-assignment stage splits each 32-bit chunk of data (i.e., a single FP32 number containing one sign bit, eight exponent bits, and 23 mantissa bits) into two low-precision numbers and assigns them to the corresponding input buffers for the multipliers in each step. In other words, the data-assignment stage divides each FP32 number (e.g., $a'_{i,j}$) into $a_{H}{}'_{i,j}$ and $a_{L}{}'_{i,j}$. As in Figure 3(a), the data-assignment stage wires the 1-bit sign and the 8-bit exponent to *both* the buffer entries representing $a_{H}{}'_{i,j}$ and $a_{L}{}'_{i,j}$. The exponent is thus artificially small for $a_{L}{}'_{i,j}$, which is why the hardware must later correct for this, post-multiplication. The data-assignment stage attaches the hidden 1 to the buffer representing $a_{H}{}'_{i,j}$ and wires the most significant 4 bits from the second half of the original FP32 number. The 12-bit mantissa field in the $a_{L}{}'_{i,j}$ completely comes from the least significant 12 bits of the second half of the original FP32 number. The same process applies to both FP32 input vectors. In the first step, each pair of buffer entries to the same multiplier will either work on the most or least significant parts of both input numbers. Then, in the second step, the data-assignment stage signals the multiplexers to flip the assignment of one of the input vectors (e.g., $b_{H}{}'_{i,j}$ and $b_{L}{}'_{i,j}$ in Figure 3(a)). This allows the multipliers to compute the products of the most significant parts of one vector and the least significant parts of another vector.

The extension to arithmetic logics As Equation 3 points out, the M^{3}XU's arithmetic logic must (1) accommodate 12 bits of mantissa computation and (2) accumulate the partial sum-of-products correctly for the case of supporting FP32. Figure 3(b) depicts the extensions M^{3}XU makes to the baseline MXU, the Tensor Core architecture of Ampere, for this.

Since existing Tensor Cores only support an 11-bit mantissa, we need to expand the arithmetic logic to support 12 bits. This 1-bit extension is much cheaper than a brute force extension to 24 bits for FP32. Modern Tensor Cores already provide native support for 8-bit exponents, so M^{3}XU does not need to extend the exponent-related logic. In addition, we need to add multiplexers next to the outputs of the multipliers that calculate $A'_H \times B'_H$ and shift the result by 24 bits, or else separately

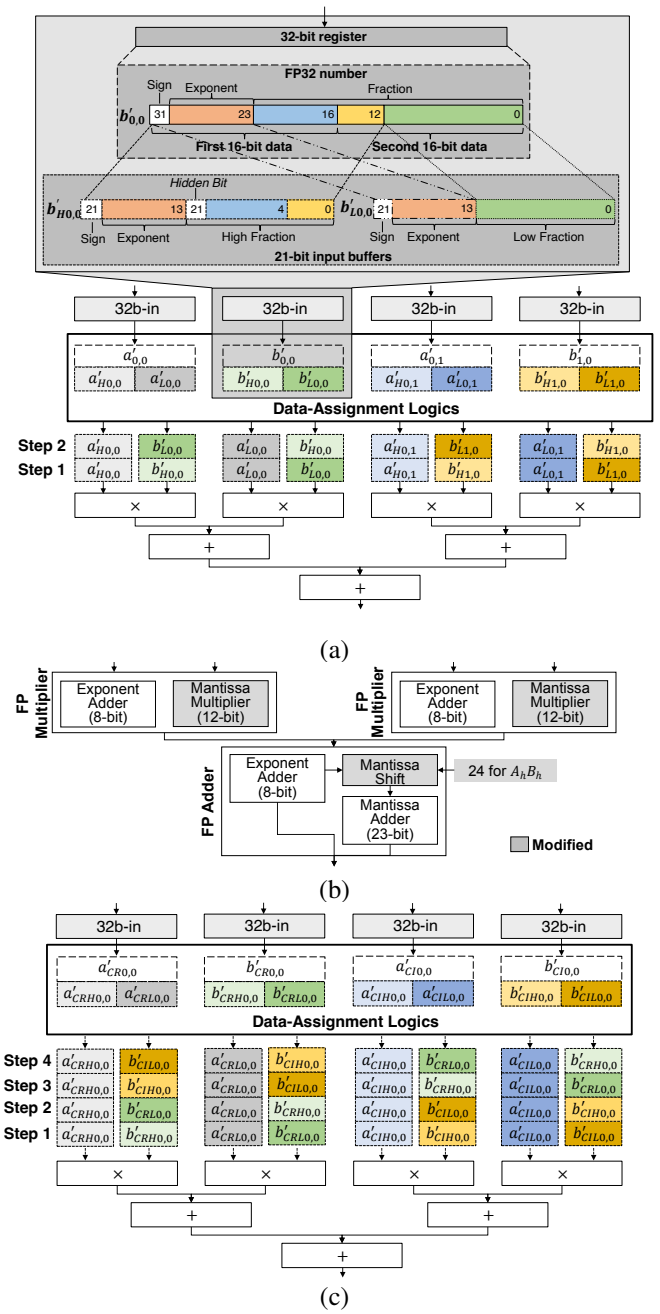

Fig. 3: The high-level design of the data-assignment stage. (a) Data-assignment stage for FP32 (b) Dot-product unit hardware modifications for FP32 (c) Data-assignment stage for FP32C

accumulate the outputs of $A'_H \times B'_H$ and $A'_L \times B'_L$ and shift the "high" result once by 24 bits. We also need 48-bit registers for the accumulation results. In Figure 3(b), we draw the former for clarity, but we implement the latter for efficiency. For the second step of the computation, all circuits remain, except that we do not shift the results of any multipliers but instead extend the multiplexers to shift the accumulation result by 16 bits, and also accumulate the result in this stage with the previous stage. Based on an $8 \times 4 \times 8$ MXU of a Tensor Core, the resulting M^{3}XU can perform $8 \times 4 \times 4$ in FP32 in each 2-step operation.

B. Extending MXUs for FP32C

Using Observation 3 in Section III and combining the earlier-described FP32 extensions for M^{3}XU, M^{3}XU can additionally support complex number arithmetic and act as an accelerator for FP32C. In addition to the modifications in Section IV-A, supporting FP32C requires (1) subtractions in parts of the sum-of-products and (2) 4-step operations where two of the steps will generate the real part and the rest generate the imaginary part. Figure 3(c) depicts the extension to FP32-M^{3}XU for FP32C.

Equation 9 indicates that M^{3}XU can perform complex number arithmetic in two steps. However, as each part of a complex number in FP32C is a FP32 number, M^{3}XU has to consider the real part, $A'_{CR} \cdot B'_{CR} - A'_{CI} \cdot B'_{CI}$, and the imaginary part, $A'_{CR} \cdot B'_{CI} + A'_{CL} \cdot B'_{CH}$, as two separate FP32 matrix multiplications. Since each FP32 multiplication takes two steps, the data-assignment stage needs to prepare four levels of inputs and store them in buffers twice the size of the ones in FP32-M^{3}XU.

Figure 3(c) illustrates the data-assignment stage in FP32C mode. M^{3}XU assumes the conventional interleaved representation of complex numbers where a pair of consecutive elements store a complex number's real and imaginary parts. Therefore, an 8×4 FP32 matrix will contain 4×4 FP32C numbers. The resulting M^{3}XU can perform an FP32C matrix multiplication of size $8 \times 4 \times 2$ in a single 4-step operation.

M^{3}XU first computes the real parts of the output, then the imaginary parts. Like in FP32 mode, M^{3}XU splits each FP32 element into two numbers, high-order and low-order parts. For the inputs in the first step, the data-assignment logic assigns either a pair of high-order parts or low-order parts together and also assigns a pair of real parts or imaginary parts together. In the case that the multiplication corresponds to two imaginary parts of numbers, the data-assignment logic flips the sign-bit for the first input such that the result will be "subtracted" when accumulated. In the case that the multiplication corresponds to two high-order parts, the output will be shifted 24 bits. For the second step, M^{3}XU swaps the high-order and low-order parts of the b input from two adjacent multipliers to complete all necessary multiplications for the real part of FP32C. The computation in this stage will again reuse the FP32 logic to shift the results by 16 bits and accumulate with the first step. For the third and fourth steps, M^{3}XU computes the imaginary parts by interleaving the real part of one number and the imaginary part from the other. However, for this set of inputs, M^{3}XU reverses the flip signed bit back as M^{3}XU does not need to perform subtraction in the corresponding stage. The data assignment logic swaps the imaginary and real parts of the b input across four adjacent multipliers (as shown), and shifts the outputs by 16 bits during accumulation. The last step swaps high-order and low-order parts of the b input from two adjacent multipliers. The 3rd and 4th steps are interchangeable. Our implementation eliminates the overhead of reassigning inputs.

C. Extending MXUs for higher bitwidth floating point number

The M^{3}XU approach, which leverages existing low-bit-width arithmetic units for seamless computation of higher-precision datatypes, extends effectively to even higher bit-width floating-point formats. FP64 computations, for instance, can mirror the FP32C process. Dot-product units receive two FP64 values, decoupling them into four components (high-high, high-low, low-high, low-low). Subsequently, they perform four dot-product operations using the same swapping policy as FP32C, but without sign bit flipping. Finally, these units accumulate the multiplier results into FP64 registers. This analogous approach easily extends to even higher bit-width floating-point formats, such as FP128, and their complex counterparts. Furthermore, the original arithmetic unit requirements remain flexible, accommodating options like 8-bit or 32-bit multipliers for composing higher bit-width datatypes, thereby broadening the design exploration space. To demonstrate the capabilities of M^{3}XU within the constraints of conventional hardware, this paper focuses on extending single-precision datatypes using 16-bit multipliers.

V. Experimental Methodology

This section describes the hardware synthesis results and the evaluation framework that we use to evaluate M^{3}XU.

A. Hardware validation

We implemented the baseline MXU and M^{3}XU using the system Verilog and synthesized them using Synopsys Design Compiler with the 45nm FreePDK45 library. We also used ModelSim to validate the correctness of our designs. The baseline MXU resembles the capability of a Tensor Core in Ampere [58] and Accel-Sim [39] as it can perform $8 \times 8 \times 4$ matrix multiplications on FP16/BF16 input elements and accumulates results in FP32.

B. Performance emulation framework

M^{3}XU's extension of the tensor instruction set does not change how the software uses the MXU. The programming model, interaction with the register file, and use of low-level instructions remain the same as the existing Tensor Cores. Therefore, we leverage existing Tensor Core MMA instructions and extend high-level CUDA GEMM libraries for performance evaluation, similar to prior works [18], [90]. Unlike previous works, our performance emulation framework does not include correctness validation and error rate checking phases for two main reasons. First, a GEMM implementation using M^{3}XU MMA instructions applies identical algorithms and optimizations compared with ones using existing Tensor Core architectures, and the computation result of M^{3}XU is exactly the same as FP32. Second, unlike software emulation approaches proposed in prior works [18], [50], [62], which remain to have between one and several bits of precision loss, M^{3}XU can retrieve standard IEEE 754 floating-point formats. Accordingly, computation results using M^{3}XU instructions introduce no additional error compared to conventional FP32 ALUs (e.g., CUDA cores). Therefore, our framework focuses on studying the performance of M^{3}XU.

Name	Description
M^{3}XU_sgemm_pipelined	FP32 GEMM Kernel by invoking 1 more MMA instruction, and 2× problem shape
M^{3}XU_sgemm	FP32 GEMM Kernel with controlled clock frequency
M^{3}XU_cgemm_pipelined	FP32 Complex GEMM Kernel by invoking 3 more MMA instructions, and 4× problem shape
M^{3}XU_cgemm	FP32 Complex GEMM Kernel with controlled clock frequency

TABLE II: M^{3}XU GEMM Kernels provided by performance emulation framework

1) Emulating performance using existing Tensor Core MMA instructions: The evaluation methodology in this paper conservatively but correctly emulates M^{3}XU performance using existing Tensor Core MMA in the following three aspects.

(a) MMA instruction latency: Since each M^{3}XU FP32 MMA instruction requires two steps of computation within the dot product unit, each M^{3}XU FP32 MMA instruction takes 2× the cycles of an FP16 Tensor Core MMA. Therefore, the emulation framework implicitly instruments 2 FP16 Tensor Core MMA instructions to emulate the latency of an M^{3}XU FP32 MMA instruction. Similarly, an M^{3}XU FP32C MMA instruction requires 4 FP16 Tensor Core MMA instructions.

(b) Instruction count: Each M^{3}XU FP32 MMA instruction computes one 16×8×8 matrix multiplication, which computes half of existing Tensor Core MMA instruction , the total instruction count of computing the same shape of FP32 matrix multiplication using M^{3}XU MMA instruction is 2× FP16 matrix multiplication using existing FP16 Tensor Core MMA instruction. Similarly, M^{3}XU FP32C matrix multiplication requires 4× total instruction count.

(c) Memory access behavior: M^{3}XU leverages the existing Tensor Core memory hierarchy. A single M^{3}XU MMA instruction incurs the same memory access latency as an FP16 Tensor Core MMA instruction, generating the same number of fragments and fetching the same amount of data from shared memory to the register file. The total memory traffic of M^{3}XU FP32 and FP32C matrix multiplication is 2× and 4× that of FP16 matrix multiplication, respectively.

2) Constructing performance emulation kernels: Our framework utilizes CUTLASS [62] to efficiently implement hierarchical blocked GEMM kernels. To assure section V-B1 (a), our framework takes advantage of PTX injection and cooperates with CUTLASS's code generator, which assures all CUTLASS kernels generate 2× or 4× more MMA instructions. To assure section V-B1 (b) and (c), for any GEMM kernel launched with a problem shape of M×K×N, our framework launches M×K×N×2 or M×K×N×4 kernels for FP32 and FP32C, respectively. Since M^{3}XU may need to operate at a lowert frequency due to extentsion of Tensor Core, our framework uses *nvidia-smi* to control GPU SM clock frequency. Table II lists all four GEMM kerenls used in our evaluation.

	Baseline MXUs		M^{3}XU		
	FP16	FP32 w/o FP32C	M^{3}XU w/o FP32C	M^{3}XU	M^{3}XU pipelined
Area	1	3.55	1.37	1.41	1.47
Cycle Time	1	1.00	1.21	1.21	1.00
Power	1	7.97	0.66	0.69	1.07

TABLE III: The relative overhead of various M^{3}XU implementations, compared with the three reference designs, the baseline FP16 MXU and two naively extended FP32-MXU with half/same amount of inputs

C. Environment configuration

We deployed our performance emulation framework on an Nvidia DGX Station. Our experiments use an installed Nvidia A100 GPU based on the Ampere architecture with 40 GB HBM. The machine hosts a DGX-specialized Ubuntu (Linux kernel version 5.4.0-81-generic) with NVIDIA's CUDA 11.4 using driver version 470.57.02. Our performance emulation framework controls the Tensor Core frequency of our testbed GPU to run at 1170 MHz. It can optionally reduce the Tensor Core frequency to 960 MHz when launching selected performance emulation kernels.

VI. Experimental Results

This section presents the performance of M^{3}XU against various approaches for FP32 and FP32C in critical kernels, including GEMM, 2D-convolution, and FFT. We also selected four representative applications as case studies. In summary, M^{3}XU delivers up to 3.89× speedup on FP32 GEMM compared to conventional vector processing units and 1.63× speedup compared to prior approaches in support of single precision GEMM. M^{3}XU can directly perform FFT calculations without approximations and achieves up to 1.99 × compared with state-of-the-art cuFFT libraries.

A. Hardware synthesis result

We presented three-versions M^{3}XU implementations that incorporate our proposed extensions: (A) An M^{3}XU that only supports FP32 MMA in addition to FP16 MMA. (B) An M^{3}XU that does not change the existing pipeline of the baseline MXU to minimize the area overhead, (C) An M^{3}XU that separates an additional pipeline stage in assigning the inputs for different phases to maintain the same clock rate as the baseline.

Table III summarizes the synthesis results. Adding the proposed FP32 MMA support in M^{3}XU incurs 37% area overhead. However, 56% of that overhead comes from the arithmetic to support the additional 1 bit of mantissa. If we extend an MXU that already supports 12-bit mantissas, the area-overhead of supporting FP32 in M^{3}XU is only 16%.

The complete M^{3}XU supports both FP32 and FP32C and incurs 4% more area overhead than just supporting FP32. However, M^{3}XU will result in a 21% increase in cycle time if we do not pipeline the data assignment stage. Despite the slowdown in supporting the baseline MXU operations, the lowered frequencies of these implementations allow the resulting M^{3}XUs to operate at 31% or 34% lower power with or without FP32C support, respectively. To maintain the same

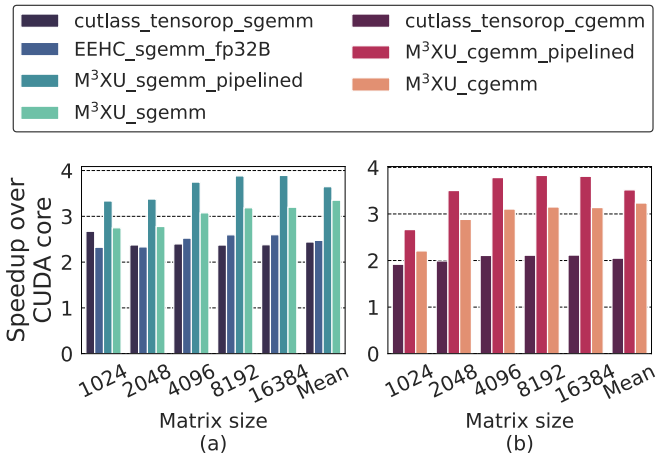

Fig. 4: Performance comparison of GEMM using different Tensor Core approaches: (a) SGEMM, (b) CGEMM.

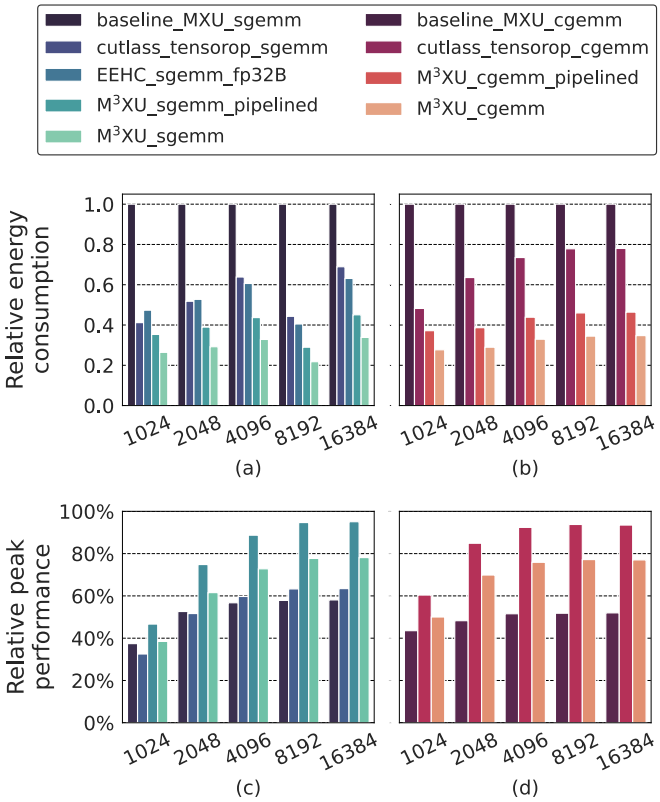

Fig. 5: Relative analysis of M^{3}XU: (a) relative energy of SGEMM, (b) relative energy of CGEMM, (c) relative performance of SGEMM, (d) relative performance of CGEMM.

cycle time, an alternative design that pipelines the data multiplexing with the two-phase computation would incur 47% area overhead to the baseline and result in a 7% increase in power. The speedup of applications can still make the pipelined design more energy-efficient than other alternatives and pay off the slight increase in power consumption. However, even with 47% area overhead, the area increase is only 4% to the SM's die size. In contrast to the area-efficiency of M^{3}XU, if we were to double the front-end memory bandwidth of Tensor Cores, completely double the bit-width of input and output data, and use FP32 multipliers, we could achieve the same throughput as FP16 MXUs. However, the design will lead to 3.55× area overhead and almost 8× power consumption but does not provide any support for FP32C as M^{3}XU does.

B. Microbenchmark

Table IV shows the five GEMM implementations we selected to represent the performance of existing approaches in single-precision GEMM. Four baseline kernels use FP32 arithmetic.

(1) `cutlass_simt_sgemm` computes using standard IEEE-754 FP32 and CUDA cores;

(2) `cutlass _tensorop_sgemm` is a vendor-provided, software emulated FP32 kernel using TF32 and Tensor Cores. It computes FP32 GEMM using 3 TF32 Tensor Core GEMMs; it's worth mentioning that perfectly emulating FP32 GEMM using TF32 Tensor Core will require 4 TF32 GEMM operations. CUTLASS omitted the 4th GEMM on two low-order portions of the FP32 inputs to reach better performance.

(3) `EEHC_sgemm_fp32B` is another software solution [50] that decouples each FP32 GEMM into 3 BF16 Tensor Core GEMM. For FP32C, we select three kernels with similar configurations as their counterparts in FP32.

GEMM performance compared with CUDA cores: Figure 4 (a) shows the performance gain of M^{3}XU and prior approaches on single precision GEMM kernels (SGEMM) over GPU SIMT GEMM kernels with problem sizes ranging from $1K \times 1K \times 1K$ to $16K \times 16K \times 16K$. M^{3}XU SGEMM achieves up to 3.89× and an average of 3.64× speedup across all SGEMM problem sizes compared with the baseline CUDA/SIMT cores. Other alternatives only achieve up to 2.67× speedup and spend 14% execution time in decoupling inputs on average. Excluding the data decoupling time, other alternatives still fall behind M^{3}XU with a maximum speedup at 3.10× due to the increased number of dynamic instructions. The performance gain of M^{3}XU saturates at about 3.89× when the SGEMM problem size is larger than $8K \times 8K \times 8K$.

Figure 4 (b) shows the evaluation result of FP32C GEMM. M^{3}XU FP32C SGEMM achieves 3.51× speedup on average, compared with baseline SIMT FP32C SGEMM. With various problem sizes, M^{3}XU achieved up to 3.82× speedup across all problem sizes. Software alternatives using three TF32 Tensor Core operations can only outperform baseline for up to 2.1×, 1.7× slower than M^{3}XU. With reduced clock frequency, non-pipelined M^{3}XU still reveals 3.35×, and 3.51× speedup over baseline kernels for FP32 and FP32C, respectively.

Energy consumption: Figure 5 (a) and (b) shows the relative energy consumption of M^{3}XU compared with baseline FP32-MXUs that implemented with full bit-width multipliers (i.e., baseline_MXU_sgemm and baseline_MXU_cgemm in

Name	Compute Type	Precision	Description
FP32 Kernels			
`cutlass_simt_sgemm`	SIMT	fp32	cutlass fp32 gemm kernel using CUDA cores
`cutlass_tensorop_sgemm`	TensorOp	fp32	cutlass software emulation fp32 gemm kernel using 3 tf32 gemm
`EEHC_sgemm_fp32B`	TensorOp	fp32-B	Prior software emulation [50] using three bf16s warp level gemm
FP32-Complex Kernels			
`cutlass_simt_cgemm`	SIMT	fp32 complex	cutlass fp32 complex gemm kernel using CUDA cores
`cutlass_tensorop_cgemm`	TensorOp	fp32 complex	cutlass software emulation fp32 complex gemm kernel using 3 tf32 complex gemm

TABLE IV: Baseline and prior GEMM Kernels

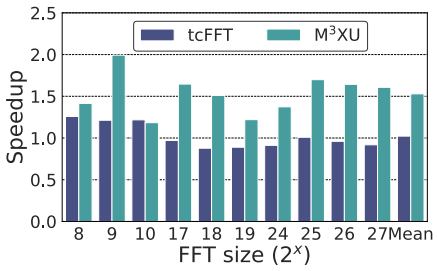

Fig. 6: Speedup of FFT over *cuFFT*

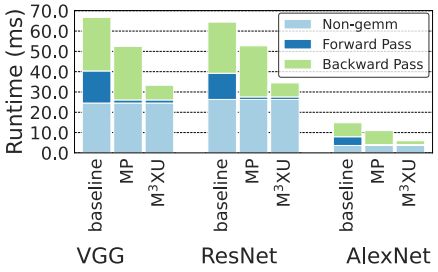

Fig. 7: End-to-end Latency of single iteration training of CNN models

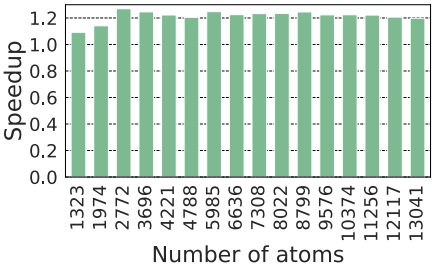

Fig. 8: Speedup of MRF dictionary generation over CUDA cores

Figure 5) and alternatives on FP16-MXUs. Despite 7% higher power consumption than FP16-MXUs, M^{3}XU's energy consumption is 61% lower than FP32-MXU and 27% lower than the most energy-efficient software solution when performing FP32 operations. The non-pipelined version of M^{3}XU enjoys lower power consumption as it operates at a lower frequency while delivering decent performance gain over other alternatives. Therefore, the non-pipelined version of M^{3}XU saves the most energy, 71% lower compared against FP32-MXUs and 45% lower than the most energy-efficient software-emulated solutions. When computing FP32 complex numbers, M^{3}XU's energy consumption is 57% lower than FP32-MXU and 36% lower than software solutions. The non-pipelined version of M^{3}XU saves the most energy, 68% lower compared to FP32-MXUs and 52% lower than software solutions.

GEMM performance compared with theoretical peak performance: As mentioned in Section III, the performance target of FP32 GEMM and CGEMM is 25% and 6.25% of FP16 Tensor Core TOPS. To demonstrate that M^{3}XU meets theoretical performance without loss of precisions, we compared the relative peak performance of M^{3}XU and other software solutions with the performance targets. Figure 5 (c) and (d) shows both M^{3}XU SGEMM and CGEMM kernels reach more than 94% of the theoretical performance, while all prior software solutions only reach up to 63% of the target.

C. Case studies

We demonstrate the impact of M^{3}XU in four real-world applications.

1) FFT: M^{3}XU can directly compute FFT using its FP32C mode to improve runtime performance. We specifically evaluated the performance of FFT implemented using M^{3}XU compared with prior GPU implementations [47], [60]. *tcFFT* [47] is the state-of-the-art Tensor Core FFT implementation, which uses 4× more operations on Tensor Core to compute each complex GEMM. Since *tcFFT* only supports FP16 complex numbers, for fair comparisons, we extended *tcFFT* to support single precision GEMM using TF32 Tensor Cores and compared the end-to-end speedup with cuFFT [60], a vendor-optimized GPU FFT library, as the baseline. Figure 6 reveals that M^{3}XU can achieve up to 1.99× and an average of 1.52× speedup over *cuFTT* across all FFT sizes. Conversely, *tcFFT* does not improve performance over *cuFTT*.

2) DNN training: This case study evaluates the performance improvements of M^{3}XU on machine learning workloads using Nebula benchmark [40]. We extended ResNet, VGG, and AlexNet. Figure 7 shows that M^{3}XU is 1.65× faster than conventional mixed-precision training.

Our proposed M^{3}XU acceleration utilizes the existing Tensor Core GEMM during the forward pass to attain the same advantages as mixed-precision training, resembling the process in Pytorch. For the backward pass, the existing implementation only applies SIMT-based kernels to mixed precision training due to the absence of FP32 Tensor Core instructions. With M^{3}XU's capability in achieving the same numerical results as standard FP32, M^{3}XU can accelerate the backward pass that accounts for 39.6%, 39.1%, and 46.5% runtime in VGG, ResNet, and AlexNet, respectively. M^{3}XU reveals 3.6× speedup for a backward pass that the existing mixed-precision method cannot improve.

3) MRF: The primary challenge in MRF is the computationally demanding reconstruction process, which relies on the accuracy of the signal model used. MRF often requires the use of high-precision complex floating point formats. Our baseline, SnapMRF [80], is a state-of-the-art GPU-based MRF approach that uses complex matrix multiplication for dictionary generation and pattern matching phase of MRF, and the dictionary generation phase takes 98.2% of total run time.

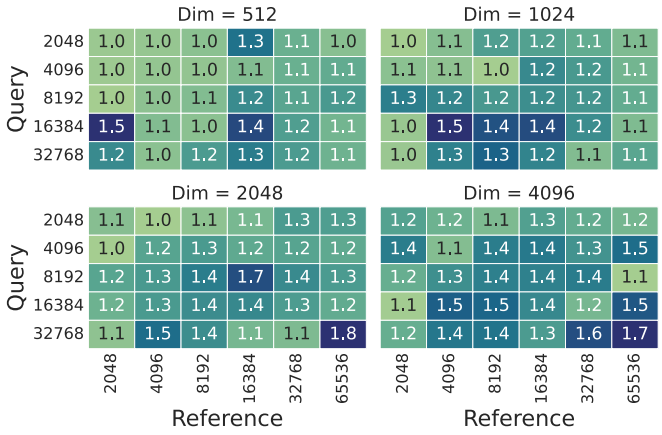

Fig. 9: KNN speedup over CUDA cores

CGEMM accounts for 22% of the runtime in the dictionary generation phase. As shown in Figure 8, M^{3}XU achieves up to 1.26× speedup in end-to-end latency of dictionary generation phase over the `cublas_cgemm`-based baseline.

4) Statistical learning: Conventional statistical learning methods, like K-Nearest Neighbor(KNN) and K-Means, are also SGEMM intensive but precision-sensitive. We evaluated KNN-CUDA [79] that intensively uses the `cuBlas_sgemm` function. Although conventional FP16 Tensor Cores can accelerate the GEMM function, the reduced precision will produce meaningless computation results for input data with extremely small values. On the other hand, M^{3}XU can accelerate FP32 matrix operations without precision loss.

Figure 9 shows the heatmaps the performance gain of KNN using M^{3}XU over the `cuBlas_sgemm`-based implementation. We evaluated KNN workloads with total reference and query points ranging from 2048 to 65536 with four dimensions ranging from 512 to 4096. We chose a fixed K of 16 as configuration as the portion of runtime contributed by GEMM increases along with input sizes, M^{3}XU reveals more performance gain and tops at 1.8× for large input sizes.

VII. Related Work

In addition to the related work described in Section I and Section II-C, several other lines of research deserve mention.

Mixed-precision Mixed-Precision Fused Multiply-Add Vector Units M^{3}XU distinguishes itself from existing multi-precision Fused Multiply-Add (FMA) floating-point units [11], [24], [33], [34], [51], [52], [75], [88] as M^{3}XU is the only design that exploits the potential of reusing multiple low-precision floating-point multipliers within MXUs. Prior work on FMA units focuses on vector processing or uses a downward-support approach that enables lower-precision arithmetic using higher-precision hardware.

Mixed-precision application-specific accelerators: Prior work has intensively investigated low-precision (under INT8) neural networks [13], [21], [27], [45], [46], [65], [81], [86], [92] and corresponding accelerator designs [12], [19], [22], [30], [37], [69], [71], [84], [85], [93] to exploit the error-tolerance aspect of neural networks and the high arithmetic density of low-bitwidth-based MXUs. Although low-precision models and corresponding accelerator designs can improve inferencing latency, training throughput, and memory efficiency, ensuring convergence and acceptable accuracy drop are still challenging. Thus, several techniques and accelerators support multiple precisions, allowing users to choose appropriate precision [12], [22], [69], [71]. Furthermore, several prior projects propose arbitrary precision support to enable various sizes of data programmer-transparently [19]. However, as previous works focus on low-bitwidth computations, naively implementing multi-precision or arbitrary-precision techniques in high-bitwidth computation might incur high computational overhead. To our knowledge, this is the first study to tailor MXUs to compute high-bitwidth computations.

Complex matrix multiplication: Without M^{3}XU's hardware support, existing projects must perform four matrix multiplications (real-real, real-imaginary, imaginary-real, and imaginary-imaginary parts) for complex numbers [17], [17], [47], [73] or they have to avoid complex number arithmetic but use software-based approximation techniques [32], [61], [62] or implement a separate accelerator or FPGA acceleration [42].

Synthesis of wider hardware using narrower function units: M^{3}XU is different from hardware synthesis that uses narrower function units to achieve functions with wider bitwidth [2]. Existing work focuses on using the exact block to create new functions, but M^{3}XU observes the similarity of desired functions and suggests extensions in existing functional units for more purpose. Therefore, existing automatic synthesization/optimization techniques cannot achieve these non-trivial extensions that M^{3}XU presents.

VIII. Conclusion

As matrix multiplications are at the core of many problems, the MXUs in AI/ML accelerators can have a broader impact than their current focuses. However, the cost of extending these low-precision MXUs prevents AI/ML accelerators from embracing more applications.

M^{3}XU provides a timely solution that allows MXUs to support standard FP32 floating point numbers and FP32C complex numbers at their theoretical throughput under current memory technologies, with relatively minor area overhead. M^{3}XU brings an average 3.64× on SGEMM, and faithful computation on CGEMM with close to 3.51× speedup.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments. This work was sponsored by Intel Labs and two National Science Foundation (NSF) awards, CNS-2007124 and CNS-2231877. This work was also supported by new faculty start-up funds from University of California, Riverside. Lastly, this work was supported by Y-BASE R&E Institute, a Brain Korea 21 four program, Yonsei University. Hung-Wei Tseng and Won Woo Ro are the co-corresponding authors.

REFERENCES

[1] AMD. AMD matrix cores. https://www.amd.com/content/dam/amd/en/documents/instinct-business-docs/white-papers/amd-cdna2-white-paper.pdf, 2023.

[2] AMD. Versal ACAP DSP Engine Architecture Manual. https://docs.xilinx.com/r/en-US/am004-versal-dsp-engine/Advanced-Math-Applications, 2023.

[3] Apple Inc. Apple M1. https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/, 11 2020.

[4] Martin Arjovsky, Amar Shah, and Yoshua Bengio. Unitary evolution recurrent neural networks. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 1120–1128. JMLR.org, 2016.

[5] Arm Corporation. Introducing the Scalable Matrix Extension for the Armv9-A Architecture. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/scalable-matrix-extension-armv9-a-architecture, 2021.

[6] David H. Bailey and Jonathan M. Borwein. High-precision arithmetic in mathematical physics. *Mathematics*, 3(2):337–367, 2015.

[7] D.H. Bailey. High-precision floating-point arithmetic in scientific computation. *Computing in Science & Engineering*, 7(3):54–61, 2005.

[8] Dominic W Berry, Graeme Ahokas, Richard Cleve, and Barry C Sanders. Efficient quantum algorithms for simulating sparse hamiltonians. *Communications in Mathematical Physics*, 270:359–371, 2007.

[9] Tamal Bose and Francois Meyer. *Digital signal and image processing*. John Wiley & Sons, Inc., 2003.

[10] S Allen Broughton and Kurt Bryan. *Discrete Fourier analysis and wavelets: applications to signal and image processing*. John Wiley & Sons, 2018.

[11] Nicolas Brunie, Florent de Dinechin, and Benoit de Dinechin. A mixed-precision fused multiply and add. In *2011 Conference Record of the Forty Fifth Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*, pages 165–169, 2011.

[12] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro*, 41(2):29–35, 2021.

[13] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, page 3123–3131, Cambridge, MA, USA, 2015. MIT Press.

[14] Ivo Danihelka, Greg Wayne, Benigno Uria, Nal Kalchbrenner, and Alex Graves. Associative long short-term memory. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 1986–1994. JMLR.org, 2016.

[15] Timothy A. Davis. Algorithm 1000: Suitesparse:graphblas: Graph algorithms in the language of sparse linear algebra. *ACM Trans. Math. Softw.*, 45(4), dec 2019.

[16] Florent de Dinechin and Gilles Villard. High precision numerical accuracy in physics research. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 559(1):207–210, 2006. Proceedings of the X International Workshop on Advanced Computing and Analysis Techniques in Physics Research.

[17] Sultan Durrani, Muhammad Saad Chughtai, Mert Hidayetoglu, Rashid Tahir, Abdul Dakkak, Lawrence Rauchwerger, Fareed Zaffar, and Wen-mei Hwu. Accelerating fourier and number theoretic transforms using tensor cores and warp shuffles. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 345–355, 2021.

[18] Boyuan Feng, Yuke Wang, Guoyang Chen, Weifeng Zhang, Yuan Xie, and Yufei Ding. Egemm-tc: Accelerating scientific computing on tensor cores with extended precision. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, pages 278–291, 2021.

[19] Boyuan Feng, Yuke Wang, Tong Geng, Ang Li, and Yufei Ding. Apnn-tc: Accelerating arbitrary precision neural networks on ampere gpu tensor cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[20] Vincent Garcia, Éric Debreuve, Frank Nielsen, and Michel Barlaud. K-nearest neighbor search: Fast gpu-based implementations and application to high-dimensional feature matching. In *2010 IEEE International Conference on Image Processing*, pages 3757–3760, 2010.

[21] Tong Geng, Ang Li, Tianqi Wang, Chunshu Wu, Yanfei Li, Runbin Shi, Wei Wu, and Martin Herbordt. O3bnn-r: An out-of-order architecture for high-performance and regularized bnn inference. *IEEE Transactions on Parallel and Distributed Systems*, 32(1):199–213, 2021.

[22] Soroush Ghodrati, Hardik Sharma, Cliff Young, Nam Sung Kim, and Hadi Esmaeilzadeh. Bit-parallel vector composability for neural acceleration. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[23] Serban Giuroiu. CUDA k-means data clustering, 2012.

[24] Mustafa Gök and Metin Mete Özbilen. Multi-functional floating-point maf designs with dot product support. *Microelectron. J.*, 39(1):30–43, jan 2008.

[25] Google LLC. Coral M.2 Accelerator Datasheet. https://coral.withgoogle.com/static/files/Coral-M2-datasheet.pdf, 2019.

[26] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.

[27] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[28] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. Extensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 319–333, New York, NY, USA, 2019. Association for Computing Machinery.

[29] Rudi Helfenstein and Jonas Koko. Parallel preconditioned conjugate gradient algorithm on gpu. *Journal of Computational and Applied Mathematics*, 236(15):3584–3590, 2012. Proceedings of the Fifteenth International Congress on Computational and Applied Mathematics (ICCAM-2010), Leuven, Belgium, 5-9 July, 2010.

[30] Brian Hickmann, Jieasheng Chen, Michael Rotzin, Andrew Yang, Maciej Urbanski, and Sasikanth Avancha. Intel nervana neural network processor-t (nnp-t) fused floating point many-term dot product. In *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)*, pages 133–136, 2020.

[31] Akira Hirose and Shotaro Yoshida. Generalization characteristics of complex-valued feedforward neural networks in relation to signal coherence. *IEEE Transactions on Neural Networks and Learning Systems*, 23(4):541–551, 2012.

[32] Jared Hoberock and Nathan Bell. Thrust: A parallel template library. http://thrust.github.io/, 2010.

[33] Libo Huang, Sheng Ma, Li Shen, Zhiying Wang, and Nong Xiao. Low-cost binary128 floating-point fma unit design with simd support. *IEEE Transactions on Computers*, 61(5):745–751, 2012.

[34] Libo Huang, Li Shen, Kui Dai, and Zhiying Wang. A new architecture for multiple-precision floating-point multiply-add fused unit design. In *18th IEEE Symposium on Computer Arithmetic (ARITH '07)*, pages 69–76, 2007.

[35] Intel Corporation. Intrinsics for Intel(R) Advanced Matrix Extensions (Intel(R) AMX) Instructions. https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-intel-advanced-matrix-extensions-intel-amx-instructions.html, 2021.

[36] Norman P Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. A Domain-specific Supercomputer for Training Deep Neural Networks. *Communications of the ACM*, 63(7):67–78, 2020.

[37] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean,

Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, 2017.

[38] TNorman P. Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Cliff Young, Xiang Zhou, Zongwei Zhou, and David Patterson. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. In *2023 ACM/IEEE 50th Annual International Symposium on Computer Architecture (ISCA)*, 2023.

[39] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. Accel-sim: An extensible simulation framework for validated gpu modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 473–486, 2020.

[40] Bogil Kim, Sungjae Lee, Chanho Park, Hyeonjin Kim, and William J. Song. The nebula benchmark suite: Implications of lightweight neural networks. *IEEE Transactions on Computers*, 70(11):1887–1900, 2021.

[41] Hyeonjin Kim, Sungwoo Ahn, Yunho Oh, Bogil Kim, Won Woo Ro, and William J. Song. Duplo: Lifting redundant memory accesses of deep neural networks for gpu tensor cores. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 725–737, 2020.

[42] Jong Hwan Ko, Burhan Mudassar, Taesik Na, and Saibal Mukhopadhyay. Design of an energy-efficient accelerator for training of convolutional neural networks using frequency-domain computation. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[43] Manny Ko, Ujjawal K Panchal, Héctor Andrade-Loarca, and Andres Mendez-Vazquez. Coshnet: A hybird complex valued neural network using shearlets. *arXiv preprint arXiv:2208.06882*, 2022.

[44] Bernd Lesser, Manfred Mücke, and Wilfried N. Gansterer. Effects of reduced precision on floating-point svm classification accuracy. *Procedia Computer Science*, 4:508–517, 2011. Proceedings of the International Conference on Computational Science, ICCS 2011.

[45] Ang Li, Tong Geng, Tianqi Wang, Martin Herbordt, Shuaiwen Leon Song, and Kevin Barker. Bstc: A novel binarized-soft-tensor-core design for accelerating bit-based approximated neural nets. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[46] Ang Li and Simon Su. Accelerating binarized neural networks via bit-tensor-cores in turing gpus. *IEEE Transactions on Parallel and Distributed Systems*, 32(7):1878–1891, 2021.

[47] Binrui Li, Shenggan Cheng, and James Lin. tcfft: A fast half-precision fft library for nvidia tensor cores. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–11, 2021.

[48] Hong Li, Nan Jiang, Zichen Wang, Jian Wang, and Rigui Zhou. Quantum matrix multiplier. *International Journal of Theoretical Physics*, 60:2037–2048, 2021.

[49] Wei Li, Des Mclernon, Kai-Kit Wong, Shilian Wang, Jing Lei, and Syed Ali Raza Zaidi. Asymmetric physical layer encryption for wireless communications. *IEEE Access*, 7:46959–46967, 2019.

[50] Zixuan Ma, Haojie Wang, Guanyu Feng, Chen Zhang, Lei Xie, Jiaao He, Shengqi Chen, and Jidong Zhai. Efficiently emulating high-bitwidth computation with low-bitwidth hardware. In *Proceedings of the 36th ACM International Conference on Supercomputing*, ICS '22, New York, NY, USA, 2022. Association for Computing Machinery.

[51] K. Manolopoulos, D. Reisis, and V.A. Chouliaras. An efficient multiple precision floating-point multiply-add fused unit. *Microelectronics Journal*, 49:10–18, 2016.

[52] K. Manolopoulos, D. Reisis, and V.A. Chouliaras. An efficient dual-mode floating-point multiply-add fused unit. In *2010 17th IEEE International Conference on Electronics, Circuits and Systems*, pages 5–8, 2010.

[53] S. Markidis, S. Chien, E. Laure, I. Peng, and J. S. Vetter. Nvidia tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531, Los Alamitos, CA, USA, may 2018. IEEE Computer Society.

[54] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training, 2017.

[55] G Niccolini, P Woitke, and B Lopez. High precision monte carlo radiative transfer in dusty media. *Astronomy & Astrophysics*, 399(2):703–716, 2003.

[56] T. Nitta. On the critical points of the complex-valued neural network. In *Proceedings of the 9th International Conference on Neural Information Processing, 2002. ICONIP '02.*, volume 3, pages 1099–1103 vol.3, 2002.

[57] Thomas Norrie, Nishant Patil, Doe Hyun Yoon, George Kurian, Sheng Li, James Laudon, Cliff Young, Norman Jouppi, and David Patterson. The design process for google's training chips: Tpuv2 and tpuv3. *IEEE Micro*, 41(2):56–63, 2021.

[58] NVIDIA. NVIDIA A100 Tensor Core GPU Architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf, 2020.

[59] NVIDIA. NVIDIA H100 Tensor Core GPU Architecture. https://resources.nvidia.com/en-us-tensor-core, 2022.

[60] NVIDIA. cuFFT. https://docs.nvidia.com/cuda/cufft/index.html, 2023.

[61] NVIDIA Corporation. CUDA Samples. https://github.com/NVIDIA/cuda-samples, 2023.

[62] NVIDIA Corporation. cuTlass 3.1. https://github.com/NVIDIA/cutlass, 2023.

[63] Hiroyuki Ootomo and Rio Yokota. Recovering single precision accuracy from tensor cores while surpassing the fp32 theoretical peak performance. *The International Journal of High Performance Computing Applications*, 36(4):475–491, 2022.

[64] Hiroyuki Ootomo and Rio Yokota. Recovering single precision accuracy from tensor cores while surpassing the fp32 theoretical peak performance. *International Journal of High Performance Computing Applications*, 36(4):475–491, 2022.

[65] Eunhyeok Park, Dongyoung Kim, and Sungjoo Yoo. Energy-efficient neural network accelerator based on outlier-aware low-precision computation. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 688–698. IEEE Press, 2018.

[66] Xiaoyue Qin, Ruwei Huang, and Huifeng Fan. An effective ntru-based fully homomorphic encryption scheme. *Mathematical Problems in Engineering*, 2021:1–9, 2021.

[67] Md Aamir Raihan, Negar Goli, and Tor M Aamodt. Modeling deep learning accelerator enabled gpus. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 79–92, 2019.

[68] Valerie Sarge and Michael Andersch. Tensor Core Performance: The Ultimate Guide. https://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/s21929-tensor-core-performance-on-nvidia-gpus-the-ultimate-guide.pdf, 2020.

[69] Ali Sazegari, Eric Bainville, Jeffry E Gonion, Gerard R Williams III, and Andrew J Beaumont-Smith. Outer product engine, March 15 2018. US Patent App. 15/264,002.

[70] Changpeng Shao. Quantum algorithms to matrix multiplication. *arXiv preprint arXiv:1803.01601*, 2018.

[71] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, Hadi Esmaeilzadeh, and Joon Kyung Kim. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 764–775, 2018.

[72] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. *SIGPLAN Not.*, 48(8):135–146, feb 2013.

[73] Anumeena Sorna, Xiaohe Cheng, Eduardo D'Azevedo, Kwai Won, and Stanimire Tomov. Optimizing the fast fourier transform using mixed precision on tensor core hardware. In *2018 IEEE 25th International Conference on High Performance Computing Workshops (HiPCW)*, pages 3–7, 2018.

[74] Dimitri Tan, Carl E. Lemonds, and Michael J. Schulte. Low-power multiple-precision iterative floating-point multiplier with simd support. *IEEE Transactions on Computers*, 58(2):175–187, 2009.

[75] Hongbing Tan, Gan Tong, Libo Huang, Liquan Xiao, and Nong Xiao. Multiple-mode-supporting floating-point fma unit for deep learning processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 31(2):253–266, 2023.

[76] Zhi-Hao Tan, Yi Xie, Yuan Jiang, and Zhi-Hua Zhou. Real-valued backpropagation is unsuitable for complex-valued neural networks. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 34052–34063. Curran Associates, Inc., 2022.

[77] Chiheb Trabelsi, Olexa Bilaniuk, Ying Zhang, Dmitriy Serdyuk, Sandeep Subramanian, Joao Felipe Santos, Soroush Mehri, Negar Rostamzadeh, Yoshua Bengio, and Christopher J Pal. Deep complex networks. *arXiv preprint arXiv:1705.09792*, 2017.

[78] Roel Van Beeumen, Daan Camps, and Neil Mehta. Qclab++: Simulating quantum circuits on gpus. *arXiv preprint arXiv:2303.00123*, 2023.

[79] Michel Barlaud Vincent Garcia, Éric Debreuve. kNN-CUDA. https://github.com/vincentfpgarcia/kNN-CUDA/, 2018.

[80] Dong Wang, Jason Ostenson, and David S. Smith. snapmrf: Gpu-accelerated magnetic resonance fingerprinting dictionary generation and matching using extended phase graphs. *Magnetic Resonance Imaging*, 66:248–256, 2020.

[81] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. Haq: Hardware-aware automated quantization with mixed precision. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[82] Yang Wang, Chen Zhang, Zhiqiang Xie, Cong Guo, Yunxin Liu, and Jingwen Leng. Dual-side sparse tensor core. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ISCA '21, page 1083–1095. IEEE Press, 2021.

[83] Scott Wisdom, Thomas Powers, John R. Hershey, Jonathan Le Roux, and Les Atlas. Full-capacity unitary recurrent neural networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, page 4887–4895, Red Hook, NY, USA, 2016. Curran Associates Inc.

[84] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. Quantized convolutional neural networks for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[85] Zhaohui Yang, Yunhe Wang, Kai Han, Chunjing XU, Chao Xu, Dacheng Tao, and Chang Xu. Searching for low-bit weights in quantized neural networks. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 4091–4102. Curran Associates, Inc., 2020.

[86] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

[87] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. Gamma: Leveraging gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 687–701, New York, NY, USA, 2021. Association for Computing Machinery.

[88] Hao Zhang, Dongdong Chen, and Seok-Bum Ko. Efficient multiple-precision floating-point fused multiply-add with mixed-precision support. *IEEE Transactions on Computers*, 68(7):1035–1048, 2019.

[89] Xin-Ding Zhang, Xiao-Ming Zhang, and Zheng-Yuan Xue. Quantum hyperparallel algorithm for matrix multiplication. *Scientific reports*, 6(1):1–7, 2016.

[90] Yunan Zhang, Po-An Tsai, and Hung-Wei Tseng. Simd2: A generalized matrix instruction set for accelerating tensor computation beyond gemm. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 552–566, New York, NY, USA, 2022. Association for Computing Machinery.

[91] Yilun Zhao, Yanan Guo, Yuan Yao, Amanda Dumi, Devin M Mulvey, Shiv Upadhyay, Youtao Zhang, Kenneth D Jordan, Jun Yang, and Xulong Tang. Q-gpu: A recipe of optimizations for quantum circuit simulation using gpus. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 726–740, 2022.

[92] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[93] Bohan Zhuang, Lingqiao Liu, Mingkui Tan, Chunhua Shen, and Ian Reid. Training quantized neural networks with a full-precision auxiliary module. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.