UNIVERSITY OF CALIFORNIA
RIVERSIDE

Democratizing Tensor Processors: Efficient and Generalized Tensor Computation with Architectural Support

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Electrical Engineering

by

Yunan Zhang

June 2024

Dissertation Committee:

Dr. Hung-Wei Tseng, Chairperson
Dr. Daniel Wong
Dr. Rajiv Gupta

The Dissertation of Yunan Zhang is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I would like to acknowledge the chair of my committee, Dr. Hung-Wei Tseng, for passing all his knowledge, experience, and ways of life to me throughout my entire graduate program. Dr. Tseng is my advisor, a model for my career, and more importantly, a friend for life.

I would like to acknowledge Dr. Po-An Tasi, who shared countless techniques and insights for my research. His encouragement and intelligence inspired me in an irreplaceable way.

I would also like to acknowledge Dr. Daniel Wong and Dr. Rajiv as my committee members. I appreciate their effort in overseeing my thesis and provide valuable feedbacks.

I would like to thank Dr. Dongho Ha, his perseverance and enthusiastic inspired me in some hard times.

I would like to thank all of my labmates in the Extreme Scale Computer Architecture Laboratory for always sharing valuable research ideas and solving countless technical issues.

Finally, I would like to thank my family for their unwavering support throughout my education.

To my mom for all the support.

ABSTRACT OF THE DISSERTATION

Democratizing Tensor Processors: Efficient and Generalized Tensor Computation with Architectural Support

by

Yunan Zhang

Doctor of Philosophy, Graduate Program in Electrical Engineering
University of California, Riverside, June 2024
Dr. Hung-Wei Tseng, Chairperson

Tensor processors, notably matrix units (MXUs), have become indispensable in accelerating matrix operations for machine learning. However, their specialized design and limited support for varying data types and operators have hindered wider adoption. This dissertation tackles these limitations by enhancing the flexibility and capabilities of tensor processors across three key areas.

First, multi-mode matrix processing units (M^3XU) are introduced, capable of efficiently handling both IEEE 754 single-precision and complex 32-bit floating-point numbers. This innovation broadens the applicability of MXUs in scientific computing without requiring significant modifications to existing systems.

Second, $SIMD^2$, a novel programming paradigm and architecture, is proposed to extend MXU capabilities beyond matrix multiplications to a wider range of generalized matrix operations. By leveraging existing tensor processor infrastructure, $SIMD^2$ offers substantial performance improvements over traditional approaches, further expanding the utility of these processors.

Finally, to address the challenges of memory-bound sparse tensor computations, a new compute dataflow, Output-stationary-Element-wise-Input-stationary (OEI), and its corresponding architecture, SIDA, are presented. This combined approach exploits inter- and intra-operator reuse opportunities, significantly reducing memory traffic and enhancing the efficiency of tensor processors in sparse linear algebra workloads.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Tensor processors, especially matrix processors, have emerged as a cornerstone of modern computing due to their ability to accelerate computationally intensive tasks involving large multi-dimensional arrays of data (tensors). These processors are specifically designed to optimize matrix operations, which are fundamental to a wide range of applications across various domains. In the realm of artificial intelligence, matrix processors power deep learning algorithms, enabling efficient training and inference of complex neural networks. They are also critical for scientific computing, where they speed up simulations in fields like computational fluid dynamics, molecular dynamics, and weather forecasting. Additionally, matrix processors play a crucial role in graphics processing, image and video processing, signal processing, and financial modeling. The importance of these processors stems from their ability to significantly reduce computation time, enhance energy efficiency, and enable breakthroughs in fields that heavily rely on matrix computations. Algorithms like matrix multiplication, convolution, and decomposition are optimized for these processors, ensur-

ing high performance and scalability. The widespread adoption of tensor processors across diverse fields underscores their significance in driving innovation and accelerating progress in computationally demanding areas.

Tensor processors and matrix units (MXUs) come in various forms, each tailored to specific applications and workloads. Google's Tensor Processing Units (TPUs), application-specific integrated circuits (ASICs) designed for accelerating machine learning workloads, are equipped with powerful matrix multiply units (MXUs) and high-bandwidth memory to handle large matrix operations efficiently. NVIDIA's GPUs, originally designed for graphics rendering, have evolved to incorporate Tensor Cores optimized for accelerating matrix operations, making them ideal for deep learning training and inference. Additionally, Intel's Advanced Matrix Extensions (AMX), a set of instructions integrated into Intel Xeon Scalable processors, significantly enhance CPU performance in matrix operations, enabling them to competently handle computationally intensive tasks involving matrices. These are just a few examples highlighting the diverse landscape of tensor processors and MXUs, each designed to address the growing demands of computationally intensive tasks that rely on efficient matrix operations.

Tensor processors are characterized by their specialized architecture designed to accelerate matrix operations, which are fundamental to various computational tasks. These processors often feature large arrays of multiply-accumulate (MAC) units, optimized for parallel execution of matrix computations. Additionally, they typically have high-bandwidth memory interfaces to efficiently feed data into the computational units and store the results. Tensor processors often support lower numerical precision formats, such as 8-bit or

16-bit floating point, as these are sufficient for many machine learning applications and allow for higher computational throughput. Furthermore, they may incorporate specialized instructions or hardware accelerators for specific operations like matrix multiplication or convolution, further enhancing their performance in targeted workloads. While tensor processors offer significant computational advantages for matrix operations, it's important to note that they often exhibit limitations in terms of domain specificity. Each tensor processor tends to be optimized for a particular set of data types, operator types, and operation types, making them highly efficient within their intended domain. For instance, some processors might excel at handling floating-point arithmetic with specific precision, while others might be tailored for integer operations. Similarly, they might be optimized for certain matrix operations like matrix multiplication or convolution, but less efficient for others. This specialization can restrict their flexibility when dealing with diverse computational tasks requiring a wider range of data and operation types.

Much like how GPUs evolved from graphics accelerators to general-purpose architectures, it is crucial for tensor processors to broaden their applicability across diverse domains, thereby democratizing their use. To achieve this, three key avenues of exploration emerge:

(1) Exploiting Data Precision: Investigating methods to enable a single tensor processor to efficiently handle multiple data precision formats is essential. This would allow for flexibility in addressing various computational tasks with varying accuracy requirements, thus expanding the processor's applicability beyond its initially designed domain.

(2) Exploiting Operator Diversity: Expanding the capabilities of tensor processors beyond multiplication and accumulation operations is pivotal. By incorporating support for a wider range of operators, these processors can cater to a broader spectrum of applications, extending their utility beyond their current niche.

(3) Exploring Operation and Data Reuse Opportunities for Sparse Applications: Sparse matrices, characterized by a significant proportion of zero elements, are prevalent in numerous real-world applications. Investigating techniques to exploit operation and data reuse opportunities within sparse matrix computations can significantly enhance the efficiency and applicability of tensor processors in these domains.

By pursuing these avenues, tensor processors can transcend their current limitations and evolve into versatile computational engines capable of accelerating a wide array of tasks across diverse fields, ultimately democratizing access to their computational power.This thesis will comprehensively explore the three key avenues of democratizing tensor processors. The following organizational structure will be employed to systematically investigate these aspects.

Chapter 2 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 3.89$\times$ speedup for 32-bit matrix multiplica-

tions and 3.8× speedup for complex number operations compared with conventional vector processing units.

Chapter 3 presents SIMD^2, a new programming paradigm to support generalized matrix operations with a semiring-like structure. SIMD^2 instructions accelerate eight more types of matrix operations, in addition to matrix multiplications. Since SIMD^2 instructions resemble a matrix-multiplication instruction, SIMD^2 architecture is built on top of any MXU architecture with minimal modifications. SIMD^2 provides up to 38.59× speedup and more than 6.94× on average over optimized CUDA programs, with only 5% of full-chip area overhead.

Chapter 4 presents Output-stationary-Element-wise-Input-stationary (OEI) datafow, a novel dataflow to capture both reuse opportunities in STA applications, and SIDA, a sparse dataflow architecture to support the OEI dataflow and maximize data reuse. Evaluation results show that SIDA with OEI dataflow is 19.99×/7.39× faster than CPU/GPU, and is 1.76× faster than an ideal sparse accelerator that cannot exploit inter-operator reuse.

Chapter 5 will provide a comprehensive review of related work, encompassing not only the landscape of existing tensor processors but also the ongoing efforts to democratize these powerful computational tools. This chapter will delve into the current state-of-the-art in tensor processor design, highlighting their architectures, capabilities, and limitations. Additionally, it will examine existing research and initiatives aimed at broadening the applicability of tensor processors by addressing challenges related to data precision, operator diversity, and sparse matrix computation.

Chapter 6 will serve as a culminating synthesis of the thesis, revisiting the three key avenues of democratization explored. This chapter will recapitulate the significant findings and contributions of each avenue, highlighting their potential to revolutionize the landscape of tensor processors. In addition, this chapter will identify and delve into potential future research directions.

Chapter 2

Exploiting Data Precision

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 [51, 73–75], present as another compute engine in modern GPU architectures [8, 121, 122], or integrate into CPUs as extensions to existing instruction set architectures (e.g., Intel AMX [69], ARM's SME [12], and Apple's Matrix Extensions [10]). 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 [75]. 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.

2.1 High precision floating point numbers for MXUs

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 [14, 15, 15, 36, 58, 117], data analytics/mining applications [43, 49], statistical learning [89], and graph analytics [35, 150] 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 [53, 56, 184], and using other formats can lead to unwanted results. Despite the error tolerance in inferencing, training NN models still rely on intensive FP32 operations [121], or require significant re-engineering to accommodate other data types [108].

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 [20, 21, 37, 93, 153] and security applications [95, 138]. Also, simulating quantum computing needs complex matrix multiplications to represent qubits and their operations [18, 94, 147, 163, 187, 190]. As multimedia signals become complex numbers after transformations, recent studies also show neural networks using complex number matrix multiplications are advantageous [11, 33, 60, 85, 86, 118, 158, 162, 173].

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 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 section 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 [39,102,105,127, 130]. As M^{3}XU enables native FP32C computations, M^{3}XU delivers better performance and more accurate results than software approximations [37, 93, 153].

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).

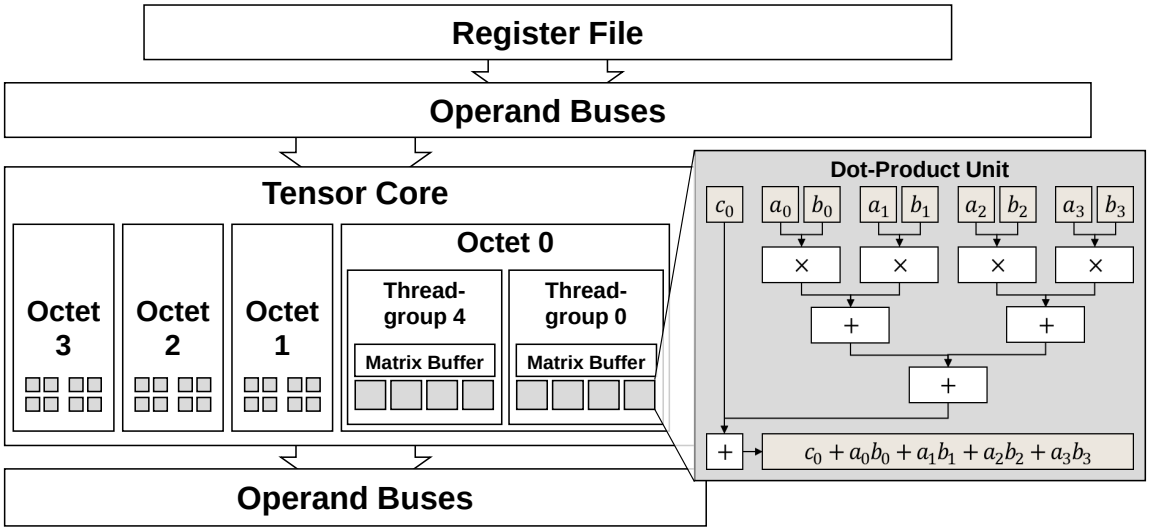


Figure 2.1: The baseline Tensor Core architecture

2.2 Target tensor processor and existing methods of extending precision

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.

2.2.1 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 [83, 140, 169]; Figure 2.1 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 operations in 8×4×8 (i.e., multiplying an 8×8 matrix by a 8×4 matrix, resulting in an 8×4 matrix) by default.

Table 2.1 excerpts the peak throughput of NVIDIA A100's Tensor Cores on various data types from the datasheet [121]. Based on the datasheet, NVIDIA's programming interface, and reverse engineering from prior work [156, 185], 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

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 2.1: A100 HMMA peak throughput

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.

2.2.2 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 2.5.1 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 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 [130, 144].

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.

2.2.3 Alternatives

Prior software-based alternatives have tried supporting the demand for matrix multiplications on FP32 [39,102,105,127,130] and FP32C [37] numbers, but all have limitations. Some MXU architectures also try to accommodate lower-precision matrix multiplications with more-precise hardware [8,120,121]. 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.

Software-based Alternatives

Despite the advantage of zero additional hardware costs, existing software alternatives [39, 102, 105, 105, 127, 130] 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 2.2 compares existing software-based FP32 GEMM solutions on FP16 MXUs and on FP32 MXUs. The same philosophy applies to software-based FP32C imple-

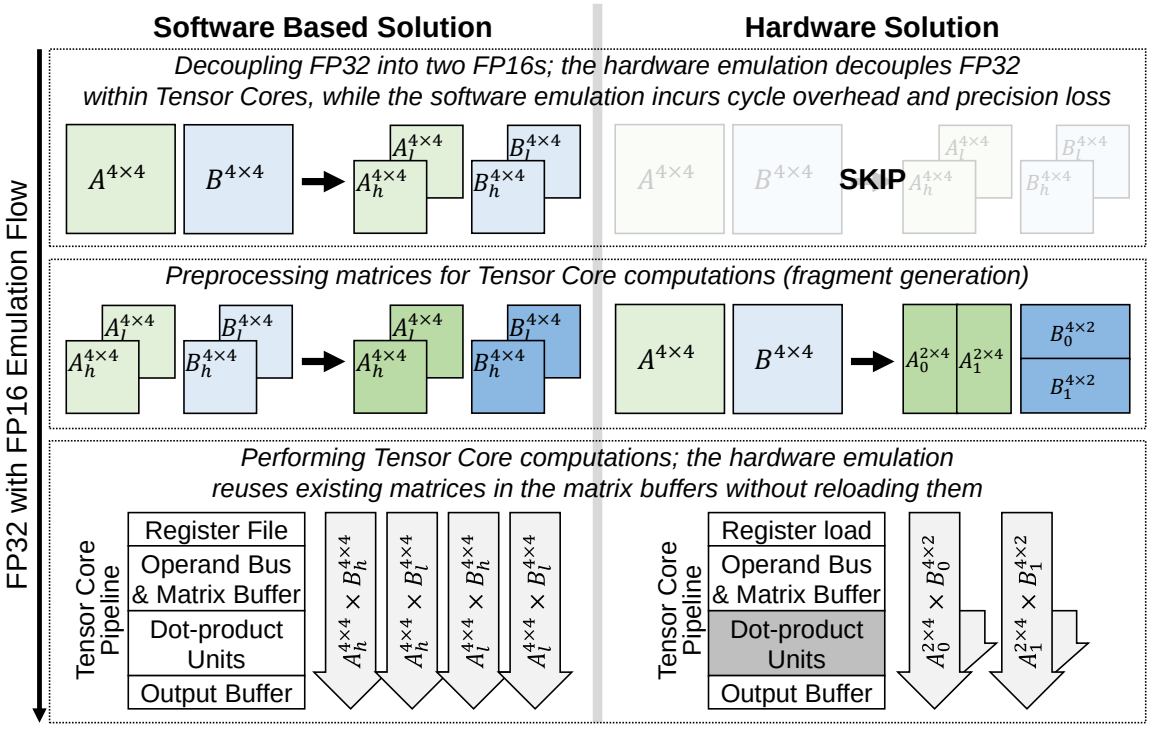


Figure 2.2: Comparison between software-based and hardware-based solutions

mentations. 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.

Hardware Solutions

Existing multi-precision hardware MXUs support FP32 MMA operations by providing logic that natively supports the highest precisions in the design [8]. 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 [120, 121]. 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.

2.3 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.

2.3.1 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 2.1 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.

$$\begin{aligned} &\forall a_{i,j} \in A, b_{i,j} \in B, c_{i,j} \in C, d_{i,j} \in D, \\ &d_{i,j} = \sum_{k=0}^{K} a_{i,k} b_{k,j} + c_{i,j} \end{aligned} \tag{2.1}$$

If we expand Equation 2.1 by separating the summation between the cases where k is odd or even, we get Equation 2.2.

$$d_{i,j} = \sum_{k=0}^{\frac{K}{2}} a_{i,2\times k} b_{2\times k,j} + \sum_{k=0}^{\frac{K}{2}} a_{i,2\times k+1} b_{2\times k+1,j} + c_{i,j} \tag{2.2}$$

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 2.3 summarizes the GEMM calculation of $D' = A' \cdot B' + C'$.

$$
\begin{aligned}
D' &= A' \cdot B' + C' \\
&= (A'_H \cdot 2^p + A'_L) \cdot (B'_H \cdot 2^p + B'_L) + C' \\
&= A'_H \cdot B'_H \cdot 2^{2p} + (A'_H \cdot B'_L + A'_L \cdot B'_H) \cdot 2^p + A'_L \cdot B'_L + C' \qquad (2.3)
\end{aligned}
$$

Again, let us create a $M \times \frac{K}{2}$ matrix A" and $\frac{K}{2} \times N$ matrix B" using the following equation.

$$
\forall a''_{i,j} \in A'', a'_{H\,i,j} \in A'_H, a'_{L i,j} \in A'_L, \begin{cases} a''_{i,2\times j} & = a'_{H i,j} \\ a''_{i,2\times j+1} & = a'_{L i,j} \end{cases} \qquad (2.4)
$$

$$
\forall b''_{i,j} \in B'', b'_{H\,i,j} \in B'_H, b'_{L i,j} \in B'_L, \begin{cases} b''_{2\times i,j} & = b'_{H i,j} \\ b''_{2\times i+1,j} & = b'_{L i,j} \end{cases} \qquad (2.5)
$$

If we perform matrix multiplication as $D'_H = A'' \cdot B''$ and apply a similar decomposition as in Equation 2.2, we can derive Equation 2.6 as below.

$$
\begin{aligned}
&\forall a''_{i,j} \in A'', b''_{i,j} \in B'', d'_{H\,i,j} \in D'_H, \\
d'_{H\,i,j} &= \sum_{k=0}^{\frac{K}{2}} a''_{i,k} b''_{k,j} \\
&= \sum_{k=0}^{\frac{K}{4}} a''_{i,2\times k} b''_{2\times k,j} + \sum_{k=0}^{\frac{K}{4}} a''_{i,2\times k+1} b''_{2\times k+1,j} \\
&= A'_H \cdot B'_H + A'_L \cdot B'_L \qquad (2.6)
\end{aligned}
$$

Equation 2.6 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 $\frac{K}{2} \times N$ matrix B"' using the following equation,

$$\forall b'''_{i,j} \in B''', b'_{Hi,j} \in B'_H, b'_{Li,j} \in B'_L, \begin{cases} b'''_{2\times i,j} = b'_{Li,j} \\ b'''_{2\times i+1,j} = b'_{Hi,j} \end{cases} \tag{2.7}$$

and perform $D'_L = A'' \cdot B'''$, we can derive Equation 2.8 as:

$$\begin{aligned} &\forall a''_{i,j} \in A'', b'''_{i,j} \in B''', d'_{Li,j} \in D'_L, \\ &d'_{Li,j} = \sum_{k=0}^{\frac{K}{2}} a''_{i,k} b'''_{k,j} \\ &\quad = \sum_{k=0}^{\frac{K}{4}} a''_{i,2\times k} b'''_{2\times k,j} + \sum_{k=0}^{\frac{K}{4}} a''_{i,2\times k+1} b'''_{2\times k+1,j} \\ &\quad = A'_H \cdot B'_L + A'_L \cdot B'_H \end{aligned} \tag{2.8}$$

Equation 2.8 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 2.6 and Equation 2.8:

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.

2.3.2 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 $B'_C = B'_{CR} + B'_{CI}i$.

$$
\begin{aligned}
D'_C &= A'_C \cdot B'_C + C'_C \\
&= (A'_{CR} + A'_{CI}i) \cdot (B'_{CR} + B'_{CI}i) + C'_C \\
&= (A'_{CR} \cdot B'_{CR} - A'_{CI} \cdot B'_{CI}) + \\
&\quad (A'_{CR} \cdot B'_{CI} + A'_{CI} \cdot B'_{CR})i + C'_C
\end{aligned} \tag{2.9}
$$

Equation 2.9 expands $D'_C = A'_C \cdot B'_C + C'_C$ with the split $A_C{}'$ and $B_C{}'$. This is almost identical to Equation 2.3, except for the subtraction. If we repeat the processing as Equation 2.4 – Equation 2.8, 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 2.9, 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 and Observation 3 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.

2.3.3 Performance Expectation on Modern Hardware

This section estimates the performance gain on modern hardware using the observations and corollaries from Sections 2.3.1 and 2.3.2 to derive this work's advantage.

Referencing the white papers from NVIDIA's Tensor Core Architectures (our baseline hardware architecture) [121,122], 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.

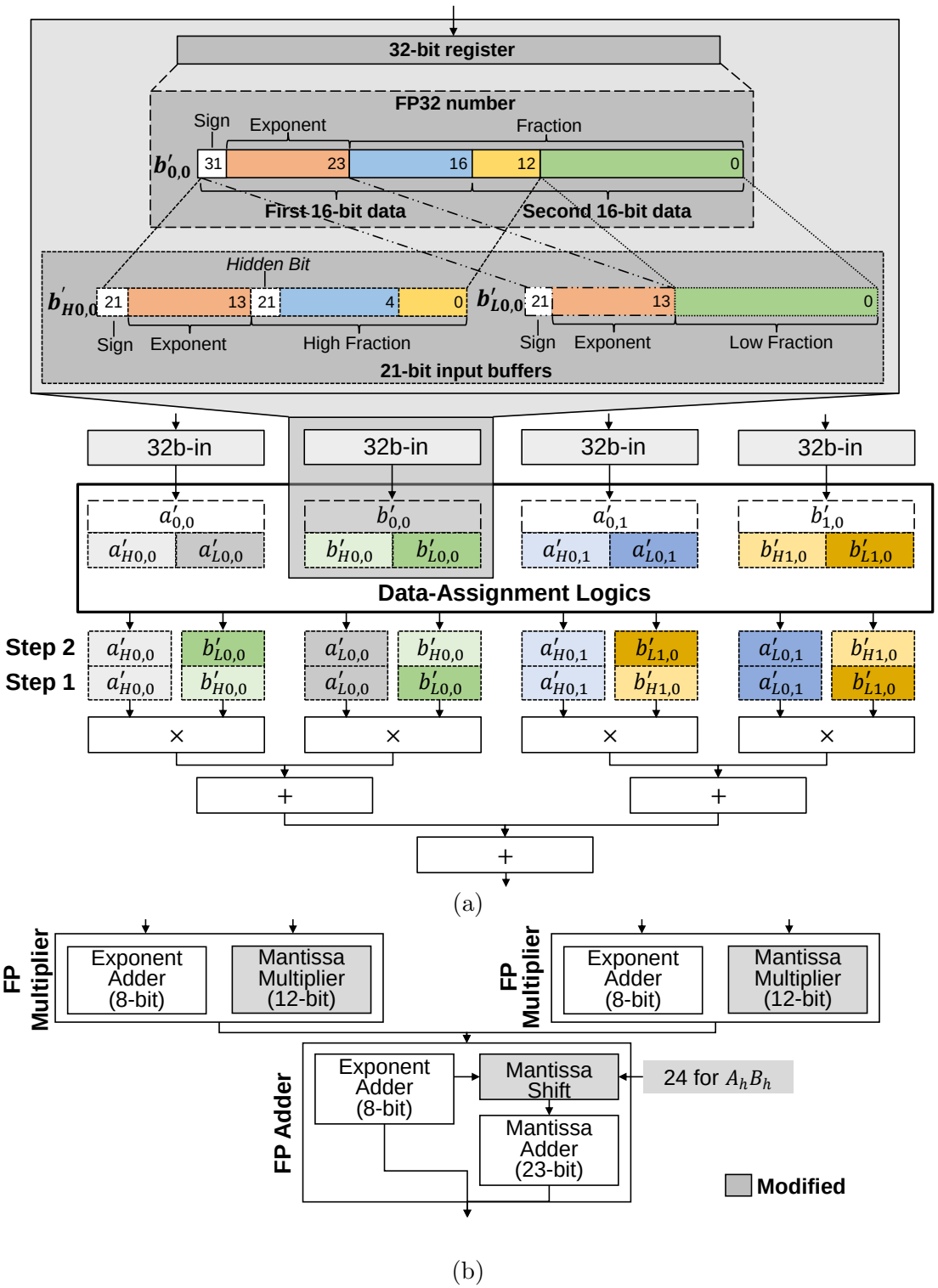


Figure 2.3: The high-level design of the data-assignment stage. (a) Data-assignment stage for FP32 (b) Dot-product unit hardware modifications for FP32

2.4 M^{3}XU Microarchitecture

Leveraging the insights from Section 2.3, M^{3}XU is built 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.

2.4.1 Extending MXUs for FP32

Summarizing Observations 1 and 2 in Section 2.3, 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.

M^{3}XU controls the dataflow of each step of an operation via multiplexers and buffers that store the inputs of each step. Figure 2.3 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 2.3(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 2.3(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 2.3 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 2.3(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 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 2.3(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.

2.4.2 Extending MXUs for FP32C

Using Observation 3 in Section 2.3 and combining the earlier-described FP32 extensions for M^{3}XU, M^{3}XU can additionally support complex number arithmetic and act

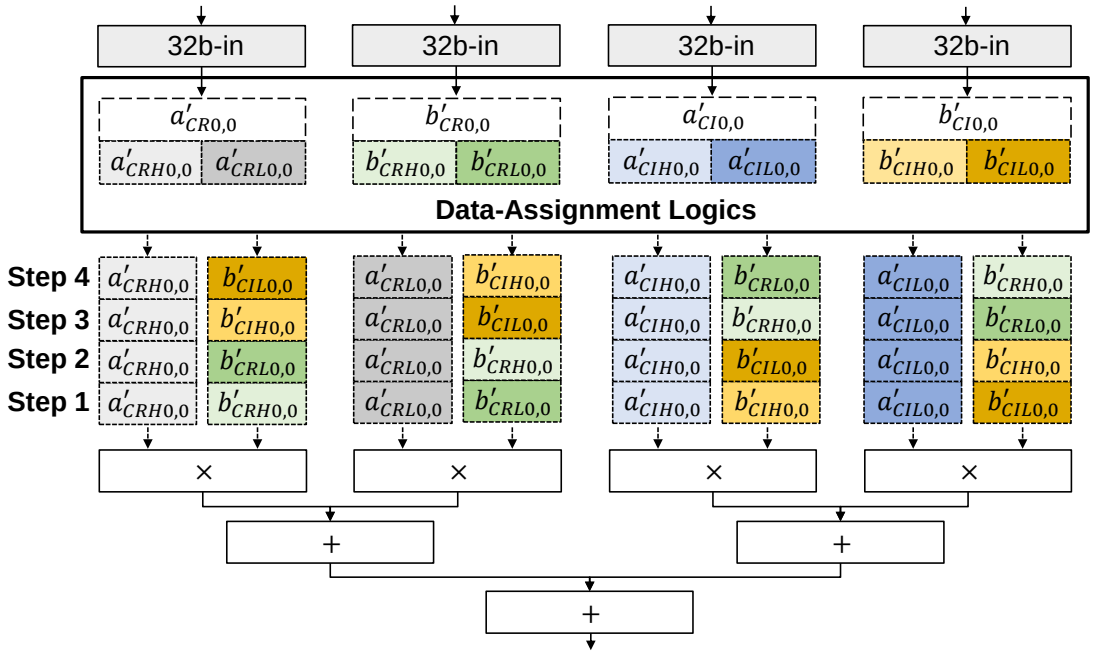


Figure 2.4: Data-assignment stage for FP32C

as an accelerator for FP32C. In addition to the modifications in Section 2.4.1, 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 2.4 depicts the extension to FP32-M^{3}XU for FP32C.

Equation 2.9 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 2.4(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

	OPcode	Output	Input	Shape
Exising Tensor Core Instructions	`HMMA.16816.F32`	FP32	FP16	16×8×16
	`HMMA.16816.F16`	FP16	FP16	16×8×16
	`HMMA.16816.F32.BF16`	FP32	BFP16	16×8×16
	`HMMA.1684.F32.TF32`	FP32	TF32	16×8×4
M^3XU extension	`HMMA.16416.F32.FP32`	FP32	FP32	16×8×8
	`HMMA.16216.F32C.FP32C`	FP32C	FP32C	16×8×4

Table 2.2: M^3XU MMA instructions and existing Tensor Cores Instructions. In existing Tensor Core Instructions, NVIDIA uses F32 for FP32.

two adjacent multipliers. The 3rd and 4th steps are interchangeable. Our implementation eliminates the overhead of reassigning inputs.

2.4.3 Instruction Set Architecture for M^3XU

M^3XU provides an instruction set architecture that extends NVIDIA's tensor cores (our baseline in this paper) to support FP32 and FP32C computations. However, M^3XU's ISA and architectural design can apply to other MXU architectures. Table 2.2 shows the opcodes for Tensor Cores' half-precision matrix multiplications in NVIDIA Ampere GPUs. Each opcode indicates the operation, output data type, input data type, and matrix multiplication shape. Shape is specified as $M \times N \times K$. As dot-product units in Ampere's Tensor Cores only support 11-bit mantissas, even though some instructions produce FP32 outputs, the computations are not true FP32 and have different results than computing on FP32 inputs using SIMD cores or a CPU.

We use the existing opcode naming scheme and add two opcodes, `HMMA.16416.F32.FP32` and `HMMA.16216.F32C.FP32C`. Since M^3XU does not widen the data path to the

Tensor Cores, it halves and quarters the K of the matrix multiplication shape, for FP32 and FP32C operations, respectively. These two instructions produce the exact same result as one gets from using SIMD cores to perform FP32/FP32C arithmetic.

2.5 Experimental Methodology

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

2.5.1 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 [121] and Accel-Sim [81] as it can perform $8 \times 8 \times 4$ matrix multiplications on FP16/BF16 input elements and accumulates results in FP32.

2.5.2 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 [39, 188]. 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 [39, 102, 127], 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.

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, thus, one 16×8×16 matrix multiplication requires 2 invocations of M^{3}XU FP32 MMA instructions. Compared to existing Tensor Core MMA instruction, which computes one FP16 16×8×16 matrix multiplication , 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.

Constructing performance emulation kernels

Our framework utilizes CUTLASS [127] to efficiently implement hierarchical blocked GEMM kernels. To assure section 2.5.2 (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 2.5.2 (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 2.3 lists all four GEMM kerenls used in our evaluation.

Name	Description
`M3XU_sgemm_pipelined`	FP32 GEMM Kernel by invoking 1 more MMA instruction, and 2× problem shape
`M3XU_sgemm`	FP32 GEMM Kernel with controlled clock frequency
`M3XU_cgemm_pipelined`	FP32 Complex GEMM Kernel by invoking 3 more MMA instructions, and 4× problem shape
`M3XU_cgemm`	FP32 Complex GEMM Kernel with controlled clock frequency

Table 2.3: M^{3}XU GEMM Kernels provided by performance emulation framework

2.5.3 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.

2.6 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

	Baseline MXUs		M^3XU		
	FP16	FP32 w/o FP32C	M^3XU w/o FP32C	M^3XU	M^3XU 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 2.4: The relative overhead of various M^3XU implementations, compared with the three reference designs, the baseline FP16 MXU and two naively extended FP32-MXU with half/same amount of inputs

can directly perform FFT calculations without approximations and achieves up to 1.99 × compared with state-of-the-art cuFFT libraries.

2.6.1 Hardware synthesis result

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

Table 2.4 summarizes the synthesis results. Adding the proposed FP32 MMA support in M^3XU 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^3XU is only 16%.

The complete M^3XU supports both FP32 and FP32C and incurs 4% more area overhead than just supporting FP32. However, M^3XU will result in a 21% increase in cycle time if we do not pipeline the data assignment stage. Despite the slowdown in supporting

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 [102] 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 2.5: Baseline and prior GEMM Kernels

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 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.

2.6.2 Microbenchmark

Table 2.5 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 [102] 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 2.5 (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 $8K \times 8K \times 8K$.

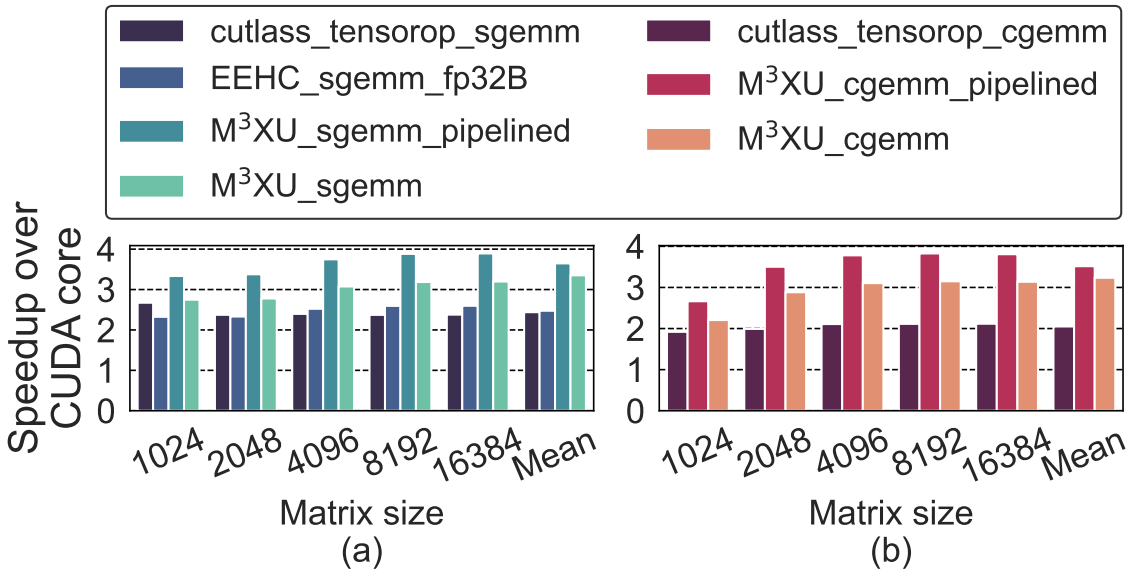


Figure 2.5: Performance comparison of GEMM using different Tensor Core approaches: (a) SGEMM, (b) CGEMM.

Figure 2.5 (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 2.6 (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 Figure 2.6) 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

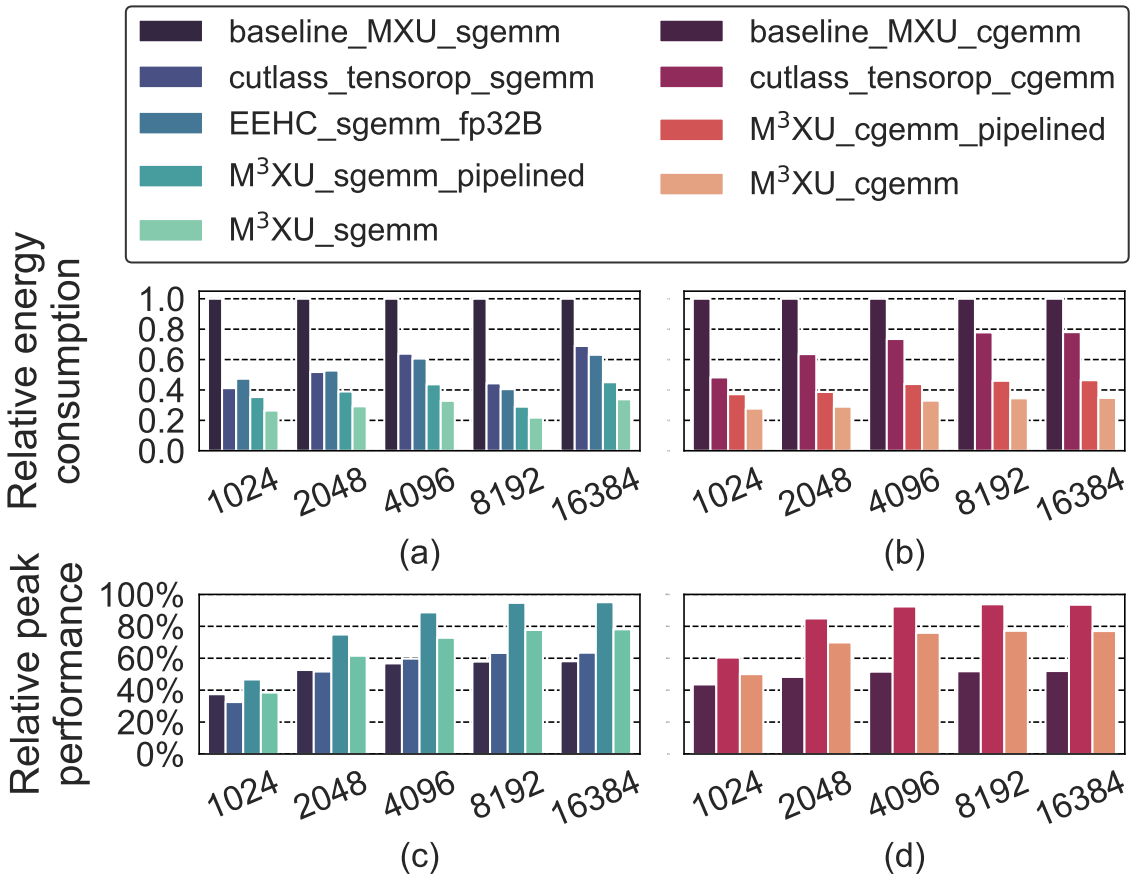


Figure 2.6: Relative analysis of M^{3}XU: (a) relative energy of SGEMM, (b) relative energy of CGEMM, (c) relative performance of SGEMM, (d) relative performance of CGEMM.

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 2.3, 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 2.6 (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.

Conv2D performance: Like our GEMM evaluation approach, Table 2.6 demonstrates seven 2D-convolution kernels, including three baseline kernels selected from CUTLASS, and four M^{3}XU kernels generated by our performance emulation framework. Our evaluation examined 12 convolution layer configurations in ResNet-50 (as Kernel 1 to Kernel 12) with a batch size of 64 for all experiments. Figure 2.7 shows that 2-D convolution using M^{3}XU can achieve 2.8× more throughput compared with baseline CUDA core kernels, which is 1.2×

Name	Description
FP32 Kernels	
`cutlass_simt_fprop`	cutlass fp32 conv2d kernel using CUDA cores
`cutlass_tensorop_fprop`	cutlass fp32 conv2d Tensor Core kernel emulated with 3xtf32
`M3XU_fprop_pipelined`	M^{3}XU fp32 conv2d Tensor Core kernel
`M3XU_fprop`	M^{3}XU fp32 conv2d Tensor Core kernel with controlled clock frequency
FP32-Complex Kernels	
`cutlass_simt_cfprop`	cutlass fp32 complex conv2d kernel using CUDA cores
`M3XU_cfprop_pipelined`	M^{3}XU fp32 complex conv2d Tensor Core kernel
`M3XU_cfprop`	M^{3}XU fp32 complex conv2d Tensor Core kernel with controlled clock frequency

Table 2.6: Baseline and M^{3}XU 2D-Convolution Kernels

better than software alternatives. Figure 2.8 shows that M^{3}XU reaches up to 2.93× more throughput compared with CUDA core kernels when executing the same set of convolution layers. Non-pipeline M^{3}XU kernel reaches up to 3.1 × and 2.5 × more throughput. Our evaluation shows the convolutional kernel size and input activation size are critical to the throughput. When dealing with a small convolutional kernel and relatively large input activation, including kernels 1,3,6 in both microbenchmarks, Tensor Core is underutilized and results in less throughput compared with other kernels.

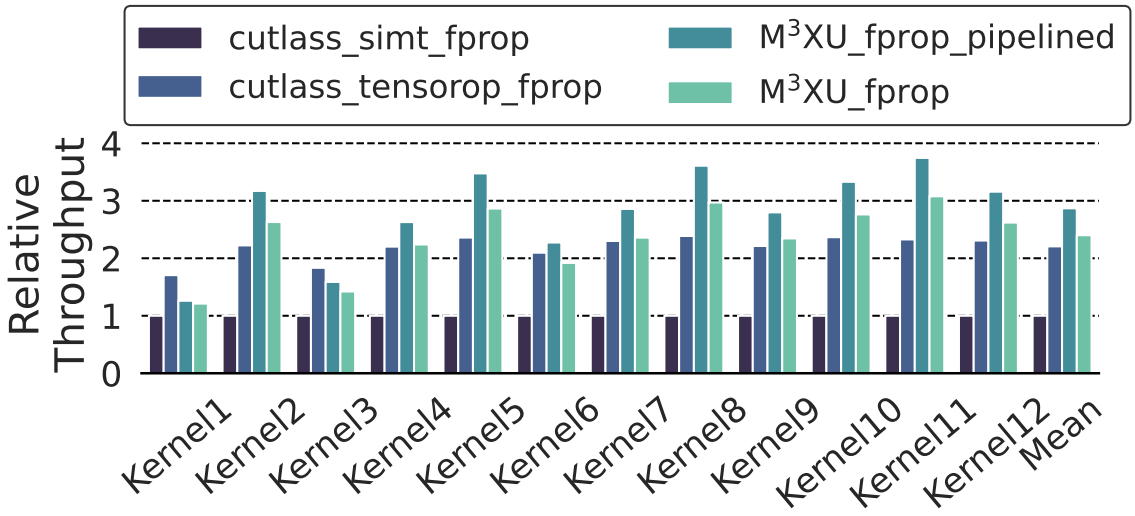


Figure 2.7: Throughput of convolution layers using FP32 Tensor Core

2.6.3 Case studies

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

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 [93, 123]. *tcFFT* [93] 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

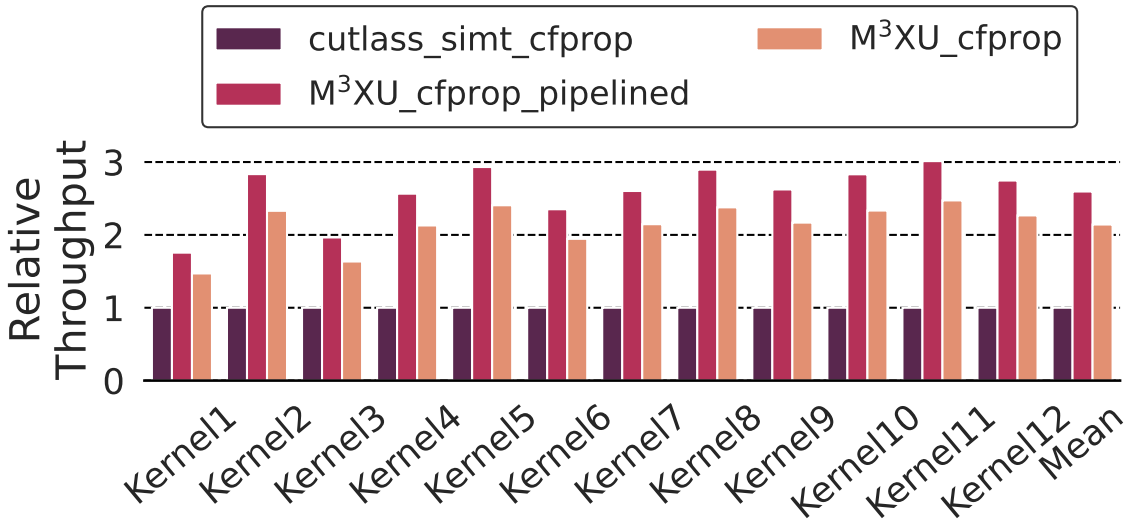


Figure 2.8: Throughput of convolution layers using FP32 complex Tensor Core

comparisons, we extended *tcFFT* to support single precision GEMM using TF32 Tensor Cores and compared the end-to-end speedup with cuFFT [123], a vendor-optimized GPU FFT library, as the baseline. Figure 2.9 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*.

DNN training

This case study evaluates the performance improvements of M^{3}XU on machine learning workloads using Nebula benchmark [82]. We extended ResNet, VGG, and AlexNet. Figure 2.10 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

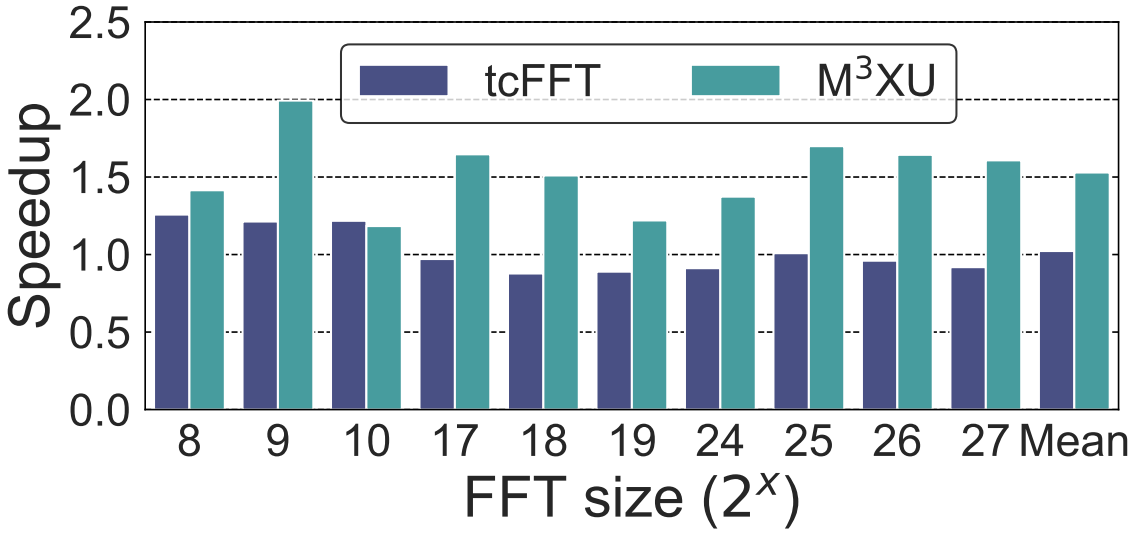


Figure 2.9: Speedup of FFT over *cuFFT*

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.

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 [167], is a state-of-the-art GPU-based MRF approach that uses complex matrix multiplication for

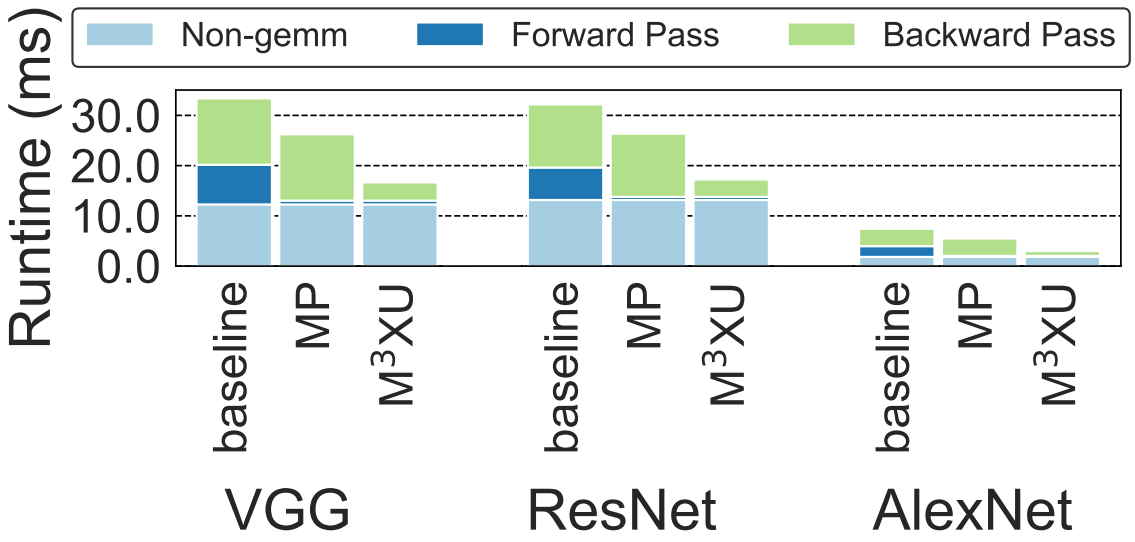


Figure 2.10: End-to-end Latency of single iteration training of CNN models

dictionary generation and pattern matching phase of MRF, and the dictionary generation phase takes 98.2% of total run time. CGEMM accounts for 22% of the runtime in the dictionary generation phase. As shown in Figure 2.11, M^{3}XU achieves up to 1.26× speedup in end-to-end latency of dictionary generation phase over the `cublas_cgemm`-based baseline.

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 [164] 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.

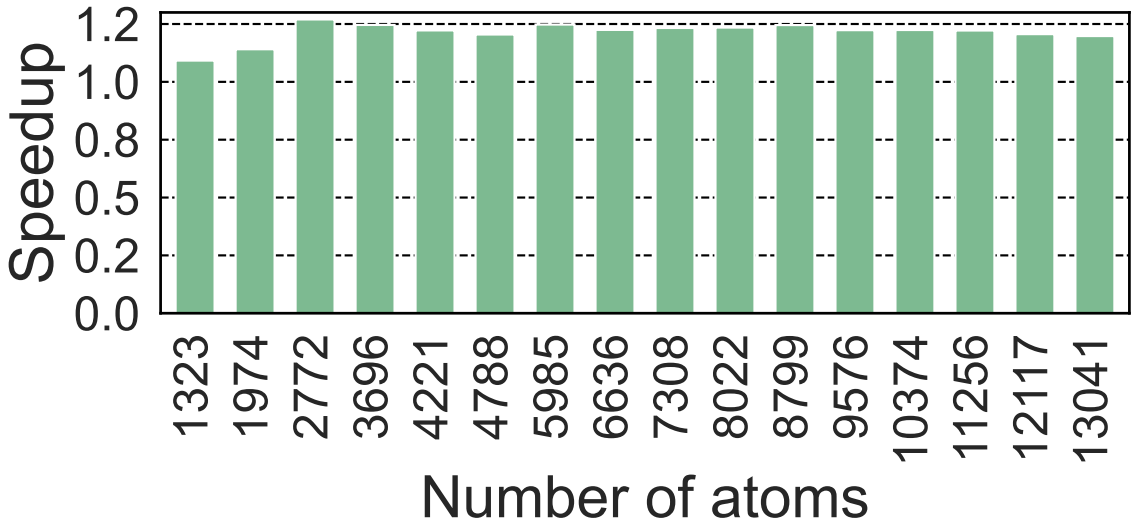


Figure 2.11: Speedup of MRF dictionary generation over CUDA cores

Figure 2.12 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.

2.6.4 FP64 Tensor Core

Ampere architecture supports double precision Tensor Core GEMM with 2× speedup [121]. Although NVIDIA provides no details regarding FP64 Tensor Core, M^{3}XU can serve as an alternative implementation of FP64 Tensor Core. Like the FP32C emulation explained in Section 2.4.2, the dot-product units reuse real and imaginary wires can receive two dou-

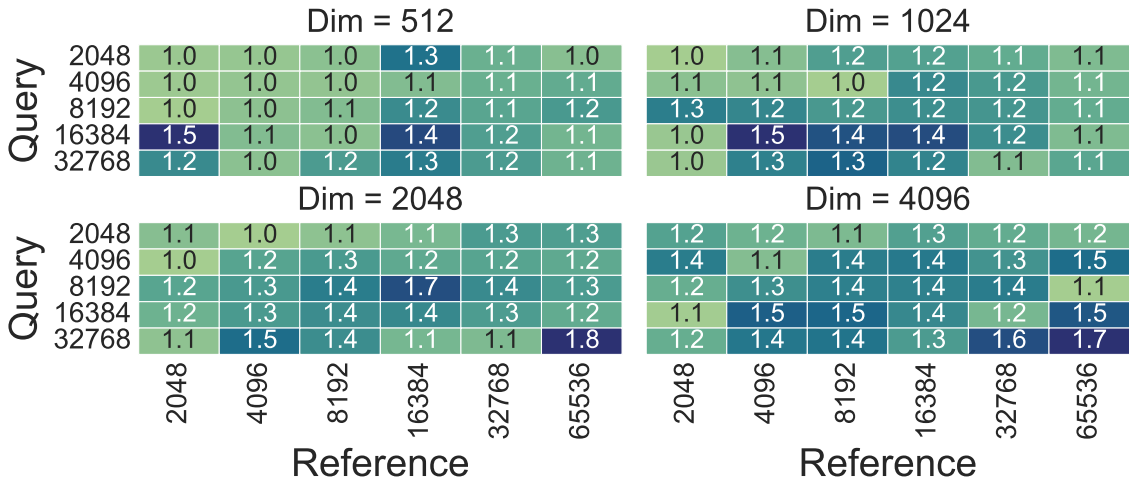


Figure 2.12: KNN speedup over CUDA cores

ble values as input. Then it decouples the two values into four parts(high-high, high-low, low-high, and low-low) and conducts four dot-product operations with the same swapping policy as FP32C. All results are accumulated from multiplier to FP64 registers similar to the FP32/FP32C mode. With minor hardware modification, we extended our performance emulation framework to evaluate the performance of M^{3}XU FP64 GEMM kernels. M^{3}XU achieved an average performance gain of 6.8× and 6.16× for pipelined and non-pipelined design, respectively, 2.9 × faster than existing Tensor Core FP64 GEMM.

2.7 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.

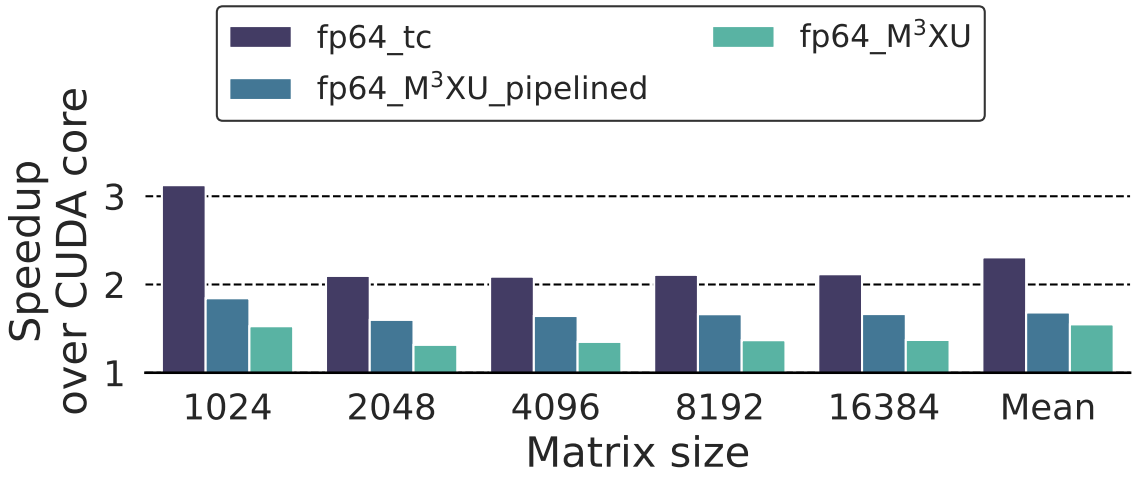


Figure 2.13: Performance of different approaches using FP64 Tensor Core GEMM

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 3.89× on SGEMM, and faithful computation on CGEMM with close to 3.8× speedup.

Chapter 3

Exploiting Operator

2-dimenional tensor/matrices are essential data structures at the core of scientific computing, data and graph analytics as well as artificial intelligence (AI) and machine learning (ML) workloads. Due to the stagnating general-purpose processor performance scaling and memory-wall problem [177], a recent trend of efficient computing on matrices focuses on building hardware accelerators. Famous examples include NVIDIA's Tensor Cores [121, 124], Google's Tensor Processing Units (TPUs) [74], and the recent IBM Power 10 MMA unit [159]. The demand of matrix-multiplication accelerators is so strong that the upcoming generations of Intel and ARM CPU processors will also provide matrix extensions and integrate MXUs [12, 69].

3.1 Overview of SIMD^2

Compared with conventional SIMD processors (e.g., GPGPUs), MXUs are more efficient in general matrix multiplication (GEMM) for two reasons. First, GEMMs are easy

to parallelize. Each MXU can take tiles from input matrices and generate an output tile, and multiple MXUs can work together to form a larger GEMM accelerator, temporally or spatially. Second, GEMMs have a higher compute intensity than vector operations (e.g., saxpy [1]). Such compute intensity alleviates the memory-wall issue in modern throughput-oriented SIMD processors and allows architects to simply add more compute throughput to scale the performance of MXU with the same on-chip and off-chip bandwidth limitation.

Besides GEMM, a wide-spectrum of problems, including all-pair-shortest-path, minimum spanning tree as well as graph problems, have matrix-based algorithms/solutions share the same computation pattern. They all follow a *semiring-like structure* – $A \oplus (B \otimes C)$, where the problem generates results (or intermediate results) by performing two-step operations ($\oplus$ and $\otimes$) on three matrix inputs (A, B and C). For example, dynamic programming methods for all-pair-shortest-path problems using All Pairs Bellman-Ford or Floyd-Warshall algorithms can be expressed in a semiring-like structure through having the $\otimes$ operator represent the addition-based distance update operations [110, 146], and the minimum operation replaces $\oplus$ operator.

However, as modern MXUs are highly specialized for just GEMM or convolutions, programmers must perform non-trivial algorithm optimizations (e.g., mapping matrix multiplications to convolutions [63, 98]) to tailor these applications for supported matrix operations. Besides, the resulting program may still under-utilize MXUs as mapping the original set of matrix operations to GEMMs that require changing the dataflow or data layout of the program before the actual computation can start. Finally, for problems including the dynamic programming algorithms, existing MXUs cannot provide native support for the

required $\oplus$ and $\otimes$ operations and have to fallback to SIMD processors (e.g., CUDA cores), even though these algorithms share the semiring-like structure with GEMM.

To address these issues, this paper presents the SIMD^2 architecture to enable more efficient matrix operations for a broader set of applications. SIMD^2 provides a *wider* set of matrix-based operations that naturally fit the application demands and abstract these functions through an appropriate set of instructions. SIMD^2 reuses and extends the function of existing MXUs and data paths to minimize the overhead in supporting additional matrix operations.

The SIMD^2 architecture brings the following benefits in accelerating matrix applications. First, programmers or compilers can leverage the richer set of instructions that naturally maps to common matrix operations without sophisticated code transformations, which facilitates matrix-based programming. By performing more matrix operations with a minimum number of instructions, the SIMD^2 instructions further reduce the control and data movement overhead over conventional SIMD instructions by exposing a matrix-based abstraction.

As an initial step in this direction, our SIMD^2 architecture introduces eight more types of instructions for matrix computation, including (1) min-plus, (2) max-plus, (3) min-mul, (4) max-mul, (5) min-max, (6) max-min, (7) or-and, and (8) plus-norm, in addition to existing mul-plus instructions. Similar to existing hardware-accelerated GEMM operations, these instructions also take tiles of matrices as inputs and update the resulting output tile. Therefore, these instructions can easily share the same infrastructure of an existing MXU, including instruction front-end, memory, and register files. As these SIMD^2 instructions

all follow the same data flow and computation pattern, they can also share the operand delivery structure and simply require a modified data path to perform new operations.

As the necessary hardware support of SIMD^2 resembles existing MXUs, a SIMD^2 architecture can be implemented on top of any matrix-multiplication accelerators, either in standalone application-specific integrated circuit (ASICs) or as processing elements in CPUs or GPUs. This paper presents SIMD^2 in the form of extending GPU architectures as this allows us to leverage existing interface/front-end of GPU programming models and mature software stacks, and focus on the benefits of the SIMD^2 model. On the other hand, since modern matrix-based applications still rely on non-matrix operations to complete all computation tasks, this architecture also offers better performance by avoiding data movements across system interconnects and taking advantage of existing high-bandwidth memory hierarchy in GPUs.

Table 3.1: Exemplary problems with their mappings to semiring-like structures and the corresponding definitions of operators to their solutions.

Type of matrix operations	1st OP $\oplus$	2nd OP $\otimes$	Representative Algorithm(s)
Plus-Multiply	$+$	$\times$	Matrix Multiplications, Matrix Inverse
Min-Plus	min	$+$	All-pairs shortest paths problem
Max-Plus	max	$+$	Maximum cost (critical path)
Min-Multiply	min	$\times$	Minimum reliability paths
Max-Multiply	max	$\times$	Maximum reliability paths
Min-Max	min	max	Minimum spanning tree
Max-Min	max	min	Maximum capacity paths
Or-And	or	and	Transitive and reflexive closure
Add-Norm	$+$	$\|a-b\|^2$	L2 Distance

3.2 The Case for SIMD^2

The motivation of proposing SIMD^2 for matrix and tensor problems comes from two sources– A family of matrix algorithms that share the same semiring pattern in computation, and the emergence of GEMM accelerators designed around the semiring pattern. Both motivate the need and the possibility of a single umbrella that covers a large set of matrix algorithms to facilitate efficient use of hardware components.

3.2.1 The Commonality among Matrix Problems

Matrices provide a natural mathematical expression for linear systems, graphs, geometric transformations, biological datasets, and so on. In addition to data representations, many applications using matrices as inputs and outputs also share the same algebraic structure in their algorithms. This algebraic structure contains two binary operators, $\oplus$ and

```
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        for(k = 0; k < N; k++) {
            D[i][j] =  C[i][j] + A[i][k] * B[k][j];
        }
    }
}
```

(a)

```
for (src = 0; src < N; src++) {
    for (dst = 0; dst < N; dst++) {
        for(k = 0; k < N; k++) {
            D[src][dst] =  min(C[src][dst], (C[src][k] + A[k][dst]));
        }
    }
}
```

(b)

Figure 3.1: Code snippet of (a) GEMM and (b) APSP.

$\otimes$. The $\oplus$ operator satisfies properties analogous to addition. The $\otimes$ operator is associative and typically has a multiplicative identity element analogous to multiplication. In other words, a large set of matrix algorithms can be formalized as:

$$D = C \oplus (A \otimes B)$$

where A, B, C are input matrices, D is the output, as well as the two customized operators, $\oplus$ and $\otimes$.

The above algebraic structure is similar to a semiring, $(R, \oplus, \otimes)$, which contains a set R equipped with two binary operators, $\oplus$ and $\otimes$. The $\oplus$ operator in a semiring satisfies properties analogous to addition. The $\otimes$ operator in a semiring has more restrictions as it must be associative, distributive as well as having a multiplicative identity element. Since some algebraic structure of matrix problems is similar, but not mathematically identical to semirings, we use the term *semiring-like structure* when referring to this identified algebraic structure.

General matrix multiplication (i.e., GEMM) is one classic example that follows this structure. To simplify the discussion, we use square matrices in the following examples. Let A be an N by N matrix and $a(i, j)$ represent the $(i,\ j)$-entry of A. Then, there also exists two other n by n matrice, B and C, where $b(i, j)$ and $c(i, j)$ represent the $(i,\ j)$-entries of B and C, respectively. General matrix multiplication consists of a set of computation for the $(i,\ j)$-entry of the resulting matrix D, $d(i, j)$, where $d(i, j) = c(i, j) + \sum_{k=0}^{N} a(i, k) \times b(k, j)$. Figure 3.1(a) illustrates the code example for matrix multiplication with $N \times N$ matrices. The matrix multiplication therefore has a semiring-like structure where the $\oplus$ operates as pair-wise addition for each pair of elements sharing the same coordinate i, j on each side of the operator, matrix C and the result of $A \otimes B$. The $\otimes$ operates as calculating the value of the $(i,\ j)$-entry in the result matrix D as $\sum_{k=1}^{n} a(i, k) \times b(k, j)$ for each i, j, k. With the aforementioned common form, a matrix multiplication problem is $D = C + A \times B$.

Besides matrix multiplications, a wide spectrum of algorithms, especially those for solving graph problems or algorithms that leverage dynamic programming, can also be formulated as a structure similar to matrix multiplications by customizing the $\oplus$ and $\otimes$

operators. For example, Figure 3.1(b) shows how the inner loops of all-piars Bellman-Ford algorithm [38] for all-pairs-shortest-path (APSP) problem is similar to the semiring-like algebraic structure as GEMM (Figure 3.1(a)). Each iteration in Line 4–5 of Figure 3.1(b) performs the computation of $d(i,j) = min\{c(i,j), min_{k=0}^{N}[c(i,k) + a(k,j)]\}$, where each $d(i,j)$, $c(i,j)$, or $a(i,j)$ represents the $(i,\ j)$-entry of matrix D, C or A, respectively. The D matrix is the result of temporal all-pairs distances after the iteration, C is the result from the last iteration, and A is the original adjacency matrix. Therefore, we can leverage the semiring-like structure to express the all-pairs Bellman-Ford algorithm for the APSP problem by replacing the $\oplus$ operator with min and the $\otimes$ operator with $+$. The core loops become $D = C\ min\ (C + A)$.

In addition to the APSP problem, there are other algorithms amenable to such a semiring-like structure. Table 3.1 illustrates a set of problems and their corresponding customizations of $\oplus$ and $\otimes$ operators in their algorithms.

Though a semiring-like structure can serve as a generic programming paradigm for matrix problems, conventional approaches in solving matrix problems require the programmers to transform matrix data into lower-ranked data representations (e.g., scalar numbers or vectors) and redesign algorithms on these data representations to fulfill the programming paradigm that modern CPUs and GPUs can support. Performance optimizations on programs solving these problems is especially challenging as they are intensive in both computation and data accesses on conventional processor architectures.

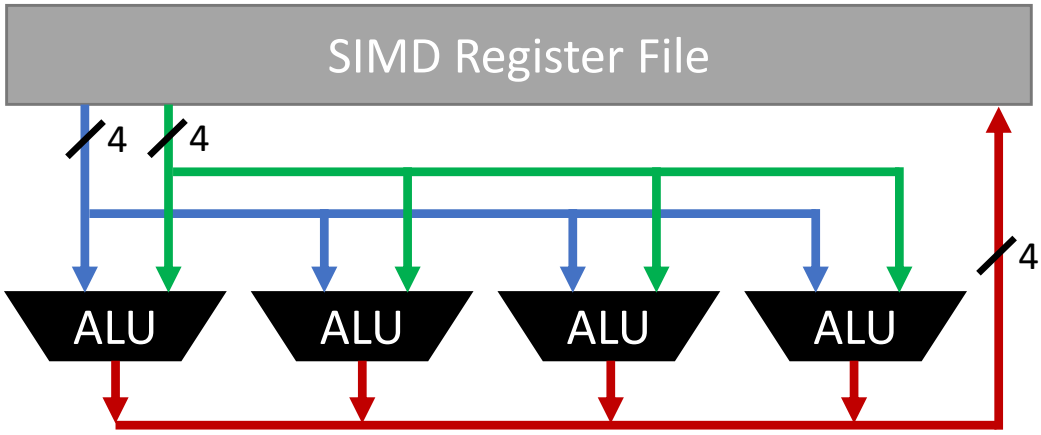


Figure 3.2: An example SIMD architecture.

3.2.2 Hardware Support for Semiring-like Structure in GEMMs Accelerators

The semiring-like algebraic structure is the key enabler behind modern tensor accelerators, like MXUs for GEMMs, which improves over conventional SIMD processors. From a hardware design point of view, conventional SIMD architectures, shown in Figure 3.2, are bottlenecked by the vector register file bandwidth. Such data transfer bottleneck (von Neumann bottleneck [165]) limits how many compute units (ALUs) can be fed by the on-chip memory. For example, a 4-wide register file can only supply to 4 ALUs at a time. Even if the degree of parallelism grows as the problem size increases, the data transfer bottleneck remains.

MXUs, instead, leverage the semiring-like algebraic structure to break such bottleneck. Figure 3.3 shows an example implementation of MXU, modeled after the matrix unit in TPUs [74]. In this MXU example, one input matrix is *broadcast* to multiple ALUs

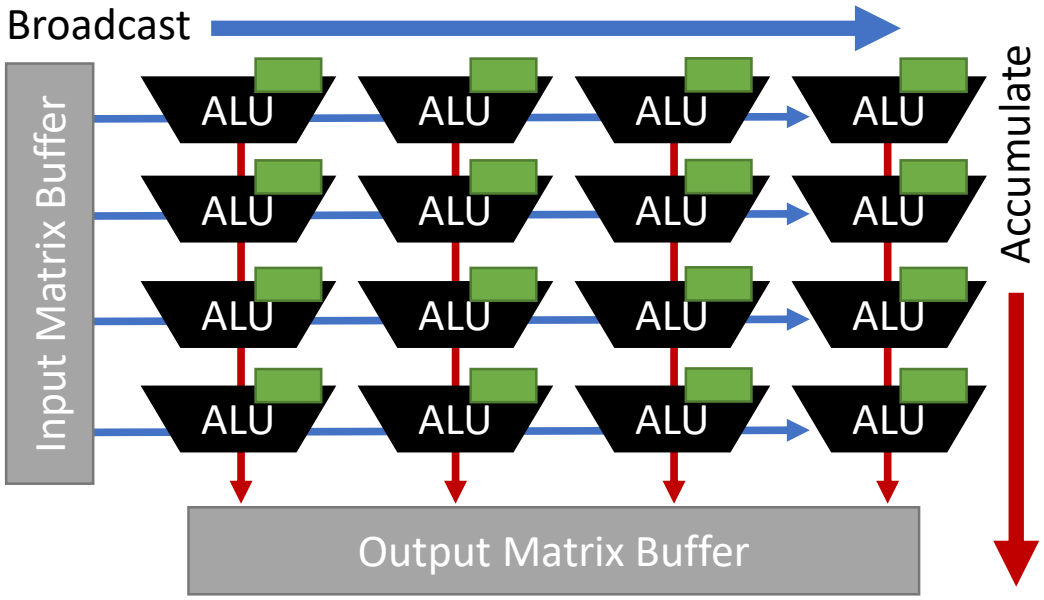


Figure 3.3: An example MXU for GEMM.

because of the intrinsic data reuse opportunities in algorithms with a semiring-like structure. The output matrix also leverages the structure (associative) and is accumulated across multiple ALUs before being stored into the output matrix buffer. With the same 4-wide memory structure, we can now supply data to 16 ALUs. More importantly, since the computation complexity is $O(N^3)$, and the data transfer is $O(N^2)$ in semiring-like algorithm, the number of ALUs can scale much more than the on-chip memory bandwidth, alleviating the memory wall issue.

As a result, modern MXUs are designed around the semiring-like structure, instead of optimizing the ALUs for multiply-add. The programming model of these MXUs also leverages the nature of the algorithm to perform work partitioning and tiling to execute a larger GEMM with multiple MXUs in a system [121] or across systems [73].

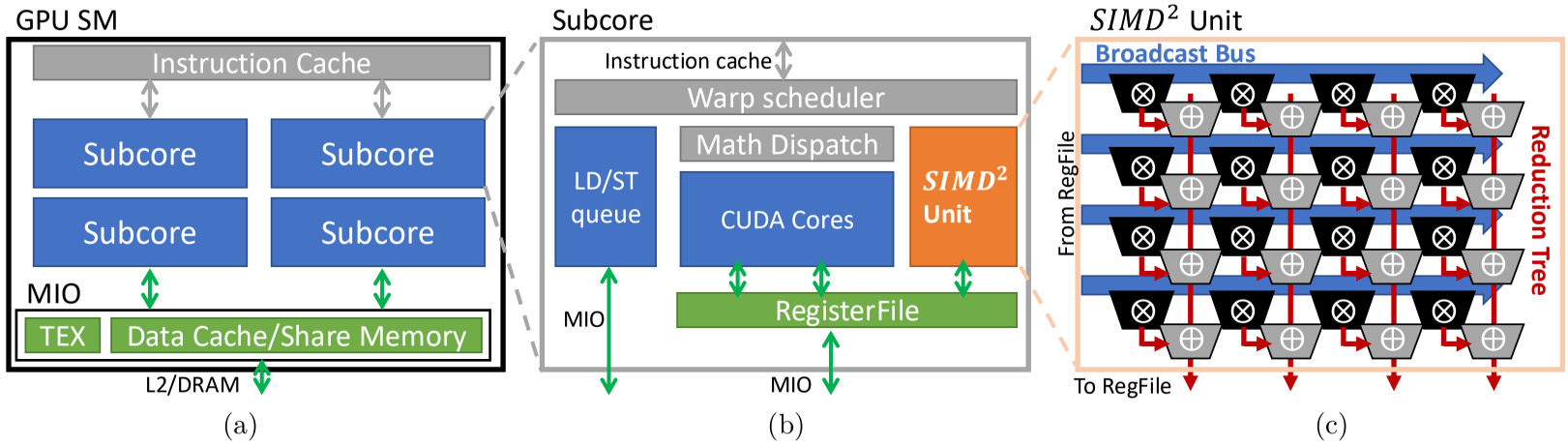


Figure 3.4: The high-level architecture of how SIMD^2 units are integrated in GPU systems and the design of an SIMD^2 unit.

For example, the *wmma* API for NVIDIA Tensor Core works at the sub-tile granularity (e.g., 16x16), and programmers can combine multiple *wmma* calls to merge sub-tile into the full problem.

Our insight is that supporting a wide range of semiring-like algorithms requires minimal changes on top of any systems with GEMM accelerators. It is clear that the ALU in Figure 3.3 is orthogonal to the hardware support (broadcast and accumulate) for a semiring-like structure. For example, if we enhance the ALU in Figure 3.3 to support $add-minimum$, then the same MXU architecture can now be used to accelerate solving APSP. That is, the recent development of MXUs for GEMM has laid the ground of supporting semiring-like algorithms, and with a better abstraction and hardware support, many more matrix algorithms can be accelerated. This motivates us to propose and design SIMD^2, a new programming paradigm and architecture for semiring-like algorithms.

3.3 SIMD^2 Architecture

We propose the SIMD^2 ISA to efficiently support matrix algorithms beyond GEMMs. SIMD^2 provides a programming paradigm and an instruction set to reflect the natural semiring-like structure in solving these matrix problems. The hardware units for SIMD^2 instructions extend existing MXU to support the proposed programming paradigm. This section will introduce both.

3.3.1 The SIMD^2 hardware architecture

Like GEMM accelerators, SIMD^2 architecture can be implemented as a standalone processor that contains SIMD^2 units only, or functional units embedded with general-purpose scalar/vector processor cores to share the same instruction front-end. In this work, we chose the latter design and prototype SIMD^2 architecture on a GPU as Figure 3.4 shows. Specifically, we build on top of the NVIDIA SM architecture [27], which integrates Tensor Core as part of the subcore in a GPU SM. The resulting high-level architecture resembles GPU SM with Tensor Cores [140] as the SIMD^2 units implementing SIMD^2 instructions are part of a streaming multiprocessor, but the rest of the architectural components (front-end, memory-subsystem, etc.) are shared with conventional GPU cores.

The SIMD^2 unit in Figure 3.4(c) extends conventional MXUs to use different $\otimes$ and $\oplus$ operators. Each SIMD^2 unit can perform an SIMD^2 arithmetic instruction using $\otimes$ operation on fixed-size matrix tiles (e.g., 4x4 in Figure 3.4(c)) and produce an output matrix by reducing the result from $\otimes$ operation with the $\oplus$ operator. Unlike tensor cores that only support multiply and accumulation, the $\otimes$ ALU supports *multiply*, *min/max*, *add/and*, and *L2 dist*, and the $\oplus$ ALU supports *add*, *min/max*, *or*, and *subtract*. Both ALUs are configured by decoding SIMD^2 instructions, as shown in Figure 3.5.

We chose to build SIMD^2 architecture on top of GPUs for the following reasons. First, since matrix operations just serve as the core computation in matrix applications, applications typically rely on scalar or vector processors to preprocess or postprocess matrix data structures. Collocating SIMD^2 units with other processing elements enables efficient and fine-grained data exchange and synchronization among heterogeneous computing units.

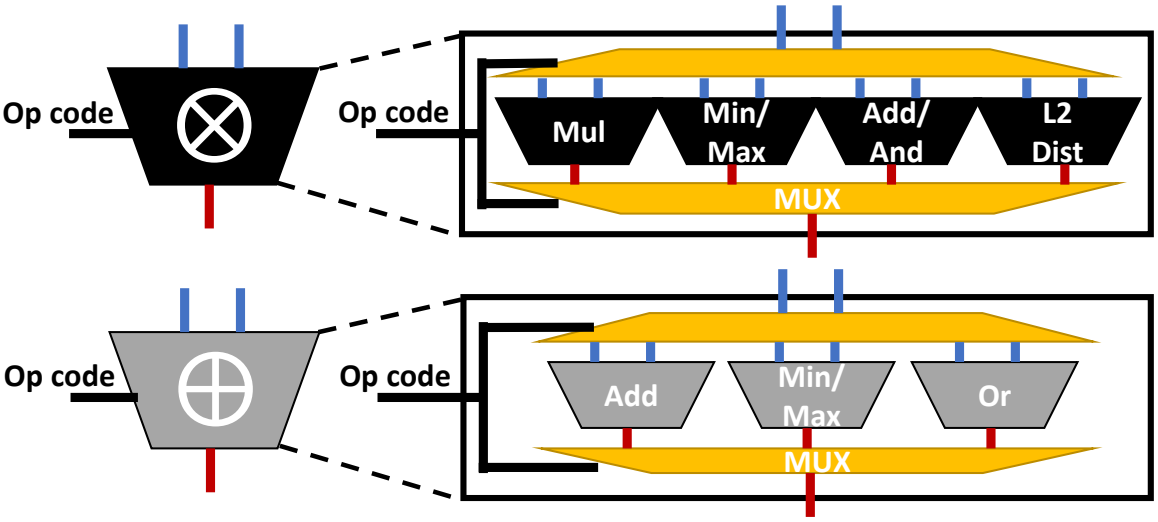


Figure 3.5: $\otimes$ ALU and $\oplus$ ALU in an SIMD^2 unit.

Second, GPU's memory architecture design is more bandwidth-oriented and serves better for the purpose since each SIMD^2 unit would consume/produce large amounts of data at once. Finally, there already exists Tensor Cores in NVIDIA's GPU architecture that allow us to leverage as a baseline design and an emulation framework.

Alternatively, we have also explored implementing the SIMD^2 unit by building a dedicated hardware unit for each semiring-like algorithm. For example, in addition to the MXU for GEMM, we can add a hardware unit for *min-add*, another unit for *add-norm*, and so on. Nonetheless, this design introduces 300% area overhead (See Section 4.6.7) to the GEMM-only MXU, which is $> 4\times$ of the overhead introduced by the combined design in Figure 3.4.

While we chose GPUs as the baseline system, building an SIMD^2 architecture on other GEMM-based accelerators, such as TPUs [74], should be straightforward and low overhead.

3.3.2 The SIMD^2 ISA

Table 3.2: A summary of the PTX instruction set architecture for SIMD^2

Data Movement Instructions	**Data Types**	**Matrix Shape**	**Source → Destination**
SIMD^2.load	fp16	16x16	Shared Memory → Register File
SIMD^2.store	fp32	16x16	Register File → Share Memory
Arithmetic Instructions	$\oplus$ **OP**	$\otimes$ **OP**	**Algorithm**
SIMD^2.mma	$+$	$\times$	GEMM
SIMD^2.minplus	min	$+$	All-pairs shortest paths problem
SIMD^2.maxplus	max	$+$	Maximum cost (critical path)
SIMD^2.minmul	min	$\times$	Minimum reliability paths
SIMD^2.maxmul	max	$\times$	Maximum reliability paths
SIMD^2.minmax	min	max	Minimum spanning tree
SIMD^2.maxmin	max	min	Maximum capacity paths
SIMD^2.orand	or	and	Transitive and reflexive closure
SIMD^2.addnorm	$+$	$\|a-b\|^2$	L2 Distance

The SIMD^2 instruction extension builds on top of the warp-level matrix-multiply-accumulate (`wmma` [125]) instructions for GPUs and extends it to support new arithmetic instructions. Table 3.2 lists these SIMD^2 instructions.

The `load` instruction moves a chunk of data from the 1D shared memory address space as a fixed-size (16x16) matrix to the per-thread register file. Like the `wmma` abstraction, each thread in the warp stores part of the matrix in the register file and contributes to the whole warp-level operation. The `store` instruction instead moves the matrix segments in the register file back to the 1D shared memory address space.

In our implementation, we assume input operands are always in 16-bit, half-precision floating-point format (fp16), while the output data is always in 32-bit, single-precision floating point format (fp32). While supporting other formats (e.g., `int8`) is possible, for many algorithms, we find fixed-precision format cannot converge to the same result as baseline fp32 implementations without SIMD^2 instructions.

For the arithmetic operations, we introduced eight more $\oplus$-$\otimes$ ops, in addition to the classic matrix-multiply-accumulate (`mma`). These nine instructions map to the frequently used matrix problem patterns in Table 3.1. The SIMD^2 arithmetic instruction shares the same register file as the vector processor, and uses arguments that specify register locations of input and output matrices. The latency of each SIMD^2 instructions depends on the actual hardware implementation of the SIMD^2 unit, and in our implementation, we provision the SIMD^2 unit to be the same throughput as the conventional MXUs so that all SIMD^2 arithmetic instructions have the same latency.

Similar to our changes for hardware architecture, we expect adding the SIMD^2 instructions to other ISAs that already support GEMMs, such as Intel AMX [69], to be straightforward. These matrix extensions already support matrices as input or output operands and provide data movement instructions for matrices (load/store matrix). SIMD^2 simply adds more arithmetic instruction on top of them. We align our SIMD^2 design point with modern GPU architectures to facilitate our evaluation, but this is not fundamental.

3.4 Programming Model

Table 3.3: Sample Low-level Matrix Operations

Sample Low-level Synopsis	**Description**
`simd2::matrix<matrix_type, m, n, k, data_type>`	Declaration function, declare the matrix will be applied in the mxnxk matrix-matrix operation.
`simd2::fillmatrix(simd2::matrix, value)`	Fill the target matrix with given value.
`simd2::loadmatrix(simd2::matrix, source, ld)`	Load value from source memory location to the target matrix, load with the step of leading dimension.
`simd2::mmo(simd2::matrix, simd2::matrix, simd2::matrix, simd2::matrix, simd2::opcode)`	Performs the matrix-matrix operation with given opcode.
`simd2::storematrix(target, simd2::matrix, ld)`	Store value to source memory location from the target matrix, store with the step of leading dimension.

The SIMD^2 units in our proposed architecture can perform matrix operations on a set of predefined matrix shapes and data types. Therefore, the native programming interface reflects the abstraction by which these SIMD^2 units expose through the SIMD^2 ISA. To further facilitate programming at the application level, the framework can provide higher-level library functions that decouple the programmability from architecture-dependent parameters.

Table 3.3 summarizes the available functions from SIMD^2's low-level programming interface. Each of these functions maps directly to a set of instructions that Section 3.3.2 describes. The exemplary programming interface resembles the C++ warp matrix operations that NVIDIA's Tensor Cores use to smooth the learning curve, but not a restriction from the SIMD^2 architecture.

Since the low-level interface reflects the architecture of SIMD^2 units, these functions must operate on a set of matrix shapes and data types that the underlying SIMD^2 hardware natively supports. The program needs to first declare the desired matrix shapes and reserve the register resources for input matrices using the `simd2::matrix` function. Then, the program can load input matrices into these reserved resources using the `simd2::loadmatrix` function or set values using the `simd2::fillmatrix` function. The `simd2::mmo` function receives arguments describing the desired SIMD^2 operation to perform on the input matrices and the location of the destination matrix.

After the code finishes necessary computation on these matrices, the `simd2::storematrix` can reflect the updated values to a memory location. In case the source dataset does not fit the supported formats, the program typically needs to explicitly partition datasets into tiles of matrices and aggregate partial results appropriately.

To facilitate programming and alleviate the burden of programmers, our framework provides a set of high-level functions as an alternative programming interface. Each maps to a specific type of SIMD^2 arithmetic operations. These functions are essentially composed using the aforementioned low-level functions. In contrast to the low-level interface with limitations on inputs, these high-level functions allow the programmer to simply specify the memory locations of datasets and implicitly handle the tiling/partitioning of datasets and algorithms.

Figure 3.6 provides an example code that implements a high-level interface function that solves the `min-plus` matrix problems. The compute kernel starts by identifying the logical SIMD^2 unit of the instance itself is occupying (Lines 6–7). The compute kernel then allocates resources on the SIMD^2 units (Lines 9–11). The code then loads the current partial result of the target tile into one of the allocated matrix storage (Line 13). The following for-loop (Lines 15–21) loads different pairs of tile matrices from the raw input (Lines 17–18) and performs `min-plus` operations (Line 20) on these tile matrices together with tile loaded in Line 13.

To use the compute kernel from Figure 3.6 or the low-level SIMD^2 interface, the programming model still requires a host program to control the workflow, coordinate the computation on various types of processors and move datasets among memory locations

```
1  void simd2_minplus( half *A, half *B,
2                      float *C, float *D,
3                      int m, int n, int k){
4    // set tile ID
5    int tile_id_y = get_tile_id_y();
6    int tile_id_x = get_tile_id_x();
7    // Declare simd2 matrices
8    simd2::matrix<simd2::matrixa,16,16,16,half> mat_A;
9    simd2::matrix<simd2::matrixb,16,16,16,half> mat_B;
10   simd2::matrix<simd2::accum,16,16,16,float> mat_C;
11   // load C to c_tile
12   simd2::loadmatrix(mat_C, C, 16)
13   // loop over K, each time do 16x16x16 mmo
14   for(int tile_id_k=0;tile_id_k<k;tile_id_k+=16){
15     // load A/B into a_tile/b_tile
16     simd2::loadmatrix(mat_A, A, 16)
17     simd2::loadmatrix(mat_B, B, 16)
18     // performe mmo
19     simd2::mmo(mat_C, mat_A, mat_B, mat_C, minplus);
20   }
21   // store back results
22   simd2::storematrix(D,mat_C, 16);
23 }
```

Figure 3.6: Tiled minplus MM on some architecture with SIMD^2 supports

on heterogeneous computing devices. Figure 3.7 shows an example code that solves the all pair shortest path problem using the All-pairs Bellman Ford algorithm. As SIMD^2 units are

```
1  float * adj_mat_d;
2  float * dist_d_delta;
3  float * dist_d;
4  cudaMalloc(..., adj_mat_d, ...);
5  cudaMalloc(..., dist_d, ...);
6  cudaMalloc(..., dist_d_delta, ...);
7
8  cudaMemcpy(adj_mat_d, ..., H2D);
9  cudaMemcpy(dist_d_delta, ..., H2D);
10 cudaMemcpy(dist_d, ..., H2D);
11
12 bool converge = true;
13 while(converge){
14     simd2_minplus(adj_mat_d, dist_d, dist_d, dist_d_delta, v, v, v);
15     converge = check_convergence(dist_d, dist_d_delta, ...);
16 }
17 cudaMemcpy(... , dist_d, ... , D2H);
```

Figure 3.7: CUDA kernel implenmentation of APSP using SIMD^2 API

auxiliary computing resources to a GPU, the program code will need to explicitly allocate GPU device memory (Line 4–10) and move data to the allocated space before invoking the high-level `simd2_minplus` function that Figure 3.6 implements (Line 14). The naïve SIMD^2 implementation of All-pairs Bellman Ford algorithm would require V iterations of Line 14. The naïve implementation assumes the diameter of the graph is always the same as the number of vertices, the worst case scenario. However, the diameter of a real-world graph is way lower than that and a majority of iterations in Line 14 repeatedly generate

identical results. Therefore, the implementation in Figure 3.7 added a convergence check (i.e., the `check_convergence` function call) in Line 15 to compare if any element in the result matrix changes from the last iteration. If the result remains the same, the algorithm can terminate earlier. The `check_convergence` (Line 15) is a pure GPU kernel. Because both SIMD^2 units and conventional GPU cores share the same device memory and registers, the program does not need additional data movements between Line 14 and Line 15.

In Figure 3.7, we use All-pairs Bellman Ford algorithm as the inputs of SIMD^2 computation in this algorithm are easier to understand. In practice, the Leyzorek's Algorithm can solve APSP problem with fewer SIMD^2 operations [90]. Leyzorek's Algorithm still uses SIMD^2, but computes $C = C \oplus (C \otimes C)$ in Line 14 instead. In this way, Leyzorek's Algorithm only requires $lg|V|$ iterations to solve an APSP problem in the worst case scenario.

3.5 Experimental Methodology

As SIMD^2 promotes matrix-based algorithms, the SIMD^2-ized implementations of our benchmark applications may use different algorithms compared to their state-of-the-art implementations, typically using vectorized or scalar-based algorithms, on alternative platforms. Therefore, we designed a framework that allows us to validate the correctness of SIMD^2-ized programs and emulate the performance of SIMD^2-ized programs with or without SIMD^2 hardware acceleration presented. This section will describe these aspects in detail.

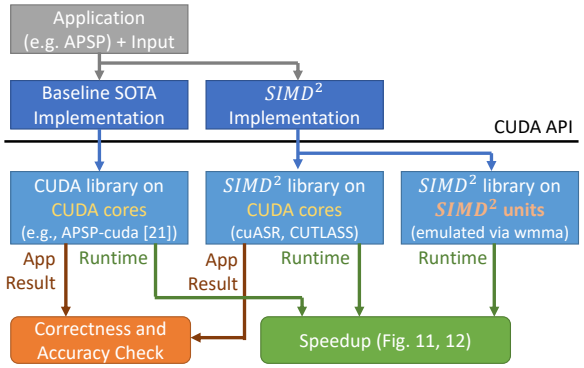


Figure 3.8: The workflow of the emulation framework for SIMD^2 evaluation.

3.5.1 Emulation framework

To evaluate SIMD^2, we developed a framework that evaluates the correctness and performance for each program under test on top of a testbed using a state-of-the-art GPU architecture.

Hardware configuration

Our validation and emulation framework uses a machine with NVIDIA's RTX 3080 GPU based on the Ampere architecture with 10 GB device memory. This machine has an 8-core, 16 threads AMD RyZen 3700X processor with peak clock rate at 4.4 GHz and 16 GB physical main memory installed. The machine hosts an Ubuntu 20.04 (Linux kernel version 5.13) with NVIDIA's CUDA 11.1 using driver version 470.103.01.

Evaluation Process

Figure 3.8 illustrates the workflow of our process in evaluating applications. For baseline applications, we executed each application directly on the hardware platform with-

out any modification to their source code, datasets and invoked library functions. For SIMD^2-ized applications, their implementations leverage semiring-like algorithms using the programming model and SIMD^2 API functions described in Section 3.4. The evaluation framework takes three types of inputs: (1) the compiled program and command line arguments, (2) the dataset used in the baseline application, and (3) the output of the baseline application with the input dataset and command line arguments. Once the emulation framework receives these three sets of inputs, the emulation framework can dynamically change the linked library that implements SIMD^2 API functions to perform (1) correctness validation by using a backend that leverages conventional vector processors and compare the output with the output that the baseline version produced, or (2) performance emulation by using a backend that generates instructions to Tensor Cores residing on the hardware platform. The following paragraphs will describe the correctness validation and performance emulation process in detail.

Correctness validation In this work, we need to validate correctness in addition to performance emulation for the following reasons. First, as we need to alter the compute kernels to efficiently use SIMD^2 units and in many cases, using a different algorithm (e.g., Semiring-based vs. Kruskal's Algorithm in Minimum Spanning Tree problems), we need to verify if the change of implementation still delivers the same outcome as the baseline implementation. Second, as existing hardware accelerators only support MMA operations that cannot generate the correct output for other SIMD^2 operations this paper proposes to extend, we need to verify if implemented semiring-based algorithms can generate the desired output after mapping the computation into the proposed SIMD^2 units. Finally, this process

can help collect the statistics regarding the total amount of various matrix operations and provide the input for performance emulation.

During the validation process, we linked the backend of the SIMD^2 programming interface to a library that we extended from cuASR [70]. This library implements exactly the same functionality as the proposed low-level SIMD^2 functions, except that the library can simply leverage CUDA cores through NVIDIA's high-performance CUTLASS library, but not use Tensor Cores. When implementing low-level SIMD^2 functions for validation purposes, we carefully partitioned the inputs and outputs to fit the exact shape of matrix inputs and outputs of proposed SIMD^2 units (i.e., the input/output sizes of each Tensor Core in our testbed) when invoking corresponding SIMD^2 function calls. We also used reduced/mixed precision inputs/outputs to match the data types that our SIMD^2 units support. Therefore, the validation process can help us access the accuracy of SIMD^2 units. For each program under test, we can optionally count the number of iterations, threads, and low-level SIMD^2 function calls that are necessary to finish running the program and compare each program's output with its state-of-the-art implementation on the alternative architecture.

Performance emulation The design of SIMD^2 allows this work to leverage existing Tensor Cores that are available on the GPU of our emulation hardware for exact performance evaluation for the two main reasons. First, adding SIMD^2 instructions do not increase the timing of an existing MMA unit (e.g., a Tensor Core) as Section 4.6.7 reports. Second, the low-level instructions, register files, memory hierarchy as well as the interaction with the host machine can be made almost identical to those of Tensor Cores, except for the exact output after each computation.

When performing performance emulation, the framework links the backend of the low-level SIMD^2 API library that implements through using equivalent Tensor Cores' WMMA low-level interface. As this paper simply proposes to extend the ALU functions of Tensor Cores, the memory operations remain the same in SIMD^2 units compared with Tensor Cores. Therefore, each `simd2::loadmatrix` and `simd2::storematrix` invocation are identical in its counterpart in CUDA's WMMA API. However, since the state-of-the-art Tensor Cores can only perform MMA operations, the performance emulation backend library maps each invocation of `simd2::mmo` to a CUDA's `WMMA::mma` function call on the same size of inputs. This is also the main reason why the performance emulation backend cannot produce correct/meaningful computation outcomes. The performance emulation process can optionally receive statistics from the corresponding validation process to compare if the performance emulation backend generates the exact amount of simd2 and WMMA operations as desired. This performance emulation methodology is similar with prior work in extending Tensor Cores [39] to support different precisions.

Table 3.4: Source and input data size of baseline implenmentation for each selected applications.

Application	**Baseline Source**	**Input Dimension**	
All Pair Shortest Path (APSP)	ECL-APSP [79, 97]	Small	4096
		Medium	8192
		Large	16384
All Pair Critical Path (APLP)	ECL-APSP [79, 97]	Small	4096
		Medium	8192
		Large	16384
Maximum Capacity Path (MCP)	CUDA-FW [101, 106]	Small	4096
		Medium	8192
		Large	16384
Maximum Reliability Path (MAXRP)	CUDA-FW [101, 106]	Small	4096
		Medium	8192
		Large	16384
Minimum Reliability Path (MINRP)	CUDA-FW [101, 106]	Small	4096
		Medium	8192
		Large	16384
Minimum Spanning Tree (MST)	CUDA MST [55, 61, 71, 141, 151]	Small	1024
		Medium	2048
		Large	4096
Graph Transitive Colsure (GTC)	CUBOOL [131]	Small	1024
		Medium	4096
		Large	8192
K-Nearest Neighbor (KNN)	KNN-CUDA [164]	Small	4096
		Medium	8192
		Large	16384

3.5.2 Applications

To demonstrate the performance of SIMD^2, we ran two types of workloads on the aforementioned evaluation framework. The first type is a set of microbenchmark workloads that only iteratively invoke SIMD^2 functions and accept synthetic datasets to help us to understand the pure performance gain of SIMD^2 instructions over alternative implementations.

The other is a set of full-fledged benchmark applications where each program contains not only SIMD^2 functional, but also interacts with other types of processors to

complete the tasks. These benchmark applications can accept real-world datasets and generate meaningful outputs accordingly for us to assess the quality of results if appropriate. For each workload, we evaluate three implementations.

State-of-the-art GPU baseline This version of code serves as the baseline of our workloads. We tried our best to collect implementations from publicly available open-source code hosting websites and select the best-performing implementation on our testbed as the state-of-the-art baseline version for each workload. These implementations simply leverage CUDA cores, but not Tensor Cores to accomplish their tasks. In fact, without a work like SIMD^2, none of the selected benchmark can leverage Tensor Cores due to the limited MMA functions available on such hardware units.

SIMD^2 in CUDA cores This version of code serves as another baseline of our workloads. This set of programs implement SIMD^2-ized algorithms only using CUDA cores, but not Tensor Cores. Our implementations try to leverage the highly optimized functions from cuASR or CUTLASS whenever appropriate. Different from backend functions used in Section 3.5.1, this version of code does not manually partition the algorithms based on our proposed SIMD^2 hardware configuration but allow the code to fully exploit the performance from CUDA cores. This version helps us to identify the performance variance by naively applying matrix algorithms without the presence of appropriate matrix accelerations.

SIMD^2 using Tensor Cores This version of code use identical algorithms to the version of SIMD^2 in CUDA cores except that we replace these algorithms' matrix operations to SIMD^2 ones when appropriate. As existing hardware does not support our proposed SIMD^2

operations yet, we evaluate the performance and validate the result of this version through the framework that Section 3.5.1 describes.

Table 3.4 lists the set of benchmark applications. Each of these applications represents a use case for a proposed SIMD^2 instruction as follows.

All-Pairs Shortest Path (APSP) and All-Pairs Critical (Longest) Path (APLP) APSP and APLP are graph problems that can be solved via `min-plus` and `max-plus` SIMD^2 instructions. Without SIMD^2, the most efficient implementation, ECL-APSP [97], applied a phase-based-tiled Floyd Warshall algorithm to exploit massive parallelism using CUDA. We implemented APLP by extending the ECL-APSP with reversing the input weights on DAG to support the desired recurrence relation. For SIMD^2 version, the implementation simply changes the function calls to use `min-plus` and `max-plus`.

Maximum Capacity Path (MaxCP), Maximum Reliability Path (MaxRP) and Minimum Reliability Path (MinRP) MaxCP, MaxRP and MinRP represent another set of graph problems with solutions based on transitive-closure. We select CUDA-FW as the state-of-the-art GPU baseline for these problems and apply different operations in each iteration of their algorithms. These applications' SIMD^2 kernels simply require invoking `max-min`, `max-mul` and `min-mul` instructions.

Minimum Spanning Tree (MST) Minimum spanning tree or minimum spanning forest (MSF) has rich applications in real-life network problems. However, conventional MST or MSF algorithms cannot efficiently take advantage of GPU architectures due to limited parallelism. The best-performing GPU implementation that we know of is CUDA MST and we use this one as our baseline. MST and MSF map perfectly to the `min-max` SIMD^2

instruction. Our SIMD^2 version of code thus leverages `min-max` instruction to investigate the efficiency of SIMD^2 in this type of problem.

Graph Transitive Closure (GTC) GTC is also a graph analytics workload. Unlike other graph algorithms, GTC simply checks the connectivity between all vertices rather than reporting a route to fulfill the goal of optimization. Therefore, GTC can use library functions from cuBool [131] for efficient implementation on GPUs. In SIMD^2 version, we used `or-and` instruction to implement the solution.

K-Nearest Neighbor (KNN) Solving pair-wise L2 distance is at the core of K-nearest neighbor and K-means problems, and can leverage SIMD^2's `add-norm` instruction. For the state-of-the-art GPU baseline, we use KNN-CUDA.

3.6 Results

This section summarizes our evaluation of SIMD^2. SIMD^2 delivered up to 38.59× speedup in benchmark applications with simply 5% of total chip area overhead.

3.6.1 Area and Power

We implemented the proposed SIMD^2 unit in RTL and synthesize them using Synopsis design compiler and the 45nm FreePDK45 library. We extended a baseline MMA unit that can simply perform MMA functions like conventional MXUs presented in Tensor Cores. The baseline MMA unit features 4x4 matrix multiplications on 16-bit input elements and accumulates results in 32-bit elements. This configuration resembles the architecture used by Tensor Cores [121] and Accel-Sim [81]. We carefully design the proposed extensions

Table 3.5: The area overhead of supporting SIMD^2 instructions through (a) adding instructions to the MMA unit, (b) individual accelerators, (c) extension to the MMA unit with various precisions, compared to the baseline 16-bit MMA Unit.

Supported Ops.	Area
MMA + All SIMD^2 Insts.	1.69
MMA + Min-Plus	1.21
MMA + Max-Plus	1.21
MMA + Min-Mul	1.12
MMA + Max-Mul	1.12
MMA + Min-Max	1.01
MMA + Max-Min	1.01
MMA + Or-And	1.04
MMA + Add-Norm	1.18

(a)

Supported Ops.	Area
Min-Plus	0.26
Max-Plus	0.26
Min-Mul	1.03
Max-Mul	1.03
Min-Max	0.06
Max-Min	0.06
Or-And	0.08
Add-Norm	0.19
Total	2.96

(b)

	8-bit	16-bit	32-bit	64-bit
MMA only	0.25	1	4.04	11.17
MMA + All SIMD^2 Insts.	0.69	1.69	6.42	17.01

(c)

to make the timing of the SIMD^2 unit the same as the baseline. We empirically observe that our the modification for the SIMD^2 unit never increases the critical path delay.

Table 3.5(a) lists the area overhead of adding SIMD^2 instructions into the baseline MMA unit. The baseline MMA unit is 11.52 mm^2 in size. Adding each individual instruction results in 1.34% – 21.25% overhead. The full-fledged SIMD^2 unit has an area overhead of 69.23%. We inspected the public die photo of an NVIDIA 3080 GPU and found that SMs account for 50.2% of the 628.4 mm^2 die area, and each SM is 3.75 mm^2. If we scale the 69.23% overhead from the 45nm process to the Samsung 8N process used for our 3080 NVIDIA GPU baseline, a SIMD^2 unit introduces only 0.378 mm^2, which is only 10% of the SM area and 5% of the total die area.

Table 3.5(b) also lists the case where we only implement a processing element to support a specific SIMD^2 instruction without the MMA function (i.e., as an individual

accelerator). If we implement each SIMD^2 instruction separately as an individual accelerator, the total area of these accelerators will require additional 2.96x space of the baseline MMA unit. In contrast, the design of SIMD^2 unit allows these instructions to reuse common hardware components and saves area. For example, we found that for the processing elements supporting Min-Mul and Max-Mul operations, the area is almost the same as an MMA unit. However, combining their functions into a single SIMD^2 unit only results in 11.82% of area overhead, showing these instructions can share a large amount of circuits that were originally used for MMA operations. The baseline MMA unit consumes 3.74 W power. Extending the baseline as a SIMD^2 unit only adds 0.79 W to the active power.

If we extend the baseline MMA to support 32-bit numbers, the size of the MMA unit becomes 4.03x larger than a 16-bit MMA unit as Table 3.5(c) lists. A SIMD^2 unit supporting 32-bit inputs occupies 59% more area than the 32-bit MMA unit. If we further extend the MMA to support 64-bit numbers, the size of the MMA unit becomes 11x larger than the 16-bit MMA. Extending the 64-bit MMA unit as a 64-bit SIMD^2 unit will add 52% area overhead. If we make both the baseline MMA and SIMD^2 units in supporting 8x8 matrix operations in 16-bit inputs, the MMA unit will become 7.5x larger than the 4x4 baseline. The area overhead over the baseline MXU stays constant and scales well.

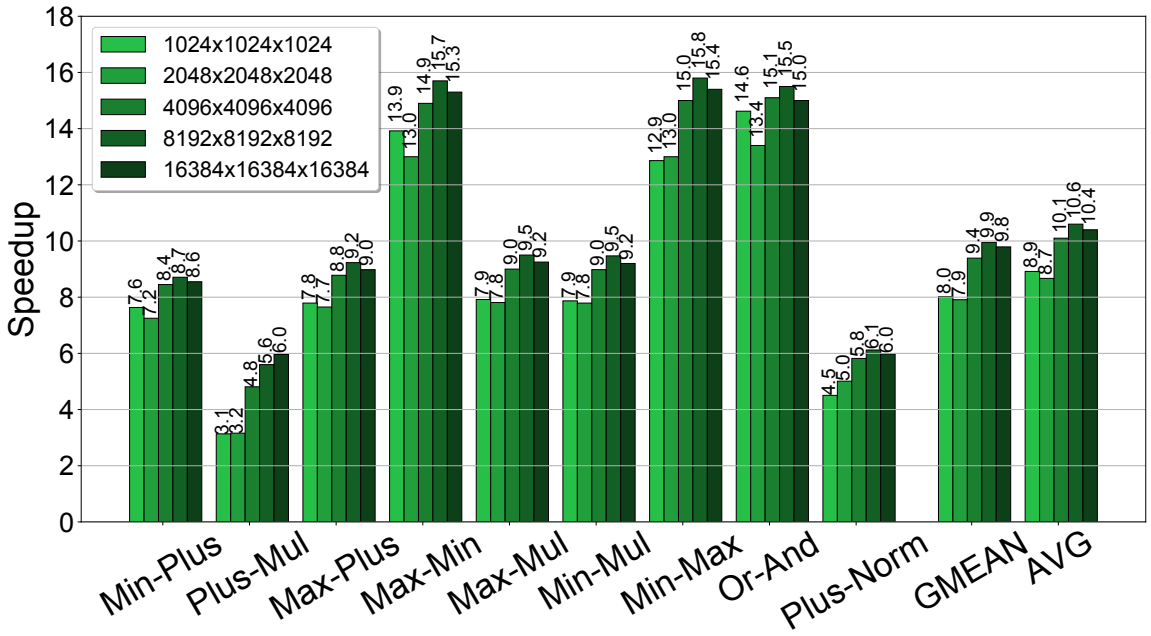


Figure 3.9: Performance of microbenchmark with square matrices using SIMD^2 API

3.6.2 Microbenchmarks

We used microbenchmark workloads that repetitively invoke SIMD^2 the same instructions to gauge the performance gain of using SIMD^2 units compared against equivalent GPU implementations. The result shows up to 15.8× speedup in evaluated scenarios.

Figure 3.9 shows the performance gain of SIMD^2 over the equivalent GPU baseline implementations when using square matrices as inputs. SIMD^2 reveals up to 15.8× speedup compared with using CUDA cores to achieve the desired matrix operation on the same dataset. The geometric mean (gmean) that discounts the outlier also shows a strong 7.9×–9.9× speedup, depending on the input set sizes. When input matrices are larger than

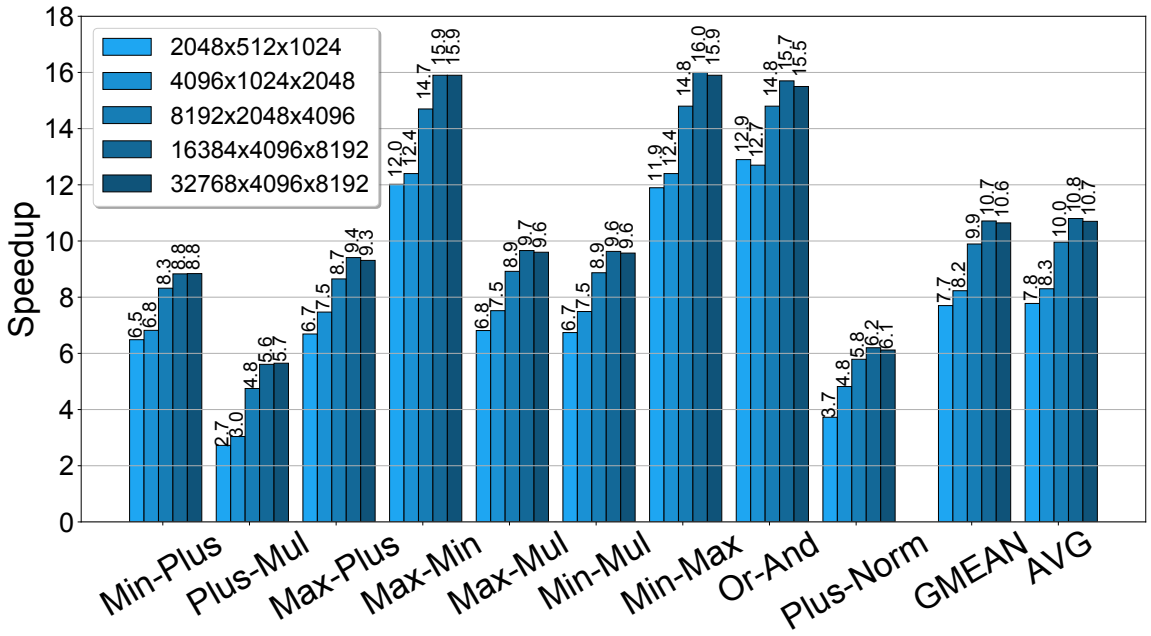


Figure 3.10: Performance of microbenchmark with nonsquare matrices using SIMD^2 API

4,096×4,096 ones, the performance gain saturates at about 10×, representing the level of peak performance gain of these instructions. Figure 3.10 shows the performance gain of SIMD^2 instructions on different shapes of matrices. The performance gain still saturates at the level of 10× when matrices are large, regardless of their shapes.

From both results, SIMD^2 has the largest performance gains for `min-max`, `max-min`, and `or-and` instructions, by up to 15.8×. Such improvement is larger than the peak throughput difference between vector units and SIMD^2 units. We suspect the extra benefit from SIMD^2 units is due to the structural hazard in the GPU SM architecture, where `min` and `max` operations share the same hardware resources(e.g., ALU port), and so are `or` and

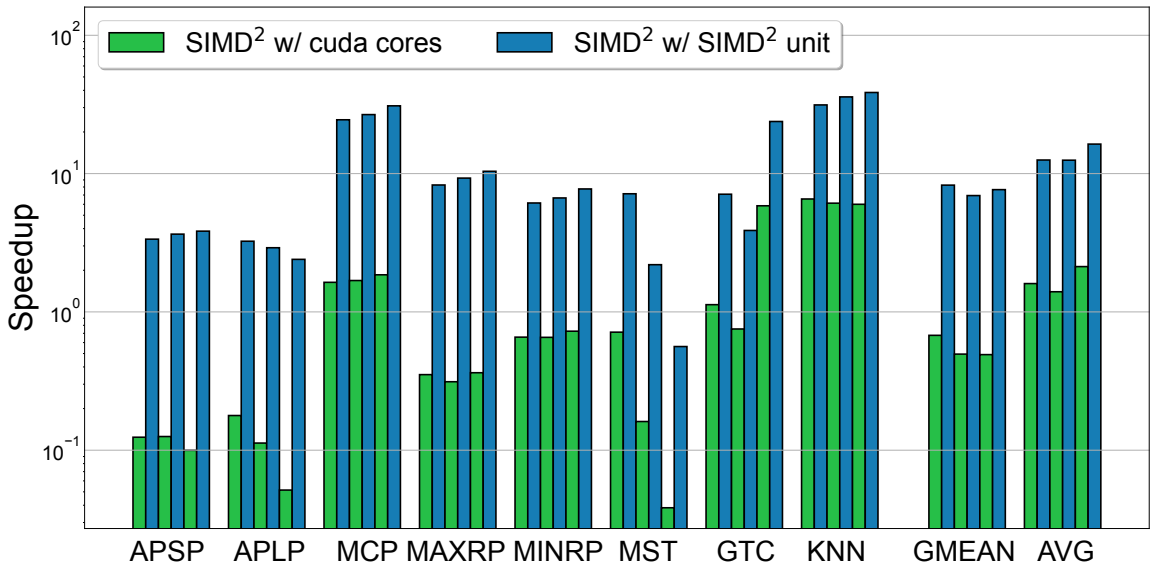


Figure 3.11: Performance of applications using SIMD^2 API

`and` operations. By fusing these operations in a single instruction, SIMD^2 unit avoids this bottleneck and results in much higher speedup.

The speedups of `Plus-Mul` and `Plus-Norm` operations are relatively low compared with others, but still enjoy a 3.1× speedup over using CUDA cores. This is because CUDA cores provide support for fused multiply-add (FMA) that allow the GPU to complete `plus-mul` operations with a single instruction. We expect that supporting more instructions similar to FMA would also provide similar performance boost to the class of problems that SIMD^2 addresses. Nevertheless, SIMD^2 still has a significant advantage, obtaining a speedup of up to 5.96× for larger matrix operations. We conclude that the SIMD^2 architecture has larger potential than fusing more vector operations, which we leave to future work.

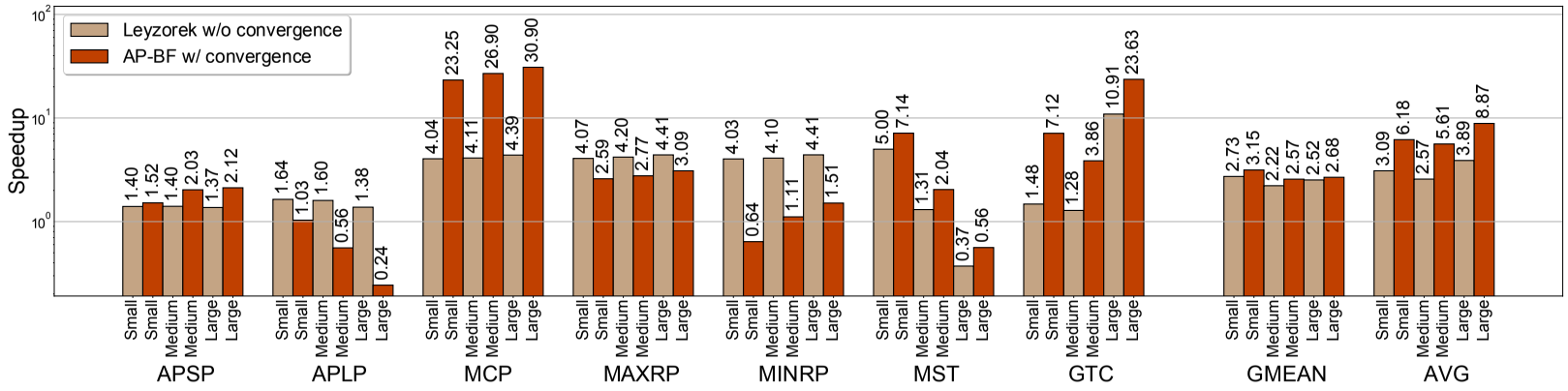


Figure 3.12: Performance of different algorithmic optimizations

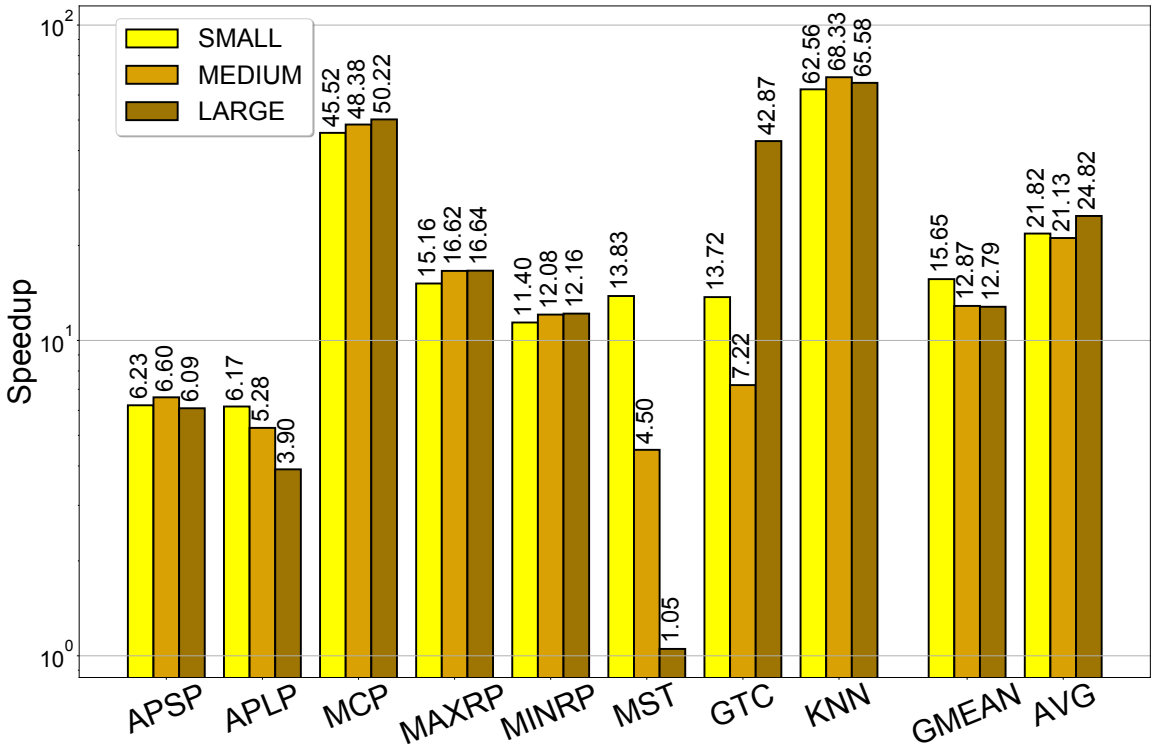


Figure 3.13: Performance of applications using Sparse SIMD^2 unit

3.6.3 Benchmark Applications

Figure 3.11 shows the speedup of kerenl latency of applications using SIMD^2 (SIMD^2 w/ SIMD^2 units) over the baseline, optimized GPU implementations. SIMD^2 achieves a geometric mean of 6.94× – 8.25×, with speedup as large as 38.59×. The performance gain of SIMD^2 in 7 out of the 8 applications remains strong even when dataset sizes increased.

Compared with implementing the same matrix-based algorithms without SIMD^2 presented (SIMD^2 w/ CUDA cores), all applications show significant slow down when

SIMD^2 units are absent. For APLP, MST, MaxRP, MinRP, and APSP, these applications can never take advantage of matrix-base algorithms due to their higher computational complexities when SIMD^2 units are absent. This result explains why these algorithms were not favorable in conventional architectures. However, the introduction of SIMD^2 makes these matrix algorithms feasible. The matrix processing power from the SIMD^2 unit can compensate or even improve the performance of the applications as our experimental results tell. In fact, these algorithms can potentially take advantage of the embarrassingly parallel nature of matrix multiplication to parallelize hard-to-parallelize problems.

For MCP, GTC, and KNN, their SIMD^2 implementations out-perform their baseline, state-of-the-art implementations, even without the presence of SIMD^2 units. For KNN, the computational complexity is the same for both SIMD^2 and the baseline implementations. However, the SIMD^2 kernel can still achieve a maximum speedup of 6.55× without the help of SIMD^2 units. This is because the baseline implementation uses customized functions to implement the algorithm, but the backend library of SIMD^2 without SIMD^2 units leverages CUTLASS that is more optimized and adaptive to modern GPU architectures. However, the performance gap between configurations with or without SIMD^2 units ranges between 4.79× and 6.43×. The performance advantage is more significant when we use the largest dataset. Therefore, even we revisit the design of the GPU baseline and make that as efficient as SIMD^2 on CUDA cores, such implementation still has a huge performance gap to catch up with the performance using SIMD^2 units. For MCP and GTC, SIMD^2 w/ CUDA cores can outperform their baseline implementations even though the computational complexity is higher in SIMD^2 implementations for two reasons. The first reason is similar to the case in KNN that SIMD^2 w/ CUDA cores benefits from more optimized library functions than the baseline ones.

The other reason is that the rich parallelism of these matrix-based algorithms allow these implementations to scale better on modern GPU architectures – considering that the RTX 3080 GPU has twice as many CUDA cores than that of the previous generation of GPU architecture. However, the state-of-the-art baseline implementation cannot take advantage of this architectural improvement. On the other hand, this result also reveals that SIMD^2 programming model can make programs more adaptive to various underly-

ing architectures since these architectural optimizations on SIMD^2 operations will remain without the demand of further code optimization.

The performance of APLP and MST using SIMD^2 degrades when datasets become larger. This is because both APLP and MST using SIMD^2 require additional convergence checks that are sensitive to input data values to determine the completion of the solution. As the input dataset grows, the variance in the content also becomes more significant and needs more iterations for the algorithm to converge. However, if the number of iterations do not increase with the growth of dataset sizes, the program can still show performance gain over conventional CUDA cores since SIMD^2 still makes each iteration faster. For MST, the baseline GPU solution uses Kruskal's algorithm that can solve MST/MSF problems with computational complexity at $O(E \log E)$ [30,87], where E is defined as the number of edges in the input graph. In contrast, each iteration of the matrix-based SIMD^2 solution has the complexity of $O(V^3)$ [30, 42], where V is the number of vertices in the input graph. Therefore, SIMD^2 becomes slower than the baseline implementation in each iteration for MST when dataset size is larger.

3.6.4 Discussion on algorithmic optimizations

In Figure 3.11, our implementations use Leyzorek's algorithm and convergence checks to optimize the number of SIMD^2 operations, except for KNN. As the proposed SIMD^2 architecture improves the performance of supported semiring-like operations, SIMD^2 still allows these matrix-based algorithms to outperform the baseline state-of-the-art GPU implementations without these algorithmic optimizations.

The effect of convergence checks is sensitive to inputs. For each compute kernel using Leyzorek's algorithm on graph problems with V vertices, the implementation will take $lg|V|$ SIMD^2 operations in the worst-case scenario. To evaluate the worst-case performance, we implemented a version of these applications without convergence checks. Figure 3.12 illustrates the performance of these implementations with bars labeled as Leyzorek w/o convergence. The baseline remains the same as Figure 3.11. Despite the increasing numbers of iterations, all applications still outperform their baseline GPU implementations, ranging from 1.11× to 10.91×.

In Figure 3.12, we also present implementations of these applications using the less efficient all-pair Bellman-Ford algorithm (AP-BF w/ convergence). As Bellman-Ford algorithm can take up to —V— SIMD^2 operations, using Bellman-Ford algorithm can slow down APLP and MST when datasets become large. MINRP can never beat GPU implementations if we use Bellman-Ford algorithm-based implementations. However, the performance gain remains significant for other applications as the advantage of SIMD^2 architecture out-weighted the shortcomings of increased computational complexity.

3.6.5 SIMD^2 for Sparse Workloads

SIMD^2 on architectural support for sparsity. The idea of SIMD^2 can be applied to architecture support for sparse inputs, too. As an initial look of the SIMD^2 model, we extend our emulation framework and build on top of the `cuSparselt` library to model the benefit of applying the SIMD^2 idea to the sparse Tensor Cores in the RTX 3080 GPU, which supports structured sparsity and provides up to 2× throughput. We assume

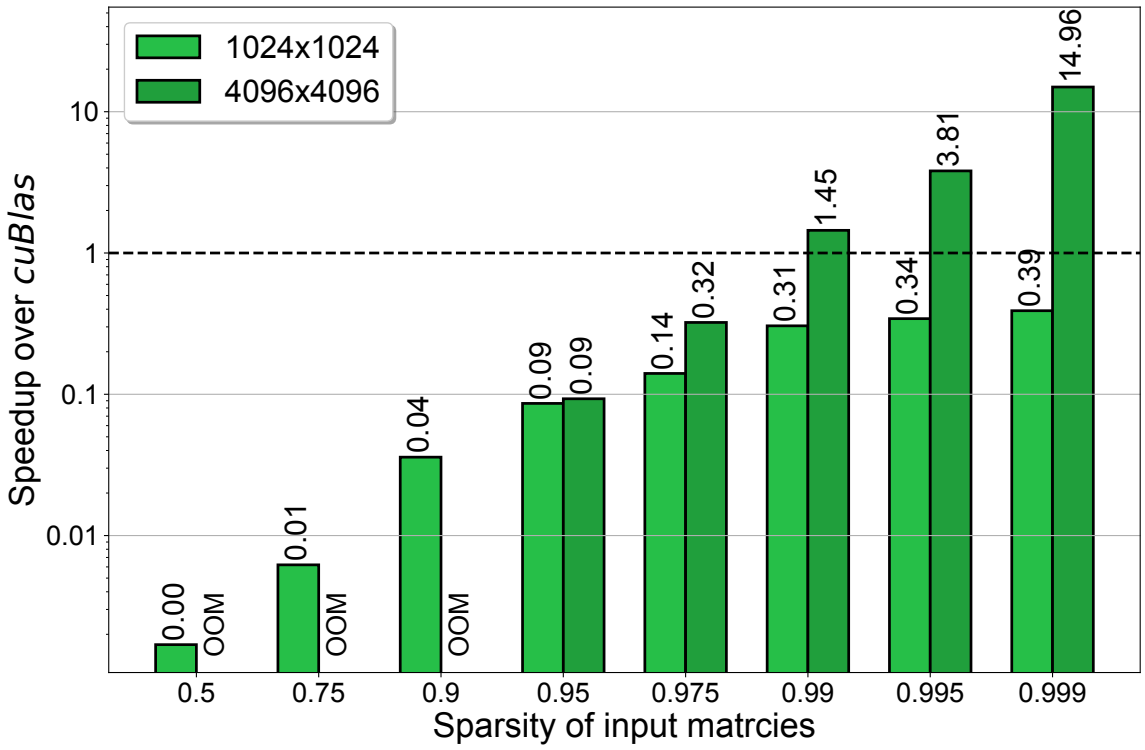


Figure 3.14: Performance of sparse matrix multiplication

the inputs are pre-processed and stored in the format required by the sparse Tensor Core, excluding the processing overhead when reporting the speedup.

Figure 3.13 shows the speedup over baseline implementation when using a sparse SIMD^2 unit. We performed experiments using datasets with densities at 1%. Using sparse SIMD^2 units can improve performance by up to 68.33×, with geometric means ranging from 12.79×–15.65×. Compared with the baseline SIMD^2, SIMD^2 on sparse Tensor Cores is 1.67×–1.9× faster.

SIMD^2 for extremely sparse inputs. Some applications often have extremely sparse inputs, especially for graph algorithms. For these sparse inputs, a dense SIMD^2 unit

might provide less performance improvement over implementations that are designed for sparse inputs, such as the `cuSparse` library. We therefore explore at what sparsity SIMD^2 can still provide benefits, which is illustrated in Figure 3.14. The x-axis in Figure 3.14 shows the sparsity of inputs, meaning the ratio of zeros to non-zeros in each dataset. The y-axis shows the speedup of using NVIDIA's `spGemm` function, a sparse GEMM function optimized for Tensor Cores, from `cuSparse` library compared against `gemmEx` function, a dense GEMM function for Tensor Cores, from `cuBlas` library. The results show that *cuSparse* does not outperform *cuBlas* for matrices of size 1024×1024, and for matrices of size 4096×4096, *cuSparse* can outperform *cuBlas* when the sparsity of the input matrix exceeds 99%. This result shows that while many applications need to process sparse inputs, there is still a range of sparsity where SIMD^2 can provide benefit. Such range also covers a number of real graph datasets that do not exceed the sparsity indicated in the results [107], implying that it is more efficient to use the dense matrix processing method for these cases if appropriate architectural support for sparse matrix operations are absent.

To handle extremely sparse inputs (sparsity >= 99%) on larger graphs, we can apply SIMD^2 sparse accelerators for spGEMM, which also use multiply-and-add for the ALU, such as GAMMA [183]. For example, a GAMMA PE uses FP64 multiplier and adder, and an SIMD^2 GAMMA PE will use two FP64 ALUs, one supports the $\oplus$ op, and the other supports the $\otimes$ op. This SIMD^2 GAMMA accelerator would then be able to run APSP on sparse graphs. In fact, extending sparse accelerators with SIMD^2 would incur less overheads, as compute units contribute to less area than dense accelerators. For example, in GAMMA, only 10% of the total area is due to the FP64 MAC unit. We leave this extension to future work.

It is worth mentioning that while libraries like `cuSparse` have an advantage in terms of space complexity when dealing with extremely sparse matrices, the compressed matrix format may consume more device memory when storing relatively dense matrices. Experimental results show that `cuSparse` requires more memory than a single RTX 3080 GPU can provide when processing matrices with sparsity less than 90% (the OOM result in Figure 3.14) and size more than 16384×16384. However, when using a dense processing method, a GPU with 10GB of device memory can accommodate a matrix multiplication of at least 32768×32768 in size.

3.7 Conclusion

Recent advance in hardware accelerators that accelerate matrix multiplications in AI/ML workloads encourage us to take a new look at other matrix problems. As many matrix problems share a similar computation pattern with matrix multiplications that existing

hardware accelerators already optimize for, a more generalized matrix processor will allow these matrix problems to benefit from hardware acceleration.

This paper introduces SIMD^2 to investigate the potential of this research avenue. We leverage the common computation pattern of significant matrix problems to design the SIMD^2 instruction set and implement a feasible, exemplary hardware architecture supporting these SIMD^2 instructions with 5% total chip area overhead. We demonstrate the effectiveness of SIMD^2 using a set of benchmark applications, some of them are rewritten with algorithms that are traditionally considered inefficient due to the lack of hardware support like SIMD^2. Our evaluation results show that the proposed SIMD^2 architecture achieves more than 6.94× speedup on average across eight applications with various tensor computation patterns.

Chapter 4

Exploiting Data reuse with Inter-operator Dataflow

Sparse tensor algebra (STA) is the key building block of scientific computing, graph analytics, and machine learning applications. STA operators such as SpMSpM (sparse matrix-sparse matrix multiplication), SpMM (sparse matrix-dense matrix multiplication), and SpMV (sparse matrix-dense vector multiplication) contribute to the majority of the runtime of these applications [4, 57]. Unlike dense tensor algebra, the data movement limits performance of STA applications due to their low arithmetic intensity. Thus, maximizing data reuse is the key to accelerating STA applications.

4.1 Overview of data resue and SIDA

Prior research focuses on exploiting *intra-operator data reuse* to reduce data movement. For example, SpMSpM accelerators [57, 112, 183] propose dataflows with specialized

format and microarchitecture support to minimize data movement in SpMSpM operations. Other work improves cache locality [16, 17] of SpMV operations in graph analytics. These ideas push the system closer to the roofline [172], where the available memory bandwidth is fully utilized for the STA operation. However, even with these improvements, due to the intrinsic low arithmetic intensity in STA operations, many applications still reside in the bandwidth-bound region of the roofline [128], and any further data movement reduction can be directly translated into performance.

Conventionally, implementing STA applications requires hand-written and format-specific code with nested loops and application-specific logic, muddling opportunities to reduce data movements. Fortunately, recent developments in domain-specific STA languages and compilers [19, 41, 84, 142] alleviate the programming burden by generating low-level code for STA applications. Frameworks with tensor and dataflow abstractions, such as TensorFlow [6], PyTorch [68], GraphBLAS [34], and ALP [181], offer new abstractions for STA applications. Such novel abstraction with a dataflow graph presents data reuse opportunities *beyond a single operator*.

We find that there are two unexplored, inter-operator reuse opportunities for STA applications. First, **producer-consumer data reuse** reduces data movement by combining multiple tensor operations into a single, large fused operation. Prevalent in dataflow graphs, producer-consumer reuse is typically captured by forming pipelines of operations to keep intermediate results in on-chip buffers. While producer-consumer data reuse has been exploited widely in dense tensor algebra [7], limited work has explored this reuse opportunity in STA applications.

Second, **cross-iteration data reuse** is a *new reuse opportunity* revealed in this paper, which extends beyond single or adjacent STA operations. By unrolling loops or stages in STA applications, it is possible to fuse multiple identical operations (e.g. SpMV in a while-loop) and reduce memory traffic across iterations. No prior work has identified this opportunity, and harnessing cross-iteration data reuse requires a novel dataflow (Section 4.3) and corresponding hardware supports (Section 4.4).

To exploit these inter-operator reuse opportunities, this chapter proposes (a) OEI datflow, which facilitates inter-operator data reuse, and (b) SIDA–Sparse Inter-operator Dataflow Architecture, which incorporates key features:

- A dynamic execution pipeline with compute cores for each stage of the OEI dataflow. These cores support the diverse semiring operations prevalent in common STA applications, extending its applicability beyond HPC/DNN.
- An efficient on-chip buffer that streamlines the data supply to compute cores in the OEI dataflow.
- A set of intelligent control and management policies to schedule computation and data access tasks in sub-tensor manners to maximize data reuse, targeting the producer-consumer and cross-iteration reuse opportunities.
- A sparse tensor preprocessing algorithm, including blocking and reordering, to improve inter-operator reuse.

4.2 STA applications and Challenges of data reuse

In this section, we first review how to represent STA applications in modern sparse tensor framework. With this representation, we then identify the required hardware architecture ingredients and potential data reuses to motivate the proposed OEI dataflow and our SIDA architecture.

Implementing high-performance STA applications traditionally requires various manual optimizations and is rarely portable. To improve programmers' productivity, recent advances in language and compiler design thus leverage and extend the abstraction of BLAS [1] and Einsum [84] to allow programmers to represent their STA applications as tensor dataflow graph. For example, in Fig. 4.2, we compare two implementations of a classic STA application, `PageRank`, one in standard C, and the other in the GraphBLAS [80], tensor-based abstraction.

There are three advantages of the dataflow representation. First, the building block is a set of well-define, *semiring* tensor operators, such as `vxm` or `mxm` (vector/matrix matrix multiplication), and a series of `e-wise` (element-wise) operations.

4.2.1 STA applications as tensor dataflow graphs

For example, in `PageRank`, the used operators are `vxm` with `Mul-Add` as the semiring operation, and `set`, `fold`, `dot` (vector-vector dot product), `swap` as `e-wise` operations, and other STA applications use different combinations (see Table. 4.2 in Sec. 4.5). The implementation details of the operators, including storage format and tensor traversal order, are hidden from the programmer. This separation of concerns [139] lets programmers focus on expressing the applications and leaves how to optimize operators to system designers.

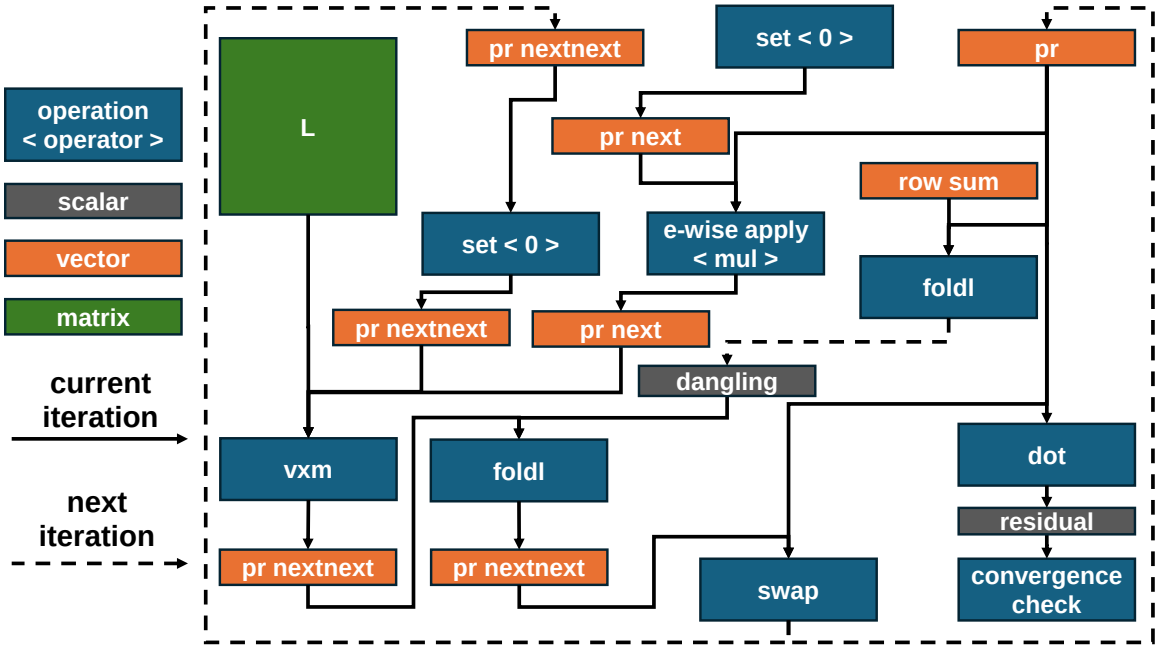


Figure 4.1: Inner loop dataflow graph of Pagerank algorithm implemneted by ALP/graph-bas. Few small operations are ommited to increase readibility.

Second, under this abstraction, the tensor dataflow and dependency between operators is clear. Compared to an non-fused cpmute graph shown in Fig. 4.1, Fig. 4.3 shows an abstracted compute graph of `PageRank`'s inner loop. The `vxm` operator takes vector `pr_next` and matrix `L` as input and produces vector `pr_nextnext` as output, while other vectors

serve as inputs, outputs, or intermediates for fused `e-wise`. As shown in Fig. 4.3 (b), two groups of `e-wise` can be fused by identifying connected components of operations and data nodes, yielding two new sub-graphs. This abstracted compute graph enhances visibility of data dependencies across each loop iteration, facilitating exploration of cross-iteration data reuse opportunities.

Lastly, under the dataflow representation, STA applications contain multiple iterations/stages of *the same subgraph*, with each iteration advancing towards a convergent result. The loop body can generally be divided into a BLAS-2/3 operation (`vxm` or `mxm`), and a series of `e-wise` operations. Very often, the matrix in the `vxm` or `mxm` operator is a constant sparse tensor (e.g., the graph `L` in `PageRank`), which accounts for the most of the data movement and is shared across iterations.

```
// L: Input graph  d: Damping factor
// pr, pr_next, pr_nextnext, row_sum: PageRank vector buffer
// dangling: Caching the contribution of random jumps from dangling nodes
// res: Residual value
// Mul-Add, Abs-Diff, Add, Mul: Semiring/Monoid operator
```

Standard C code	GraphBLAS code

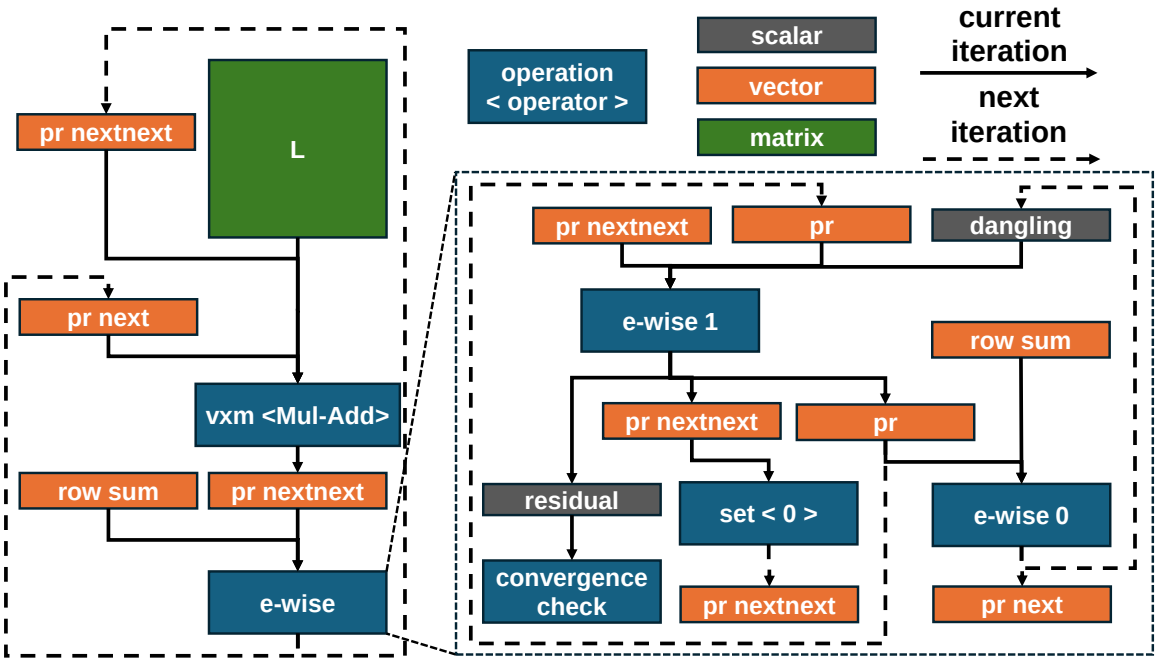
`for( int i = 0; i < n; i++ ) {`
`  dangling += pr[i] + row_sum[i];`
`  pr_next[i] = pr[j] * row_sum[j];`
`}`
	`foldl(dangling, pr, row_sum, Add);`
	`set(pr_next, 0);`
	`eWiseApply(pr_next, pr, row_sum, Mul);`

`dangling = (d * dangling + 1 - d)/n;`
`for( int i = 0; i < n; i++ ) {`
`  sum = dangling`
`  for( int j = 0; j < n; j++ ) {`
`  sum += pr_next[i] * L[i][j];`
`  }`
`  pr_nextnext[i] = sum;`
`}`
	`dangling = (d * dangling + 1 - d) / n;`
	`set(pr_nextnext, 0);`
	`vxm(pr_nextnext, pr_next, L, Mul-Add);`
	`foldl( pr_nextnext, dangling, Add);`

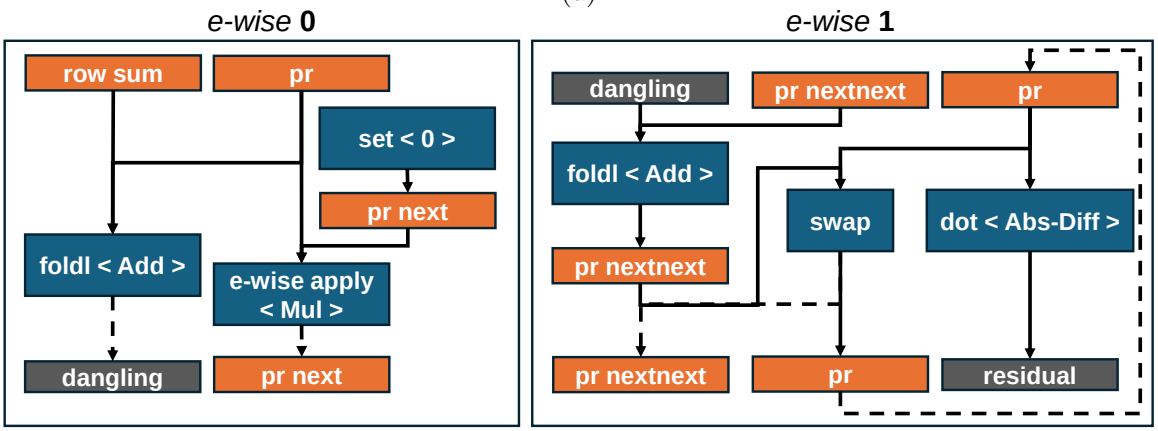
`for ( int i = 0; i < n; i++ ) {`
`  res += fabs(pr[i]-pr_nextnext[i]);`
`}`
	`dot(res, pr, pr_nextnext, add, Abs-Diff);`

`pr = pr_next;`	`swap(pr, pr_nextnext);`

Figure 4.2: Inner loop of `PageRank` algorithm. For simplicity, the c implementation assumes dense tensors.



(a)



(b)

Figure 4.3: Inner loop compute graph of `PageRank` algorithm, (a) abstracted compute graph fusing `e-wise` operations, (b) further break down of `e-wise` operations.

4.2.2 Architectural support to accelerate STA applications

To accelerate dataflow-based STA application like `PageRank` shown in Fig. 4.3, we identify several key architectural supports missing from existing solutions.

First, supporting and accelerating configurable *semiring* tensor operations is a must. STA applications implemented in frameworks like GraphBLAS require a larger set of semiring operators (similar to algorithms mentioned in Table. 3.2). Prior sparse accelerators, such as GAMMA [183] for scientific applications, SCNN [132] for sparse DNNs, optimize the data reuse within a STA operator, but only support multiply-add as the basic computation. SIMD^2 [188] extends dense tensor accelerator for general semiring computation, but no prior work in sparse accelerators has the required semiring support.

Moreover, simply combining ideas in optimizing intra-operator reuse and SIMD^2 does not capture the full reuse opportunity in Fig. 4.3. To fully capture the inter-operator reuse, the sparse architecture should support an explicit data staging between operators. ISOSceles [179] proposes hardware support to capture such producer-consumer reuse, but only for sparse CNNs. ALP and Graphblas's nonblocking execution method lets the programmer exploits producer-consumer reuse of STA applications in CPUs, but the lack of an explicit buffer control in hardware hinders the programmer to exploit the reuse opportunity. Such hardware support is similar to prior accelerators [135] for dense tensor dataflow graphs, but need to specialize for the dataflow and dynamism of STA applications.

Finally, despite that the sparse matrix is very often reused across multiple iterations or stages in STA applications, the footprint of this sparse matrix and the long reuse distance (i.e., cross-iteration) prevent any prior on-chip buffer or cache optimizations to cap-

ture such reuse. To address this, the system needs to store only a small portion of sparse tensors at any time and *executes work in different iterations in a short time window* to exploit the potential reuse across iterations, ensuring that stored data is quickly consumed to make room for new data. Therefore, the system must closely monitor the on-chip buffer and schedule work to maximize reuse.

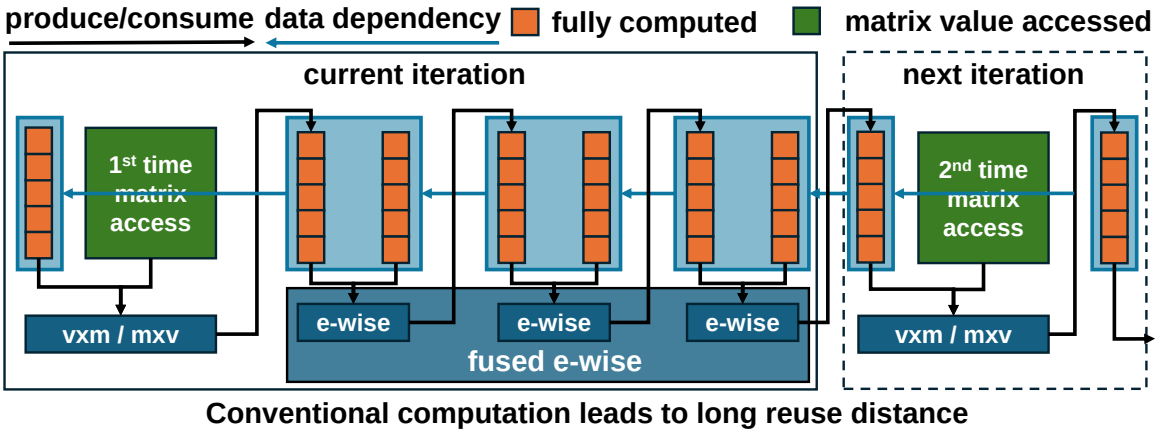
Implementing all above required support in purely software can be both challenging and inefficient, negating the potential benefits of inter-operator data reuse. These opportunities and challenges motivate us to propose the OEI dataflow and develop the SIDA architecture.

4.3 Exploiting cross-iteration data reuse

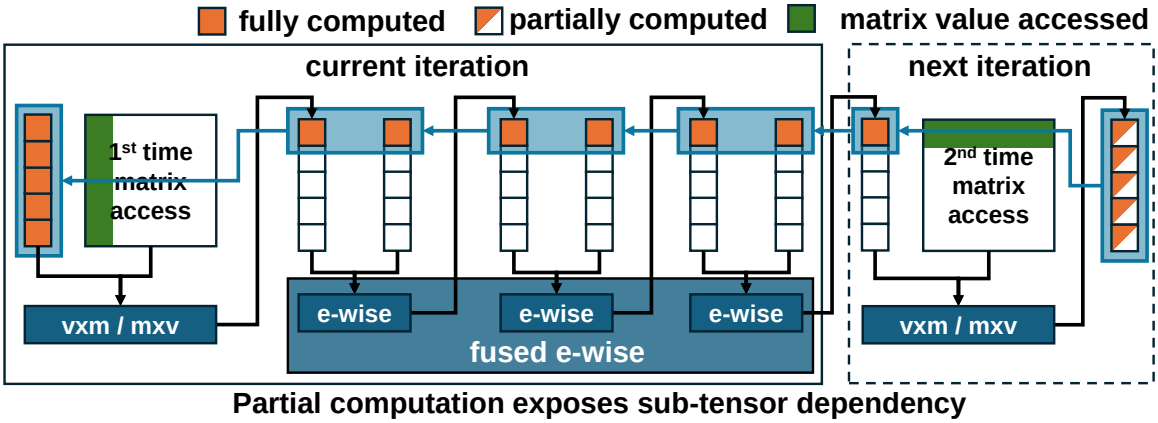
This section details a novel dataflow tailored for cross-iteration data reuse. We start by formulating an abstract view of STA applications to identify cross-iteration data reuse.

4.3.1 Abstracting sparse algorithms

STA algorithms typically can be decomposed into two components: leading matrix (e.g.,`vxm`/`mxv`) operations and subsequent `e-wise` operations. By fusing all `e-wise` operations, the originally complex compute graph simplifies, revealing clear data dependencies between input and output tensors.



(a)



(b)

Figure 4.4: Generalized compute graph of STA applications. (a) Data dependencies of STA application using conventional computation. (b) Data dependencies of STA application using partial computation.

To exploit cross-iteration reuse, a dataflow schedule must simultaneously execute operations spanning multiple loop iterations. That is, to reuse the input sparse matrix, it is crucial that the `vxm` and fused `e-wise` from the current iteration are fused with the `vxm` of the subsequent iteration.

However, conventional dataflow schedule of STA applications executes operators sequentially (i.e., `vxm` has to finish before `e-wise` starts). Such schedule leads to a long reuse distance between two consecutive `vxm` operations. Figure 4.4 (a) shows the unrolled compute graph of an arbitrary STA application including `vxm` in two iterations. To provide the input vector of the second `vxm`, the current iteration needs to access the entire input matrix for the first `vxm` to produce the output vector, and fully compute fused `e-wise` operations. Data dependencies on the entire output vector lead to long reuse distance, preventing efficient fusing of two `vxm` operations.

Fortunately, for STA in dataflow representation, the loop traversal order is hidden from the programmer. The system can optimize the schedule arbitrarily and performs partial computation, so long as it acknowledge the finest-data dependency (as small as a scalar). Figure 4.4 (b) demonstrates the advantage of partial computation, which reveals finer granularity of data dependencies. If the schedule aims to produce *just a single input element* for the subsequent `vxm`, the current iteration only needs to partially access the input matrix and computes corresponding elements of fused `e-wise` operation. We define such finer data dependency as **sub-tensor dependency**.

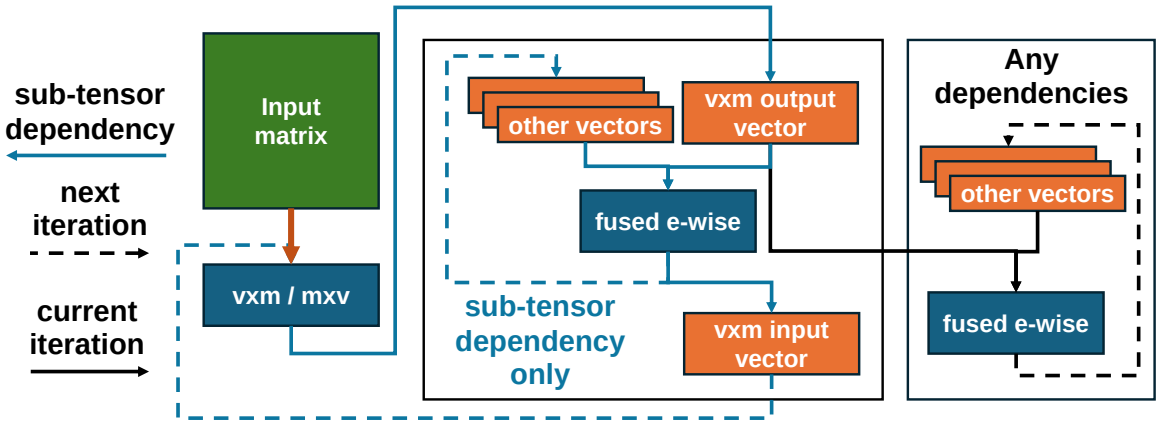


Figure 4.5: Abstracted STA compute graph with isolation of sub-tensor dependency only region.

Sub-tensor dependency reduces the reuse distance of two consecutive `vxm` operations, as the top-left element of the sparse matrix can be reused after the first `vxm` access only a column of the sparse matrix, instead of the full matrix. Isolating the **sub-tensor dependency-only-region** of the dataflow thus reveals cross-iteration data reuse opportunity in any STA applications. Figure 4.5 shows the generalized STA compute graph after fusing `e-wise` operations. For any STA compute graph, if there exists a subgraph that includes both input and output vector of `vxm`, and all operations within the subgraph exhibit sub-tensor dependency, fusing two `vxm` can leverage cross-iteration data reuse. For example, in `PageRank`, a valid subgraph can be structured by `vxm` $\rightarrow$ `e-wise` 1 $\rightarrow$ `e-wise` 0 $\rightarrow$ `vxm`, which exposes sub-tensor dependency for all operations fused in `e-wise` 1 and `e-wise` 0.

Several other structurally different compute graphs also reveal the benefits of this generalized STA abstraction. KNN (K-nearest neighbors), as shown in Figure 4.6, incorporates two `vxm` (or `mxv`) within the same iteration. Despite its unique structure, the circular

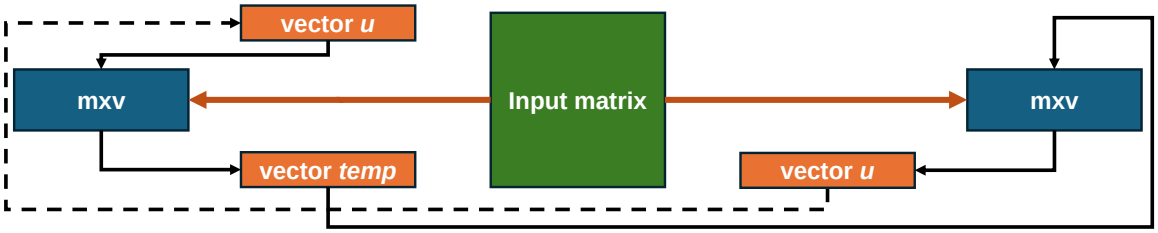


Figure 4.6: Inner loop compute graph of `KNN`.

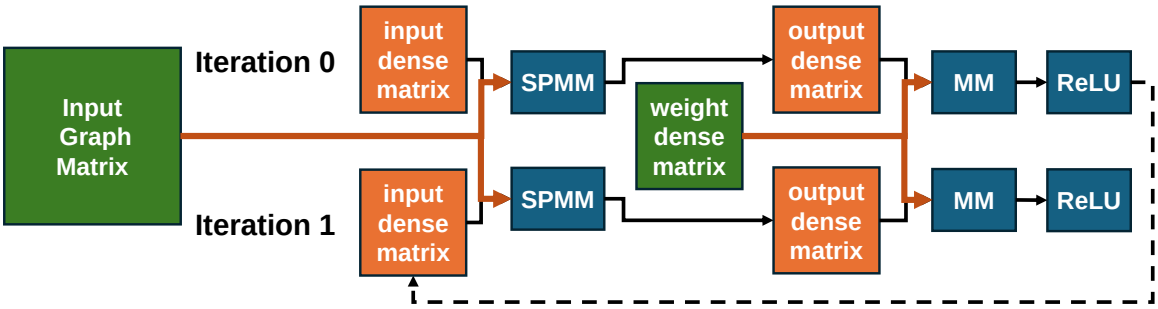
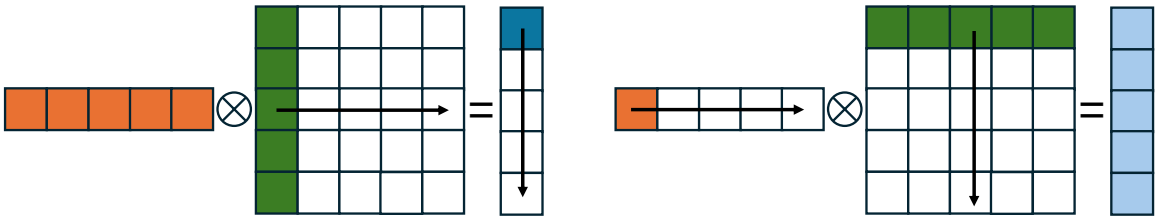


Figure 4.7: Inner loop compute graph of `GCN`.

dependency between the two `vxm` across iterations forms a subgraph: `vxm` → `no-op` → `vxm`, making reuse of input matrix possible.

In addition, as shown in Figure 4.7, Graph Convolutional Neural Networks (GCNs) can be represented as subgraphs of `SpMM` → `MM` (Dense Matrix Multiplication) → `ReLU`. Since no value in the input dense matrix is blocked by `MM` and `ReLU`, and `SpMM` can be implemented as multiple `vxm`, it is possible to fuse SpMM operations from different stages to exploit cross-iteration data reuse.

Based on this observation, so long as an STA algorithm can be abstracted with the generalized compute graph, cross-iteration data reuse applies regardless of whether the fused operations occur within a single loop iteration or span across multiple iterations.



(a) (b)

Figure 4.8: `vxm` dataflow: (a) Output stationary `vxm` dataflow. (b) Input stationary `vxm` dataflow.

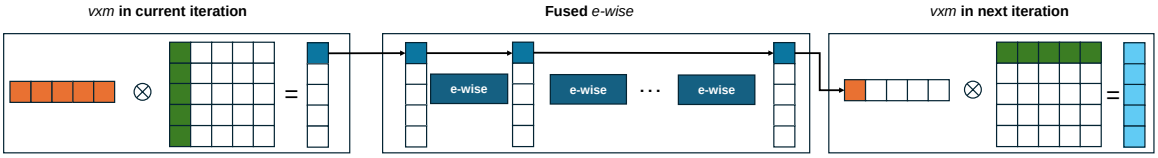


Figure 4.9: Overview of OEI dataflow, fusing OS `vxm` and IS `vxm`.

4.3.2 OEI dataflow

Within the isolated subgraph exposes sub-tensor dependency, the subsequent `vxm` must execute concurrently with the preceding `vxm` without causing any blockages. Given that all other operations within the subgraph do not interfere with each other, any output generated by the earlier `vxm` can be immediately consumed by the subsequent `vxm`.

However, single `vxm` can only choose stationarity between the input vector or the output vector. As illustrated in Figure 4.8, `vxm` operations exhibit two prevalent compute dataflows: (a) Output Stationary (OS) dataflow, which generates a single element in the output vector at a time, requiring access to all input vector elements, and (b) Input Stationary (IS) dataflow, which yields partial results for all output vector elements but requires only a single input vector element at a time.

Conventional implementations of STA algorithms adopt one dataflow type across all iterations, which prevents the cross-iteration data reuse of multiple `vxm`. For instance, when fusing two `vxm` with OS dataflow, the first `vxm` produces one output element at a time, but the second `vxm` requires entire vector output from the first `vxm` to start. Namely, the first `vxm` does no expose sub-tensor dependency with the second `vxm`. Conversely, when choosing IS dataflow for both `vxm`, the input vector elements for the second `vxm` are not fully available until the completion of the first `vxm`, which also prevents the sub-tensor dependency between two `vxm`.

To address this, **our insight is to employ OS dataflow for the first `vxm` and IS dataflow for the second `vxm`.** This mixed dataflow meets the necessary condition to reuse the data from the sparse matrix. As depicted in Figure 4.9 (a), the first `vxm` with the OS dataflow generates an output vector element, subsequently consumed by the fused `e-wise` operation. The element produced by `e-wise` operations can be directly consumed by the second `vxm` with the IS dataflow, without any hindrance from preceding operations.

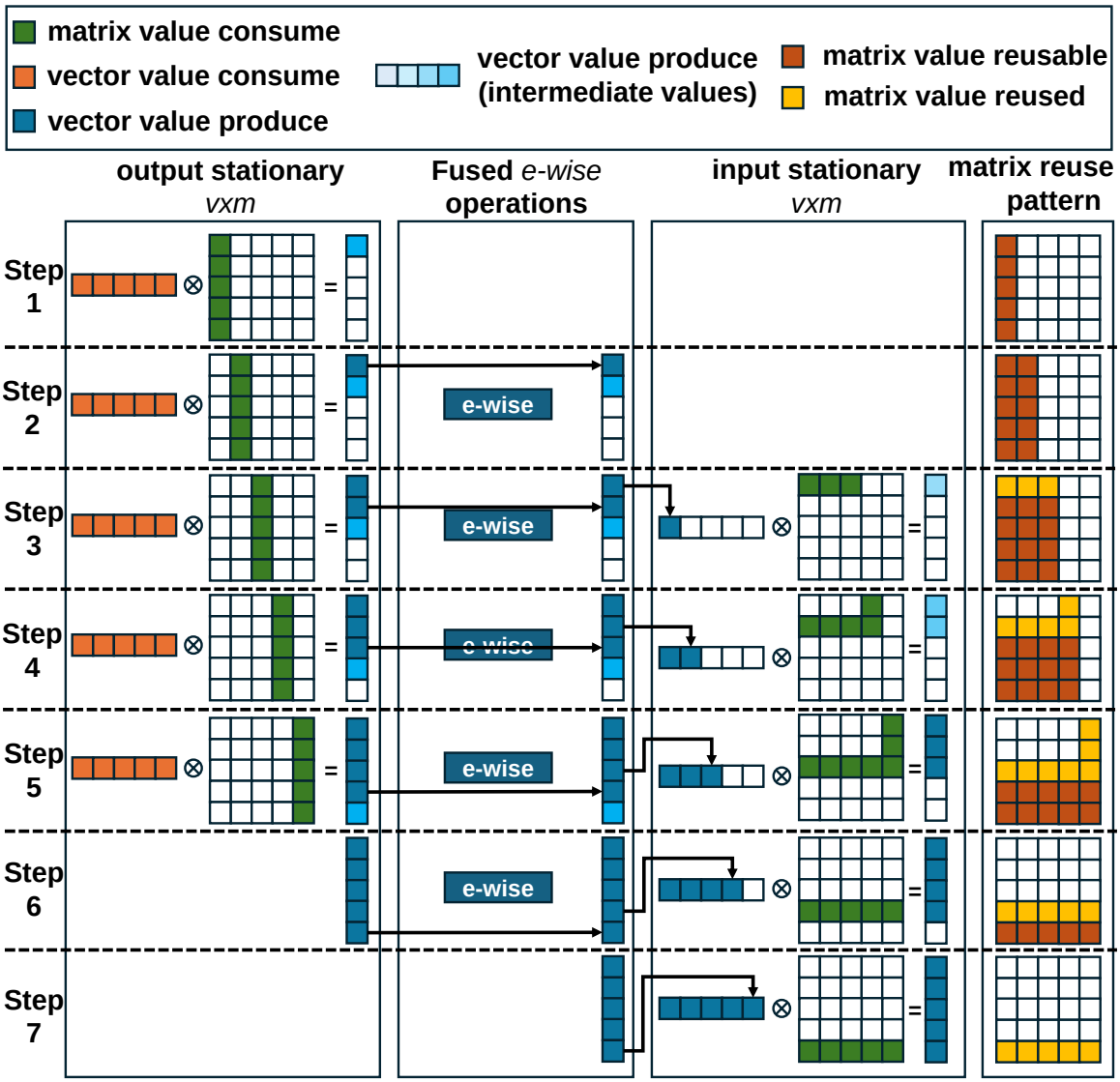


Figure 4.10: Illustration of OEI dataflow for dense matrices.

We name such dataflow as OEI (OS-ewise-IS) dataflow, which facilitates the simultaneous execution of operations in the subgraph, enabling the reuse of large input matrices in two `vxm` across iteration.

Table 4.1: Portion of sparse matrix need to be stored on-chip to enable OS-ewise-IS dataflow (smaller % is better)

matrix	**row/col**	**nnz**	**max (%)**	**avg (%)**
ca	18772	198110	98802 (49.9%)	65124 (32.9%)
gy	17361	178896	8661 (4.8%)	3321 (1.9%)
g2	150102	438388	15448 (3.5%)	7304 (1.7%)
co	434102	16036720	2143362 (13.7%)	1155196 (7.2%)
bu	513351	10360701	9329007 (90%)	4944897 (47.7%)
wi	3566907	45030389	17422630 (38.7%)	10450514 (23.2%)
ad	6815744	13624320	1143568 (9.4%)	694064 (5.1%)
ro	23947347	28854312	557694 (1.9%)	281769 (1.0%)
eu	50912018	54054660	2338567 (4.3%)	1419430 (2.6%)

Fig. 4.10 demonstrates the OEI dataflow with a 5×5 dense matrix as an example. In each step, the OS `vxm` computes one output vector element by accessing the entire input vector and a single column of the input matrix. The fused `e-wise` is delayed by one step relative to the OS `vxm` because the input vector elements required for `e-wise` are not fully available until the OS `vxm` completes its previous step. Similarly, IS `vxm` lags two steps behind the OS `vxm`, as it awaits the completion of both the OS `vxm` and the fused `e-wise`. IS `vxm` scatters the partial sum from the multiplication for each pair of elements, where the matrix value has been previously use by OS `vxm`. Therefore, the IS `vxm` avoids the computation of the full outer product in each step.

To show the data reuse, the matrix reuse pattern in Fig. 4.10 indicates matrix values that are either ready for reuse or are currently being reused. Figure 4.9 (b) highlights the matrix reuse pattern for selected execution steps after applying the OEI dataflow to a sparse matrix. For extremely sparse input matrices in STA applications, only a small portion of the matrix is required to be buffer at a time, potentially fitting within a standard on-chip buffer.

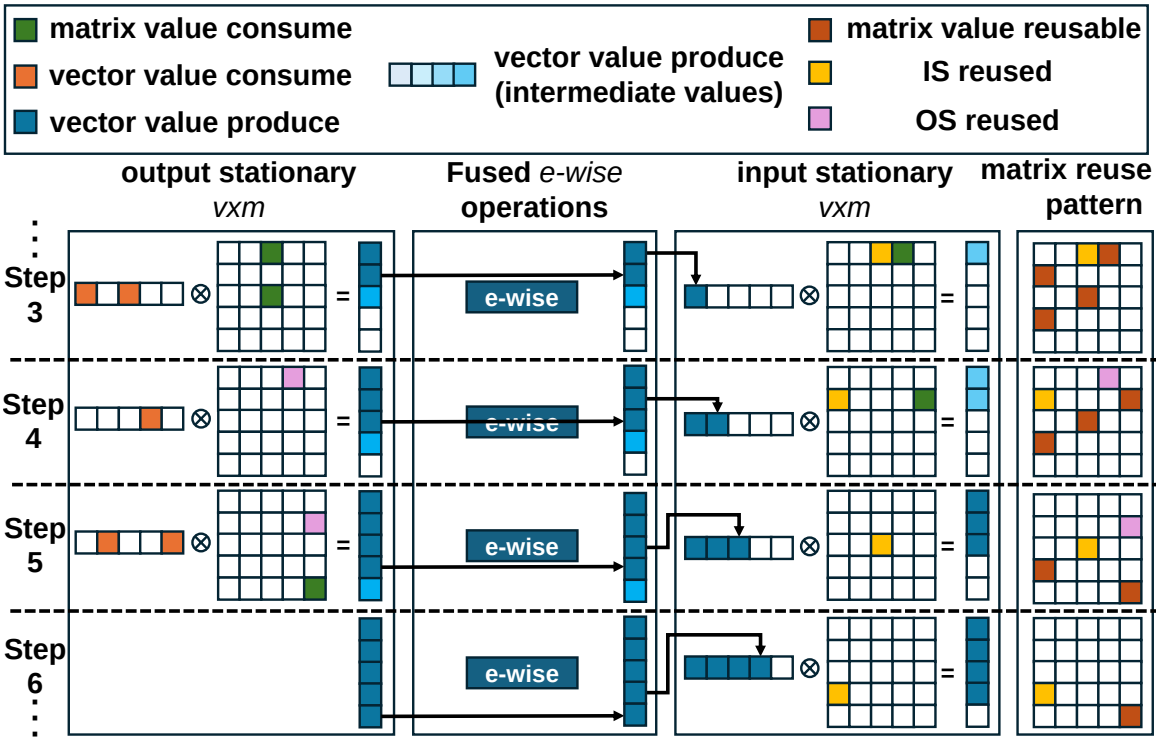


Figure 4.11: Illustration of OEI dataflow for sparse matrices with eager IS execution.

Table 4.1 demonstrates our simulation results for the maximum and average percentage of the nonzero values in a sparse matrix to be stored on-chip using the OEI dataflow. For a vast majority of the examined matrices, maintaining only a small fraction (<10%) of values is sufficient to capture the reuse opportunity in the OEI dataflow.

When the on-chip buffer can hold all the reusable matrix values across all steps, the IS `vxm` need not to load any matrix element from memory. However, given the uneven distribution of non-zero values in sparse matrices, it becomes challenging to ensure that OS `vxm` uniformly loads data from the main memory in each step.

This uneven data distribution can lead to load imbalance between OS and IS stages and under-utilization of memory bandwidth in certain steps. To address this, figure 4.11 shows an enhancement to the OEI dataflow, using the same sparse matrix referenced in Figure 4.9 (b). In step 3, the initial element of the IS `vxm` input vector is generated by fused `e-wise` operations. Besides computing a partial sum with reusable matrix elements, the IS `vxm` also proactively loads another matrix value from memory, instead of idling. The matrix value loaded by IS `vxm` is instead reused by OS `vxm` in step 4, eliminating the need for additional main memory loading. Similar dynamics occur between steps 4 and 5. This refined approach addresses the issues of load imbalance and bandwidth utilization.

4.4 Sparse Inter-operator Dataflow Architecture

We now presents SIDA, an dataflow architecture that exploits the reuse opportunities of sparse tensor algebra. This section will overview the proposed architecture and describe key components and optimization in the software framework.

4.4.1 Overview of SIDA microarchitecture

Figure 4.12 presents the high-level functional blocks of SIDA. SIDA efficiently supports sparse tensor algebra while enabling data reuse opportunities in the following aspects.

- Dual sparse storage in on-chip buffers to efficiently accommodate various data access demands in different stages of OEI dataflow.
- A dynamic scheduling pipeline that natively supports OEI dataflow and sparse tensor

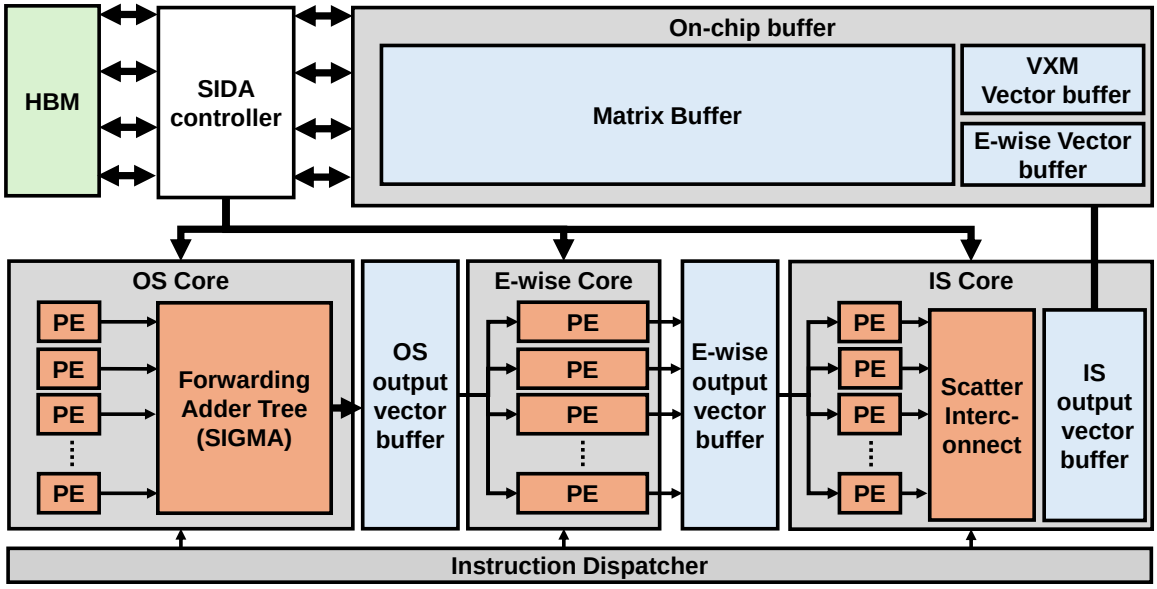


Figure 4.12: High level architecture of SIDA: Pipelined OS Core, IS Core, and E-Wise Core share an on-chip buffer.

semiring operations through compute cores, dedicated to the demand of each stage in the OEI dataflow: the OS Core, E-Wise Core, and IS Core.

- A dataflow-aware controlling logic that schedules memory accesses, dispatches computation tasks, and prefetches data in units of sub-tensors in the control logic to efficiently use memory bandwidth and buffer space.

SIDA leverages the compiler behind existing STA programming frameworks to generate efficient tensor semiring instructions. SIDA and a SIDA-compliant language framework partition data and schedule computation tasks in sub-tensors to more efficiently use memory bandwidth and reduce memory footprint. SIDA initiates a stream of computation tasks from loading input sub-tensors that typically contain multiple columns of data for the OS Core and propagates the intermediate sub-tensors to the E-Wise Core. In the

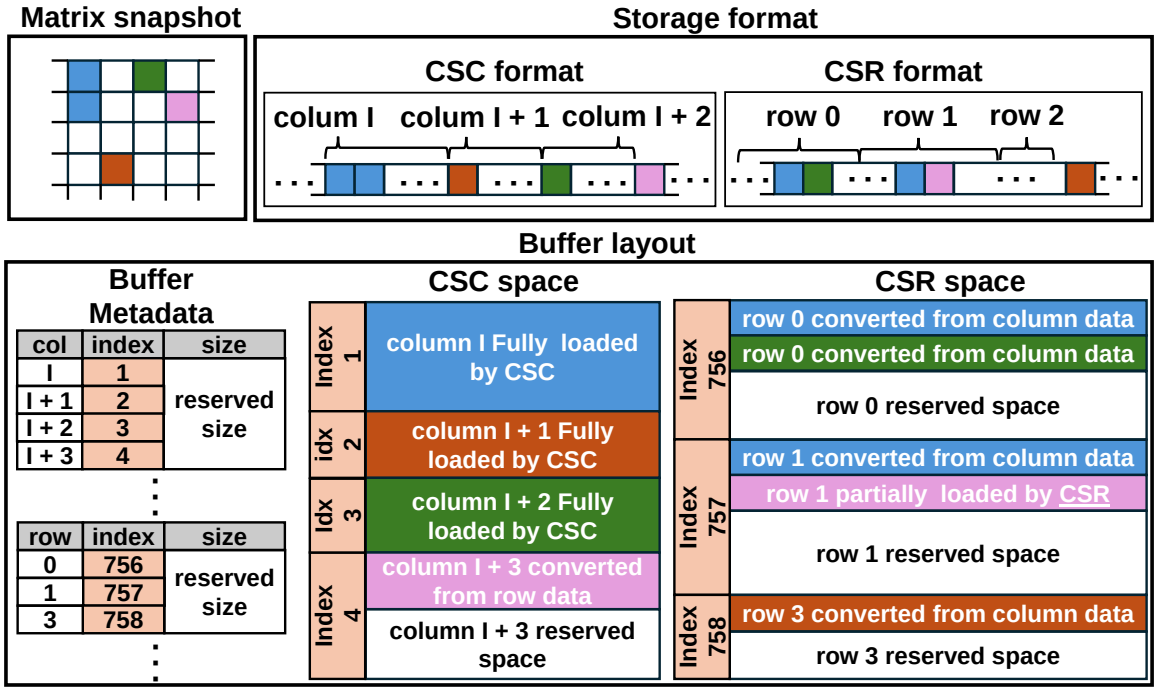


Figure 4.13: Dual sparse storage and memory layout of SIDA on-chip buffer

meantime, SIDA can initiate another stream of computation tasks by loading another set of sub-tensors into the buffer. SIDA can perform OS operations for the later stream as soon as the previous stream advanced to the E-wise operations. By executing streams of sub-tensor computation tasks in a pipeline manner among OEI compute cores, SIDA exploits pipeline parallelism and cross-iteration data reuse opportunities. In addition to the architectural supports and optimizations, the language framework can further optimize data structures that facilitate program execution and improve space efficiency. In the rest of this section, we will describe these architectural and software components.

4.4.2 Dual sparse storage

In SIDA architecture, both the OS and IS Cores require sub-tensor inputs from the same sparse tensor but with opposite traversal orders. Specifically, the `vxm` operations in OS Cores need sub-tensors from the input matrix in column order for the desired semiring operations with the input vector. In the meantime, IS `vxm` demands row-wise access for scatter multiplication with a matrix row to yield partial outputs. A design option of the on-chip buffer is to cache input tensor elements using an orientation-neutral format like coordinate Format (COO) that does not favor column-based or row-based access patterns; such a design can only ensure efficient access for the sorted dimension.

Therefore, SIDA's on-chip buffers store input sub-tensors in a dual storage strategy that utilizes both CSC and CSR formats to optimize data access for both OS and IS dataflows to address that limitation that no single sparse matrix storage format optimally supports both row and column data access simultaneously. Figure 4.13 depicts the high-level idea of the dual storage on-chip buffers. Each buffer contains a CSC space for data in CSC format and CSR space for data in CSR format.

As CSC format consecutively places column data between *col_start* and *col_end* of the coordinate and data arrays, each *val* shares the same *col_idx* but exhibits unique *row_coord*. SIDA stores the same column consecutively in the on-chip buffer, providing straightforward access for OS Cores. The practice of consecutive fetching and storing column data fetched from CSC format seamlessly extends to managing row data fetched from CSR format. Storing row data fetched from CSR format into the CSR space follows a similar strategy.

As data in the same column always belongs to unique rows, the on-chip buffer must store the converted row data non-consecutively in the CSR space the original input is in CSC format. To address this, SIDA determines the necessary space for each row using *row_start* - *row_end* from the CSR index array, reserving space upon receiving the first converted row data from CSC data. Specifically, when column data (*col_idx, row_coord, val*) from CSC format is converted to row data as (*row_coord, col_idx, val*), equivalent to (*row_idx, col_coord, val*) in CSR format, SIDA calculates and reserves the required space for newly fetched rows, assigning a buffer index for subsequent access.

As STA applications demand column data in the order from lower to higher *col_idx*, the row data in which an earlier fetching operation brings should always have lower *col_coord* compared the later ones. Therefore, the first non-zero element of any row can always trigger space reservation in advance, allowing for consecutive and ascending storage of subsequently fetched row data within its reserved space. Additionally, when row data are eagerly loaded from the CSR format to utilize the remaining memory bandwidth, they are converted to column data and stored in the CSC space following the same logic.

4.4.3 The OEI compute pipeline

The compute pipeline of SIDA features the OS Core, the E-Wise Core, and the IS Core for efficiently executing the OEI dataflow. Fig. 4.14 presents the runtime exmaple of SIDA pipeline.

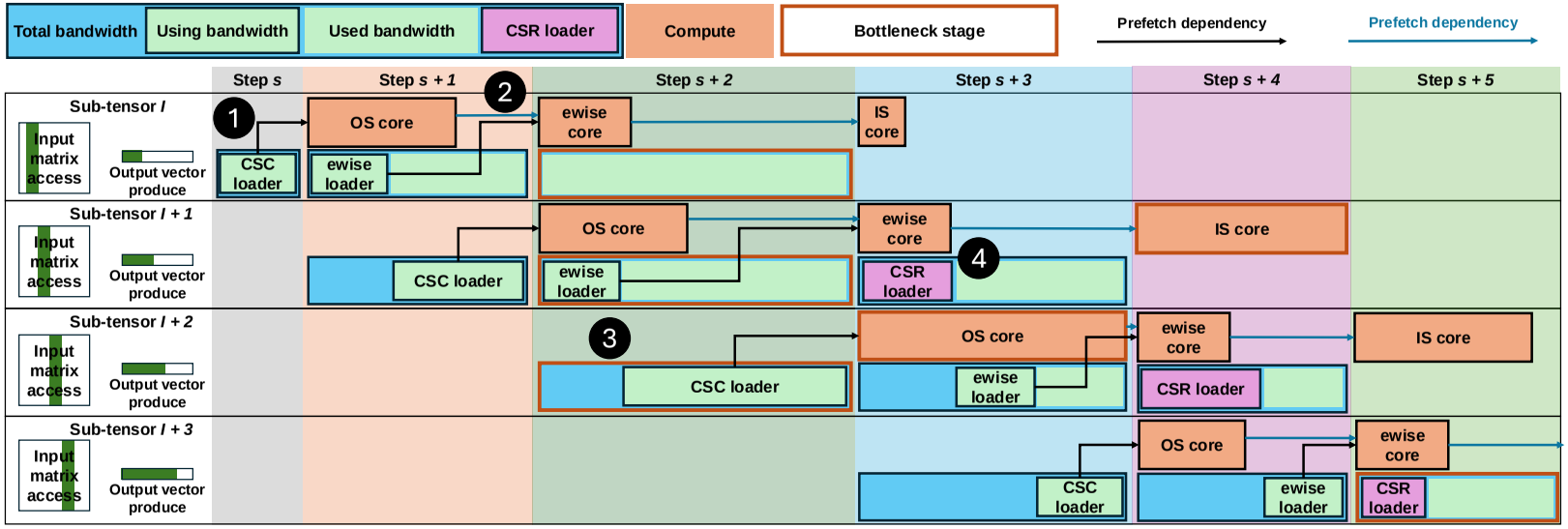


Figure 4.14: SIDA controll logic

OS Core

The OS Core executes operations on each matrix column and vector pair like dot-products. Beyond the `mul-add` operation, SIDA extends each processing element (PE) to additionally support frequently used semiring operations in sparse tensor algebra, including `and-or`, `min-add`, `aril-add`. Each PE in the OS Core can execute of a semiring operation on a sub-tensor/column simultaneously with other PEs. SIDA uses the Forwarding Adder Tree from SIGMA [137] to handle the varying number of non-zero elements per column, allowing flexible sets of PEs to communicate during the reduction phase of each vector-column dot product. The OS Core stores the generated intermediate sub-tensors (vectors) in a buffer for the later stage.

E-Wise Core

E-Wise Core processes `e-wise` operations on the OS `vxm`'s output buffer, where the sub-tensor size directly corresponds to the number of elements managed by the E-Wise Core in each step of OEI dataflow. The E-Wise Core processes sub-tensor elements concurrently in Single Instruction, Multiple Data (SIMD) model. SIDA uses offline compilation to pre-generate instructions for fused `e-wise` operations specific to an application. Each PE in the E-Wise Core is identical to the PEs in the OS Core. E-Wise Core stores the results of sub-tensors into the e-wise vector buffer.

IS Core

The IS Core performs outer product calculation, where a single vector element multiplies against multiple row elements and accumulates with intermediate values gener-

ated in E-Wise Cores. SIDA includes an output vector buffer to store partial results from the IS Core and employs a scatter network to integrate the most recent products. SIDA writes each fully computed output vector element back to the main memory. Similar to OS Core, IS Core configures semiring operators specific to each application before execution, eliminating any additional runtime overhead.

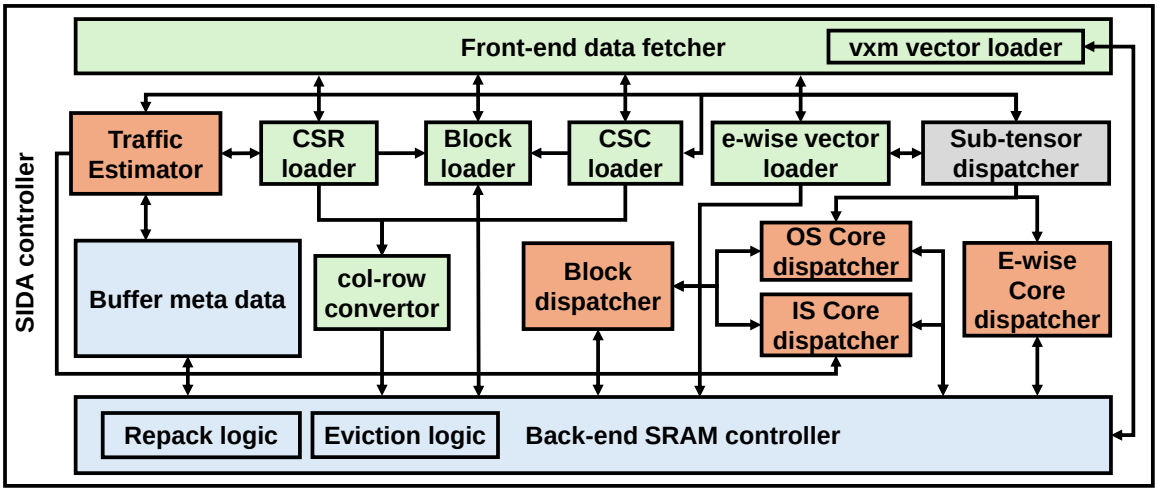


Figure 4.15: SIDA controll logic

4.4.4 Control logic

As memory bandwidth limits the roofline of STA applications, the control unit in SIDA targets utilizing all available memory bandwidth beyond just controlling compute resource operations. SIDA's controller serves the following purposes:

- Estimating bottleneck component of SIDA at each step within the OEI dataflow.
- Managing the selection of CSC/vector sub-tensors for loading in the forthcoming steps of the OEI dataflow.
- Determining when to load CSR data to maximize memory bandwidth utilization.
- Regulating the compute phase for each step of the OEI dataflow.

Pipeline control

The pipeline control logic generates control signals for each datapath element in the OEI pipeline. Unlike OS `vxm` and `e-wise`, which operates on determinent sub-tensor in each step of OEI dataflow, the runtime behavior of the IS `vxm` can vary based on several conditions: (1) SIDA opts to load CSR data only when there's leftover bandwidth. (2) The IS dataflow limits its computation to element-row scatter multiplication exclusively for the rows in the on-chip buffer. (3) Scatter multiplication computations by the IS dataflow are contingent upon the completion of OS `vxm` and `e-wise`.

Figure 4.16 shows the concept of SIDA's pipeline control. The sub-tensor dispatcher orchestrates the control stage by maintaining a state machine that tracks the index I. The sub-tensor dispatcher generates signals based on I to coordinate other components immediately before initiating the next step. Assuming $I-2$ represents the sub-tensor fully processed by both the OS and element-wise operations, the sequence for the upcoming step is as follows:

- Index I is allocated to sub-tensors for element-wise operation execution.
- Index $I+1$ is designated for sub-tensors under OS dataflow processing and loading of element-wise vectors.
- Index $I+2$ indicates CSC data that will be prefetched.

Initially, the sub-tensor dispatcher sends the index $I+2$ to the traffic estimator to calculate the total non-zero count of CSC data required for the next step by accessing a buffered CSC index array stored in the buffer metadata. Leveraging this non-zero count, the traffic

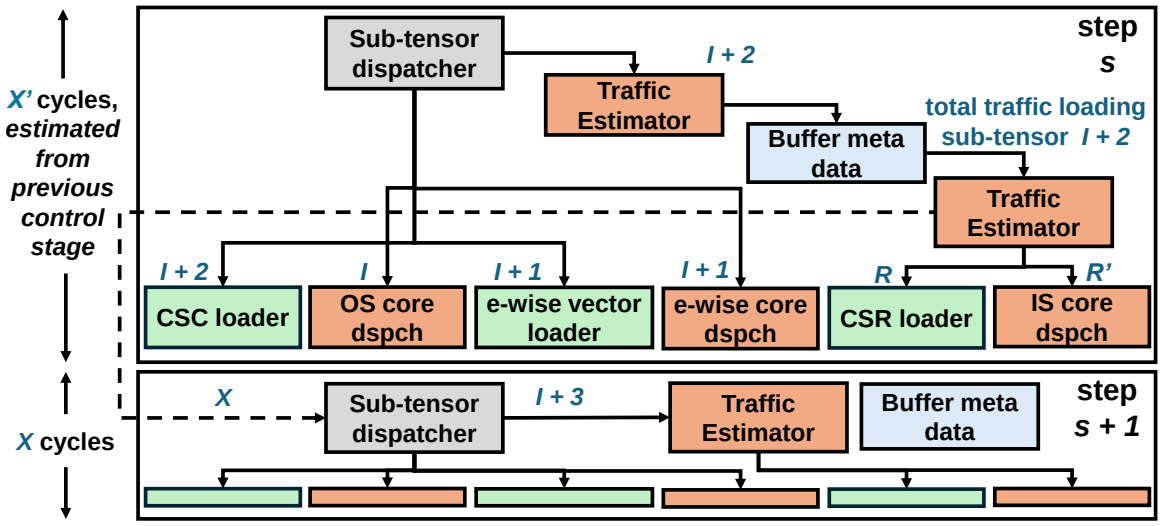


Figure 4.16: Runtime behavior of pipeline control, sub-tensor index assignment to each of SIDA component.

estimator computes the cycle count X, by comparing the loading latency of CSC and element-wise data against compute latency of OS operations and E-wise operations. If the computation time determines the X value, the traffic estimator will proceed to calculate R, the quantity of row data that the IS Core can fetch to optimize memory bandwidth utilization in the subsequent step. Sub-tensor dispatcher issues the relevant sub-tensor indexes to other components after X' cycles, as estimated by the preceding control stage.

The pipeline control sets the operations of each core unit using the following rules. At any given step s, if the E-Wise Core dispatcher receives sub-tensor index I, then the OS Core dispatcher receives sub-tensor index $I+1$, as the OS dataflow naturally progresses one step ahead of element-wise operations. The prefetch stage ensures all necessary data are prefetched and stored in the on-chip buffer, allowing the OS and E-Wise Core dispatchers

to directly access the SRAM controller and transmit data to the OS and E-Wise Cores, respectively. When using a sub-tensor of size T, the OS Core dispatcher loads columns and vector elements with indexes ranging from $(I + 1) * T$ to $(I + 2) * T$, whereas the E-Wise Core dispatcher loads elements of e-wise vectors with indexes ranging from $I * T$ to $(I + 1) * T$. On the other hand, IS Core dispatcher, theoretically, can eagerly compute scatter multiplication up to the s-th rows. However, to prevent bottlenecks in the IS Core from affecting other stages and to align with data prefetched by the CSR loader, the IS Core conservatively fetches up to R (received from the traffic estimator) row data in all rows along with the corresponding vector elements.

Sub-tensor loading and prefetching

SIDA contains three dedicated data loaders for each phase of the OEI dataflow: the CSC loader for the OS Core, E-wise vector loader, and CSR loader for the IS Core. Both the CSC and e-wise vector loaders directly receive sub-tensor indexes from the sub-tensor dispatcher and generate fetch commands for CSC and vector inputs, respectively. The loader issues commands to the HBM if a demanding portion of the sub-sensor does not exist in on-chip buffers.

Conversely, the CSR loader may receive a parameter R from the traffic estimator, indicating the quantity of necessary sub-tensor data. As on-chip buffer stores all row data in consecutive and ascending from each row's first non-zero element, the CSR loader can access all rows eligible for IS `vxm` computation from the buffer. Suppose memory bandwidth saturates due to the loading of CSC data for the OS dataflow and vector data for E-wise operations. In that case, the IS Core will prioritize computing scatter multiplications solely

on data already fetched or converted to CSR format. When bottlenecks arise during the compute phase and accesses to the CSC and vector data under-utilizes the available memory bandwidth, SIDA prioritizes loading CSR data to mitigate this imbalance.

SIDA also prefetches CSR data heuristically. For each row r has an index greater than the highest fully computed row index S by the IS `vxm`, yet smaller than the fully computed index E, the CSR loader decides the number of elements to prefetch per row by considering (a) the total number of fetched row data does not surpass R and, (b) for each row r, where $S < r < E$, the CSR loader calculates $P(r) = \frac{\sum_{i=S}^{r} T(i)}{r}$, where $T(i)$ denotes the count of already fetched row data for row i. This heuristic ensures that (a) the loaded row data fully utilizes the remaining memory bandwidth, and (b) load balance of IS `vxm`. The CSR loader then issues fetch commands for each row r to retrieve consecutive data starting from the next non-zero at *col_coord* to *col_coord*+$P(r)$.

If a column in CSC space has reserved its space but a newly converted column data is not the next non-zero element the computation task needs, SIDA discards this conversion. Even if storing the converted column were feasible, any not-requested data with a lower column index would lead to separate fetches, potentially causing the CSC loader to initiate multiple commands for non-consecutive data in the same column, thus diminishing memory utilization efficiency.

Upon receiving a fetch command from CSC and CSR loaders, the front-end data fetcher retrieves the data and computes a set of *vxm* vector indexes by intersecting all received *row_idx* and *row_coord*. It retrieves the necessary vector data from memory and stores them in the on-chip buffer. Lastly, the e-wise vector loader directly issues fetch commands since e-wise PEs use all e-wise data in a single step of the dataflow.

Data eviction and repacking

OEI dataflow facilitates data reuse by only storing a fraction of the input matrix in the on-chip buffer. SIDA further reduces the buffer size required with an efficient eviction policy and a buffer repacking mechanism.

In CSC space, SIDA immediately evicts entire column data consumed by the OS Core, freeing up reserved space without delay. Using sub-tensor execution further prevents fragmentation by fetching and evicting multiple columns concurrently.

Conversely, since IS Core operations consume each data element in a row individually, SIDA maintains additional metadata for each stored row to monitor the total count of consumed elements. Upon surpassing a predetermined threshold of total consumed elements, the controller initiates a buffer repacking process that discards fully computed sub-tensors and places remaining sub-tensors in a contiguous CSR space.

When SIDA encounters Out-Of-Memory (OOM) conditions without any available repacking opportunities, SIDA adheres to the reuse patterns outlined in Figure 4.10, prioritizes eviction for rows with higher *row_idx*. The control logic will reload the evicted row when later steps of OEI dataflow need that row.

4.4.5 Sparse tensor preprocessing

SIDA implements two offline optimizations designed to preprocess input sparse matrices.

Row reorder

SIDA employs row reordering to enhance the locality of the non-zero distribution of sparse matrices. As any converted row data may trigger CSR space reservation, string too much unconsumable row data with high *row_idx* causes frequent Out-Of-Memory in the on-chip buffer, leading to memory ping-ponging in later steps. SIDA favors fetching row data with higher *row_idx* in later steps of the OEI dataflow.

Like prior works optimizing the SpMSpM operation [183], SIDA utilizes the GraphOrder algorithm [170] to rearrange rows for input data. Additionally, SIDA incorporates a straightforward vanilla reorder algorithm as an alternative, which aims to reorder the sparse matrix towards an upper triangular matrix with simple heuristics.

Blocked sparse storage

SIDA also adopts blocked sparse storage to reduce the storage overhead due to dual sparse storage. Dual sparse storage has two main drawbacks: (a) CSC and CSR formats use redundant data arrays. (b) Overhead in indexing structure as each coordinate requires at least 4 bytes. Using the FiberTree notation proposed in Sparseloop [176], SIDA uses UOP-CP-CP format. Specifically, each value in the data array of CSC and CSR points to a non-zero block of the original sparse matrix, which offers two advantages: (a) A single byte can store a coordinate within any block that has a size up to 256; (b) quantity of non-zero

blocks is significantly less than non-zero values, allowing CSR and CSC format to have less redundancy when storing coordinate and data array.

Beyond storage efficiency, blocks of non-zeros also provide SIDA performance improvements for specific input matrices and applications. Leveraging these benefits, SIDA opts for blocked sparse storage. When utilizing blocked storage, CSC and CSR loaders additionally transmit the loaded block ID to the block loader, which then facilitates the Front-end data fetcher in loading the required non-zero blocks. Similarly, in the compute stage, OS and IS Core dispatchers load prefetched non-zero blocks and unpacked row/column data that compute cores will later use for processing.

4.5 Methodology

This paper evaluates SIDA through a custom-built simulator and a set of STA algorithms. This section highlights the simulated configurations and workloads.

4.5.1 Modeling SIDA

We developed a simulation framework to assess the performance of SIDA, accommodating various system configurations. The framework simulates an SIDA architecture with 1024 PEs for each compute core, a 64 MB on-chip buffer, and communicating with the on-device DRAM at 512GB/s bandwidth.

We evaluated the energy consumption of compute units and memory components using Cacti [115] and Accelergy [175] with the Aladdin [148] plug-in. We also scale the dynamic energy consumption based on factors reported in prior work [72].

Table 4.2: Benchmark STA applications.

Algorithm	`vxm` **Semiring**	**Reuse Pattern**	**Domain**
PageRank (pr)	Mul-Add	cross-iteration, producer-consumer	Graph Analytics
Kcore Decomposition (kcore)	Mul-Add		
Breadth First Search (bfs)	And-Or		
Single Source Shortest Path (sssp)	Min-Add		
Kmeans Initialisation (kpp)	Aril*-Add		Clustering
K-Nearest Neighbors (knn)	And-Or		
Label Propagation (label)	Mul-Add		Machine Learning
Graph Convolutional Neural network (gcn)	Mul-Add		
Generalized Minimal Residuals (gmres)	Mul-Add		Solver, HPC
Conjugate Gradient (cg)	Mul-Add	producer-consumer	
Biconjugate Gradients Stabilized (bgs)	Mul-Add		

4.5.2 Evaluated STA algorithms and systems

We evaluate a total of 10 applications from ALP/GraphBLAS. Table 4.2 lists these applications, including their semiring operations, data reuse patterns, and application domains. Eight applications can leverage cross-iteration data reuse. We also include two applications that benefit solely from inter-operator data reuse to assess the performance of SIDA on applications without OEI dataflows. 4 out of 10 applications employ graph analytics algorithms, while the remaining six power scientific computing and machine learning applications.

To thoroughly investigate the performance across all applications, we selected all nine representative sparse matrices, each with unique row/column size, sparsity, and non-zero distributions. Table 4.1 provides the list of these datasets.

We compare the performance of SIDA with two other configurations. (1) CPU-based system with large on-chip memory. We run the same workloads and datasets and collect performance counter numbers from a machine with AMD 5800X3D CPU, featuring a 96 MB 3D stacked V-cache and 128GB of dual-channel DDR4 main memory with measured memory bandwidth at 44 GB/s. (2) An idealized sparse accelerator (baseline) that utilizes the same compute and memory bandwidth as SIDA, but does not exploit inter-operator data reuse. **This idealized sparse accelerator always has the throughput as its roofline**, representing the upper bound of the prior sparse accelerator. Additionally, we chose `PageRank` as a benchmark application to compare SIDA's performance against an NVIDIA 4070 GPU with memory bandwidth at 504 GB/s.

4.6 Experimental Result

4.6.1 Performance over an idealized sparse accelerator

SIDA achieves up to 2.18 × in end-to-end latency over the baseline accelerator across all benchmark applications. For applications with OEI dataflow presented, SIDA achieves a geometric mean speedup ranging from 1.23× to 2.56×, with the highest speedup reaching 3.55×. Figure 4.17 detailed the speedup of end-to-end latency in each application.

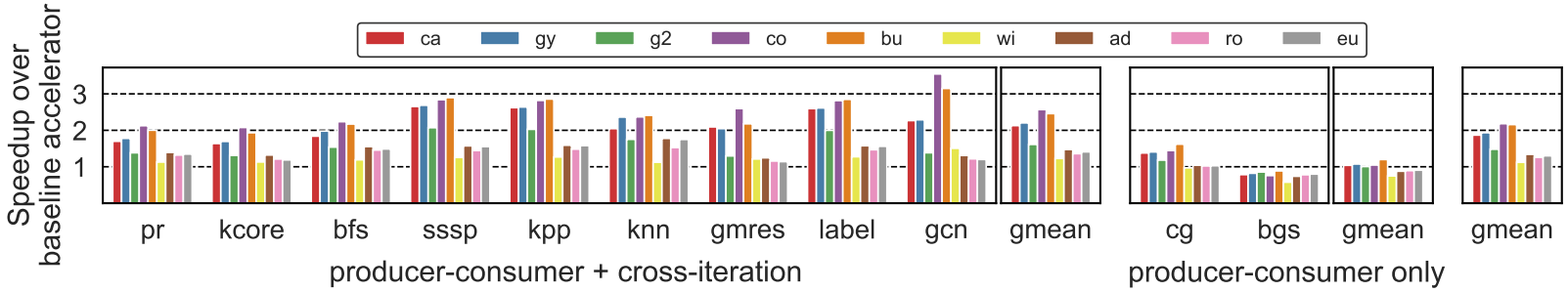


Figure 4.17: Speedup of SIDA over baseline accelerator.

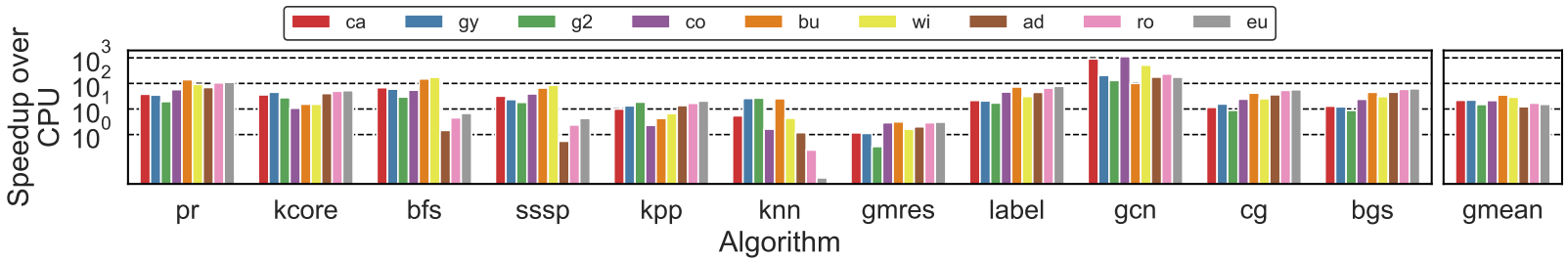
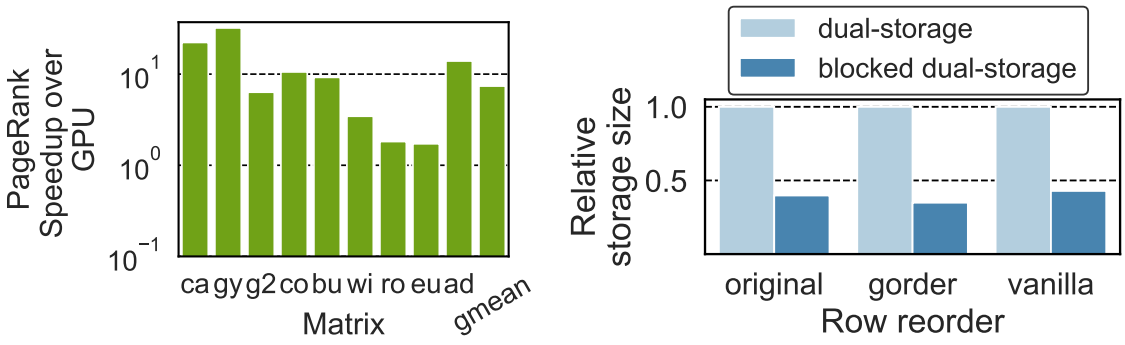


Figure 4.18: Speedup of SIDA over CPU implementation of STA algorithms.



(a) (b)

Figure 4.19: (a) GPU case study. (b) Storage improvement of blocked format.

Despite the baseline accelerator in Figure 4.17 always delivering performance at the roofline of each STA operation, the baseline does not exploit inter-operator data reuse nor data reuse in OEI dataflow. Even for applications without OEI dataflow (i.e., cg and bags), SIDA still achieves the same level speedup as the baseline accelerator, ranging from 0.75× to 1.20× as SIDA can still exploit producer-consumer reuse in these applications.

4.6.2 Performance over CPU and GPU implementations

Figure 4.18 compares SIDA with implementations using ALP/GraphBLAS running on a multicore CPU-based STA framework. The CPU implementations exploit non-blocking execution patterns for producer-consumer data reuse. In contrast, SIDA benefits from both OEI dataflow and a higher memory bandwidth utilization. Excluding graph convolution neural networks (GCN), where SIDA also benefits from dp4a-like instructions, SIDA achieves up to a 174.37× speedup. Across all applications and matrices, SIDA per-

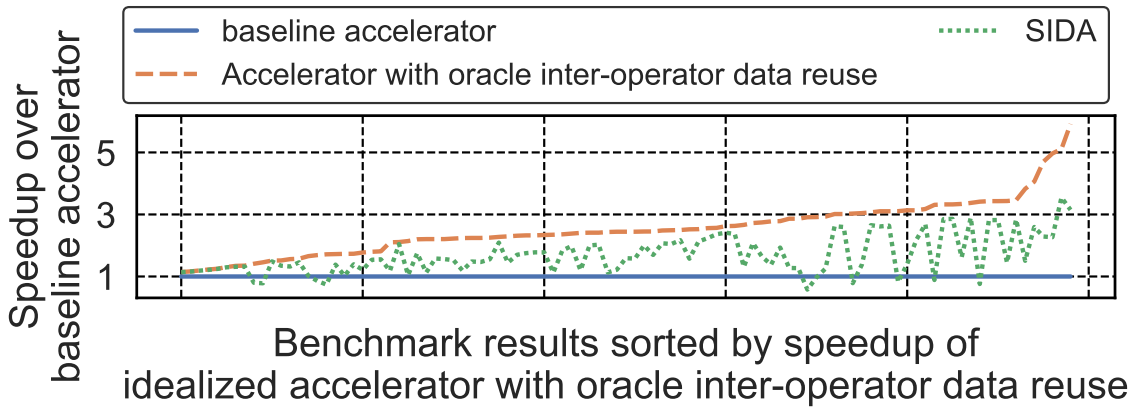


Figure 4.20: The performance of SIDA compared with an accelerator with perfect inter-operator reuse.

forms 12.28× to 35.16× better, surpassing the theoretical benefit ratio of 11.6× on system configuration with higher memory bandwidth.

While our focus primarily lies in presenting evaluation results compared to accelerator-based approaches, we also chose the PageRank algorithm as a representative benchmark to study SIDA's advantages over GPUs. We utilized the implementation from Graph-BLAST [178], a GPU-based STA framework. As Figure 4.19 (a) demonstrates, SIDA achieves up to a 32.08× speedup.

4.6.3 Effectiveness in exploiting cross-iteration data reuse

To evaluate the effectiveness of SIDA in exploiting cross-iteration and OEI dataflow data reuse opportunities, we modeled an oracle STA accelerator that assumes that all elements of the input sparse matrix are always ready when reuse opportunities across iterations present, fully exploiting all inter-operator data reuse opportunities irrespective of on-chip buffer size. Such an oracle accelerator presents the theoretical performance upper limit for

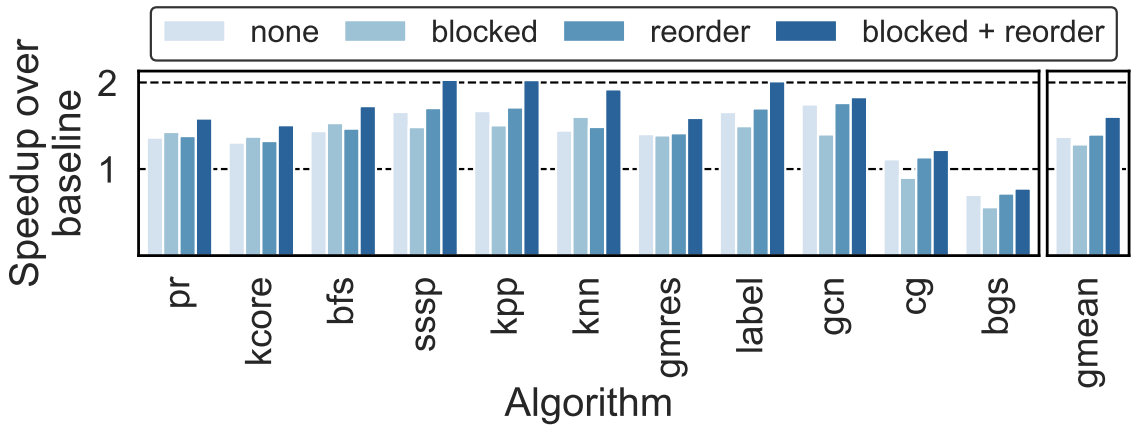


Figure 4.21: Sensitivity study for the benefit of data optimization.

STA applications. As shown in Figure 4.20, on average, SIDA achieves 66.78% of the oracle accelerator's performance, utilizing only a 64MB on-chip buffer to process sparse matrices as large as 1.3GB (with 64-bit datatype).

4.6.4 Impact of sparse tensor preprocessing

The accelerator architecture of SIDA presents a skeleton of STA accelerators. Without any optimization, Figure 4.21 shows that SIDA can still achieve 1.37× speedup over the baseline accelerator.

Encoding data using the blocked sparse format can help SIDA to improve performance by up to 1.11×. Employing solely the optimal row reorder technique slightly boosts SIDA's performance from 1.01× to 1.03×. With both optimizations increasing the locality of non-zero values, SIDA can have a more efficient data access pattern during the OS-wise-IS dataflow, leading to 1.10× to 1.33× speedup from the SIDA without data optimizations.

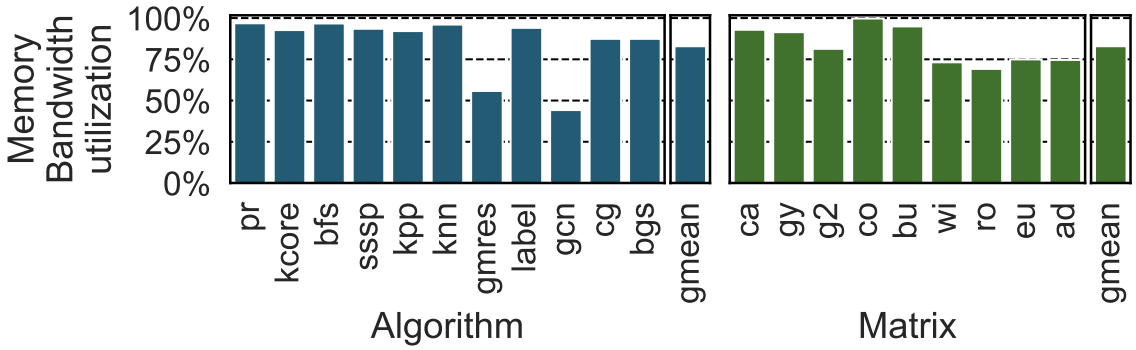


Figure 4.22: Bandwidth utilization of SIDA, geometric mean across algorithms and sparse matrices.

Using the blocked sparse format reduces the size of the dual sparse storage of input matrices. Figure 4.19 (b) illustrates the storage benefits of the blocked dual sparse format. Regardless of the reorder technique employed, blocked formats enhance the storage efficiency of dual storage, decreasing the storage requirements to 39.2% of the non-blocked dual-storage format.

4.6.5 Memory bandwidth utilization in SIDA

SIDA can effectively use available memory bandwidth for bandwidth-sensitive STA applications. Figure 4.22 shows that SIDA maintains 82.93% memory bandwidth utilization. When considering only naturally memory-bound applications (excluding *gmres* and *gcn*), SIDAfficiently utilized a geometric mean of 92.94% of system memory bandwidth.

4.6.6 Energy savings with SIDA

The primary advantage of SIDA lies in reducing the memory traffic of STA applications through cross-iteration data reuse. Since memory operations dominate the energy

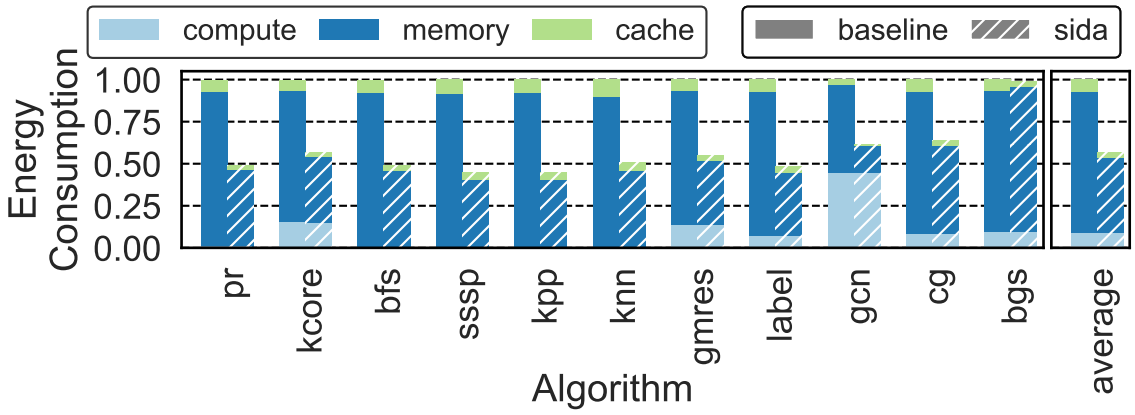


Figure 4.23: Relative energy consumption of SIDA separating compute, memory, and cache operations.

consumption for a significant portion of the selected STA applications, SIDA achieves a significant energy saving. It reduces energy consumption by an average of 55.11% across all applications compared with the baseline accelerator. Specifically, SIDA saves 50.38% of energy on memory operations and 38.35% on cache/on-chip buffer operations.

4.6.7 Area Estimation of SIDA

The simulated configuration resembles the size of a consumer grade, mid-tier GPU. We estimated SIDA's area using RTL and synthesized it with the Synopsys Design Compiler using the 45 nm technology library and scaled the design to the Samsung 8N process. Under the identical hardware configurations as our performance simulation, SIDA takes 348mm^2, close to that of an RTX 3070 GPU. The on-chip buffer contributes 200mm^2 of the chip area.

4.7 Conclusion

Reducing data movement and maximizing data reuse are the key to accelerate STA applications. While prior work exploits solely intra-operator reuse, this paper identifies two other inter-operator reuse opportunities in sparse dataflow graphs, consumer-producer reuse and cross-iteration reuse. We further propose the OEI dataflow to capture inter-operator reuse, and the SIDA architecture to support the OEI dataflow and maximize reuse with limited buffer space. Our evaluation shows that SIDA is significantly faster than CPU/GPU for STA applications, and exploiting inter-operator reuse provides notable advantage even over an idealized sparse accelerator.

Chapter 5

Related works

This section will provide a comprehensive review of related work pertaining to all democratization techniques explored in this thesis. Specifically, it will delve into existing literature and research efforts aimed at expanding the applicability of tensor processors by addressing challenges related to data precision, operator diversity, and sparse matrix computation. By examining prior work in these areas, this section will establish a solid foundation for understanding the current state-of-the-art and identifying potential avenues for further innovation.

Mixed-precision Mixed-Precision Fused Multiply-Add Vector Units M^3XU distinguishes itself from existing multi-precision Fused Multiply-Add (FMA) floating-point units [22, 50, 66, 67, 103, 104, 157, 185] as M^3XU 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 [31, 46, 54, 91, 92, 133, 168, 182, 192] and corresponding accelerator designs [26,40,47,59,74,145,149,174,180,194] 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 [26, 47, 145, 149]. Furthermore, several prior projects propose arbitrary precision support to enable various sizes of data programmer-transparently [40]. 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 [37, 37, 93, 153] or they have to avoid complex number arithmetic but use software-based approximation techniques [61, 126, 127] or implement a separate accelerator or FPGA acceleration [85].

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 [9]. 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 func-

tional units for more purpose. Therefore, existing automatic synthesization/optimization techniques cannot achieve these non-trivial extensions that M^3XU presents.

Matrix extensions and instructions Instruction-level support for matrix-matrix multiplication can be dated back to the 90s. MOM [29] proposes to leverage MXU to accelerate multi-media applications. As neural networks become one of the most critical workloads, commercial general processors now also include matrix instructions as well as MXUs to accelerate tiled-matrix-multiplication. NVIDIA Tensor Core [121, 124], Intel AMX [69], and Arm SME [12] all provide instructions for GEMM. Our SIMD^2 architecture is compatible with these prior work and modern designs. SIMD^2 reuses the existing hardware and software infrastructure to accelerate matrix operations beyond GEMM.

Dense tensor accelerators SIMD^2 builds on top of recent dense tensor accelerators for matrix-multiplication [25, 73, 74, 96, 121, 159] to efficiently share data across datapath and reduce the bandwidth requirement of SIMD^2 instructions. While we implement our SIMD^2 microarchitecture using systolic-array-like hardware structure, other matrix-multiplication accelerator architecture, such as the IBM MMA [159] unit, can be extended to support SIMD^2 instructions. In addition to matrix-multiplication, prior work also proposes accelerators for other dense linear algebra algorithms with different data sharing patterns. For example, Weng et al. [171] propose a hybrid systolic-dataflow architecture for inductive matrix algorithms (e.g., linear algebra solver). Tithi et al. [161] propose a spatial accelerator for edit distance algorithms. While these algorithms have a different data sharing pattern than SIMD^2 instructions support, we expect they can be implemented as *CISC-SIMD*^{2} instructions with variable latency. We nonetheless leave this extension to prior work.

Graph algorithm accelerators While many graph algorithms can be expressed as tensor operations and linear algebra [80] and accelerated by tensor accelerators, prior work has also proposed hardware accelerators to speed up graph algorithms and analytics in their classic form. Graphicionado [53], GraphR [152], GraphP [186], and GraphQ [195] leverage processing-in-memory (PIM) architecture to alleviate the bandwidth bottleneck in graph algorithms. PHI [114] and HATS [113] instead enhance conventional multi-core processors to accelerate common operations in graph analytics, such as commutative reduction and traversal scheduling. These hardware acceleration techniques focus on leveraging properties in graph algorithms to reduce data movement and bandwidth requirement. In contrast, SIMD^2 proposes a new instruction set for tensorized graph algorithms to leverage tensor accelerators ubiquitous in all compute platforms.

Sparse tensor algebra framework and compiler In addition to ALP and GraphBlast, there are serveral system software proposals to assist STA programming. For example, the Sparse Abstract Machine [64] argues for an Einsum-based programming model to generate hardware accelerator for STA. Sparse MLIR [19], TACO [28, 84], and COMET [116, 160] provide compiler support to generate performant, format-optimized implementations across CPU/GPUs. SIDA shares the same tensor abstraction programming model as these prior proposals, and our proposed OEI dataflow could be implemented in the framework as a software solution, albeit the potential overhead in buffer and work management.

Sparse dataflows and accelerators Much prior work in sparse accelerator has focused on intra-operator reuse for SpMSpM. This includes DRT [128], OuterSPACE [4], SpArch [189], MatRaptor [154], ExTensor [57], and Gamma [183]. SIDA instead explores inter-operator reuse and is therefore orthogonal to these prior accelerators. Other work in sparse accelerators also focuses on reconfigurable hardware and specialized storage format to accelerate STA. For example, ALRESCHA [13] accelerates SpMV, SMASH [76] targets both SpMV and SpMM, and Flexagon [112] accelerates SpMSpM in DNNs. Qin et. al [136] proposes flexible hardware to support multiple sparse formats. SIGMA [137] and MAERI [3] exploits configurable network for sparse DNNs. Symphony [134] proposes a coarse-grained reconfigurable architecture for sparse and dense tensor algebra. Since they still optimize for a single sparse operator, the proposed OEI dataflow can be combined to these prior accelerators.

Exploiting inter-operator reuse Prior work in DNNs has investigated how to exploit producer-consumer data reuse in dense tensor algebra. On the one hand, prior research has proposed special dataflows to optimize for producer-consumer reuse. For example, FLAT [77] optimizes for the multi-head attention kernel in Transformers, Fused-Layer CNN Accelerators [7] optimized for for CNNs. Some other work, such as Pipelayer [2], TANGRAM [5], ARCHON [129], and Atomic dataflow [191], proposes to optimize producer-consumer reuse across spatial hardware units and optimize for the whole DNNs. On the other hand, as optimizing for fusion requires a large design space search, prior work has also explored tools and modeling techniques for fusion. Convfusion [166], DNNFuser [78], MultiFuse [24], and Stream [155] provide optimization tools to automatically search for the best fusion decisions. SET [23] and LoopTree [48] propose polyhedral abstraction to rea-

son about fusion decision more effectively. All techniques above optimizes for dense tensor algebra, while SIDA targets STA instead.

Exploit specific sparse tensor applications Finally, very recent work has also looked into producer-consumer reuse for a specific STA domain. For example, ISOSceles [179] targets sparse CNNs, GOGETA [44] targets Conjugate Gradient (CG) in HPC, and Raveesh et. al [45] exploits pipeline dataflows in GNNs. These techniques specialize for only the targeted STA domain. In contrast, SIDA supports a wide range of STA applications and exploits cross-iteration reuse, a new type of data reuse.

Chapter 6

Conclusions and Future Work

This thesis presents a comprehensive exploration of three key avenues for democratizing tensor processors, primarily focusing on their application in accelerating deep neural networks within the realm of artificial intelligence and machine learning (AI/ML). In addition to accelerating NNs, recent projects have demonstrated the strong potential of using NN/MMA accelerators for a broader spectrum of applications. Both Tensor Cores and TPUs can help improve the performance of linear algebra beyond GEMM [52, 63], database queries [32,62,65], cryptography [88] and scientific computing problems [37,39,93, 98–100,111,119]. Ray tracing accelerators are also useful for Monte Carlo simulations [143], robotics navigation [109] and nearest neighbor search problems [193]. However, due to the domain-specific nature of these accelerators, programmers have to intensively re-engineer the algorithm implementation to make use of these hardware accelerators. The resulting program may also incur overhead when transforming data structures to fulfill the demand of the target accelerator. By extending the hardware features, SIMD^2,M^{3}XU provides better

programmability to reduce the overhead of remapping algorithms and allows applications that are not possible on conventional NN/MMA accelerators. With hardware accelerators lifting the roofline, a critical issue is designing a memory hierarchy that streamlines the data inputs/outputs for computational logic. The overarching aim of this thesis is to contribute to the democratization of domain-specific architectures (DSAs). This involves elevating their programmability and extending their computational capabilities to a wider array of applications, thereby benefiting diverse scientific fields through the advancements of modern computer architecture.

Akin to the evolutionary trajectory of GPUs, which transitioned from specialized graphics accelerators to versatile general-purpose GPUs (GPGPUs) now bolstering nearly all scientific applications, many DSAs could also find broader utility if their functionality and programmability were enhanced and their "killer applications" identified. This necessitates a dual effort: re-examining classic algorithms and applications to uncover potential for acceleration, and efficiently designing new microarchitectures with minimal resource expenditure.

In fact, modern CPUs and GPUs already demonstrate this approach. x86 architectures have integrated vector (SSE and AVX) and matrix (AMX) instructions alongside dedicated hardware units, while streaming multiprocessors now feature specialized components for real-time ray tracing (RT cores) and neural network acceleration (Tensor Cores). These distinct components are seamlessly integrated into the original, straightforward designs through shared memory hierarchies and unified programming models, all driven by the need for acceleration in specific domains.

As research delves deeper into democratization strategies, we envision the emergence of more unified architecture designs that cater to a broader range of scientific domains. This thesis explores several promising research directions and areas within this landscape, with the ultimate goal of inspiring further investigation and encouraging contributions from experts in the field. The hope is that these collective efforts will lead to the development of more accessible and versatile DSAs, fostering innovation and accelerating progress across a multitude of scientific disciplines.

Bibliography

[1] BLAS (Basic Linear Algebra Subprograms). `http://www.netlib.org/blas/`, 2004.

[2] Pipelayer: A pipelined reram-based accelerator for deep learning. In *EEE International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '17. IEEE Press, 2017.

[3] Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. In *ACM SIGPLAN Notices*. IEEE Press, 2018.

[4] Outerspace: An outer product based sparse matrix multiplication accelerator. In *IEEE International Symposium on High Performance Computer Architecture*, HPCA, '18. IEEE Press, 2018.

[5] Tangram: Optimized coarse-grained dataflow for scalable nn accelerators. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19. IEEE Press, 2019.

[6] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[7] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer CNN Accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

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

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

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

[11] 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.

[12] 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.

[13] Bahar Asgari, Ramyad Hadidi, Tushar Krishna, Hyesoon Kim, and Sudhakar Yalamanchili. ALRESCHA: A Lightweight Reconfigurable Sparse-Computation Accelerator. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

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

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

[16] Vignesh Balaji, Neal Crago, Aamer Jaleel, and Brandon Lucia. P-opt: Practical Optimal Cache Replacement for Graph Analytics. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[17] Vignesh Balaji, Neal C Crago, Aamer Jaleel, and Stephen W Keckler. Community-based Matrix Reordering for Sparse Linear Algebra Optimization. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 214–223. IEEE, 2023.

[18] 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.

[19] Aart Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. Compiler Support for Sparse Tensor Computations in MLIR. *ACM Trans. Archit. Code Optim.*, 19(4), 2022.

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

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

[22] 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.

[23] Jingwei Cai, Yuchen Wei, Zuotong Wu, Sen Peng, and Kaisheng Ma. Inter-layer Scheduling Space Definition and Exploration for Tiled Accelerators. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, 2023.

[24] C. Chang, J. Liou, C. Huang, W. Hsu, and J. Lu. MultiFuse: Efficient Cross Layer Fusion for DNN Accelerators with Multi-level Memory Hierarchy. In *2023 IEEE 41st International Conference on Computer Design (ICCD)*, 2023.

[25] Karam Chatha. Qualcomm® Cloud Al 100: 12TOPS/W Scalable, High Performance and Low Latency Deep Learning Inference Accelerator. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, 2021.

[26] 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.

[27] Jack Choquette, Olivier Giroux, and Denis Foley. Volta: Performance and Programmability. *IEEE Micro*, 38(2):42–52, 2018.

[28] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Format Abstraction for Sparse Tensor Algebra Compilers. *Proc. ACM Program. Lang.*, 2(OOPSLA), 2018.

[29] Jesus Corbal, Roger Espasa, and Mateo Valero. MOM: a matrix SIMD instruction set architecture for multimedia applications. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, 1999.

[30] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[31] 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.

[32] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu. Accelerating reduction and scan using tensor core units. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, pages 46–57, 2019.

[33] 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.

[34] Timothy A Davis. Algorithm 1000: SuiteSparse: GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Transactions on Mathematical Software (TOMS)*, 45(4):1–25, 2019.

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

[36] 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.

[37] 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.

[38] Jeff Erickson. *Algorithms*. 2019.

[39] 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.

[40] 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.

[41] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, and Tianqi Chen. TensorIR: An Abstraction for Automatic Tensorized Program Optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 804–817, New York, NY, USA, 2023. Association for Computing Machinery.

[42] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, page 345, jun 1962.

[43] 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.

[44] Raveesh Garg, Michael Pellauer, Sivasankaran Rajamanickam, and Tushar Krishna. Exploiting Inter-Operation Data Reuse in Scientific Applications using GOGETA. *arXiv preprint arXiv:2303.11499*, 2023.

[45] Raveesh Garg, Eric Qin, Francisco Muñoz-Matrínez, Robert Guirado, Akshay Jain, Sergi Abadal, José L. Abellán, Manuel E. Acacio, Eduard Alarcón, Sivasankaran Rajamanickam, and Tushar Krishna. Understanding the Design-Space of Sparse/Dense Multiphase GNN dataflows on Spatial Accelerators. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 571–582, 2022.

[46] 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.

[47] 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.

[48] Michael Gilbert, Yannan Nellie Wu, Angshuman Parashar, Vivienne Sze, and Joel S. Emer. LoopTree: Enabling Exploration of Fused-layer Dataflow Accelerators. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023.

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

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

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

[52] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J. Higham. Harnessing gpu tensor cores for fast fp16 arithmetic to speed up mixed-precision iterative refinement solvers. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 603–613, 2018.

[53] 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.

[54] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[55] Pawan Harish, P.J. Narayanan, Vibhav Vineet, and Suryakant Patidar. Chapter 7 - fast minimum spanning tree computation. In Wen mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, pages 77–88. Morgan Kaufmann, 2012.

[56] 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.

[57] 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*, pages 319–333, 2019.

[58] 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.

[59] 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.

[60] 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.

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

[62] Pedro Holanda and Hannes Mühleisen. Relational queries with a tensor processing unit. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, DaMoN'19, 2019.

[63] Kuan-Chieh Hsu and Hung-Wei Tseng. Accelerating Applications using Edge Tensor Processing Units. In *SC: The International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC 2021, 2021.

[64] Olivia Hsu, Maxwell Strange, Ritvik Sharma, Jaeyeon Won, Kunle Olukotun, Joel S. Emer, Mark A. Horowitz, and Fredrik Kjølstad. The Sparse Abstract Machine. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, 2023.

[65] Yu-Ching Hu, Yuliang Li, and Hung-Wei Tseng. TCUDB: Accelerating Database with Tensor Processors. In *the 2022 ACM SIGMOD/PODS International Conference on Management of Data*, SIGMOD 2022, 2022.

[66] 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.

[67] 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.

[68] Sagar Imambi, Kolla Bhanu Prakash, and GR Kanagachidambaresan. PyTorch. *Programming with TensorFlow: Solution for Edge Computing Applications*, pages 87–104, 2021.

[69] 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.

[70] Jeff Hammond. cuASR: CUDA Algebra for Semirings. `https://github.com/hpcgarage/cuASR`, 2021.

[71] Jiacheng Pan. CUDA MST. `https://github.com/jiachengpan/cudaMST`, 2016.

[72] 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. Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2021.

[73] 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.

[74] 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.

[75] 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.

[76] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu. SMASH: Co-designing Software Compression and Hardware-Accelerated Indexing for Efficient Sparse Matrix Operations. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, 2019.

[77] Sheng-Chun Kao, Suvinay Subramanian, Gaurav Agrawal, Amir Yazdanbakhsh, and Tushar Krishna. FLAT: An Optimized Dataflow for Mitigating Attention Bottlenecks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, 2023.

[78] Kao, Sheng-Chun and Huang, Xiaoyu and Krishna, Tushar. DNNFuser: Generative pre-trained transformer as a generalized mapper for layer fusion in dnn accelerators. *arXiv preprint arXiv:2201.11218*, 2022.

[79] Gary J. Katz and Joseph T. Kider. All-pairs shortest-paths for large graphs on the gpu. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pages 47–55, 2008.

[80] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, et al. Mathematical foundations of the graphblas. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9, 2016.

[81] 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.

[82] 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.

[83] 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.

[84] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017.

[85] 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.

[86] 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.

[87] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 1956.

[88] Wai-Kong Lee, Hwajeong Seo, Zhenfei Zhang, and Seong Oun Hwang. Tensorcrypto: High throughput acceleration of lattice-based cryptography using tensor core on gpu. *IEEE Access*, 10:20616–20632, 2022.

[89] 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.

[90] M Leyzorek, RS Gray, AA Johnson, WC Ladew, SR Meaker Jr, RM Petry, and RN Seitz. Investigation of model techniques–first annual report–6 june 1956–1 july 1957–a study of model techniques for communication systems. *Case Institute of Technology, Cleveland, Ohio*, 1957.

[91] 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.

[92] 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.

[93] 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.

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

[95] 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.

[96] Heng Liao, Jiajin Tu, Jing Xia, Hu Liu, Xiping Zhou, Honghui Yuan, and Yuxing Hu. Ascend: a scalable and unified architecture for ubiquitous deep neural network computing: Industry track paper. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[97] Yiqian Liu and Martin Burtscher. ECL-APSP v1.0. `https://userweb.cs.txstate.edu/~burtscher/research/ECL-APSP/`, 2021.

[98] Tianjian Lu, Yi-Fan Chen, Blake Hechtman, Tao Wang, and John Anderson. Large-scale discrete fourier transform on tpus. *IEEE Access*, 2021.

[99] Tianjian Lu, Thibault Marin, Yue Zhuo, Yi-Fan Chen, and Chao Ma. Accelerating mri reconstruction on tpus. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9, 2020.

[100] Tianjian Lu, Thibault Marin, Yue Zhuo, Yi-Fan Chen, and Chao Ma. Nonuniform fast fourier transform on tpus. In *2021 IEEE 18th International Symposium on Biomedical Imaging (ISBI)*, pages 783–787, 2021.

[101] Ben D. Lund and Justin W. Smith. A multi-stage cuda kernel for floyd-warshall. *ArXiv*, abs/1001.4108, 2010.

[102] 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.

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

[104] 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.

[105] 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.

[106] Mateusz Bojanowski. Cuda Floyd Warshall implementation. `https://github.com/MTB90/cuda-floyd_warshall`, 2018.

[107] Guy Melancon. Just how dense are dense graphs in the real world? a methodological note. In *Proceedings of the 2006 AVI Workshop on BEyond Time and Errors: Novel Evaluation Methods for Information Visualization*, BELIV '06, pages 1–7, 2006.

[108] 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.

[109] Heajung Min, Kyung Min Han, and Young J. Kim. Accelerating probabilistic volumetric mapping using ray-tracing graphics hardware. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5440–5445, 2021.

[110] Mehryar Mohri. Semiring frameworks and algorithms for shortest-distance problems. *Journal of Automata, Languages and Combinatorics*, 7(3):321–350, 2002.

[111] Alan Morningstar, Markus Hauru, Jackson Beall, Martin Ganahl, Adam G. M. Lewis, Vedika Khemani, and Guifre Vidal. Simulation of Quantum Many-Body Dynamics with Tensor Processing Units: Floquet Prethermalization. *arXiv preprint arXiv:2111.08044*, 2021.

[112] Francisco Muñoz Martínez, Raveesh Garg, Michael Pellauer, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. Flexagon: A Multi-dataflow Sparse-Sparse Matrix Multiplication Accelerator for Efficient DNN Processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, 2023.

[113] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling. In *the 51st Annual IEEE/ACM international symposium on Microarchitecture (MICRO)*, 2018.

[114] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. PHI: Architectural Support for Synchronization-and Bandwidth-Efficient Commutative Scatter Updates. In *the 52nd Annual IEEE/ACM international symposium on Microarchitecture (MICRO)*, 2019.

[115] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing NUCA Organizations and Wiring alternatives for Large Caches with CACTI 6.0. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, 2007.

[116] Erdal Mutlu, Ruiqin Tian, Bin Ren, Sriram Krishnamoorthy, Roberto Gioiosa, Jacques Pienaar, and Gokcen Kestor. Comet: A Domain-specific Compilation of High-performance Computational Chemistry. In *International Workshop on Languages and Compilers for Parallel Computing*, 2020.

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

[118] 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.

[119] Ricardo Nobre, Aleksandar Ilic, Sergio Santander-Jiménez, and Leonel Sousa. Exploring the binary precision capabilities of tensor cores for epistasis detection. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 338–347, 2020.

[120] 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.

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

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

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

[124] NVIDIA Corporation. NVIDIA T4 TENSOR CORE GPU. `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-t4/t4-tensor-core-datasheet-951643.pdf`, 2019.

[125] NVIDIA Corporation. Warp Level Matrix Multiply-Accumulate Instructions. `https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#warp-level-matrix-instructions`, 2021.

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

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

[128] Toluwanimi O. Odemuyiwa, Hadi Asghari-Moghaddam, Michael Pellauer, Kartik Hegde, Po-An Tsai, Neal C. Crago, Aamer Jaleel, John D. Owens, Edgar Solomonik, Joel S. Emer, and Christopher W. Fletcher. Accelerating Sparse Data Orchestration via Dynamic Reflexive Tiling. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, 2023.

[129] MohammadHossein Olyaiy, Christopher Ng, and Mieszko Lis. Accelerating DNNs Inference with Predictive Layer Fusion. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '21, 2021.

[130] 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.

[131] Egor Orachyov, Pavel Alimov, and Semyon Grigorev. cuBool: sparse Boolean linear algebra for NVIDIA CUDA. `https://github.com/JetBrains-Research/cuBool`, 2021. Version 1.2.0.

[132] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 27–40, 2017.

[133] 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.

[134] Michael Pellauer, Jason Clemons, Vignesh Balaji, Neal Crago, Aamer Jaleel, Donghyuk Lee, Mike O'Connor, Anghsuman Parashar, Sean Treichler, Po-An Tsai, Stephen W. Keckler, and Joel S. Emer. Symphony: Orchestrating Sparse and Dense Tensors with Hierarchical Heterogeneous Processing. *ACM Trans. Comput. Syst.*, 41(1–4), dec 2023.

[135] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W Keckler, Christopher W Fletcher, and Joel Emer. Buffets: An Efficient and Composable Storage Idiom for Explicit Decoupled Data Orchestration. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[136] Eric Qin, Geonhwa Jeong, William Won, Sheng-Chun Kao, Hyoukjun Kwon, Sudarshan Srinivasan, Dipankar Das, Gordon E. Moon, Sivasankaran Rajamanickam, and Tushar Krishna. Extending Sparse Tensor Accelerators to Support Multiple Compression Formats. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021.

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

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

[139] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *Acm Sigplan Notices*, 48(6), 2013.

[140] 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.

[141] Scott Rostrup, Shweta Srivastava, and Kishore Singhal. Fast and memory-efficient minimum spanning tree on the gpu. *International Journal of Computational Science and Engineering*, 2013.

[142] Amit Sabne. XLA : Compiling Machine Learning for Peak Performance, 2020.

[143] Justin Salmon and Simon McIntosh-Smith. Exploiting hardware-accelerated ray tracing for monte carlo particle transport with openmc. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 19–29, 2019.

[144] 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.

[145] 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.

[146] Stanislav G. Sedukhin and Marcin Paprzycki. Generalizing matrix multiplication for efficient computations on modern computers. In *Parallel Processing and Applied Mathematics*, pages 225–234, 2012.

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

[148] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Aladdin: A Pre-RTL, Power-Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures. *ACM SIGARCH Computer Architecture News*, 42(3), 2014.

[149] 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.

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

[151] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: The problem based benchmark suite. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 68—70, 2012.

[152] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. Graphr: Accelerating graph processing using reram. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 531–543, 2018.

[153] 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.

[154] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[155] A. Symons, L. Mei, S. Colleman, P. Houshmand, S. Karl, and M. Verhelst. Stream: A Modeling Framework for Fine-grained Layer Fusion on Multi-core DNN Accelerators. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023.

[156] 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.

[157] 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.

[158] 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.

[159] Brian W Thompto, Dung Q Nguyen, José E Moreira, Ramon Bertran, Hans Jacobson, Richard J Eickemeyer, Rahul M Rao, Michael Goulet, Marcy Byers, Christopher J Gonzalez, et al. Energy Efficiency Boost in the AI-Infused POWER10 Processor. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[160] Ruiqin Tian, Luanzheng Guo, Jiajia Li, Bin Ren, and Gokcen Kestor. A High Performance Sparse Tensor Algebra Compiler in MLIR. In *2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, 2021.

[161] Jesmin Jahan Tithi, Neal C Crago, and Joel S Emer. Exploiting spatial architectures for edit distance algorithms. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[162] 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.

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

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

[165] John Von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4), 1993.

[166] Luc Waeijen, Savvas Sioutas, Maurice Peemen, Menno Lindwer, and Henk Corporaal. ConvFusion: A Model for Layer Fusion in Convolutional Neural Networks. *IEEE Access*, 9, 2021.

[167] 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.

[168] 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.

[169] 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.

[170] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. Speedup Graph Processing by Graph Ordering. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1813–1828, New York, NY, USA, 2016. Association for Computing Machinery.

[171] Jian Weng, Sihao Liu, Zhengrong Wang, Vidushi Dadu, and Tony Nowatzki. A hybrid systolic-dataflow architecture for inductive matrix algorithms. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[172] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[173] 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.

[174] 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.

[175] Yannan Nellie Wu, Joel S Emer, and Vivienne Sze. Accelergy: An Architecture-level Energy Estimation Methodology for Accelerator Designs. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019.

[176] Yannan Nellie Wu, Po-An Tsai, Angshuman Parashar, Vivienne Sze, and Joel S. Emer. Sparseloop: An Analytical Approach To Sparse Tensor Accelerator Modeling. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1377–1395, 2022.

[177] Wm A Wulf and Sally A McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[178] Carl Yang, Aydin Buluç, and John D. Owens. GraphBLAST: A high-performance linear algebra-based graph framework on the GPU. *CoRR*, abs/1908.01407, 2019.

[179] Yifan Yang, Joel Emer, and Daniel Sanchez. ISOSceles: Accelerating Sparse CNNs through Inter-Layer Pipelining. In *Proceedings of the 29th international symposium on High Performance Computer Architecture (HPCA-29)*, February 2023.

[180] 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.

[181] A. N. Yzelman, D. Di Nardo, J. M. Nash, and W. J. Suijlen. A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation. Preprint, 2020.

[182] 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.

[183] 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 international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-26)*, April 2021.

[184] 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.

[185] 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.

[186] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. Graphp: Reducing communication for pim-based graph processing with efficient data partition. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

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

[188] 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.

[189] Zhekai Zhang, Hanrui Wang, Song Han, and William J. Dally. SpArch: Efficient Architecture for Sparse Matrix Multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[190] 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.

[191] S. Zheng, X. Zhang, L. Liu, S. Wei, and S. Yin. Atomic Dataflow based Graph-Level Workload Orchestration for Scalable DNN Accelerators. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

[192] 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.

[193] Yuhao Zhu. RTNN: Accelerating Neighbor Search Using Hardware Ray Tracing. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, pages 76–89, 2022.

[194] 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.

[195] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. Graphq: Scalable pim-based graph processing. In *the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 712–725, 2019.