SIMD^2: A Generalized Matrix Instruction Set for Accelerating Tensor Computation beyond GEMM

Yunan Zhang
University of California, Riverside
USA
yzhan828@ucr.edu

Po-An Tsai
NVIDIA Research
USA
poant@nvidia.com

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

ABSTRACT

Matrix-multiplication units (MXUs) are now prevalent in every computing platform. The key attribute that makes MXUs so successful is the semiring structure, which allows tiling for both parallelism and data reuse. Nonetheless, matrix-multiplication is not the only algorithm with such attributes. We find that many algorithms share the same structure and differ in only the core operation; for example, using add-minimum instead of multiply-add. Algorithms with a semiring-like structure therefore have potential to be accelerated by a general-purpose matrix operation architecture, instead of common MXUs.

In this paper, we propose 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, we are able to build SIMD^2 architecture on top of any MXU architecture with minimal modifications. We developed a framework that emulates and validates SIMD^2 using NVIDIA GPUs with Tensor Cores. Across 8 applications, 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.

ACM Reference Format:
Yunan Zhang, Po-An Tsai, and Hung-Wei Tseng. 2022. SIMD^2: A Generalized Matrix Instruction Set for Accelerating Tensor Computation beyond GEMM. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3470496.3527411

1 INTRODUCTION

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 [72], a recent trend of efficient computing on matrices focuses on building hardware accelerators. Famous examples include NVIDIA's Tensor Cores [52, 53], Google's Tensor Processing Units (TPUs) [27], and the recent IBM Power 10 MMA unit [67]. 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 [3, 23].

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 [47, 62], 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 [21, 40]) 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.

ISCA '22, June 18–22, 2022, New York, NY, USA

ACM ISBN 978-1-4503-8610-4/22/06.
https://doi.org/10.1145/3470496.3527411

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.

We evaluate the proposed SIMD^2 architecture and hardware units through software emulation and hardware synthesis. We also made the emulation framework and hardware design publicly available through a web-hosted repository[1]. We demonstrate 8 applications that can naturally leverage these operations in their core algorithms. With the proposed SIMD^2 MXUs, these applications enjoy up to 38.59× speedup and more than 6.94× speedup on average. Synthesis results show that over a conventional MXU that supports only multiply-and-accumulate, SIMD^2 MXU adds 69% area overhead while supporting 8 different operations under the same clock period. This area overhead is 5% of the total chip area according to public die shot photos.

In presenting the SIMD^2 architecture, this paper makes the following contributions.
(1) It identifies a set of matrix applications with semiring-like structure and reveals strong potential in performance gain if SIMD^2 support is available in hardware.
(2) It proposes SIMD^2 architecture, programming model, instructions, and hardware units to accelerate semiring-like applications.
(3) It evaluates the performance benefit of the proposed SIMD^2 architecture, and the cost of SIMD^2 hardware units over a common MXU to demonstrate the opportunity of a SIMD^2 programming paradigm.

[1]You may find the code repository at https://github.com/escalab/SIMD2

Table 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	+	×	Matrix Multiplications, Matrix Inverse
Min-Plus	*min*	+	All-pairs shortest paths problem
Max-Plus	*max*	+	Maximum cost (critical path)
Min-Multiply	*min*	×	Minimum reliability paths
Max-Multiply	*max*	×	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

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.

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 $\otimes$. 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

```
1 for (i = 0; i < N; i++)
2   for (j = 0; j < N; j++)
3     for(k = 0; k < N; k++){
4       D[i][j] =  C[i][j]
5         + A[i][k] * B[k][j];
6     }
7                          (a)
```

```
1 for (src = 0; src < N; src++)
2   for (dst = 0; dst < N; dst++)
3     for(k = 0; k < N; k++){
4       D[src][dst] =  min(C[src][dst],
5                (C[src][k] + A[k][dst]));
6     }
7                          (b)
```

Figure 1: Code snippet of (a) GEMM and (b) APSP. See Section 4 for full APSP implementation.

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 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 1(b) shows how the inner loops of all-piars Bellman-Ford algorithm [10] for all-pairs-shortest-path (APSP) problem is similar to the semiring-like algebraic structure as GEMM (Figure 1(a)). Each iteration in Line 4–5 of Figure 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 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.

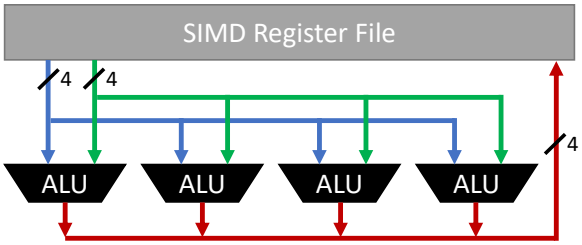

Figure 2: An example SIMD architecture.

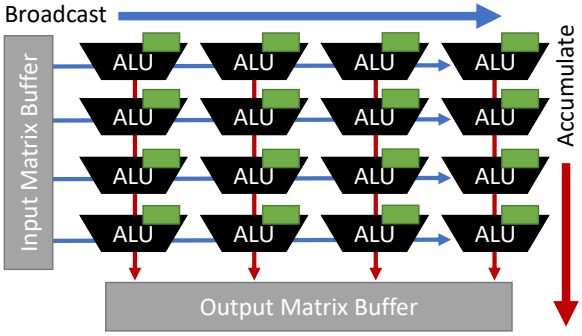

Figure 3: An example MXU for GEMM.

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 2, are bottlenecked by the vector register file bandwidth. Such data transfer bottleneck (von Neumann bottleneck [70]) 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 shows an example implementation of MXU, modeled after the matrix unit in TPUs [27]. In this MXU example, one input matrix is *broadcast* to multiple ALUs 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.

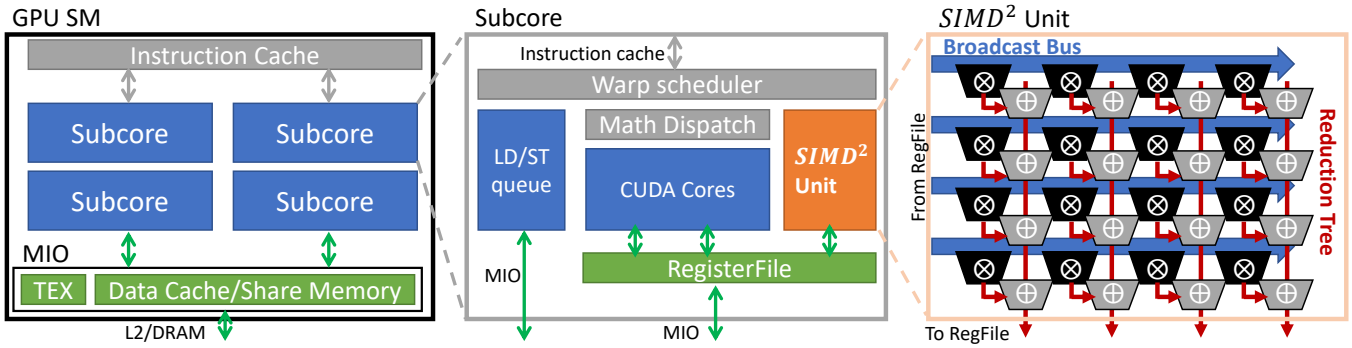

Figure 4: The high-level architecture of how SIMD^2 units are integrated in GPU systems and the design of an SIMD^2 unit.

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 [52] or across systems [26]. 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 is orthogonal to the hardware support (broadcast and accumulate) for a semiring-like structure. For example, if we enhance the ALU in Figure 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 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.

3.1 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 4 shows. Specifically, we build on top of the NVIDIA SM architecture [5], which integrates Tensor Core as part of the subcore in a GPU SM. The resulting high-level architecture resembles GPU SM with Tensor Cores [59] as the SIMD^2 units implementing SIMD^2 instructions are part of a streaming multiprocessor, but the rest of

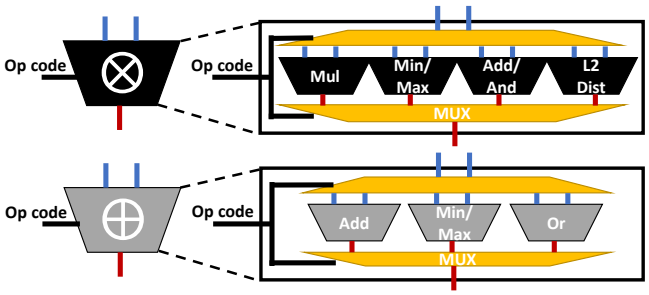

Figure 5: $\otimes$ ALU and $\oplus$ ALU in an SIMD^2 unit.

the architectural components (front-end, memory-subsystem, etc.) are shared with conventional GPU cores.

The SIMD^2 unit in Figure 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 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 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. 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 6.1) to the GEMM-only MXU, which is $> 4\times$ of the overhead introduced by the combined design in Figure 4.

Table 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	**⊕ OP**	**⊗ OP**	**Algorithm**
SIMD^2.mma	+	×	GEMM
SIMD^2.minplus	*min*	+	All-pairs shortest paths problem
SIMD^2.maxplus	*max*	+	Maximum cost (critical path)
SIMD^2.minmul	*min*	×	Minimum reliability paths
SIMD^2.maxmul	*max*	×	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

Table 3: Example API of SIMD^2 programming model

Sample Low-level Synopsis	Description
simd2::matrix<matrix_type, m, n, k, data_type>	Declaration function, declare the matrix will be applied in the m×n×k 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.

While we chose GPUs as the baseline system, building an SIMD^2 architecture on other GEMM-based accelerators, such as TPUs [27], should be straightforward and low overhead.

3.2 The SIMD^2 ISA

The SIMD^2 instruction extension builds on top of the warp-level matrix-multiply-accumulate (`wmma` [54]) instructions for GPUs and extends it to support new arithmetic instructions. Table 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 ⊕-⊗ ops, in addition to the classic matrix-multiply-accumulate (`mma`). These nine instructions map to the frequently used matrix problem patterns in Table 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 [23], 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.

4 PROGRAMMING MODEL

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 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 3.2 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.

```
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 6: Tiled minplus MM on some architecture with SIMD^2 supports

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 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.

```
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 7: CUDA kernel implenmentation of APSP using SIMD^2 API

To use the compute kernel from Figure 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 on heterogeneous computing devices. Figure 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 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 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 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 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 [35]. 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.

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

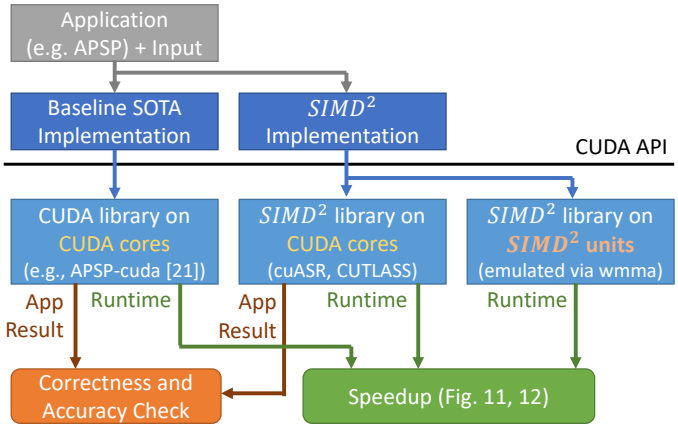

Figure 8: The workflow of the emulation framework for SIMD^2 evaluation.

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.

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.

5.1.1 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.

5.1.2 Evaluation Process. Figure 8 illustrates the workflow of our process in evaluating applications. For baseline applications, we executed each application directly on the hardware platform without 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 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 [24]. 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 6.1 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

Table 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 [28, 38]	Small	4096
		Medium	8192
		Large	16384
All Pair Critical Path (APLP)	ECL-APSP [28, 38]	Small	4096
		Medium	8192
		Large	16384
Maximum Capacity Path (MCP)	CUDA-FW [43, 44]	Small	4096
		Medium	8192
		Large	16384
Maximum Reliability Path (MAXRP)	CUDA-FW [43, 44]	Small	4096
		Medium	8192
		Large	16384
Minimum Reliability Path (MINRP)	CUDA-FW [43, 44]	Small	4096
		Medium	8192
		Large	16384
Minimum Spanning Tree (MST)	CUDA MST [17, 19, 25, 60, 63]	Small	1024
		Medium	2048
		Large	4096
Graph Transitive Colsure (GTC)	CUBOOL [56]	Small	1024
		Medium	4096
		Large	8192
K-Nearest Neighbor (KNN)	KNN-CUDA [69]	Small	4096
		Medium	8192
		Large	16384

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 [11] to support different precisions.

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 5.1.2, 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 5.1 describes.

Table 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 [38], 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 [56] 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

Table 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)

leverage SIMD^2's `add-norm` instruction. For the state-of-the-art GPU baseline, we use KNN-CUDA.

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.

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 [52] and Accel-Sim [30]. We carefully design the proposed extensions 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 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 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

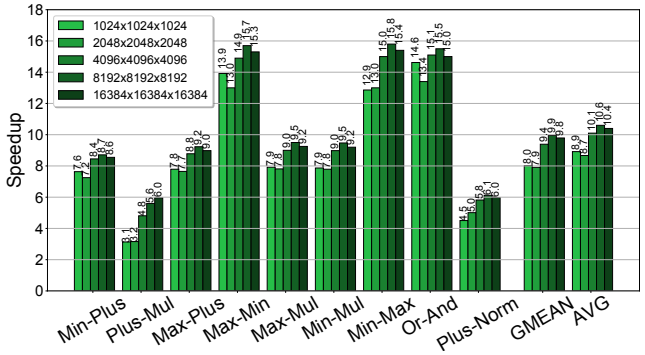

Figure 9: Performance of microbenchmark with square matrices using SIMD^2 API

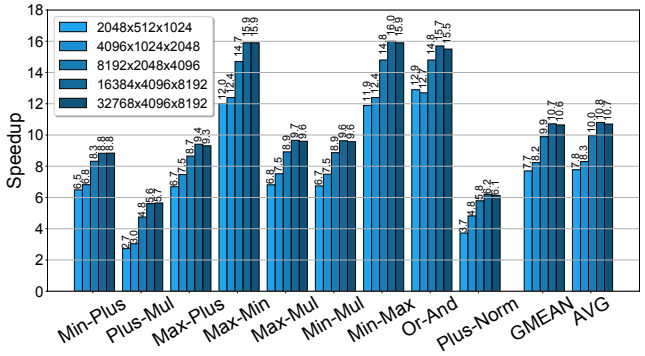

Figure 10: Performance of microbenchmark with nonsquare matrices using SIMD^2 API

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 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.

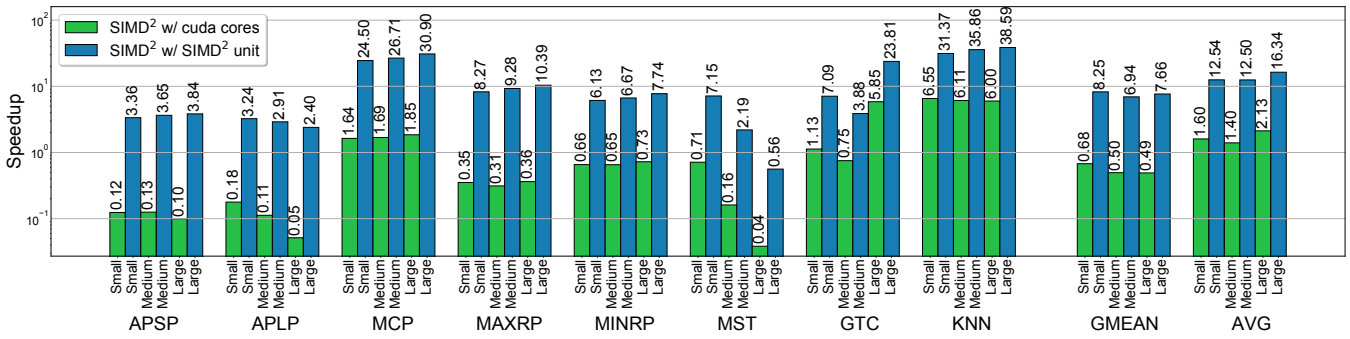

Figure 11: Performance of applications using SIMD² API

6.2 Microbenchmarks

We used microbenchmark workloads that repetitively invoke SIMD² the same instructions to gauge the performance gain of using SIMD² units compared against equivalent GPU implementations. The result shows up to 15.8× speedup in evaluated scenarios.

Figure 9 shows the performance gain of SIMD² over the equivalent GPU baseline implementations when using square matrices as inputs. SIMD² 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 4,096×4,096 ones, the performance gain saturates at about 10×, representing the level of peak performance gain of these instructions. Figure 10 shows the performance gain of SIMD² 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, SIMD² 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 SIMD² units. We suspect the extra benefit from SIMD² 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 `and` operations. By fusing these operations in a single instruction, SIMD² 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 SIMD² addresses. Nevertheless, SIMD² still has a significant advantage, obtaining a speedup of up to 5.96× for larger matrix operations. We conclude that the SIMD² architecture has larger potential than fusing more vector operations, which we leave to future work.

6.3 Benchmark Applications

Figure 11 shows the speedup of kerenl latency of applications using SIMD² (SIMD² w/ SIMD² units) over the baseline, optimized GPU implementations. SIMD² achieves a geometric mean of 6.94× – 8.25×, with speedup as large as 38.59×. The performance gain of SIMD² in 7 out of the 8 applications remains strong even when dataset sizes increased.

Compared with implementing the same matrix-based algorithms without SIMD² presented (SIMD² w/ CUDA cores), all applications show significant slow down when SIMD² 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 SIMD² units are absent. This result explains why these algorithms were not favorable in conventional architectures. However, the introduction of SIMD² makes these matrix algorithms feasible. The matrix processing power from the SIMD² 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 SIMD² implementations outperform their baseline, state-of-the-art implementations, even without the presence of SIMD² units. For KNN, the computational complexity is the same for both SIMD² and the baseline implementations. However, the SIMD² kernel can still achieve a maximum speedup of 6.55× without the help of SIMD² units. This is because the baseline implementation uses customized functions to implement the algorithm, but the backend library of SIMD² without SIMD² units leverages CUTLASS that is more optimized and adaptive to modern GPU architectures. However, the performance gap between configurations with or without SIMD² 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 SIMD² on CUDA cores, such implementation still has a huge performance gap to catch up with the performance using SIMD² units. For MCP and GTC, SIMD² w/ CUDA cores can outperform their baseline implementations even though the computational complexity is higher in SIMD² implementations for two reasons. The first reason is similar to the case in KNN that SIMD² 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

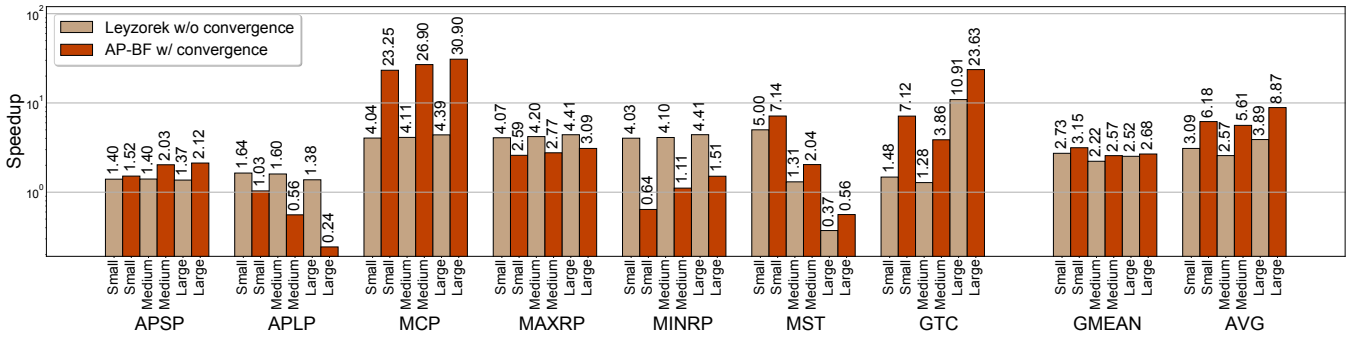

Figure 12: Performance of different algorithmic optimizations

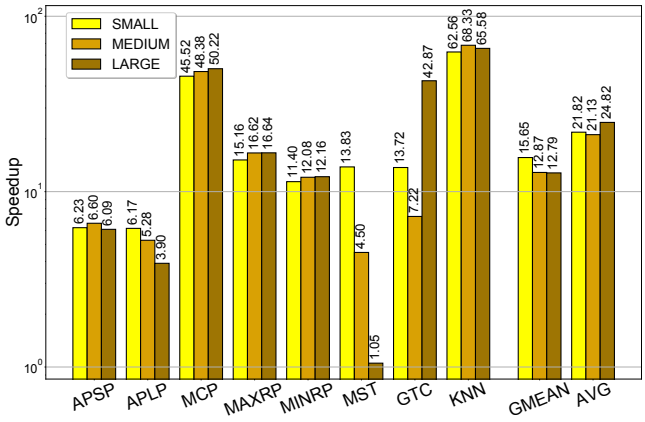

Figure 13: Performance of applications using Sparse SIMD^2 unit

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 underlying 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 $O(E \log E)$ [7, 31], 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 $O(V^3)$ [7, 12], 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.

6.4 Discussion on algorithmic optimizations

In Figure 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 12 illustrates the performance of these implementations with bars labeled as Leyzorek w/o convergence. The baseline remains the same as Figure 11. Despite the increasing numbers of iterations, all applications still outperform their baseline GPU implementations, ranging from 1.11× to 10.91×.

In Figure 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 outweighted the shortcomings of increased computational complexity.

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

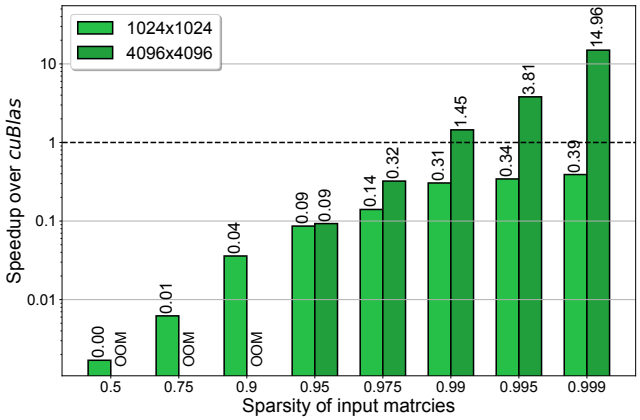

Figure 14: Performance of sparse matrix multiplication

Figure 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 14. The x-axis in Figure 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 [45], 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 [73]. 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 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.

7 RELATED WORK

In addition to the related work that motivates SIMD^2 in Section 2, several other lines of research that are relevant to SIMD^2 deserve mention.

7.1 Matrix extensions and instructions

Instruction-level support for matrix-matrix multiplication can be dated back to the 90s. MOM [6] 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 [52, 53], Intel AMX [23], and Arm SME [3] 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.

7.2 Dense tensor accelerators

SIMD^2 builds on top of recent dense tensor accelerators for matrix-multiplication [4, 26, 27, 37, 52, 67] 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 [67] 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. [71] propose a hybrid systolic-dataflow architecture for inductive matrix algorithms (e.g., linear algebra solver). Tithi et al. [68] 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.

7.3 Sparse tensor accelerators

Since sparse matrices are common for many applications, such as HPC workloads, there is also ample prior work in sparse tensor accelerators [2, 13, 14, 18, 33, 57, 58, 65, 66, 75–77]. These accelerators propose various sparse optimizations to skip ineffectual computations to speed up the tensor algorithms with sparse inputs. They leverage various hardware support for gather/scatter operation and intersection to transform sparse tensor algebra into dense tensor algebra, improving conventional dense tensor accelerators.

These techniques are therefore orthogonal to SIMD^2, and we expect SIMD^2 can be extended to support sparse tensor operation by applying similar techniques, as discussed previously.

7.4 Graph algorithm accelerators

While many graph algorithms can be expressed as tensor operations and linear algebra [29] and accelerated by tensor accelerators, prior work has also proposed hardware accelerators to speed up graph algorithms and analytics in their classic form. Graphicionado [16], GraphR [64], GraphP [74], and GraphQ [79] leverage processing-in-memory (PIM) architecture to alleviate the bandwidth bottleneck in graph algorithms. PHI [50] and HATS [49] 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.

7.5 Democratizing Domain-Specific Accelerators

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 [15, 21], database queries [8, 20, 22], cryptography [34] and scientific computing problems [9, 11, 36, 40–42, 48, 51]. Ray tracing accelerators are also useful for Monte Carlo simulations [61], robotics navigation [46] and nearest neighbor search problems [78]. 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 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. Potential solutions include bringing hardware accelerators closer to large memory arrays [32] or using other hardware accelerators to produce the demanding data structures for the target computing resource [39, 55].

8 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.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their helpful comments. This work was sponsored by the two National Science Foundation (NSF) awards, CNS-1940048 and CNS-2007124. This work was also supported by new faculty start-up funds from University of California, Riverside.

REFERENCES

[1] BLAS (Basic Linear Algebra Subprograms). http://www.netlib.org/blas/, 2004.
[2] Dennis Abts, Jonathan Ross, Jonathan Sparling, Mark Wong-VanHaren, Max Baker, Tom Hawkins, Andrew Bell, John Thompson, Temesghen Kahsai, Garrin Kimmell, et al. Think fast: a tensor streaming processor (TSP) for accelerating deep learning workloads. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 145–158, 2020.
[3] 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.
[4] 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.
[5] Jack Choquette, Olivier Giroux, and Denis Foley. Volta: Performance and Programmability. *IEEE Micro*, 38(2):42–52, 2018.
[6] 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.
[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.
[8] 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.
[9] 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.
[10] Jeff Erickson. *Algorithms.* 2019.
[11] 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.
[12] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, page 345, jun 1962.
[13] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, et al. AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 922–936, 2020.
[14] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and T. N. Vijaykumar. SparTen: A sparse tensor accelerator for convolutional neural networks. In *2019 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, MICRO '52, 2019.
[15] 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.
[16] 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)*, 2016.
[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

[32] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. TensorDIMM: A practical near-memory processing architecture for embeddings and tensor operations in deep learning. In *the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, MICRO '52, pages 740–753, 2019.

[33] Sangwon Lee, Gyuyoung Park, and Myoungsoo Jung. TensorPRAM: Designing a scalable heterogeneous deep learning accelerator with byte-addressable prams. In *12th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2020, July 13-14, 2020*, 2020.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

[39] Yu-Chia Liu and Hung-Wei Tseng. NDS: N-Dimensional Storage. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 2021, pages 28–45, 2021.

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

[41] 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.

[42] 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.

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

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

[45] 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.

[46] 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.

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

[48] 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.

[49] 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.

[50] 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.

[51] Ricardo Nobre, Aleksandar Ilic, Sergio Santander-Jim?mnez, 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.

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

[53] 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.

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

[55] NVIDIA Corporation. NVIDIA Hopper Architecture In-Depth. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/, 2022.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. MatRaptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 766–780, 2020.

[66] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 689–702, 2020.

[67] 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.

[68] 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.

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

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

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] 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)*, pages 261–274, 2020.

[76] Xuda Zhou, Zidong Du, Qi Guo, Shaoli Liu, Chengsi Liu, Chao Wang, Xuehai Zhou, Ling Li, Tianshi Chen, and Yunji Chen. Cambricon-S: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 15–28, 2018.

[77] Maohua Zhu, Tao Zhang, Zhenyu Gu, and Yuan Xie. Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern GPUs. In *the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 359–371, 2019.

[78] 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.

[79] 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.