SMASH: Sparse Matrix Atomic Scratchpad Hashing

A Thesis Presented

by

Kaustubh Shivdikar

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements

for the degree of

Master of Science

in

Electrical and Computer Engineering

Northeastern University

Boston, Massachusetts

April 2021

To the people trying to make the architectures of today, history tomorrow.

Contents

List of Figures

List of Tables

List of Acronyms

ABM *Advanced Bit Manipulation*

ATT Address translation tables

ADX *Multiprecision Add Carry*

AESI *Advanced Encryption Instructions*

AMD *Advanced Micro Devices*

AMD *Central Processing Unit*

DGAS *Distributed Global Address Space*

AVX2 *Advanced Vector Instructions 2*

BFS *Breadth First Search*

BLAS *Basic Linear Algebra Subprograms*

BMI2 *Bit Manipulation Instruction Set 2*

C-RAM *Computational RAM*

ELL *ELLPACK*

EMMX *Extended MMX Extension*

CLMUL *Carry-less Multiplication Extension*

CPU *Central Processing Unit*

CSR *Compressed Sparse Row*

CSC *Compressed Sparse Column*

DARPA Defense Advanced Research Projects Agency

DFFT *Dense Fast Fourier Transform*

DMA *Direct Memory Access*

DP *Double Precision*

DRAM *Dynamic Random Access Memory*

F16C *16-bit Floating Point Conversion*

FMA *Floating-point Multiply and Add*

FMA3 *3-Operand Fused-Multiply-Add*

GCN *Graph Convolutional Network*

GEMM *General Matrix Multiplication*

GNN *Graph Neural Networks*

GPGPU *General-Purpose Graphic Processing Unit*

GPU *Graphics Processing Unit*

HBM *High Bandwidth Memory*

HP *Half Precision*

HIVE Hierarchical Identify Verify Exploit

IPC *Instructions per Cycle*

ISA *Instruction Set Architecture*

MLP *Multi-layer Perceptron*

MKL *Math Kernel Library*

MLP *Multi-Layer Perceptrons*

MTC *Multi-threaded Core*

PIUMA Programmable Integrated Unified Memory Architecture

RF Register File

SP *Single Precision*

OMP *OpenMP*

OS *Operating System*

PIM *Processor in Memory*

QP *Quadruple Precision*

RMAT *Recursive Matrix*

SFFT *Sparse Fast Fourier Transform*

SIMD *Single Instruction Multiple Data*

SMASH *Sparse Matrix Atomic Scratchpad Hashing*

SPAD *Scratchpad*

SPMD *Single Program, Multiple Data*

SpGEMM *Sparse Matrix-Matrix Multiply*

SpMV *Sparse Matrix-Vector*

SSE4.2 *Streaming SIMD Extensions 4.2*

STC *Single-threaded Core*

Acknowledgments

I would like to start with thanking my Ph.D. advisor, Prof. David Kaeli, for his dedicated support and motivation in this thesis as well as the project. Would like to thank Dr. Fabrizio Petrini for providing me with this extra-ordinary opportunity to experiment with the latest and greatest tools as well as his guidance on SpGEMM. In addition, would like to thank Dr. Fabio Checconi for his valuable feedback and inspiration for SMASH kernels. Thank you Intel for exposing me to some of the brightest minds in the valley. Thank you all members of NUCAR, Dr. Norm Rubin, Dr. Yifan Sun, Elmira Karimi, Malith Jayaweera, Zlatan Feric, Derek Rodriguez, Julian Gutierrez, Trinayan Baruah, and Yuhui Bao for your guidance and support. A special thanks to thank Nicolas Agostini, the source of inspiration and motivation, throughout this project.

With the world coming to a grinding halt with an epidemic that showed no signs of an end, I would like to thank my parents Chandrakant and Anagha and my brother Saumil for the relentless support in all aspects over these last years. Last but not least, I would like to thank all my friends in US for their constant motivation, who succeeded in making this journey memorable.

Abstract of the Thesis

SMASH: Sparse Matrix Atomic Scratchpad Hashing

by

Kaustubh Shivdikar

Master of Science in Electrical and Computer Engineering

Northeastern University, April 2021

Dr. David Kaeli, Adviser

Abstract: Sparse matrices, more specifically *Sparse Matrix-Matrix Multiply* (SpGEMM) kernels, are commonly found in a wide range of applications, spanning graph-based path-finding to machine learning algorithms (e.g., neural networks). A particular challenge in implementing SpGEMM kernels has been the pressure placed on DRAM memory. One approach to tackle this problem is to use an inner product method for the SpGEMM kernel implementation. While the inner product produces fewer intermediate results, it can end up saturating the memory bandwidth, given the high number of redundant fetches of the input matrix elements. Using an outer product-based SpGEMM kernel can reduce redundant fetches, but at the cost of increased overhead due to extra computation and memory accesses for producing/managing partial products.

In this thesis, we introduce a novel SpGEMM kernel implementation based on the row-wise product approach. We leverage atomic instructions to merge intermediate partial products as they are generated. The use of atomic instructions eliminates the need to create partial product matrices, thus eliminating redundant DRAM fetches.

To evaluate our row-wise product approach, we map an optimized SpGEMM kernel to a custom accelerator designed to accelerate graph-based applications. The targeted accelerator is an experimental system named PIUMA, being developed by Intel. PIUMA provides several attractive features, including fast context switching, user-configurable caches, globally addressable memory, non-coherent caches, and asynchronous pipelines. We tailor our SpGEMM kernel to exploit many of the features of the PIUMA fabric.

This thesis compares our SpGEMM implementation against prior solutions, all mapped to the PIUMA framework. We briefly describe some of the PIUMA architecture features and then delve into the details of our optimized SpGEMM kernel. Our SpGEMM kernel can achieve $9.4\times$ speedup as compared to competing approaches.

Chapter 1

Introduction

1.1 The Development of Matrix-based Methods

In 1812, a French mathematician named Jacques Philippe Marie Binet pointed out several important computations involved the multiplication of two matrices [53]. On November 30 of the same year, he provided a lecture on his observation and further extended his work, leading to the Cauchy-Binet formula [54]. This is one of the oldest known sources of the discovery of matrix multiplication. Matrix multiplication was described as a method of multiply data arranged in rows. Later, in the year 1850, Arthur Cayley applied matrix multiplication to solve a system of linear equations [15], showing applications of this idea to solve an important class of mathematical problems.

1.2 High Performance Computing and Matrices

The 20^{th} century witnessed developments in computer technology. Computers, which were initially developed to crunch numbers for tabulating the United States census [117], were soon being used to perform calculations for a variety of physics and mathematics problems, many that included matrix multiplication [117].

Use-cases for matrix multiplication applications were so widespread that they demanded standardization [49]. In 1979, the BLAS Technical forum published a report on standardization of a few of the common linear algebra operations (also known as subroutines), which they referred to as *Level-1 Basic Linear Algebra Subroutines* (BLAS) [57]. Later, in 1986 and 1988, BLAS was further augmented with Level-2 and Level-3 subroutines, respectively. The Level-3 subroutines included

the matrix multiplication subroutine (also known as GEMM kernel). The *General Matrix Multiplication* (GEMM) kernel implementations were designed to work with dense matrices (matrices with mostly non-zero elements).

Matrix multiplication is a key operation in many scientific computations. One such computation is graph analysis. Graph analysis commonly represents graphs using an adjacency matrix and then performs matrix multiplication operations on these matrices. The associated adjacency matrices are inherently sparse [27] (with very few non-zeros) due to many graphs' structures. Early library implementations of GEMM performed poorly on such sparse matrices. By 2002, the BLAS Technical forum adopted a new standard for such sparse data [29]. They presented the *Sparse Basic Linear Algebra Subprograms* (Sparse BLAS), which included subroutines that included SpGEMM (also known as the SpGEMM kernel), and focused on optimizations required for sparse matrix multiplication [29].

1.3 Applications

Today, GEMM and SpGEMM kernels have found their way into many important applications. Some of these applications include:

1. Data encryption: AES, SHA1, SHA2, Twofish [16, 46, 24, 67, 25, 101, 58, 98];
2. Data compression and Decompression: zip files, JPEG and PNG image compression [80, 59];
3. Image processing: filters for real-time image processing, such as Sobel filtering, image sharpening, image blurring [82, 71, 88, 91];
4. Pathfinding: *Breadth First Search* (BFS) and Dijkstra's Algorithm [118];
5. Signal processing: *Dense Fast Fourier Transform* (DFFT), *Sparse Fast Fourier Transform* (SFFT) [44, 86, 81];
6. Simulations: N-body, raytracing, and Monte-Carlo [4, 97, 112, 107]; and
7. Machine learning: various supervised and unsupervised learning algorithms are implemented using GEMM kernels Deep learning utilizes GEMM kernels for convolution layers [78, 66, 22, 75, 102, 8, 89, 103, 108].

This thesis's motivation lies in improving the performance of SpGEMM kernels, which will have a significant impact on many important applications.

Recently, we have seen growth in graph-based applications in the industry.

Facebook [11], Google [73], Twitter [41], Amazon [93], Netflix [13], Cora [52], Citeseer [35] and many other large companies use graphs to analyze social networks, citation networks, and even recommend products.

The currently available computational frameworks have not kept up with the ever-increasing demands of graph-based workloads [106]. One of the key components of such applications is the sparse matrix multiplication kernel (SpGEMM kernel). As compared to their dense counterparts, SpGEMM kernels are complex and harder to optimize. Traditional multi-core CPUs and many-core GPU architectures provide limited performance over SpGEMM kernels, mainly due to their irregular memory access patterns and unbalanced work distribution.

1.4 Motivation

One graph-analysis application that is growing in popularity is the *Graph Neural Networks* (GNN). GNNs represent features of a node in the graph with a vector. These vectors are then recursively aggregated and transformed, based on the features of the neighboring nodes [110]. These features can then be used to classify nodes or perform inference on datasets. Unlike traditional neural networks that work with dense data structures, such as *Multi-Layer Perceptrons* (MLP), GNNs operate on sparse graph structures [12]. They have been becoming increasingly popular, given their high accuracy for node classification on graph-structured datasets [115, 116]. From a computational

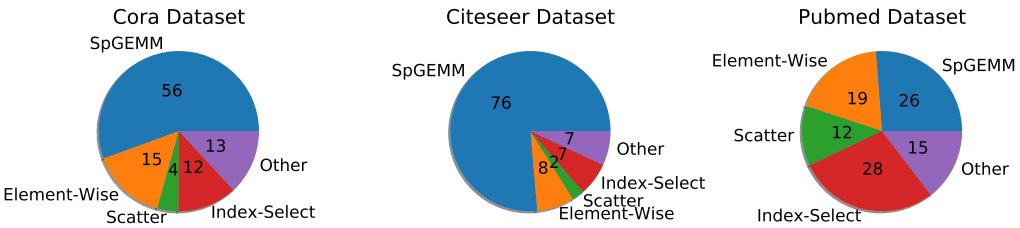

Figure 1.1: Graph Convolutional Network (GCN) kernel execution time breakdown.

perspective, GNNs are comprised of a mix of kernels, including element-wise operations, transpose operations, dense matrix multiplication, sparse matrix multiplication, index selection, reduction, and batch normalization. One example of a GNN is a Graph Convolutional Network or *Graph Con-*

volutional Network (GCN). A GCN is similar to a Convolutional Neural Network (CNN), except that it performs convolution operations on a graph instead of image-based operations on pixels [21]. Figure 1.1 shows a breakdown of time spent in each of these operations in a GCN application.

The time spent in computing SpGEMM kernels is a function of two factors: 1) the sparsity of the dataset and 2) the sparsity pattern. Although it is difficult to compare sparsity patterns, Table 1.1 presents the degree of sparsity for various graph datasets. Many workloads, including graph convolution [23], node classification [5], path planning [62], use the datasets shown in Table 1.1 to analyze graphs and derive useful information. SpGEMM remains an integral kernel used in such workloads, processing highly sparse datasets, thus providing us with an opportunity to exploit this sparsity using an optimized kernel.

Dataset	**Vertices**	**Edges**	**Degree of Sparsity**
Citeseer	3327	9464	99.914
Cora	2708	10,858	99.851
Pubmed	19,717	88,676	99.977
Wikipedia RfA	113,80	188,077	99.854
Epinions	75,888	508,837	99.991
Slashdot	82,144	549,202	99.991
Astro Physics Collaborations	18,772	792,320	99.775
NotreDame	325,729	1,497,134	99.998
Amazon	334,863	1,851,744	99.998
Google Page Hyperlinks	916,428	5,105,039	99.999
Youtube	1,134,890	11,950,496	99.999
Patent Citations	3,774,768	16,518,948	99.999
Stack Overflow	2,601,977	36,233,450	99.999
Orkut	3,072,441	486,740,332	99.994
Twitter Follower Network	41,652,230	1,468,365,182	99.999

Table 1.1: Sparse Graph datasets

Given the dominance of SpGEMM execution in GNN applications, we focus on accelerating SpGEMM kernels to reduce their execution time. In this thesis, we focus on identifying the underlying bottlenecks of SpGEMM kernels and improving these workloads' performance with datasets with varying degrees of sparsity. We target the Intel PIUMA parallel accelerator, providing us with a state-of-the-art target to demonstrate our approach's utility.

1.5 Dataflow in SpGEMM

There are four common approaches used to multiply two matrices (as shown in Figure 1.2) [96]:

1. Inner product approach: $Row(A) \times Col(B) = Element(C)$
2. Outer product approach: $Col(A) \times Row(B) = Partial\ Products\ of\ Matrix(C)$
3. Row-wise product approach: $Row(A) \times Corresponding\ Rows(B) = Row(C)$
4. Column-wise product approach: $Corresponding\ Cols(A) \times Col(B) = Col(C)$

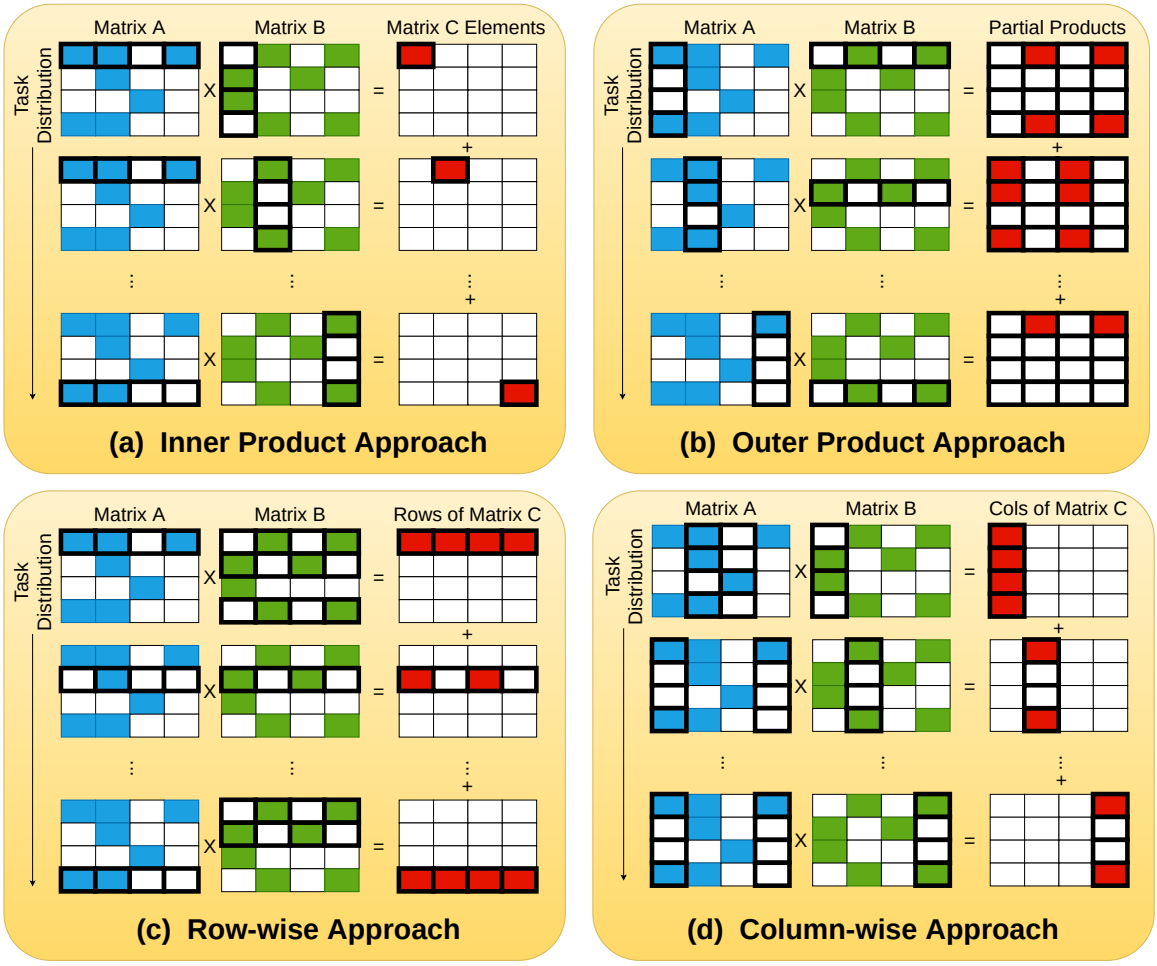

Figure 1.2: Matrix multiplication approaches

The most widely used approach for matrix multiplication is the inner product approach. Inner-product methods are based on computing a dot product of a row of Matrix A with a column of Matrix B, generating a single output element of Matrix C (Figure 1.2 (a)).

This leads to multiple reads of the input matrices but results in a single write of the output matrix elements. Thus, inner-product based methods exhibit poor input reuse, but good output reuse. Equation 1.1 denotes the operations of the inner product method:

$$c_{i,j} = \sum_{k=0}^{N-1} a_{i,k} \times b_{k,j} \tag{1.1}$$

where A and B are input matrices, $c_{i,j}$ is an element of the i^{th} row and j^{th} column of the output matrix C, N is the number of columns in the A matrix, subsequently $a_{i,k}$ and $b_{k,j}$ are elements of the corresponding rows and columns of matrices A and B, respectively.

Method	Input Reuse	Output Reuse	Intermediate Size	Disadvantage
Inner Product	Poor	Good	Small	Redundant input reads
Outer Product	Good	Poor	Large	Large intermediate size
Row-wise	Poor	Good	Small	Load imbalance
Col-wise	Poor	Good	Small	Load imbalance

Table 1.2: Matrix Multiplication Methods

In contrast, the outer-product method multiplies a single row of Matrix A with all the rows of matrix B to produce partial products (Figure 1.2 (b)). These partial products are stored in intermediate matrices and are later merged to form the output matrix [61, 85]. This leads to a single read of the input matrices but multiple writes of the partial product output matrices. Thus, in contrast to the inner-product method, the outer-product method exhibits good input reuse but poor output reuse. Computation of the output matrix using an outer product approach is expressed in Equation 1.2:

$$\begin{aligned} Output\ Matrix &= \sum_{n=0}^{N-1} C_n \\ C_n &= a_n b_n \end{aligned} \tag{1.2}$$

where C_n is a partial product matrix of output matrix C, A and B represent input matrices, N is the number of columns in matrix A and $a_n b_n$ is a cross-product of n^{th} column of A and n^{th} row of B.

The row-wise approach consists of a scalar product of every element of a row of matrix

A with corresponding rows of the matrix B (see Figure 1.2 (c) and Equation 1.3). The case for the column-wise approach is similar, where a single column of the matrix B is multiplied by corresponding columns of matrix A (as seen in Figure 1.2 (d) and Equation 1.4). Both row-wise and column-wise products have similar dataflow properties. They both exhibit poor input reuse due to redundant accesses made to one of the two input matrices. As opposed to an outer product approach, they do not generate a large number of intermediate products because partial products are immediately merged after generation. Thus, both of these approaches produce high output reuse. In addition, the inner and outer product approaches require the input matrices to be stored in opposite storage formats, matrix A in row-major and matrix B in column-major for the inner product, and vice versa for the outer product. On the other hand, both row-wise and column-wise require both input matrices to be stored in the same storage format. A significant disadvantage of the row-wise and column-wise approach is a skewed matrix (matrix with unevenly distributed non-zeros) can cause load imbalance in computations. This problem of load imbalance, and a solution for the same, are further described in Section 5.2 of this thesis.

$$C[i,:] = \sum_{k=0}^{N} A[i,k] * B[k,:] \tag{1.3}$$

$$C[:,j] = \sum_{k=0}^{N} A[:,k] * B[k,j] \tag{1.4}$$

where $C[i,:]$ and $C[:,j]$ represent i^{th} row and j^{th} column, respectively, of output matrix C. A and B are the two input matrices, and N is number of rows of matrix A from Equation 1.3 and number of columns of matrix B from Equation 1.4.

1.6 Introduction of SMASH

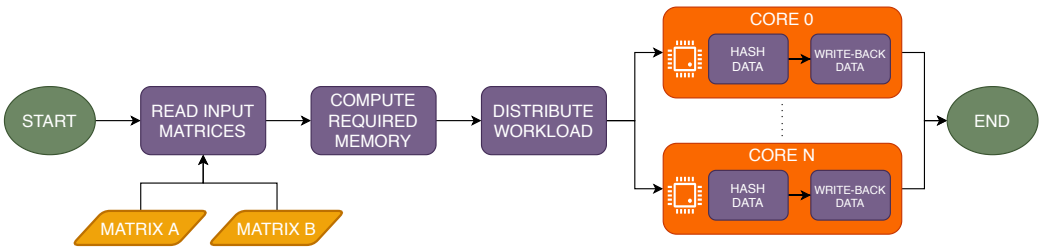

Figure 1.3: SMASH Overview

In this thesis, we present Sparse Matrix Atomic Scratchpad Hashing or SMASH, a row-wise product method that uses the Compressed Sparse Row (CSR) format [92]. An overview of our algorithm is shown in Figure 1.3. The SMASH algorithm is designed to adapt to varying sparsity patterns in the input matrices. We address issues related to the row-wise product approach by designing our kernel to merge partial products using a custom implementation of a hashtable.

This thesis also discusses the unique features of a novel architecture from Intel called PIUMA, designed to speedup graph-based workloads. We further describe our implementation of SpGEMM that exploits these features of this new architecture. Then we describe several improvements to our SpGEMM algorithm. We discuss design decisions and their impact on the resulting algorithm. Finally, we compare our optimized SpGEMM kernel performance on the Intel PIUMA accelerator architecture and provide an in-depth analysis of the results.

1.7 Contributions

The contributions of this thesis include:

- Analysis of the problems exhibited by sparse matrix multiplication kernels.
- A comparison of architectures that support sparse matrix multiplications.
- A comparison of previous implementations of SpGEMM kernels.
- An architectural overview of Intel's novel PIUMA graph accelerator.
- A novel SpGEMM kernel implementation that makes the best use of the PIUMA accelerator.

Chapter 2

Background

This chapter reviews the background information on CPU and GPU architectures required to place this thesis in context. We also cover common approaches on improving the performance of SpGEMM workloads. We include discussion on hardware designs of domain-specific accelerators for such workloads. Finally, we describe related work on SpGEMM kernels and their implementations.

2.1 CPU

The history of the *Central Processing Unit* (CPU) dates back as far as 1971 when Intel introduced the first microprocessor in the market, the Intel 4004 [10], capable of performing 60,000 operations per second. Since then, there have been rapid advances in this field regarding clock speed and transistor technology, enabling today's AMD Ryzen CPUs to execute 2.3 teraoperations per second (a teraoperation is 10^{12} operations per second).

A CPU can be defined as a computational device that, fundamentally, reads instructions from program memory and performs calculations. These instructions are fetched from the main memory of the computer (typically *Dynamic Random Access Memory* (DRAM)) and undergo 3 stages of computations:

- Fetch: Retrieve the instructions from memory. The control unit usually sends a signal through the address bus to retrieve instructions.
- Decode: The control unit splits the instruction into two parts, the opcode and the operands.

- Execute: The command represented by opcode is executed on the operand in the execute stage.

As far as CPU architecture is concerned, a large portion of the chip area is dedicated to control logic. With fewer cores and a large control-logic chip area, the chip real estate dedicated to control logic per core is high. In addition, every single core of the CPU is faster than the GPU. These factors allow the CPU to excel at certain workloads compared to the GPU. The CPU is designed to handle a wide range of tasks efficiently but is heavily limited when running tasks concurrently. The larger control-logic area and faster cores provide the CPUs with an advantage over the GPUs for control-dominated general-purpose workloads containing multiple conditional branches. We compare the CPU chip area with that of GPU in Figure 2.1

Given that a CPU devotes more logic to control (i.e., branch handling) and is clocked faster than the GPU, it allows the CPU to excel at executing workloads with complex single-threaded tasks, such as operating system services and database engine operations.

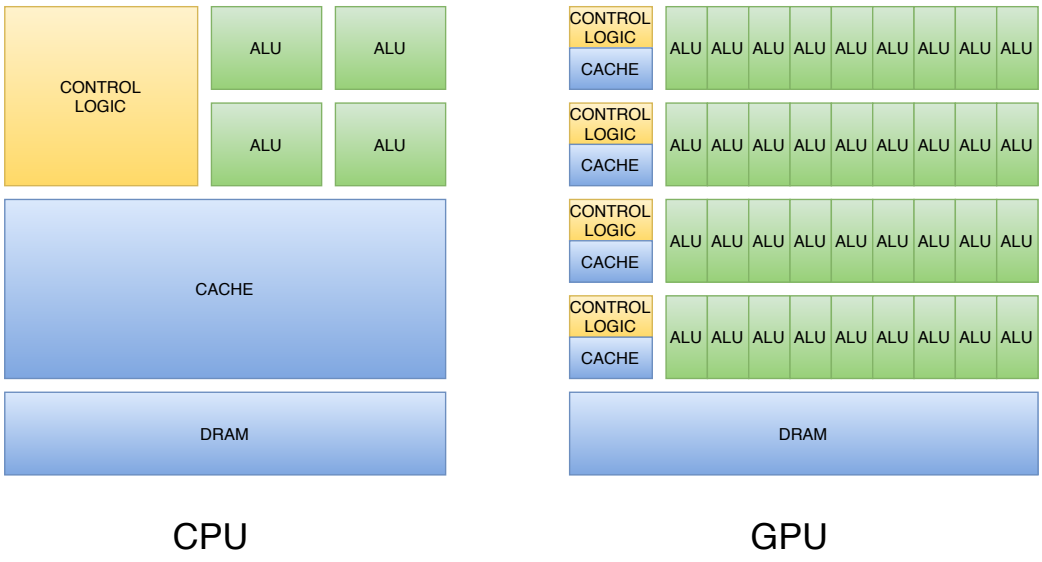

Figure 2.1: CPU and GPU Architectural Overview

2.2 GPU

Graphics Processing Unit (GPU)s were traditionally designed to render graphics and videos to displays. They were used in appliances that need a display, like the personal computer, mo-

biles, and embedded systems. Modern GPUs are now capable of accelerating a variety of tasks that were previously executed on CPUs. These devices led to the birth of a new field called GPU Compute, where the GPU is equipped with programmable shaders. Today, GPUs commonly run a range of compute-oriented workloads, including encryption, decryption, physics simulations, pathfinding, and machine learning.

Figure 2.1 compares the chip area distribution between a CPU and a GPU. The high number of cores on a GPU allows this device to perform parallel tasks efficiently. A single core of a GPU, compared to a CPU, is much slower in terms of clock rate. The GPU amortizes this slower clock speed by running thousands of tasks in parallel. Hence, workloads that possess a high degree of parallelism are better suited to run on a GPU.

A GPU can outperform a CPU in many workloads that express such parallelism. Dense matrix multiplication is one such application. The high number of cores present on a single GPU allow it to run tasks in parallel.

Although GPUs excel at accelerating data-parallel tasks by utilizing high levels of concurrency, they do not perform well on workloads that involve control flow. Control flow statements, such as "if-else" clauses, require control-flow logic in hardware to boost performance. With a large number of cores on a GPU, the amount of chip area per core dedicated to control logic is limited. Hence, workloads that contain a significant number of conditional branches tend to perform poorly on GPUs [51].

2.3 Branch and Control Flow Logic

A computer program consists of a set of instructions. These instructions are executed in a specific order (commonly referred to as the control flow of the program). Control flow statements (e.g., `if-else, for, while`) allow programmers to create algorithms with divergent execution sequences/paths. The decision of choosing a path is evaluated based on some conditions. The "branch" instruction is a special instruction that can change the execution path by altering the program counter (hence called branching). Modern-day CPUs and GPUs overlap instruction execution in a single stream to gain speedup. This overlap of instructions is called pipelining. Branching makes it difficult to overlap instructions because the next instruction to be executed might depend upon the previous instruction's output.

The CPU uses techniques such as branch prediction, where if the branch is predicted correctly, it incurs a little to no execution penalty, but if the branch is mispredicted, the CPU is

forced to squash the contents of the pipeline and continue execution on the correct branch. In addition, CPUs have a deeper pipeline (with more stages) as compared to GPUs. Hence the penalty for mispredicting a branch is higher in a CPU, as more instructions will be squashed. The large control logic area enables the CPU to reduce this penalty of mispredicting branches.

A GPU executes workload in a warp (the basic unit of execution) [1]. A warp is a collection of GPU threads (NVIDIA GPUs contain 32 threads in a warp). GPUs do not have the resources to have each of their threads execute divergent branches simultaneously. When executing a conditional branch instruction, a single warp in the GPU computes both sides of the branch (sequentially) and discards one of them based on the correct branch path. This is referred to as predication. Predication works for cases when each branch's size is considerably small but incurs a large penalty otherwise. In summary, the CPU is capable of handling workloads with a large number of conditional statements, while the GPU will encounter significant slowdown for such workloads.

2.4 Computational Performance

Floating Point Operations per Second (FLOPS) is a key metric for comparing the performance of different hardware designs. This metric captures the number of floating-point operations that a device can complete in one second. Floating-point numbers can be stored in memory in different formats based on the precision required. They can be stored in *Half Precision* (HP) format (which occupies 16 bits in memory), *Single Precision* (SP) format (which occupies 32 bits), *Double Precision* (DP) format (which occupies 64 bits), or *Quadruple Precision* (QP) format (which occupies 128 bits). SP format stores floating-point values using 32 bits in memory, whereas the DP format occupies 64 bits.

A modern CPU, such as Intel's Xeon 8180 Platinum Processor (from the Skylake microarchitecture family), has many cores (the Xeon 8180 has 24 cores running at a maximum turbo frequency of 2.3 GHz). If all cores execute AVX-512 instructions, the Xeon 8180 can reach a peak performance of 4.12 TFLOPS [104] single-precision performance, and about 2.06 TFLOPs of double-precision performance [104] ($1\ \texttt{TFLOPS} = 10^{12}\ \texttt{FLOPS}$).

A modern GPU, such as NVIDIA's A100 GPU (based on NVIDIA Ampere architecture), has many single and double-precision cores (the A100 has 6912 single-precision cores and 3456 double-precision cores). The A100 can reach a peak performance values of 19.5 TFLOPs [74] single-precision and 9.7 TFLOPs [74] double precision.

2.5 The Properties of Spatial Locality

Spatial locality is the property that instructions and data entities tend to be stored relatively close together in an address space. Workloads exhibit high spatial locality when they request data from neighboring memory locations frequently. A sparse matrix multiplication kernel (SpGEMM kernel) consists of accessing elements in random rows and columns of the input matrix (most of the rows and columns contain few non-zero elements), which results in a somewhat random memory access pattern. A large number of non-zero elements in every row can lead to high spatial locality in matrix multiplication kernels, but sparse matrices tend to have fewer non-zero elements per row, thus the resulting access-pattern of rows leads to accessing few non-zero elements scattered across the input matrix. Hence SpGEMM kernels exhibit low spatial locality.

This makes the SpGEMM kernel more difficult to optimize. In an ideal case, for efficient SpGEMM computations, we would require a large control logic area per core (present on the CPU), along with a large number of CPU cores, to achieve the high computational performance possible on a single GPU.

2.6 Sparse Matrix Storage Formats

Matrices are generally stored in a one-dimensional linear array of contiguous memory, organized in either a row-major or column-major format. Row-major format stores subsequent elements of a row sequentially in the address space, whereas column-major stores subsequent elements of a column sequentially [99]. While these storage formats work efficiently when working with dense matrices, they encounter performance issues when working with sparse matrices [28]. The large number of zeros present in sparse matrices cause the row-major and column-major formats to store mostly zeros in memory, thus leading to inefficient usage of valuable memory space.

The *Compressed Sparse Row* (CSR) storage format stores only the non-zeros of a sparse matrix, recording the index of only the non-zero elements in each row. CSR packs the non-zeros of each row in a single linear array (data array) and the indices corresponding to each element in another linear array (column-index array). A third array is used to track the number of elements in each row (i.e., the row-pointer array). Hence, in order to access subsequent non-zeros in a sparse matrix, one can iterate over these dense arrays of non-zeros (i.e., the data array and column-index array). The resulting CSR format is efficient in terms of storage, as well as in terms of computation, as compared to using dense storage formats. Similar concepts of compressed sparse storage format

can be extended to storing elements of columns together. Such a matrix storage format is referred to as *Compressed Sparse Column* (CSC) format. We extensively use the CSR storage format in our SpGEMM kernel implementation to pack sparse matrices to fit in our on-chip memory.

Chapter 3

Related Work

Equipped with a brief introduction on the architecture of the CPU and GPU, and informed with an understanding of spatial locality described above, in this chapter, we discuss the class of computations that are the target of this thesis. We begin by discussing libraries that are commonly used in high-performance computing and take a deeper look into prior implementations of SpGEMM workloads.

3.1 SpGEMM on CPU

The *Basic Linear Algebra Subprograms* (BLAS) [72] is a standard set of libraries that provide high-performance application programming interfaces (APIs) to perform linear algebra operations. Many hardware vendors provide their own performance-tuned implementations for BLAS, providing advantages for their own architecture. For example, Intel provides Intel *Math Kernel Library* (MKL) [105] for their x86 processors. Intel's implementation of these APIs exploits unique architectural features (i.e., hardware extensions) present on their CPUs to boost their performance. Some of these extensions include:

- *Streaming SIMD Extensions 4.2* (SSE4.2)
- *Advanced Vector Instructions 2* (AVX2)
- *Advanced Bit Manipulation* (ABM)
- *Bit Manipulation Instruction Set 2* (BMI2)
- *3-Operand Fused-Multiply-Add* (FMA3)

- *Advanced Encryption Instructions* (AESI)
- *Multiprecision Add Carry* (ADX)
- *Carry-less Multiplication Extension* (CLMUL)
- *16-bit Floating Point Conversion* (F16C)

Other manufacturers provide similar extensions. For the Zen architecture, *Central Processing Unit* (AMD) provides the BLIS library, an optimized software implementation of the BLAS subroutines.

In this thesis, we focus on the SpGEMM APIs from the BLAS library in our analysis. In prior work, Xie et al. [109] described an optimized SpGEMM kernel, evaluated on both a CPU and a GPU. They used deep learning to train their model (called MatNet) to learn the data distribution patterns of a matrix. Their algorithm chooses the best format to represent the input data based on the MatNet model's decisions. By performing input data transformations with MatNet, they were able to accelerate a SpGEMM kernel by $3.27\times$ over Intel's MKL platform and $13.17\times$ speedup over AMD's platform.

Nagasaka et al. [68] compare the performance of most publicly available implementations of SpGEMM kernels and propose their own implementation based on hashing and heap-based algorithms. They concluded that specific implementations work better based on the pattern of non-zeros in the input data set.

3.2 SpGEMM on GPU

One of the largest GPU chip manufacturers is Nvidia. Similar to the CPU vendors, Nvidia provides its own library for linear algebra kernels. In this thesis, we focus on efficient SpGEMM kernels. Nvidia has developed its own libraries that provide SpGEMM kernels, released as part of cuSparse and CUSP packages. cuSparse is Nvidia's BLAS implementation to support all sparse operations. CUSP is an open-source C++ library of generic parallel algorithms used for sparse linear algebra and graph computations on CUDA architecture GPUs.

Many other attempts have been made to optimize SpGEMM kernels on a GPU. Nagasaka et al. [68] present an algorithm for efficient sparse matrix multiplication on a Pascal GPU. Their approach uses a row-counting method (i.e., counting the number of intermediate partial-products and then grouping rows based on the number of partial-products in each row) [68]. In addition to

accelerating the kernel, they also try to minimize the total memory required for this operation. They achieved a $4.3\times$ speedup on single-precision and $4.4\times$ speedup on double-precision compared to existing SpGEMM libraries. They also reduced memory usage by 14.7% for single precision and 10.9% for double-precision, on average.

In general, it is difficult to optimize SpGEMM workloads on GPUs due to the random data access patterns and the large memory footprint of the intermediate data generated. We look at accelerators designed specifically for SpGEMM in the next section.

3.3 SpGEMM Accelerators

Next, we focus on previous attempts made in designing domain-specific accelerators for SpGEMM workloads. Table 3.1 provides a comparison between the various accelerators and kernel implementations developed specifically for SpGEMM workloads.

Zhang et al. propose the SpArch accelerator [114], an accelerator designed to speed up SpGEMM kernels. They designed a kernel using the outer-product method for matrix multiplication. The issue with outer product multiplication is the large number of intermediate partial products produced by this approach. Their work's key contribution addresses the partial product generation problem by designing a streaming-based merger into the processing pipeline, combining the multiplies with the merge stage of the partial products. This allows the partial products to be merged on-chip immediately after they are produced. In addition to this optimization, they also proposed a condensed matrix representation and a Huffman tree scheduler to gain further speedup. They report an average speedup of $18\times$ over Intel MKL, cuSparse, and CUSP libraries. Although their implementation provides considerable speedup compared to other libraries, the merge-tree implementation occupies a large portion of the overall chip area. Approximately 60% of the chip area is dedicated to the merge tree implementation, while just 1.6% of the chip area is devoted to a multiplication array. In terms of energy, 55% of the energy is spent on merging partial products on-chip. The extra hardware area and energy requirements devoted to partial product merging leave room for further optimization, both in terms of accelerator design and the SpGEMM algorithm.

Qin et al. [83] propose SIGMA, an accelerator that tackles irregular memory accesses in SpGEMM kernels. The fundamental block of their architecture, known as the Flexible Dot Product Engine (Flex-DPE), consists of switchable interconnects that allow them to build a flexible network topology. Leveraging a flexible and scalable network topology allows them to keep the utilization of their processing elements high. They reported that SIGMA could obtain approximately

Research Papers	SpGEMM Kernel	Accelerator	Features
SpGEMM on GPU [68]	Outer Product	NVIDIA Pascal GPU	On-Chip shared memory merging Hashtable for partial products
OuterSPACE [76]	Outer Product	OuterSPACE	Algorithm-Hardware co-design
ExTensor [42]	Inner Product	ExTensor	Hierarchical elimination of computation in the presence of sparsity
MatRaptor [96]	Row-wise Product	MatRaptor	New sparse storage format C^2SR Hardware Sorting
Sunway Taihu-Light [20]	Partitioned Outer Product	Sunway	Novel partitioning method
SpArch [114]	Outer Product	SpArch	Streaming based merger Condensed matrix representation Huffman tree scheduler
ALRESCHA [9]	Inner Product	Alrescha	Data-dependent task reordering Locally-dense storage format
Synergistic CPU-FPGA [94]	Row-wise Product	CPU-FPGA	Cooperative CPU-FPGA platform New intermediate representation based on communication packets
SIGMA [83]	Row-wise Product	SIGMA	Novel reduction tree microarch. Flexible Interconnects
SMASH (Our approach)	Row-wise Product	PIUMA	Hashtable based on-chip merge Dynamic load balancing In-memory computation using PIM modules

Table 3.1: SpGEMM Accelerator Comparison

a $3\times$ speedup compared to other state-of-the-art accelerators, including a TPU [36], EIE [40], SCNN [77], OuterSPACE [76], Eyeriss v2 [19], Packed Systolic [56], and Cambricon-X [113].

In 2018, Pal et al. introduced their SpGEMM accelerator called OuterSPACE [76]. They took a two-phase approach to implement their SpGEMM kernel. In their first phase, called the *multiply phase*, they perform an outer product of the two input matrices to produce partial products. In the subsequent phase, called the *merge phase*, they merge these partial products to form the output matrix. Although this approach is not new, their work's novelty lies in their mapping of these phases to the OuterSPACE architecture. The computation of an outer product when using sparse matrices causes poor data reuse and unbalanced workload distribution. The OuterSPACE architecture is designed, keeping in mind these problems associated with SpGEMM. With asynchronous *Single Program, Multiple Data* (SPMD) style worker cores coupled with memory hierarchies and shared

reconfigurable caches, OuterSPACE delivered an average $7.9\times$ speedup over Intel's Math Kernel Library, $13.0\times$ over cuSPARSE, and $14.0\times$ over CUSP.

Liu et al. [61] introduce another accelerator that focuses on optimizing SpGEMM kernels for mobile CNNs, using systolic arrays of Tensor Processing Elements. ALRESCHA [9] is another accelerator that differentiates itself by having two parts to its architecture; a fixed compute unit and a light-weight re-configurable engine. This allows them to adapt to input data sparsity patterns.

Many prior studies on SpGEMM accelerator development have proposed their own sparse matrix storage format, including a condensed matrix representation in SpArch [114], the REAP intermediate representation for the CPU-FPGA accelerator [94], C^2SR for MatRaptor [96], and Locally-Dense storage format for ALRESCHA [9]. A common issue with these schemes is that the degree of reformatting or data rearrangement needed in a sparse matrix, depending on the underlying architecture, can impact the speedup obtained by these accelerators. Memory access latency is a major bottleneck for SpGEMM accelerators. Dividing, rearranging, and grouping data based on the sparsity patterns and accelerator architecture reduces redundant accesses and increases data locality. Though these novel formats boost performance, they incur associated conversion costs since the sparse input matrices are typically stored in CSR, CSC, or *ELLPACK* (ELL) storage formats. These costs are either in terms of additional hardware requirements or performance degradation or both. The design of any storage format should consider whether the benefits provided can amortize the data transformation cost. Considering previous efforts to speed up sparse workloads and considering the problems faced by these accelerators, we propose our own SpGEMM kernel implementation developed on a general hardware accelerator architecture.

Chapter 4

PIUMA Architecture and Simulator

The Programmable Integrated Unified Memory Architecture (PIUMA) is being developed by Intel [3], as part of DARPA's *Hierarchical, Identify, Verify, Exploit (HIVE)* program [47]. The Hierarchical Identify Verify Exploit (HIVE) project recognizes the challenges involved with graph analytics and aims to achieve a 1,000× performance/Watt improvement over the previous state-of-the-art system [64]. The PIUMA system is a scalable systems architecture designed to accelerate graph-based applications. The system is designed to handle sparsity, bearing in mind the highly-random data access patterns present in graph workloads.

4.1 PIUMA Architecture

This section provides a detailed description of Intel's PIUMA machine, exploring some of the key features of this architecture that are exploited in our implementation of the SpGEMM kernel. We highlight some of the problems faced in SpGEMM and focus on PIUMA's components that help tackle these challenges. Primarily, we focus on the following architectural features of PIUMA [3]:

1. PIUMA Cores
    (a) Multi-Threaded Core (MTC) and
    (b) Single-Threaded Core (STC)
2. Offload Engines (OE)
    (a) DMA engine and

(b) Collective engine

3. Global Address Space

4. Network

4.1.1 PIUMA Cores

The PIUMA cores form the fundamental computational unit of this architecture. They are designed to exploit the inherent parallelism exhibited by graph-based workloads. In general, graph workloads are more memory intensive than compute-intensive workload [31]. A key principle in PIUMA is to provide a high degree of parallelism to hide memory latency. PIUMA's large number of threads are capable of keeping many memory requests in flight. The PIUMA cores can be classified into two types:

1. The *Multi-threaded Core* (MTC) and

2. The *Single-threaded Core* (STC)

We take a closer look into the architectural layout of each of these cores.

4.1.1.1 The PIUMA Multi-threaded Core

A multi-threaded core consists of one pipeline each [3]. The MTC can issue at most one instruction per cycle, providing for an energy-efficient design [3]. In addition, the MTC also has an associated register file Register File (RF). Each RF stores the context of up to 16 threads simultaneously, allowing a single MTC to be shared by up to 16 threads. Each thread represents a single stream of executions. The MTC resources are shared across these 16 threads, utilizing a round-robin resource allocation routine. When a stage of the MTC's pipeline stalls, a new thread is swapped to execute, hiding latency and keeping the pipeline stages full.

4.1.1.2 Single-Threaded Core

Unlike the MTC, the STC comprises a single thread of execution (executing a single stream of instructions). Since a single instruction stream is being executed every cycle (as opposed to the round-robin scheduling of MTCs), a higher priority is given to this single thread, making it capable of handling performance-sensitive tasks [3]. The STCs are in-order blocking (on misses),

cores designed to lower power consumption as compared to the out-of-order pipelines [3]. The primary purpose of STCs is to perform memory and thread management tasks.

Both the single-threaded and multi-threaded cores are equipped with L1 instruction caches and L1 data caches. Graph applications are known for their irregular data access patterns [31]. The resulting irregular memory access patterns lead to low L1, L2, and L3 cache utilization due to low spatial locality [50]. As graph workloads exhibit poor locality, in PIUMA architecture, no higher cache levels are included to save on power consumption [3]. All caches throughout the PIUMA system are non-coherent. Although cache coherency is helpful to maintain data uniformity, it is associated with large overheads. Caches can be made coherent using protocols such as MSI, $MESI$, $MOSI$, and $MOESI$ (Modified, Owned, Exclusive, Shared, Invalid) [39]. These protocols can cause the executing pipeline to stall for thousands of cycles to fetch the new data value from the main memory or neighboring caches. The PIUMA architecture does not provide cache coherency. It thus becomes the responsibility of the programmer to avoid modifying shared data and flush caches as required [95]. In addition, prefetching is disabled to limit power consumption [3].

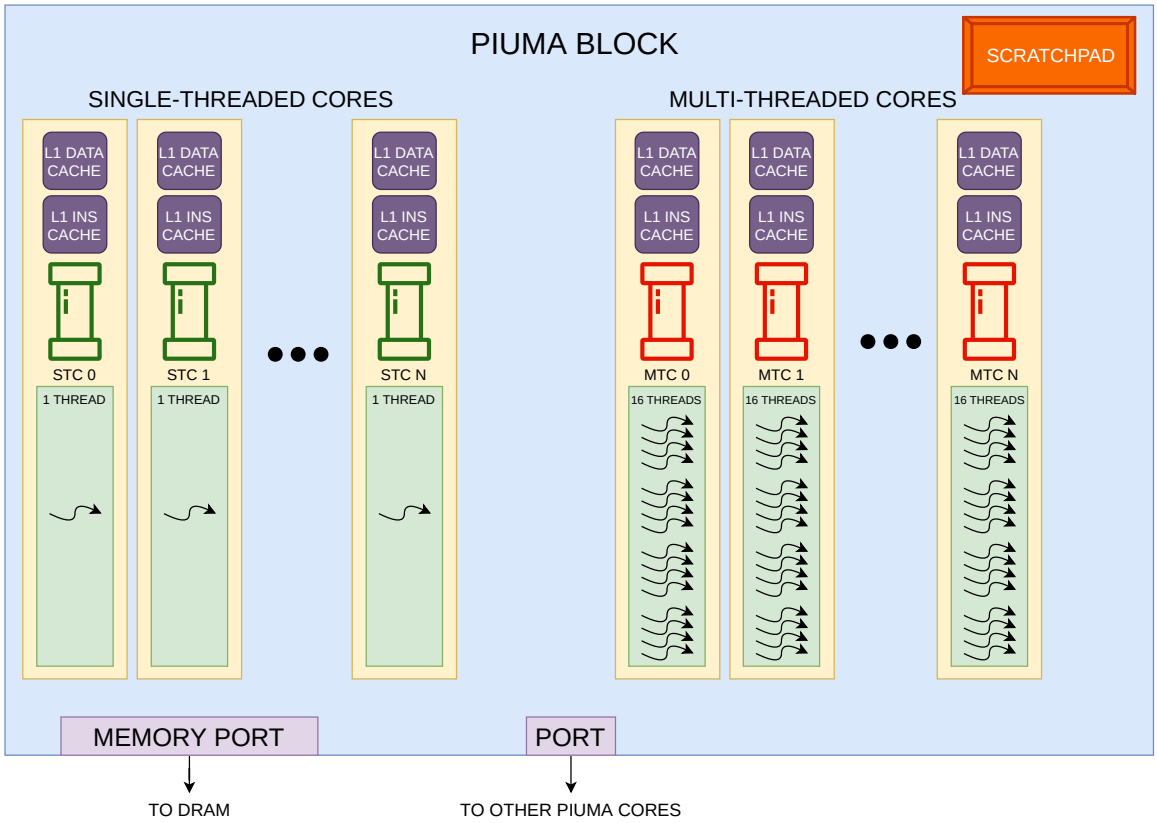

Figure 4.1: A single PIUMA block.

Multiple MTCs and STCs are grouped in a block (see Figure 4.1). Each block consists of low latency, user-accessible storage, scratchpad (SPAD) memory. Programmers can use this shared storage for storing data with high temporal locality [2].

4.1.2 Offload Engines

Traditionally, following the Von Neumann architecture [90], programs and data are commonly stored in memory and fetched by the processor for execution. This architecture will be limited by the throughput of the memory channels, commonly known as the Von Neumann bottleneck [70]. In some novel architectures [60, 69, 79], this limitation was overcome by introducing *Computational RAM* (C-RAM) technology (i.e., processors in memory). C-RAM is similar to DRAM but with a vector processing element embedded on the same chip as the memory. This enabled C-RAM chips to service instructions other than a simple load or store and perform operations such as scatter and gather.

The PIUMA architecture provides Offload Engines (OE) to support the PIUMA cores in memory-related operations. This allows selected *Single Instruction Multiple Data* (SIMD) instructions to be executed in memory. Some examples of such SIMD instructions supported by PIUMA include:

1. $Copy$: Copy a chunk of data from one section of memory to another.
2. $Strided\ Copy$: Similar to a copy, but every n^{th} element is copied.
3. $Gather$: Read an array of data and compute its sum.
4. $Scatter$: Broadcast a single value to multiple locations in memory.

We discuss two offload engines that we use to speed up our SpGEMM algorithm.

4.1.2.1 DMA Engine

Data movement forms an integral part of graph algorithms. One of the major bottlenecks in graph applications is memory throughput [31]. The *Direct Memory Access* (DMA) Engine reduces the workload on PIUMA cores by executing the memory operations (i.e., $loads$ and $stores$). A single $copy$ instruction or $gather/scatter$ instruction can replace thousands of $load$ and $store$

instructions issued by the cores. The DMA engine carries out the underlying task of copying multiple bytes of data or broadcasting data to multiple memory locations. All instructions issued to the DMA engine run in the background (non-blocking), freeing the core to execute other instructions.

4.1.2.2 Collective Engine

One important element of many parallel algorithms is the required synchronization. The PIUMA architecture is equipped with a collective engine that provides system-wide barriers and reduction operations [3].

One of the key features presented by the PIUMA architecture is the ability to offload instructions over the fabric to remote cores. Threads can wrap their instructions in network packets and forward them to remote threads for execution. These instructions are called remote or network instructions.

When a thread intends to update data present in remote memory (memory physically connected to neighboring cores), it can send remote instructions to be executed by another thread, local to that memory. Thus, instead of streaming data over the network, we stream instruction packets to the core that is physically connected to that memory chunk. Network instructions can be helpful in two ways:

1. forwarding instructions to threads that have low latency access to data can improve performance, and

2. distributing the workload among threads can improve workload balance.

Remote atomic instruction is one such networked instruction that allows atomic instructions to be executed by PIUMA cores in memory that is remote. We make use of remote atomics in our algorithm to update the partial products in our hash table.

4.1.3 Global Address Space

Distributed Global Address Space (DGAS) is a model in which the memory address space is logically divided, and sections of this address space are local to each computing thread. DGAS allows using an SPMD style of programming [111], while supporting data addressing semantics similar to a shared memory system.

The PIUMA architecture is built using DGAS, allowing data present on any core to be accessible by any other PIUMA core/thread. This allows the programmer to worry less about the

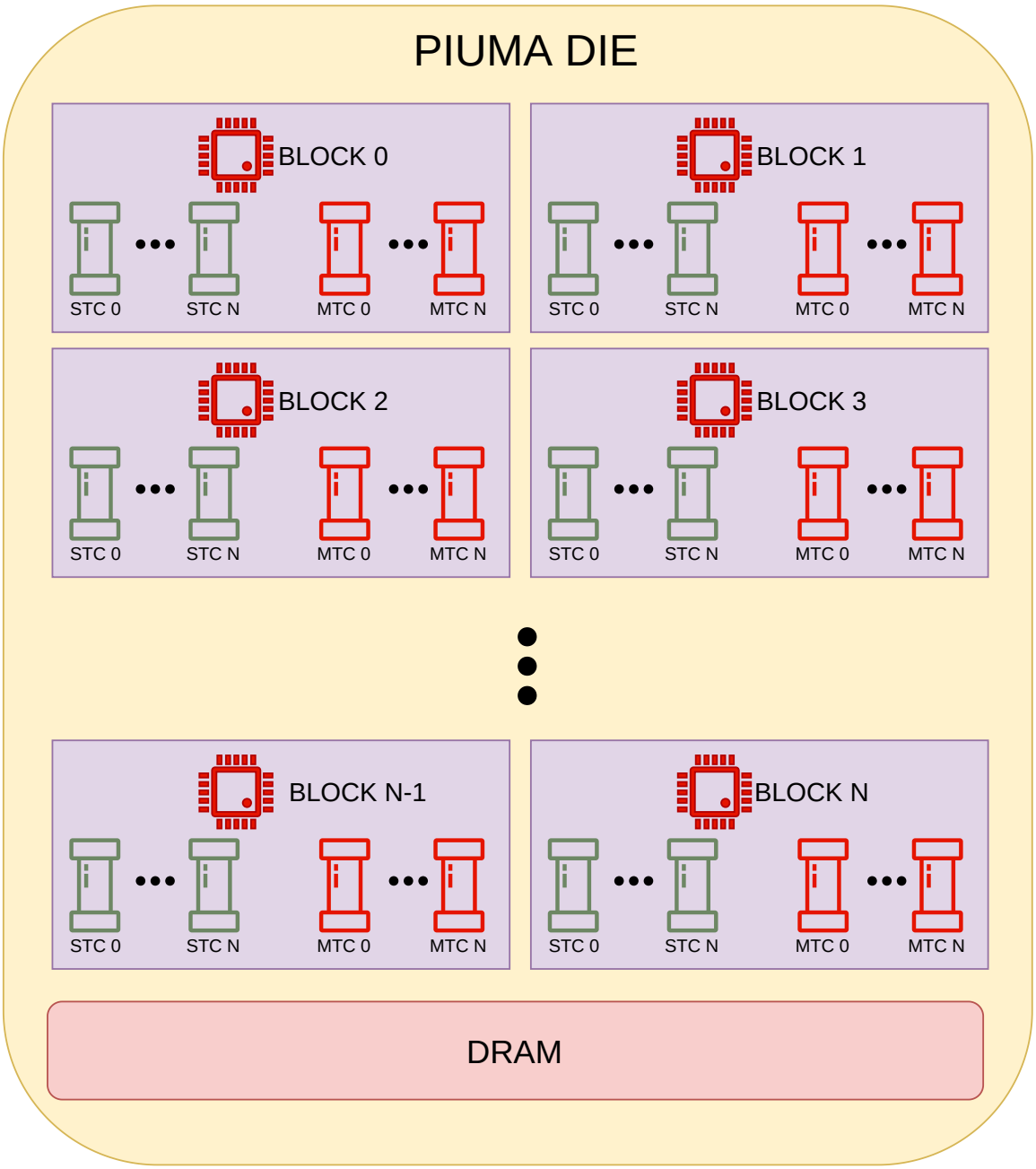

Figure 4.2: A single PIUMA die.

scope of memory access pointers and focus instead on parallelizing the workload's execution. Each PIUMA thread has an affinity to specific sections of the DGAS. Despite each thread having access to the entire address space, local memory accesses will experience lower latency than remote accesses. Thus, data stored in the DGAS partition belonging to a thread is said to have an affinity to that thread. We use DGAS when designing our sparse matrix algorithm to broadcast sections of the input matrix from the first core to all other cores involved in computing the workload. This will transfer the elements to each thread's local memory, as described in Section 5 in detail.

The PIUMA system consists of Address Translation Tables (ATT). ATT contains reconfigurable rules to translate application memory addresses to physical memory locations, enabling us to rearrange the address space as needed by the application [3].

In addition, the PIUMA memory controllers are redesigned to support native 8-byte accesses [3]. Instead of a full cache line fetch, the memory controllers can selectively fetch 8-byte words, reducing redundant memory fetches.

4.1.4 Network

The PIUMA network connects blocks (groups of MTCs and STCs) together and forwards memory requests to remote memory controllers. The PIUMA system is configured in a HyperX topology to achieve high bandwidth and low latency [3, 6]. This allows the network to have a high radix and a low diameter. The higher-level links are optical to sustain high-bandwidth at low power consumption levels.

A single PIUMA block consists of both the STCs and MTCs. Each STC and MTC is accompanied by an L1 instruction cache and an L1 data cache. Each block is accompanied by local high-bandwidth Scratchpad memory (see Figure 4.1). Multiple PIUMA blocks are laid out together to form a die, as shown in Figure 4.2). Figure 4.3 presents the layout of an entire PIUMA system at node, subnode, and die level.

The PIUMA system is a novel approach to tackle the problems present in graph-based workloads. The simple, in-order, multi-threaded cores and non-coherent caches aid in hiding latency of random memory accesses present in graph applications. The offload engines, including the DMA engine and collective engine, work alongside the PIUMA cores to help with memory operations and synchronization. The distributed shared global address space makes it convenient for programmers to implement distributed kernels. Finally, the HyperX topology of the PIUMA network with optical interconnects delivers a design that can scale out to multiple nodes, allowing computation of graphs

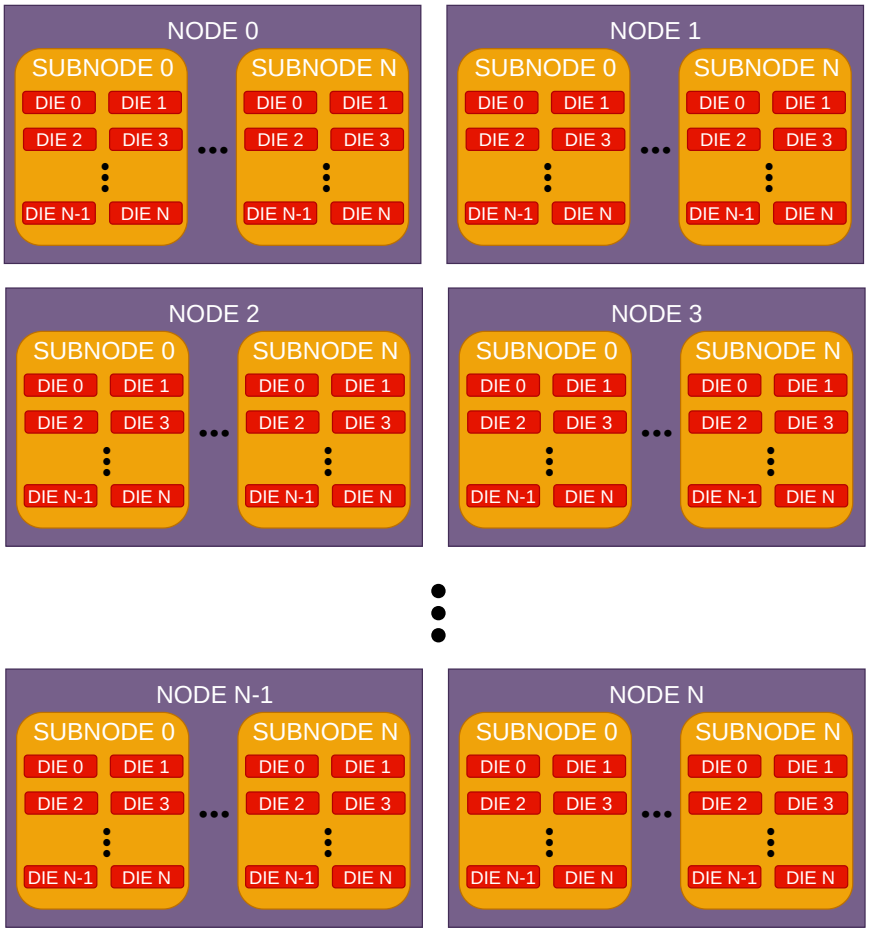

Figure 4.3: The PIUMA system.

with trillions of vertices [3].

4.2 Simulation Methodology

The evaluation of new computer architecture features is commonly evaluated pre-silicon using a simulator [7]. Simulators are software models used to simulate the behavior of various design features. In contrast to simulation models, analytical models are statistical models that can mathematically evaluate elements of a computer architecture. Owing to the complexity of today's computer architectures and the large number of configurable parameters, analytical models tend to produce inaccurate results, hence are not suitable for evaluating computer architectures [7].

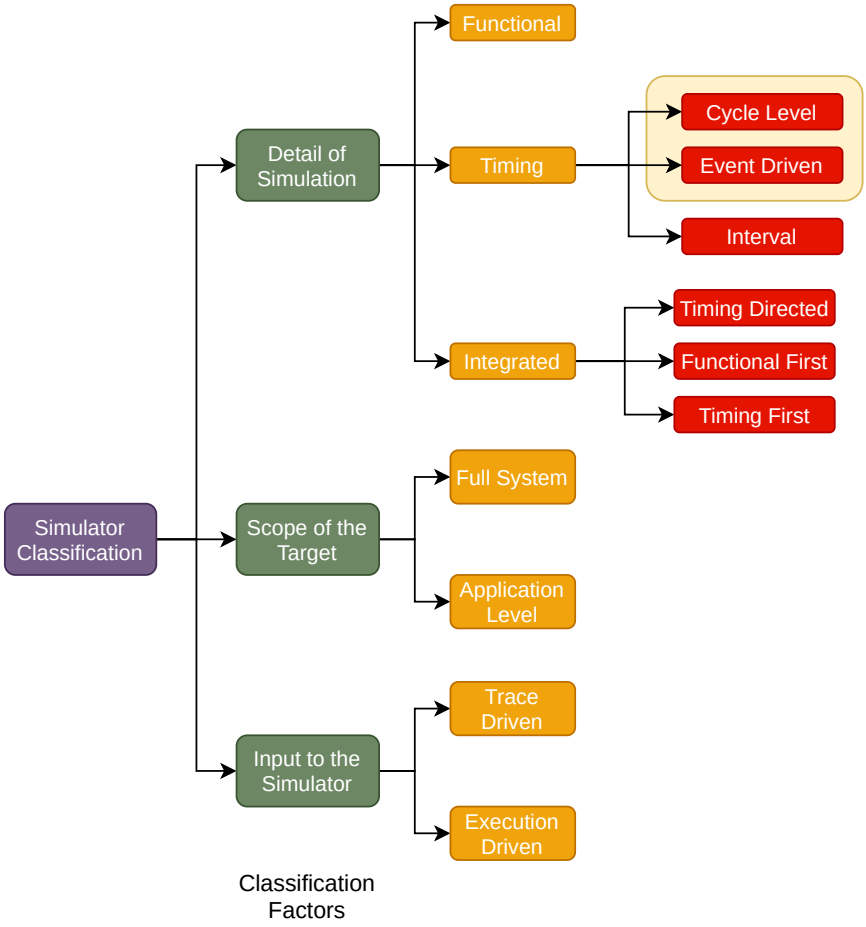

Figure 4.4: execution-driven simulator.

The design cycle and silicon fabrication process required to produce physical hardware is

both time-consuming and can incur high cost. Simulators allow architects to make design decisions before the hardware pre-silicon, thus lowering the hardware development costs [7], making them an integral part of today's hardware design process.

4.2.1 Simulator Classification

For our simulation of PIUMA architecture, we use the Sniper simulator [43]. To begin, we first review different classes of simulators to better understand the advantages of the Sniper simulator over others choices. Simulators can be grouped into various classes based on a range of factors, including the level of detail of the simulation, the scope of the target (the system that is being simulated), and input that drives the simulator. Figure 4.4 provides an overview of the various classes of simulator [7]. We briefly describe each class and discuss the advantages/disadvantages provided by each.

Our classification begin by consider the level detail of simulation:

- **Functional Simulators**: Functional simulators focus on the functionality of the modeled architecture. They provide for emulation of the target ISA, but they do not implement the underlying microarchitecture of the target system, making them faster than other simulators.
- **Cycle-level Simulators**: Cycle-level simulators model the operations of a processor cycle-by-cycle. Cycle-level simulators are relatively slower than functional simulators and consume a significant amount of memory resources.
- **Event-driven Simulators**: Event-driven simulators target events instead of cycles. Simulations steps through time to arrive at events when they are scheduled [63], thus saving time by not simulating cycles that are not scheduled.
- **Interval Simulators**: While cycle-level simulators are accurate, they are very slow. Event-driven simulators are fast, but compromise accuracy. Interval simulators strike a balance between speed and accuracy [33]. Interval simulators work on the fact that instructions flow through a pipeline can be broken down into sets of intervals, based on miss events (miss events include branch mispredictions and cache misses). A distinct branch misprediction simulator and cache miss simulator can then be used to accurately evaluate every interval's performance.
- **Integrated Timing-directed Simulators**: Functional simulators are often integrated with timing simulators. In a timing-directed simulator, the functional-simulator records the archi-

tectural state (register and memory) of the processor and forwards it to the timing simulator. The timing simulator then uses these values to perform corresponding computations. There exists heavy communication between the functional simulator and the timing simulator.

- **Integrated Functional-first Simulators**: In a functional-first simulator, a functional-simulator leads the timing simulator. The functional simulator generates an instruction trace and forwards it to the timing simulator. Functional-first simulators are similar to trace-driven simulators, with a distinguishing factor being the traces are generated by the functional simulator and forwarded to the timing simulator immediately while trace-driven simulators store traces on file after generation.

- **Integrated Timing-first Simulators**: In a timing-first simulator, the timing simulator leads the functional simulator. The timing simulators simulate the microarchitecture at cycle-level and then use functional simulators for verification purposes. In cases where the timing simulator results do not match the functional simulator, the timing simulator flushes its pipeline and restarts from the fetch cycle for that instruction.

Classification based on the scope of the target architecture is as follows:

- **Full System Simulators**: Full system simulators support booting the entire operating system (OS) and run target applications in that OS. Full system simulators are significantly slower and are generally used to simulate I/O devices.

- **Application-level Simulators**: As opposed to full system simulators, application-level simulators only simulate target applications. Since they do not suffer the high overhead of simulating the operating system or system calls. This class of simulator is significantly faster.

Classification based on input to the simulator is as follows:

- **Trace-driven Simulators**: Trace-driven simulators use trace files as input. Trace files are pre-recorded streams of instructions from a previous run of the application. Trace files are usually stored in a file system and occupy a large amount of space, which in some cases, becomes a bottleneck for simulation [48][30].

- **Execution-driven Simulators**: In contrast to trace-driven simulators, execution-driven simulators use application binaries as input instead of trace-files. These application binaries are significantly smaller in size as compared to trace files. Execution-driven simulators can simulate misspeculated instructions, unlike trace-driven simulators.

4.2.2 Sniper Simulator

Sniper is a parallel, multi-core x86 architecture simulator [43]. For functional simulation, Sniper uses the Graphite simulator, which is based on the Pin tool [65]. Sniper is classified as an Application-level simulator, making it faster than full system simulators.

Sniper is an interval-based simulator. Instead of simulating each instruction, Sniper breaks down the stream of instructions into discrete sets called intervals. These intervals are based on events, such as cache misses and branch predictions. Hence, the Sniper simulator is significantly faster owing to it's ability to '*jump*' between miss events [43]. A special branch predictor simulator coupled with a memory system simulator can then be used to evaluate miss events. The metrics of these simulators are then compiled with the analytical model's findings to estimate the duration of every interval [7].

In this thesis, to evaluate performance of our proposed scheme, we use a modified implementation of the Sniper simulator. An interval-based simulator, like Sniper, simulates processors at a higher level of abstraction. Using such a simulator allows us to simulate multi-core processors efficiently (several million instructions per second), as compared to a detailed cycle-accurate simulator. Although cycle-level simulators, such as Gem5 [14], are more accurate than high-level simulators, they tend to be significantly slower, limiting our options to simulate a range of hardware configurations [17]. To record these observations, we use a host machine based on the machine configuration described in Table 4.1. Our target architecture simulator configuration is as shown in Table 4.2.

Item	**Description**
Manufacturer	Intel Corporation
System Details	Intel Server Board S2600TP Family
CPU	Intel Xeon CPU E5-2699A
Threads per core	2
Total cores per CPU	18
Number of Sockets	2
Total threads in node	72
Max CPU Frequency	3.4 GHz
RAM	256 GB
OS	CentOS Linux release 7.7.1908 (Core)

Table 4.1: Simulator host machine specifications.

Scope	Configuration	Value	Description
Machine Global	Rack Count	1	Number of racks in the System
	Board Count	1	Number of socket units per rack
	Socket Count	1	Number of sockets
	Die Count	1	Number of dies
	Core Count	Varying (1 to 8)	Number of cores per die
Socket Global	DRAM Count	1	Number of MCs to external DRAM banks per socket
	DRAM Size	unlimited	Size in MB per MC for external DRAM
Core Global	STC Count	2	Number of STCs per core
	MTC Count	4	Number of MTCs per core
	Core SPAD Count	1	Total logical Scratchpad enteries per block
	Core SPAD Size	4,096	Size of each scratchpad in KB
	Cache Size	16	Size in KB for Cache Module
	Cache Assoc.	4	Associativity of the cache
	Cache Line Size	64	Line size in bytes of cache module
	Cache Policy	wb-wa	Replacement policies of the cache (Write-back, Write Allocate)

Table 4.2: Simulator target configuration for PIUMA architecture

Chapter 5

SMASH Kernels

We have reviewed the problems present in prior sparse matrix multiplication algorithms. We have also described the PIUMA architecture that we will target in this work. In this chapter, we will present a new SpGEMM kernel that can fully exploit the features of the PIUMA machine.

One of the key design points for our SpGEMM kernel implementation will be to choose between the four general matrix-multiplication approaches (presented in Figure 1.2). The inner product approach for sparse matrix multiplication faces issues due to the slow index-matching process, in addition to exhibiting poor input data reuse [76]. The outer-product approach generates a large number of intermediate partial products. These partial products have to be buffered somewhere for later merging. The high on-chip memory requirements of the outer-product approach make it unsuitable for multiplying extremely sparse matrices.

We introduce our novel implementation of the SpGEMM kernel based on a row-wise product method called *Sparse Matrix Atomic Scratchpad Hashing* or (SMASH). The row-wise product method is beneficial given its high input reuse behavior [83]. It gives us the ability to perform a minimum number of input matrix reads while maintaining low on-chip memory usage.

In this thesis, we present 3 different versions of our SMASH kernel. Each version improves the efficiency of a different section of our *Sparse Matrix Atomic Scratchpad Hashing* (SMASH) implementation.

We adopt an iterative improvement approach to identify bottlenecks in the current version and modify our algorithm to mitigate them in the next version. The following sections will describe in detail the 3 different versions of the SMASH kernel.

5.1 SMASH Version 1: Atomic Hashing

A row-wise product method multiples each element of the first input matrix with an entire row of the second input matrix to generate a partial product for the output matrix. These partial products are then merged to form output matrix elements. Each row of the first input matrix, when multiplied by their corresponding rows from the second input matrix, will generate a series of partial product matrices, as seen from Equation 1.3. This is one of the disadvantages of using a row-wise product method. The intermediate results (partial products) generated need to be stored in the main memory and refetched to be merged back into the output matrix. We overcome this obstacle with our first implementation of the SMASH kernel by using atomic hashing. Instead of writing the partial products back to memory, we implement a streaming mechanism to merge them on-chip, thus avoiding redundant writes to DRAM.

The SMASH implementation can be distinctively divided into three different phases.

1. the window distribution phase,
2. the hashing phase and
3. the write-back phase.

Each phase's completion is accompanied by a synchronization barrier that spans the entire PIUMA system.

5.1.1 Window Distribution Phase

Our window distribution phase begins with reading both input matrices. We store the input data matrices in the CSR format. This helps us in two key respects:

1. The $row\_pointer$ array of the CSR format allows us to compute the amount of memory required to allocate the output matrix. This computation is inspired by Gustafson's algorithm for SpGEMM [38, 32, 34, 32, 55, 87].
2. The element access pattern of the row-wise product method involves obtaining rows of input matrices. Thus, the data arrangement in the CSR storage format improves the spatial locality pattern of our solution.

After reading the input matrix arrays in CSR format, we compute the required amount of memory needed to store the output matrix by counting the total *Floating-point Multiply and*

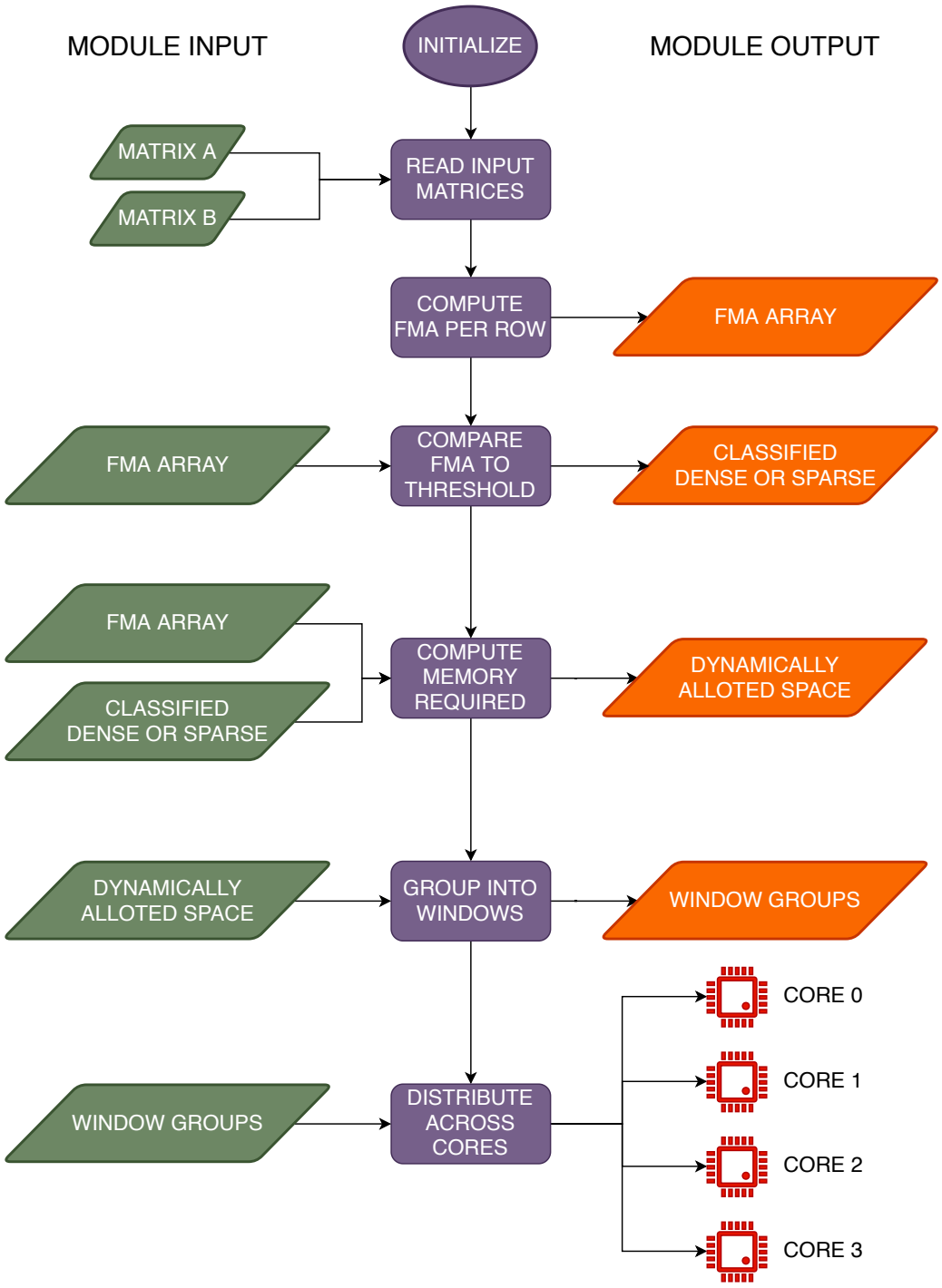

Figure 5.1: Window Distribution Algorithm.

Add (FMA) operations per row. To accomplish this, we use Gustafson's two-step algorithm [38]. The computation of the total number of FMAs per row has a computational complexity of $\mathcal{O}(n)$, where n is one of the dimensions of the input matrix.

Once an array of FMAs is generated, we decide, on a row-by-row basis, if each row should be computed as a dense row or a sparse row. The decision is made by using a threshold value specifying the maximum number of elements that need to be present in a sparse row.

Next, we group multiple rows in a single window to be computed by one PIUMA block. The size of a window is a function of the *Scratchpad* (SPAD) size. Sections of input matrices are then packaged and shipped to individual blocks in network packets using PIUMA's global address space feature. This data is then stored in the block's DRAM, ready to be processed.

Every individual PIUMA block processes its own window independently, regardless of the status of other windows. This allows us to schedule windows to blocks in random order and oversubscribe windows to blocks (Blocks with windows containing largely sparse rows can be over-subscribed as they will end up completing before other windows). Details of the window set up and distribution can be seen in the Algorithm 1.

Algorithm 1: SMASH SETUP

Input: Matrix to be multiplied: mat_A (Stored on DRAM)
Input: Matrix to be multiplied: mat_B (Stored on DRAM)
Output: Final output matrix: $mat\_C = (mat\_A \times mat\_B)$ (Stored on DRAM)
`// Read` mat_A `in CSC format`
$A\_col\_ptr \leftarrow$ Array of column pointers of matrix A in CSC format
$A\_row\_idx \leftarrow$ Array of row indices of matrix A in CSC format
$A\_data \leftarrow$ Array of data values of matrix A in CSC format
$B\_row\_ptr \leftarrow$ Array of row pointers of matrix B in CSR format
$B\_col\_idx \leftarrow$ Array of col indices of matrix A in CSR format
$B\_data \leftarrow$ Array of data values of matrix A in CSR format
$A\_col\_ptr\_copy\_1 \leftarrow$ First copy of column pointer array of A
$A\_col\_ptr\_copy\_2 \leftarrow$ Second copy of column pointer array of A
$hash\_size \leftarrow SPAD\_SIZE$
$matrix\_size \leftarrow 2^{17}$ `// Number of rows or columns in matrix A`
$window\_size$ `// Computed dynamically for each window`
$element\_size \leftarrow$ Size of one element
Launch tasks on all threads
$tid \leftarrow$ Unique thread ID for every thread
$hash\_shift \leftarrow log_2(\frac{Total\ bins\ in\ Window}{Total\ bins\ in\ SPAD})$
EMPTY $\leftarrow -1$ `// A unique flag`
for $w \leftarrow Total\ Windows$ **do**
    `// HASHING PHASE`
    barrier
    `// WRITEBACK PHASE`
    barrier
end

5.1.2 Hashing Phase

In the hashing phase, a global hashtable is created in the SPAD. A single row is allocated to one thread of each MTC in a round-robin fashion. Each element of the row from the first matrix is multiplied with an entire corresponding row of the second matrix. This leads to the creation of partial products. These partial products are hashed into the SPAD using bit-shift hashing.

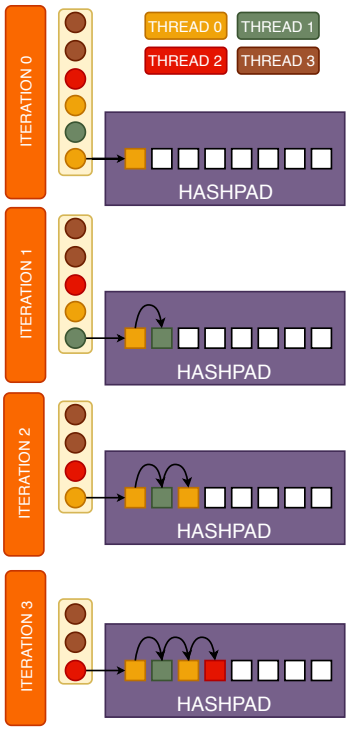

Figure 5.2: Collision Resolution.

To hash using bit-shifts, we ignore the lower n bits and store the elements based on the upper $n-1$ bits. The hashing is performed following Equation (5.1),

$$H(x) = \frac{x}{2^n} \tag{5.1}$$

where n is the number of bits shifted.

In case of collision, we resolve the conflict by adding 1 to the position tag, thus offsetting the storage location by 1 towards the right. We repeat this collision resolution until an empty space is found on the hashtable (Hashtable walk). To prevent data races, we use *atomic compare and exchange* instructions to test for empty locations in the hashtable. This collision resolution is shown

in Figure 5.2.

The use of the upper bits (the high-order bits) for hashing preserves the partial products' sorted order in the hashtable. Whenever collisions occur, the hashtable walk disrupts this order, making the table semi-sorted (most elements will be in sorted order, with only a few outliers). To merge the partial products, we employ a simple *atomic fetch and add* instruction to add partial products together. Our hashtable, with its tag and $data$ pairs, is shown in Figure 5.3. This hashtable

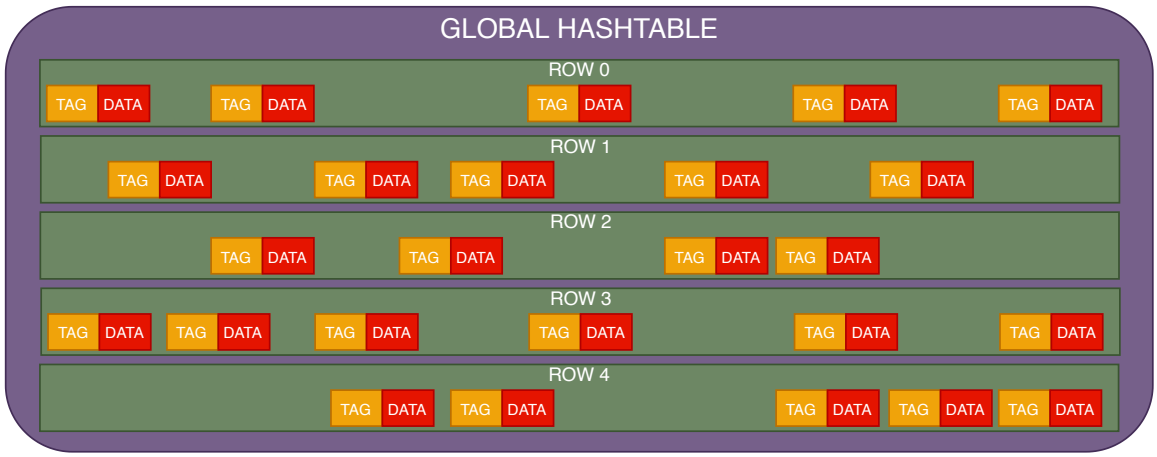

Figure 5.3: Tag-Data Hashtable.

is stored in SPAD memory for quick updates by the atomic instructions.

The pseudo-code for the entire hashing phase is shown in the Algorithm 2.

5.1.3 Write-back Phase

The write-back phase moves the partial products from the hashtable to their final output matrix, stored in DRAM in the CSR format. In the write-back phase, first, we employ a sorting mechanism to sort the partially sorted hashtable. We take advantage of the partially sorted hashtags for our implementation and sort them using a variation of insertion sort. Using insertion sort also helps us merge the remaining partial products by matching their $tags$. This enables us to stream elements from a hashtable, in ascending order, directly to the output matrix present stored in DRAM in the CSR format. Pseudocode for this phase is as shown in Algorithm 5.

An overview of the entire SMASH algorithm is presented in Figure 5.1 and Figure 5.4.

Algorithm 2: SMASH HASHING

```
    // READ PHASE
 1  while Till you reach end of window do
        // Atomically distribute work to each thread
 2      token ← Each thread will receive one unique token
 3      if token_id % 2 = 0 then
 4          row_begin ← A_col_ptr_copy_1[token_id/2]
 5      else
 6          row_begin ← A_col_ptr_copy_2[token_id/2]
 7      end
 8      row_end ← A_col_ptr[token_id/2 + 1]
 9      for i ← Iterate from row_begin to row_end do
10          if Check if we are within our assigned window then
11              col_begin ← B_row_ptr[token_id/2]
12              col_end ← B_row_ptr[token_id/2 + 1]
13              if token_id % 2 = 0 then
                    // Hash EVEN Section
14              else
                    // Hash ODD Section
15              end
16          end
17      end
18  end
19  A_col_ptr_copy_1 and A_col_ptr_copy_2 will now reflect new positions
```

Algorithm 3: SMASH HASHING Even Section

```
 1  for k ← Iterate from col_begin to (col_end − col_begin)/2 do
        // Multiply element from mat_A with that from mat_B
        and store its tag and value
 2      tag ← X coordinate from mat_A element and Y coordinate from mat_B
         element
        // Hash the Tag
 3      tag ← tag >> hash_shift
 4      if SPAD_tag[tag] = EMPTY then
 5          SPAD_tag[tag] ← tag // Store Tag on scratchpad
 6          SPAD_val[tag] ← value // Store Value on scratchpad
 7      else
 8          if SPAD_tag[tag] = tag then
 9              SPAD_val[tag]+ = value // Accumulate Value
10          else
                // Probe for empty space on Scratchpad
11          end
12      end
13  end
```

Algorithm 4: SMASH HASHING Odd Section

```
1  for k ← Iterate from col_end to (col_end − col_begin)/2 do
       // Multiply element from mat_A with that from mat_B
       and store its tag and value
2      tag ← X coordinate from mat_A element and Y coordinate from mat_B
        element
3      tag ← tag >> hash_shift
4      if SPAD_tag[tag] = EMPTY then
5          SPAD_tag[tag] ← tag // Store Tag on scratchpad
6          SPAD_val[tag] ← value // Store Value on scratchpad
7      else
8          if SPAD_tag[tag] = tag then
9              SPAD_val[tag]+ = value // Accumulate Value
10         else
               // Probe for empty space on Scratchpad
11         end
12     end
13 end
```

Algorithm 5: SMASH WRITEBACK

```
   // WRITE PHASE
   // Divide SPAD into 64 equal sections
1  scan_start ← tid × (Total Bins on SPAD)/64
   // Add an offset to scan_start to take into account
   overflow
2  scan_start ← scan_start − OFFSET_THRESHOLD
3  scan_end ← tid × ((Total Bins on SPAD)/64 + 1)
4  index ← 0 // Counter to keep track of last written element
    on C matrix
5  for i ← Iterate from scan_start to scan_end do
6      if SPAD_tag[i] ≠ EMPTY then
           // Match value on minimum heap tree
7          mat_C_tag[tid][index] ← SPAD_tag[i]// Copy tag from
            scratchpad to DRAM
8          mat_C_val[tid][index] ← SPAD_val[i]// Copy value from
            scratchpad to DRAM
9      end
10 end
```

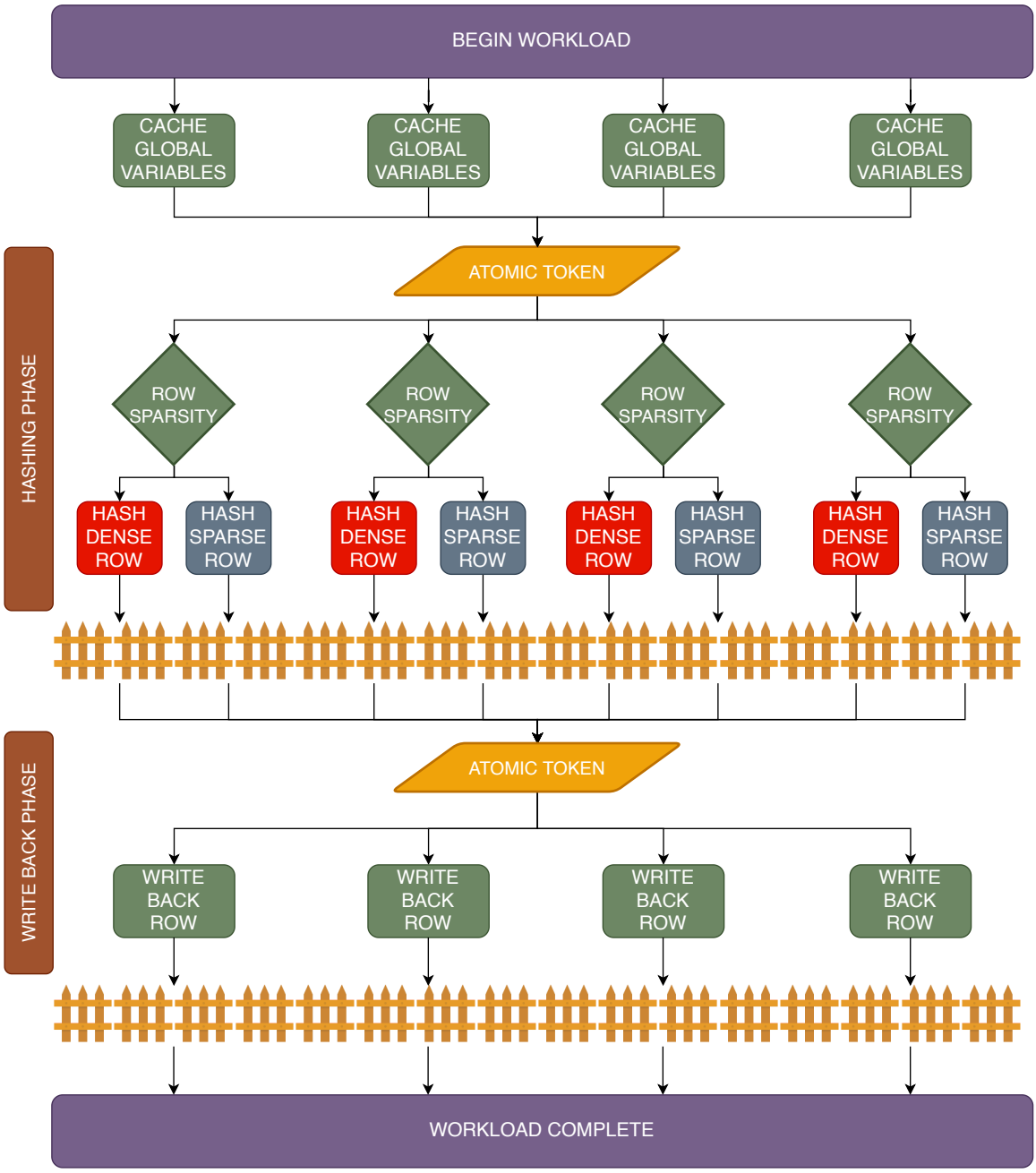

Figure 5.4: SMASH Algorithm

5.2 SMASH Version 2: Tokenization

The previous version allows one row of the input matrix to be assigned to one thread on the PIUMA block. This leads to a considerable imbalance in work across all threads in a block. The SpGEMM datasets are notorious for encountering workload imbalance during kernel execution.

We tackle the issue of workload imbalance by adding a dynamic work scheduler layer in our Hashing phase. Instead of statically allocating rows to threads in a round-robin fashion, we adopt the Producer-Consumer for model row allocation. The dynamic row allocation works as follows:

1. Generate two tokens for every single row present in the window.
2. Each PIUMA thread polls for a single token. Thus, every row is allocated 2 PIUMA threads.
3. These 2 PIUMA threads start hashing the row. The first thread starts from the beginning of the row and hashes the first half of the row (i.e., the even section). The second thread does the same over the second half of the row (i.e., the odd section).
4. Partial products from both threads are hashed into a common hashtable, stored in the SPAD memory.
5. When all of the tokens have been polled, the window execution is completed.

The split of workload between even and odd sections can be seen in Algorithms 2, 3, and 4.

Despite the overhead of polling tokens, tokenization produces great speedup over static allocation, as it achieves a near-perfect distribution of workload across threads. More details of the performance are presented in the following chapter.

Another optimization carried out in SMASH Version 2 is the change of hashing bits. In SMASH Version 1, we used the high-order bits for hashing in the hashtable. The downside of using high-order bits is that if two elements with their tag values close to each other need to be hashed, they will end up hashing to the same position, thus following the collision routine. Hashing on high-order bits groups clusters of adjacent elements together. Instead, in this version, we chose to hash on low-order bits by setting the top n high-order bits to $zero$. Using low-order bits evenly distributes a cluster of elements over the entire hashtable, thus sharply reducing the number of collisions. This can be seen in Figure 5.5

The disadvantage of using low-order bits is that the order of hashing is no longer preserved. The hashtable is no longer partially sorted, as in the case of the previous version. We

overcome this problem by merging all partial products of the same tag before writing them to the hashtable. Even though the order is not preserved and the output matrix in CSR format is not sorted, the correctness of the solution is maintained, as all partial products are properly merged.

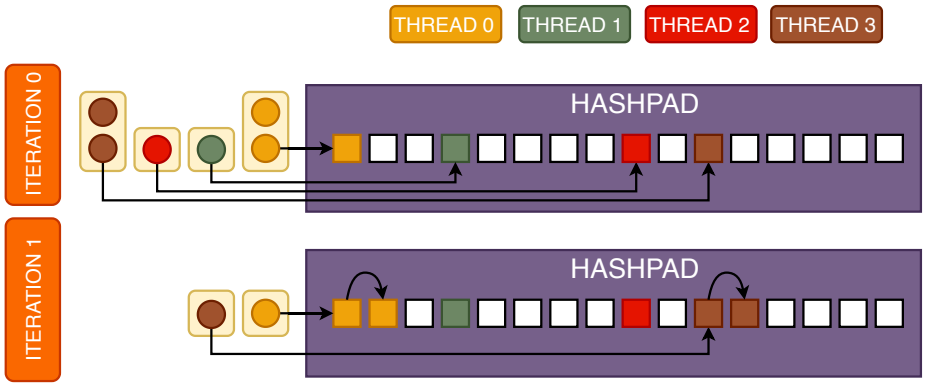

Figure 5.5: Hashing on low-order bits.

5.3 SMASH Version 3: Fragmenting Memory

Previous sections describe how *Atomic hashing* removes redundant accesses to the partial product matrices and *Tokenization* balances workload across PIUMA threads.

This section describes the integration of the DMA engine in the SMASH algorithm by fragmenting the SPAD memory. The DMA engine provides the PIUMA system the capability to move data independently within its global address space while the PIUMA blocks work on other segments of the algorithm.

We incorporate a *copy* instruction to move data from the SPAD to DRAM, as well as a *scatter* instruction to prepare the DRAM for the next window. To use the DMA engine efficiently, we make modifications to our previous version of SMASH. These modifications included:

1. In addition to hashing on a common global hashtable, the PIUMA threads also maintain a private local array that stores the tag values in a dense array.

2. Instead of storing the hashtable in the SPAD, we store the hashtable in DRAM, a memory that has lower bandwidth but more available space.

3. As every element gets hashed on the global hashtable, we store its position in the hashtable in a separate array in SPAD, called the $offset\_array$.

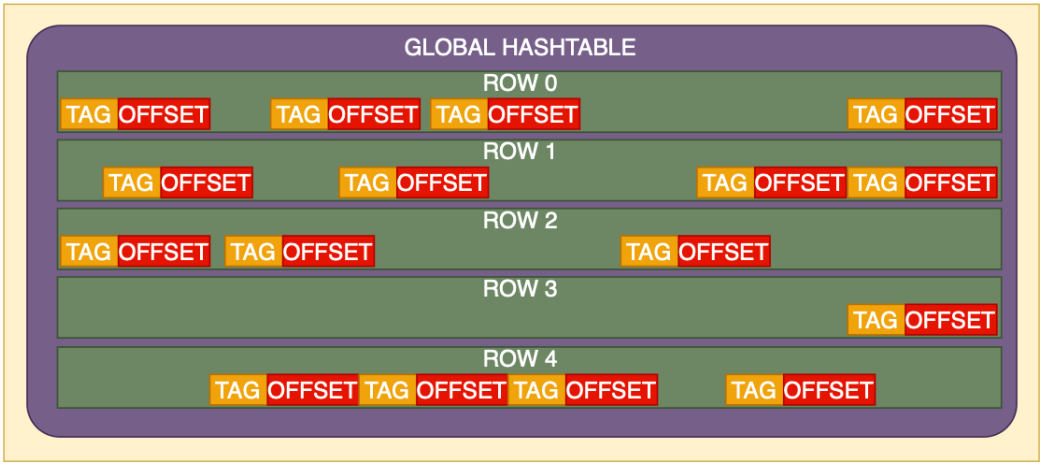

Figure 5.6: Tag-Offset Hashtable in DRAM.

The layout of the SPAD in SMASH V3 can be best described by Figure ,5.7 and the layout of DRAM can be best described by Figure 5.6.

SMASH Version 1 and SMASH Version 2 store tag and $data$ values in a hashtable, scattered across a large array. SMASH Version 3, on the other hand, stores the tag values and $data$ values in dense arrays. These dense arrays can then be easily transferred to the DRAM by simple *copy* instructions. Also, since the DMA engine runs in parallel with the MTCs, the PIUMA threads will not have to spend cycles moving these dense arrays from the SPAD to DRAM.

In the next chapter, we will look deeper into the performance results of each SMASH implementation and the advantages gained by optimizing each version.

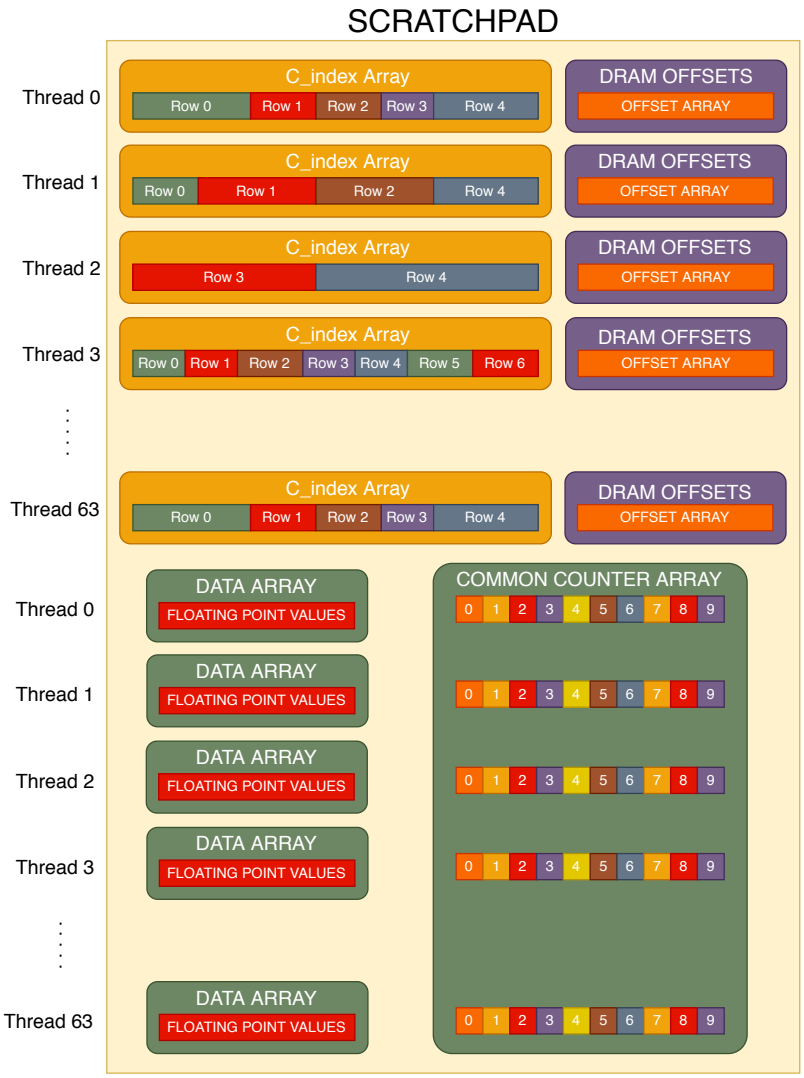

Figure 5.7: SPAD array layout

Chapter 6

Results

This chapter begins by providing more details on the evaluation methodology used in our experiments. Then, following this methodology, we compare the performance of different versions of our SMASH algorithm, thus providing insights on the improvements offered by, and the challenges remaining in, each implementation.

6.1 Experimental Methodology

For our experiments, we chose the R-MAT synthetic sparse matrix dataset [18, 45]. In prior work [18], Chakrabarti et al. proposes a synthetic graph generator that closely represents real-world graphs in multiple disciplines. In addition to their graph generator being fast, multithreaded, and robust, they successfully simulate the famous Erdos-Renyi model [26], providing a measure of the uniform probability distribution of each possible independent edge of a graph. The RMAT sparse matrices exhibit irregular sparsity patterns with a power-law distribution of non-zeros, making them notoriously difficult to balance between threads. A synthetic data generator also allows us to create graphs of specific required dimensions and varying sparsity patterns for analysis.

We generate two $16K \times 16K$ matrices using the R-MAT generator and multiply them with each other using the row-wise multiplication method. We evaluate the performance of all 3 of our kernel implementations on the same input matrices using our simulator and report various performance metrics in the next section.

6.2 Dataset Arithmetic Intensity

Before we dive deeper into the analytics of our SpGEMM implementation, it is worthwhile to explore the arithmetic intensity of multiplying two sparse matrices. The characteristics of the matrices used in this thesis are shown in Table 6.1.

Matrix	Dimensions	Total Non-zeros	Sparsity
Input Matrix A	16,384 * 16,384	254,211	99.9%
Input Matrix B	16,384 * 16,384	254,211	99.9%
Output Matrix C	16,384 * 16,384	5,174,841	98.1%

Table 6.1: Input and output data characteristics used in this thesis.

The arithmetic Intensity of SpGEMM is computed as the ratio of the total number of floating-point operations to the number of total data movement operations (reported in bytes) [37]. An AI value of 0.09 or $\frac{9}{100}$ means for 9 floating-point operations, at least 100 bytes of data need to be moved.The arithmetic intensity (AI) for multiplying sparse matrix A with sparse matrix B to produce output matrix C is given by equation 6.1:

$$AI \leq \frac{nnz(C) * cf}{[nnz(A) + nnz(B) + nnz(C)] * b} \leq \frac{cf}{b} \tag{6.1}$$

where nnz is the total number of non-zeros in the matrix, b is the total number of bytes required to store one element of the input matrix, cf is the compression factor computed as a ratio of FLOPs to nonzeros in the output matrix as seen in Equation 6.2.

$$cf = \frac{flop}{nnz(C)} \tag{6.2}$$

Matrix Parameters	Data Type	Elements	Size (Bytes)	Size (in KB)
Row Pointer	INT 4 Bytes	16,385	65,540	64 KB
Column Index	INT 4 Bytes	2,54,211	1,016,844	993 KB
Data Array	Double 8 Bytes	2,54,211	2,033,688	1,986 KB
Total	-	5,683,263	3,116,072	3,043 KB

Table 6.2: CSR matrix arrays for input matrices A and B.

Matrix Parameters	Data Type	Elements	Size (Bytes)	Size (in KB)
Row Pointer	INT 4 Bytes	16,385	65,540 B	64 KB
Column Index	INT 4 Bytes	5,174,841	20,699,364	20,214 KB
Data Array	Double 8 Bytes	5,174,841	41,398,728	40,428 KB
Total	-	10,366,067	62,163,632	60,706 KB

Table 6.3: CSR matrix arrays for the output matrix C.

In our case, we consider one particular example to compute cf and AI using data metrics, as shown in Table 6.1, Table,6.2 and Table 6.3.

For our implementation, we compute the compression factor, $cf = 1.23$. We further compute our arithmetic intensity AI using this cf. The arithmetic intensity of our SMASH Version 3 implementation is $AI = 0.09$.

6.3 DRAM Performance

Both the input matrices and the output matrix are stored in DRAM. The DRAM bandwidth is a measure of the rate at which the input matrices are read, and the output matrices are written [84]. The DRAM bandwidth is considered a bottleneck for SpGEMM implementations [100]. We compare the DRAM bandwidth utilization for all our SMASH implementations. DRAM bandwidth helps us decide if our SpGEMM kernel is memory-bound or compute-bound. Changes in bandwidth utilization over time also help us narrow down algorithm phases that produce a bottleneck. Considering we are using a row-wise product approach, the DRAM is utilized to read the row pointers, column indices, and data values for input matrices A and B. As version 3 of our SpGEMM implementation stores the hashtable in DRAM, this hashtable contributes to the DRAM bandwidth demands as well. The DRAM bandwidth demands are compared in Table 6.4.

SMASH Versions	DRAM Bandwidth
Version 1	55.2% (3.03 GB/s)
Version 2	73.9 % (4.06 GB/s)
Version 3	95.9% (5.26 GB/s)

Table 6.4: Aggregated DRAM bandwidth demands.

6.4 Cache Performance

Cache performance and utilization play a crucial role in the speed achieved by our SpGEMM kernels. We maintain temporal locality across the first input matrix elements and spatial locality across the second input matrix. This reuse of elements from the first matrix, combined with the access to neighboring elements from the second matrix, allows us to achieve high data-cache hit rates. The L1 data-cache hit rates for all 3 versions of our SMASH algorithm are presented in Table 6.5.

SMASH Versions	L1 Data Cache Hit Rate
Version 1	88.7%
Version 2	92.2%
Version 3	94.1%

Table 6.5: Cache performance of our 3 SMASH implementatinos.

6.5 Workload Distribution

We achieved a significant performance improvement when leveraging *tokenization* in SMASH Version 2. This leads to a near-perfect balance of workload across PIUMA threads. This avoided idle PIUMA cores, with individual cores waiting for other cores to finish. This, in turn, significantly boosted the *Instructions per Cycle* (IPC), as shown in the following section.

We analyze the performance of SMASH Version 1 and SMASH Version 2 on a single window, using a single PIUMA block. Results are shown in Figures 6.1, 6.2, 6.3 and 6.4.

Figures 6.1 and 6.2 provide information on the utilization of each thread in a block, measured over time. These figures represent the thread utilization, with the x-axis plotting the time in milliseconds and the y-axis reporting the associated thread utilization.

Figure 6.1 shows thread utilization of SMASH V1 kernel. As observed in this figure, some of the threads do not achieve high thread utilization, indicating that the multi-threaded core is underutilized as some of the threads stall during execution (they stall on barriers, waiting for other threads to complete).

Figure 6.2 shows the same workload on the SMASH V2 kernel. All threads in this figure achieve close to 100% thread utilization. This shows that our later implementation has mitigated the cause of under-utilization of the multi-threaded cores.

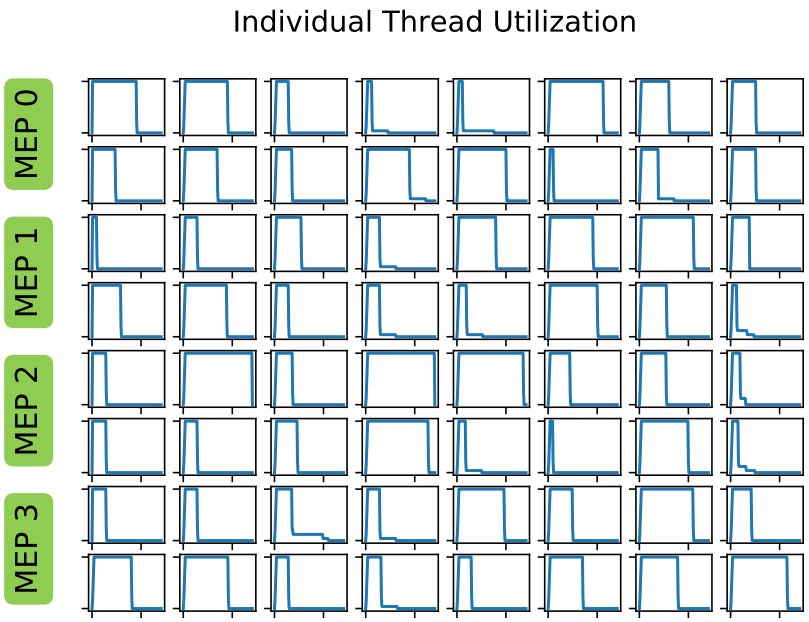

Figure 6.1: SMASH V1: Thread utilization plots for unbalanced workload.

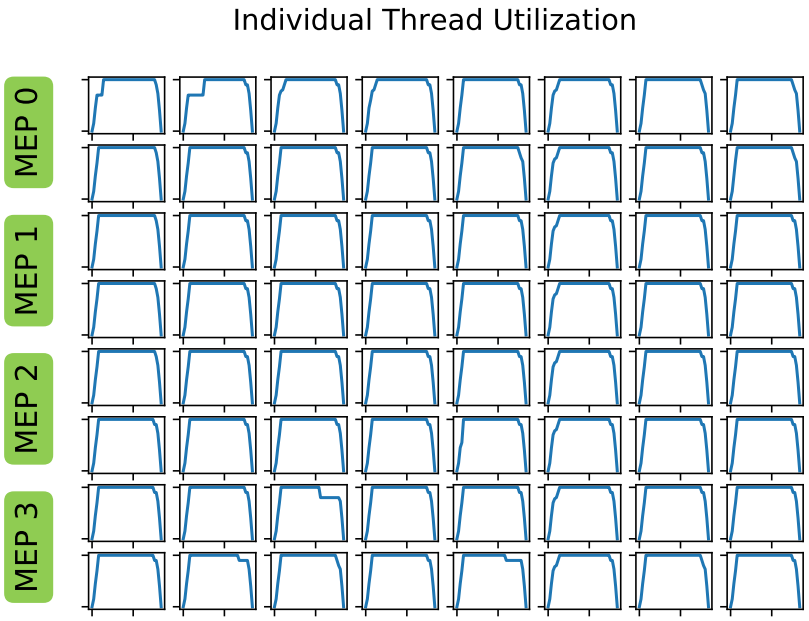

Figure 6.2: SMASH V2: Thread utilization plots for balanced workload.

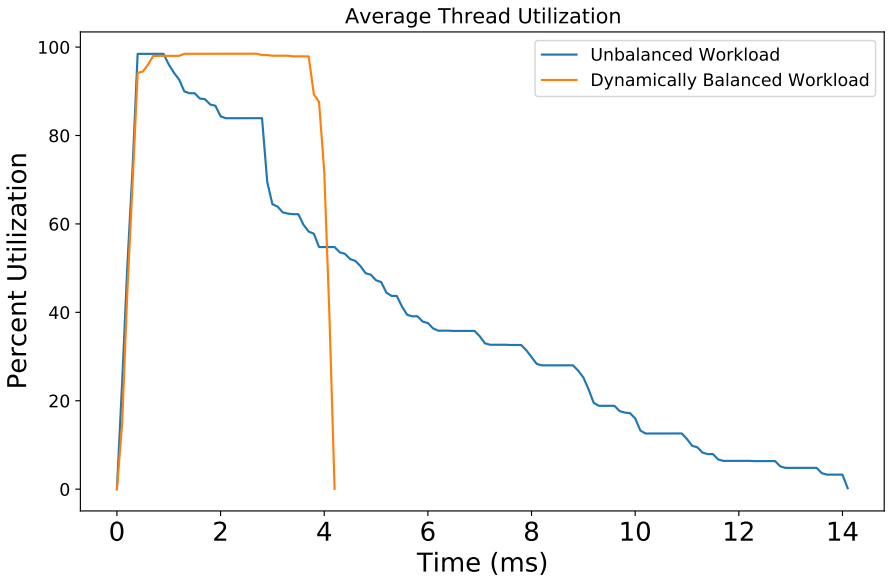

Figure 6.3: Average thread utilization.

Figure 6.3 reports the average thread utilization for each workload. Using dynamic allocation prevents threads from stalling while waiting for other threads to complete, thus maintaining a higher IPC value.

In Figure 6.4 we provide a normalized histogram to report on thread utilization, showing the performance improvement in SMASH V2 as compared to SMASH V1. The balanced workload shown on the right exhibits more threads achieve nearly 100% utilization, as opposed to the unbalanced workload, where multiple threads idling.

With the use of $tokenization$, we managed to not only distribute workload evenly across all threads, but we also end up reducing the overall execution time required for this window from 14.15 ms to 4.09 ms, despite the presence of overhead introduced when creating and polling tokens.

6.6 Instruction Throughput

IPC describes the total number of instructions being executed in the system for every cycle. We compare our SMASH implementations by comparing aggregate IPC over the entire execution of the workload while considering all PIUMA threads. Ideally, the max value of the IPC that can be achieved is equal to the number of MTCs present in the block.

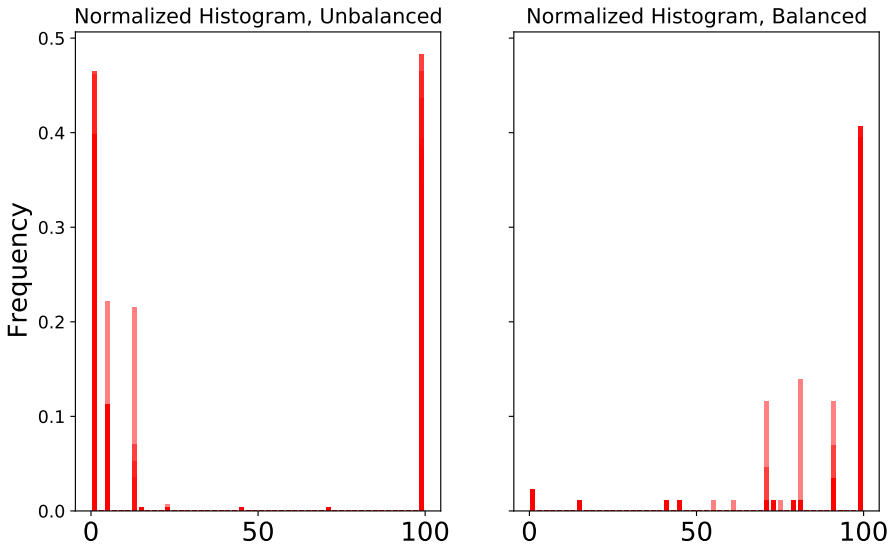

Figure 6.4: Thread utilization histogram comparison between balanced and unbalanced workloads.

$$Aggregate\ IPC = \frac{Total\ Instructions\ Executed}{Total\ Cycles} \tag{6.3}$$

Aggregate IPC can be computed using the equation 6.3.

SMASH Versions	Aggregate IPC
Version 1	0.9 IPC
Version 2	1.7 IPC
Version 3	2.3 IPC

Table 6.6: Aggregate IPC Comparisons

We provide the aggregate IPC values for each of our 3 SMASH implementations in Table 6.6.

6.7 Application Speedup

We simulate the total time required by each of our SMASH versions on our interval simulator, simulating the PIUMA hardware. We consider the time required to run the entire SpGEMM

workload on a single PIUMA block. The runtime comparison for all 3 versions of SMASH are presented in Table 6.7.

SMASH Versions	Runtimes	Speedup over Version 1
Version 1	986.7 ms	1.0×
Version 2	432.5 ms	2.3×
Version 3	105.4 ms	9.4×

Table 6.7: Runtime for an entire SpGEMM workload on 64 PIUMA threads.

6.8 Summarry of Results

The SMASH V3 kernel is a state-of-the-art SpGEMM implementation built on the PIUMA architecture. It employs various optimizations over previous SpGEMM kernel implementations, as well as previous SMASH versions. The new kernel is capable of utilizing 95.9% of DRAM bandwidth, nearly saturating the available bandwidth. With the deployment of the producer-consumer model, the SMASH V3 kernel is able to deliver almost 100% multi-threaded core utilization. This optimization translates to a $9.4\times$ speedup, as well as a 155% increase in instruction throughput, as seen in Tables 6.6 and 6.7.

Chapter 7

Conclusions and Future Work

SpGEMM workloads are well known to test the limits of both hardware and software. The software kernel implementation can play a key role in the resulting irregular memory access pattern. For this reason, most general-purpose architectures, including CPUs and GPUs, typically fail to achieve high speedups when executing SpGEMM-based applications.

In this work, we described the many uses of SpGEMM kernels, helping to motivate the need for domain-specific architectures for such workloads. We identified key issues that need to be addressed when designing SpGEMM kernels. We further investigated prior research that has pursued these same issues. This helped shape the design of our approach pursued in this thesis while optimizing the mapping of the SpGEMM kernel to the underlying PIUMA architecture. We utilized a row-wise product approach in each of our four implementations.

We explored some of the novel features present in the PIUMA architecture, designed to tackle sparse graph and sparse matrix applications. We designed 3 different SpGEMM kernel implementations called SMASH for the PIUMA system, focusing on the key features available on this accelerator, including DGAS, networked instructions, DMA Engines, and multi-threaded cores.

Our set of optimizations focused primarily on improving DRAM bandwidth utilization of the SpGEMM kernel. But an increase in DRAM bandwidth utilization by itself is not an indication of improved performance, as multiple factors can impact the resulting performance. Avoiding redundant reads to memory, poor reuse of input matrices, and increases in metadata size can all offset bandwidth utilization improvements. Our SMASH V3 kernel implementation stores the hashtable in memory instead of using an on-chip Scratchpad. Thus, in addition to reading input matrices, the kernel also has to read intermediate partial-products from the hashtable stored on DRAM. Thus the DRAM bandwidth is shared between the input data reads and the partial-product reads. But in

addition to an increase in DRAM bandwidth utilization, we also observed a significant speedup of Version 3 over previous versions. We were able to achieve a speedup of $9.4\times$ over Version 1 by iterative improvements, performing tokenization, and memory defragmentation.

To summarize, the SMASH kernel improvements are as follows.

- We successfully built an implementation using atomic hashing that eliminated the need for partial product matrices in row-wise product methods, thus preventing redundant accesses to DRAM memory.
- We improved workload balance by adding a layer of dynamic work allocation, leveraging a producer-consumer model.
- Finally, we leveraged PIUMA's DMA engine, which enabled us to move data from the SPAD to DRAM without wasting precious cycles of the MTCs.

7.1 Contributions of this Thesis

The main contributions of this thesis include:

- An in-depth analysis of the inherent problems exhibited by sparse matrix multiplication kernels.
- A comparison study of prior architectures that support SpGEMM workloads.
- A comparative study on previous implementations of SpGEMM kernels.
- An architectural overview of Intel's novel PIUMA graph accelerator.
- A state-of-the-art SpGEMM kernel implementation that uses the features present in the PIUMA accelerator architecture to speedup sparse matrix operations.

7.2 Future Work

Sparse matrix-matrix multiplication kernel optimizations will continue to be an active research area. A key problem to deal with in any SpGEMM kernel implementation is the resulting workload imbalance. In our implementation, we explored applying a uniform work distribution by estimating floating-point operations based on the number of non-zeros in each row. Although this

method improves the algorithm's performance for our dataset, it leaves room for optimization for other sparsity patterns.

In our work, to store and merge partial products, we employed an in-memory hashtable. Such a hashtable allows us to ensure that the partial products are merged immediately as they are produced. One of the drawbacks of using hashtables is that they can cause memory hotspots. Based on the hashing mechanism in our implementation, we used either the high-order bits or low-bits bits for hashing. This resulted in some sparsity patterns to generate hotspots (multiple elements getting hashed to the same hash class) in our hashtable. Such patterns will cause our algorithm to run the collision resolution subroutine, leading to degraded performance. In our next iteration, we plan to avoid collisions by incorporating a better hashing algorithm, one that is not solely based on restricting the bits selected. We will consider a dynamic hashing algorithm, developing one that can adapt to different sparsity patterns.

The PIUMA architecture provides a rich platform where we can further explore the acceleration of linear algebra operations. We want to extend this work well beyond the present focus on the SpGEMM kernel. We intend to explore other linear algebra subroutines (GraphBLAS) and consider how to optimize performance given the unique features of this architecture.

Bibliography

[1] Tor M Aamodt, Wilson Wai Lun Fung, and Timothy G Rogers. General-purpose graphics processor architectures. *Synthesis Lectures on Computer Architecture*, 13(2):1–140, 2018.

[2] S. Aananthakrishnan, R. Pawlowski, J. Fryman, and I. Hur. Efficient sparse matrix-vector multiplication on intel piuma architecture. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–2, 2020.

[3] Sriram Aananthakrishnan, Nesreen K Ahmed, Vincent Cave, Marcelo Cintra, Yigit Demir, Kristof Du Bois, Stijn Eyerman, Joshua B Fryman, Ivan Ganev, Wim Heirman, et al. Piuma: Programmable integrated unified memory architecture. *arXiv preprint arXiv:2010.06277*, 2020.

[4] Sverre J Aarseth. *Gravitational N-body simulations: tools and algorithms*. Cambridge University Press, 2003.

[5] Sami Abu-El-Haija, Amol Kapoor, Bryan Perozzi, and Joonseok Lee. N-gcn: Multi-scale graph convolution for semi-supervised node classification. In *Uncertainty in Artificial Intelligence*, pages 841–851. PMLR, 2020.

[6] Jung Ho Ahn, Nathan Binkert, Al Davis, Moray McLaren, and Robert S Schreiber. Hyperx: topology, routing, and packaging of efficient large-scale networks. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, 2009.

[7] Ayaz Akram and Lina Sawalha. A survey of computer architecture simulation techniques and tools. *Ieee Access*, 7:78120–78145, 2019.

[8] Moustafa Alzantot, Yingnan Wang, Zhengshuang Ren, and Mani B Srivastava. Rstensorflow: Gpu enabled tensorflow for deep learning on commodity android devices. In *Proceedings*

of the 1st International Workshop on Deep Learning for Mobile Systems and Applications, pages 7–12, 2017.

[9] 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)*, pages 249–260. IEEE, 2020.

[10] William Aspray. The intel 4004 microprocessor: What constituted invention? *IEEE Annals of the History of Computing*, 19(3):4–15, 1997.

[11] Michael Bailey, Rachel Cao, Theresa Kuchler, Johannes Stroebel, and Arlene Wong. Social connectedness: Measurement, determinants, and effects. *Journal of Economic Perspectives*, 32(3):259–80, 2018.

[12] Trinayan Baruah, Kaustubh Shivdikar, Shi Dong, Yifan Sun, Saiful Mojumder, Kihoon Jung, José L. Abellán, Yash Ukidave, Ajay Joshi, John Kim, and David Kaeli. Gnnmark: A benchmark suite to characterize graph neural network training on gpus. 2021.

[13] James Bennett, Stan Lanning, et al. The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35. New York, 2007.

[14] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

[15] Encyclopaedia Britannica. Matrix.

[16] David Canright and Lejla Batina. A very compact "perfectly masked" s-box for aes. In *International Conference on Applied Cryptography and Network Security*, pages 446–459. Springer, 2008.

[17] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014.

[18] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446. SIAM, 2004.

[19] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):292–308, 2019.

[20] Yuedan Chen, Guoqing Xiao, and Wangdong Yang. Optimizing partitioned csr-based spgemm on the sunway taihulight. *Neural Computing and Applications*, 32(10):5571–5582, 2020.

[21] Zhaodong Chen, Mingyu Yan, Maohua Zhu, Lei Deng, Guoqi Li, Shuangchen Li, and Yuan Xie. fusegnn: Accelerating graph convolutional neural network training on gpgpu. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2020.

[22] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[23] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 257–266, 2019.

[24] Joan Daemen and Vincent Rijmen. The rijndael block cipher: Aes proposal. In *First candidate conference (AeS1)*, pages 343–348, 1999.

[25] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002.

[26] J-J Daudin, Franck Picard, and Stéphane Robin. A mixture model for random graphs. *Statistics and computing*, 18(2):173–183, 2008.

[27] Michael DeLorimier, Nachiket Kapre, Nikil Mehta, Dominic Rizzo, Ian Eslick, Raphael Rubin, Tomas E Uribe, F Thomas Jr, Andre DeHon, et al. Graphstep: A system architecture for sparse-graph algorithms. In *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 143–151. IEEE, 2006.

[28] Jack Dongarra. Sparse matrix storage formats. *Templates for the Solution of Algebraic Eigenvalue Problems: a Practical Guide. SIAM*, 11:445–448, 2000.

[29] Iain S Duff, Michael A Heroux, and Roldan Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum. *ACM Transactions on Mathematical Software (TOMS)*, 28(2):239–267, 2002.

[30] Lieven Eeckhout. Computer architecture performance evaluation methods. *Synthesis Lectures on Computer Architecture*, 5(1):1–145, 2010.

[31] Stijn Eyerman, Wim Heirman, Kristof Du Bois, Joshua B Fryman, and Ibrahim Hur. Many-core graph workload analysis. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 282–292. IEEE, 2018.

[32] Dimitar Filev, Olga Georgieva, P Angelov, and A Kasabov. An extended version of the gustafson-kessel algorithm for evolving data stream clustering. *Evolving intelligent systems: Methodology and applications*, pages 273–300, 2010.

[33] Davy Genbrugge, Stijn Eyerman, and Lieven Eeckhout. Interval simulation: Raising the level of abstraction in architectural simulation. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12. IEEE, 2010.

[34] Olga Georgieva and Dimitar Filev. Gustafson-kessel algorithm for evolving data stream clustering. In *Proceedings of the International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing*, pages 1–6, 2009.

[35] C Lee Giles, Kurt D Bollacker, and Steve Lawrence. Citeseer: An automatic citation indexing system. In *Proceedings of the third ACM conference on Digital libraries*, pages 89–98, 1998.

[36] Google. Cloud tpu. *https://cloud.google.com/tpu*, 2019.

[37] Zhixiang Gu, Jose Moreira, David Edelsohn, and Ariful Azad. Bandwidth-optimized parallel algorithms for sparse matrix-matrix multiplication using propagation blocking. *arXiv preprint arXiv:2002.11302*, 2020.

[38] Fred G Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS)*, 4(3):250–269, 1978.

[39] Daniel Hackenberg, Daniel Molka, and Wolfgang E Nagel. Comparing cache architectures and coherency protocols on x86-64 multicore smp systems. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on microarchitecture*, pages 413–422, 2009.

[40] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News*, 44(3):243–254, 2016.

[41] Carl Lee Hanson, Ben Cannon, Scott Burton, and Christophe Giraud-Carrier. An exploration of social circles and prescription drug abuse through twitter. *J Med Internet Res*, 15(9):e189, Sep 2013.

[42] 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.

[43] Wim Heirman, Trevor Carlson, and Lieven Eeckhout. Sniper: Scalable and accurate parallel multi-core simulation. In *8th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES-2012)*, pages 91–94. High-Performance and Embedded Architecture and Compilation Network of ..., 2012.

[44] Sung-Hsien Hsieh, Chun-Shien Lu, and Soo-Chang Pei. Sparse fast fourier transform by downsampling. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 5637–5641. IEEE, 2013.

[45] Lorenz Hübschle-Schneider and Peter Sanders. Linear work generation of r-mat graphs. *Network Science*, 8(4):543–550, 2020.

[46] Ken Ivanov. Autonomous collision attack on ocsp services, 2016.

[47] Bryan Jacobs. Hierarchical identify verify exploit (hive).

[48] Lizy Kurian John. 8.2 performance evaluation: Techniques, tools, and benchmarks. *The Computer Engineering Handbook*, 8:21, 2002.

[49] Bo Kågström, Per Ling, and Charles Van Loan. Gemm-based level 3 blas: high-performance model implementations and performance evaluation benchmark. *ACM Transactions on Mathematical Software (TOMS)*, 24(3):268–302, 1998.

[50] Mike Kelly. Gyrfalcon starts shipping ai chip. *electronics-labs.com*, Oct 2018.

[51] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. Modeling gpu-cpu workloads and systems. In *Proceedings of the 3rd workshop on general-purpose computation on graphics processing units*, pages 31–42, 2010.

[52] Thomas N Kipf and Max Welling. Variational graph auto-encoders. *arXiv preprint arXiv:1611.07308*, 2016.

[53] Oliver Knill. When was matrix multiplication invented? *Harvard Mathematics Department*, Jun 2009. Available at `http://people.math.harvard.edu/~knill/history/matrix/`.

[54] Oliver Knill. Cauchy–binet for pseudo-determinants. *Linear Algebra and its Applications*, 459:522–547, 2014.

[55] Raghu Krishnapuram and Jongwoo Kim. A note on the gustafson-kessel and adaptive fuzzy clustering algorithms. *IEEE Transactions on Fuzzy systems*, 7(4):453–461, 1999.

[56] HT Kung, Bradley McDanel, and Sai Qian Zhang. Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 821–834, 2019.

[57] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[58] Hua Li and Zachary Friggstad. An efficient architecture for the aes mix columns operation. In *2005 IEEE International Symposium on Circuits and Systems*, pages 4637–4640. IEEE, 2005.

[59] Chun-Yuan Lin, Yeh-Ching Chung, and Jen-Shiuh Liu. Efficient data compression methods for multidimensional sparse array operations based on the ekmr scheme. *IEEE Transactions on Computers*, 52(12):1640–1646, 2003.

[60] Rong Lin and Martin Margala. Multiplier-based processor-in-memory architectures for image and graphics processing, January 23 2007. US Patent 7,167,890.

[61] Zhi-Gang Liu, Paul N Whatmough, and Matthew Mattina. Systolic tensor array: An efficient structured-sparse gemm accelerator for mobile cnn inference. *IEEE Computer Architecture Letters*, 19(1):34–37, 2020.

[62] Adam Lugowski, David Alber, Aydm Buluç, John R Gilbert, Steve Reinhardt, Yun Teng, and Andrew Waranis. A flexible open-source toolbox for scalable complex graph analysis. In *Proceedings of the 2012 SIAM International Conference on Data Mining*, pages 930–941. SIAM, 2012.

[63] Milo MK Martin, Daniel J Sorin, Bradford M Beckmann, Michael R Marty, Min Xu, Alaa R Alameldeen, Kevin E Moore, Mark D Hill, and David A Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *ACM SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[64] Dan McCreary. Intel's incredible piuma graph analytics hardware, Nov 2020.

[65] Jason E Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12. IEEE, 2010.

[66] Duncan JM Moss, Srivatsan Krishnan, Eriko Nurvitadhi, Piotr Ratuszniak, Chris Johnson, Jaewoong Sim, Asit Mishra, Debbie Marr, Suchit Subhaschandra, and Philip HW Leong. A customizable matrix multiplication framework for the intel harpv2 xeon+ fpga platform: A deep learning case study. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 107–116, 2018.

[67] Sean Murphy and Matthew JB Robshaw. Essential algebraic structure within the aes. In *Annual International Cryptology Conference*, pages 1–16. Springer, 2002.

[68] Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. High-performance and memory-saving sparse general matrix-matrix multiplication for nvidia pascal gpu. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 101–110. IEEE, 2017.

[69] Sarath Mohanachandran Nair, Christopher Münch, and Mehdi B Tahoori. Defect characterization and test generation for spintronic-based compute-in-memory. In *2020 IEEE European Test Symposium (ETS)*, pages 1–6. IEEE, 2020.

[70] Matthew Naylor and Colin Runciman. The reduceron: Widening the von neumann bottleneck for graph reduction using an fpga. In Olaf Chitil, Zoltán Horváth, and Viktória Zsók, editors, *Implementation and Application of Functional Languages*, pages 129–146, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[71] Geneviève Ndour, Tiago Trevisan Jost, Anca Molnos, Yves Durand, and Arnaud Tisserand. Evaluation of approximate operators case study: sobel filter application executed on an approximate risc-v platform. In *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 146–149, 2018.

[72] NetLib. Blas, basic linear algebra subprograms. *NetLib publication*, 2004.

[73] Natasha Noy, Yuqing Gao, Anshu Jain, Anant Narayanan, Alan Patterson, and Jamie Taylor. Industry-scale knowledge graphs: Lessons and challenges. *Communications of the ACM*, 62 (8):36–43, 2019.

[74] A100 NVIDIA. Nvidia a100 gpus power the modern data center, May 2020.

[75] Kazuki Osawa, Akira Sekiya, Hiroki Naganuma, and Rio Yokota. Accelerating matrix multiplication in deep learning by using low-rank approximation. In *2017 International Conference on High Performance Computing & Simulation (HPCS)*, pages 186–192. IEEE, 2017.

[76] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. Outerspace: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 724–736. IEEE, 2018.

[77] 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. *ACM SIGARCH Computer Architecture News*, 45(2):27–40, 2017.

[78] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, et al. Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications. *arXiv preprint arXiv:1811.09886*, 2018.

[79] Xiaochen Peng, Shanshi Huang, Yandong Luo, Xiaoyu Sun, and Shimeng Yu. Dnn+ neurosim: An end-to-end benchmarking framework for compute-in-memory accelerators with versatile device technologies. In *2019 IEEE International Electron Devices Meeting (IEDM)*, pages 32–5. IEEE, 2019.

[80] William B Pennebaker and Joan L Mitchell. *JPEG: Still image data compression standard*. Springer Science & Business Media, 1992.

[81] Nikos P Pitsianis, Gerald G Pechanek, and Ricardo E Rodriguez. Efficient complex multiplication and fast fourier transform (fft) implementation on the manarray architecture, January 4 2005. US Patent 6,839,728.

[82] C Pradabpet, N Ravinu, S Chivapreecha, B Knobnob, and K Dejhan. An efficient filter structure for multiplierless sobel edge detection. In *2009 Innovative Technologies in Intelligent Systems and Industrial Applications*, pages 40–44. IEEE, 2009.

[83] 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. IEEE, 2020.

[84] N. Rafique, W. Lim, and M. Thottethodi. Effective management of dram bandwidth in multicore processors. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 245–258, 2007.

[85] Marcel Richter and Gudula Runger. Symbolic matrix multiplication for multithreaded sparse gemm utilizing sparse matrix formats. In *2018 International Conference on High Performance Computing & Simulation (HPCS)*, pages 523–530. IEEE, 2018.

[86] Jörn Schumacher. High performance sparse fast fourier transform. Master's thesis, ETH Zurich, Department of Computer Science, 2013.

[87] Jeong Bong Seo and Dae-Won Kim. A parallel implementation of the gustafson-kessel clustering algorithm with cuda. *IEICE TRANSACTIONS on Information and Systems*, 95(4):1162–1165, 2012.

[88] Khyati Shah. Performance analysis of sobel edge detection filter on gpu using cuda & opengl. *HGPU ORG*, 2013.

[89] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. In *Advances in Neural Information Processing Systems*, pages 10414–10423, 2018.

[90] Charles Shipley and Stephen Jodis. Programming languages classification. In Hossein Bidgoli, editor, *Encyclopedia of Information Systems*, pages 545–552. Elsevier, New York, 2003.

[91] Kaustubh Shivdikar, Ahan Kak, and Kshitij Marwah. Automatic image annotation using a hybrid engine. In *2015 Annual IEEE India Conference (INDICON)*, pages 1–6. IEEE, 2015.

[92] FS Smailbegovic, Georgi N Gaydadjiev, and Stamatis Vassiliadis. Sparse matrix storage format. In *Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing*, pages 445–448, 2005.

[93] Brent Smith and Greg Linden. Two decades of recommender systems at amazon. com. *Ieee internet computing*, 21(3):12–18, 2017.

[94] Mohammadreza Soltaniyeh, Richard P Martin, and Santosh Nagarakatte. Synergistic cpu-fpga acceleration of sparse linear algebra. *arXiv preprint arXiv:2004.13907*, 2020.

[95] Mohit Srinivasan, Ahan Kak, Kaustubh Shivdikar, and Chirag Warty. Dynamic power allocation using stackelberg game in a wireless sensor network. In *2016 IEEE Aerospace Conference*, pages 1–10. IEEE, 2016.

[96] 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)*, pages 766–780. IEEE, 2020.

[97] Mark Stock and Adrin Gharakhani. Toward efficient gpu-accelerated n-body simulations. In *46th AIAA Aerospace Sciences Meeting and Exhibit*, page 608, 2008.

[98] Swadhin Thakkar, Kaustubh Shivdikar, and Chirag Warty. Video steganography using encrypted payload for satellite communication. In *2017 IEEE Aerospace Conference*, pages 1–11. IEEE, 2017.

[99] Jeyarajan Thiyagalingam, Olav Beckmann, and Paul HJ Kelly. An exhaustive evaluation of row-major, column-major and morton layouts for large two-dimensional arrays. In *Performance Engineering: 19th Annual UK Performance Engineering Workshop*, pages 340–351, 2003.

[100] Jesmin Jahan Tithi, Fabio Checconi, Douglas Doerfler, and Fabrizio Petrini. Performance optimization of su3_bench on xeon and programmable integrated unified memory architecture. *arXiv preprint arXiv:2103.00571*, 2021.

[101] Elena Trichina, Domenico De Seta, and Lucia Germani. Simplified adaptive multiplicative masking for aes. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 187–197. Springer, 2002.

[102] Michael Tschannen, Aran Khanna, and Animashree Anandkumar. Strassennets: Deep learning with a multiplication budget. In *International Conference on Machine Learning*, pages 4985–4994, 2018.

[103] Aravind Vasudevan, Andrew Anderson, and David Gregg. Parallel multi channel convolution using general matrix multiplication. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 19–24. IEEE, 2017.

[104] Andrey Vladimirov. A survey and benchmarks of intel xeon gold and platinum processors, Aug 2020.

[105] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*, pages 167–188. Springer, 2014.

[106] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. Gnnadvisor: An efficient runtime system for gnn acceleration on gpus. *arXiv preprint arXiv:2006.06608*, 2020.

[107] Darren J Wilkinson and Stephen KH Yeung. A sparse matrix approach to bayesian computation in large linear models. *Computational statistics & data analysis*, 44(3):493–516, 2004.

[108] Tobias Würfl, Florin C Ghesu, Vincent Christlein, and Andreas Maier. Deep learning computed tomography. In *International conference on medical image computing and computer-assisted intervention*, pages 432–440. Springer, 2016.

[109] Zhen Xie, Guangming Tan, Weifeng Liu, and Ninghui Sun. Ia-spgemm: An input-aware auto-tuning framework for parallel sparse matrix-matrix multiplication. In *Proceedings of the ACM International Conference on Supercomputing*, pages 94–105, 2019.

[110] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[111] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, et al. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 24–32, 2007.

[112] Rio Yokota and Lorena Barba. Hierarchical n-body simulations with autotuning for heterogeneous systems. *Computing in Science & Engineering*, 14(3):30–39, 2012.

[113] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-x: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[114] 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. IEEE, 2020.

[115] Zhihui Zhang, Jingwen Leng, Lingxiao Ma, Youshan Miao, Chao Li, and Minyi Guo. Architectural implications of graph neural networks. *IEEE Computer Architecture Letters*, 19(1):59–62, 2020.

[116] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.

[117] Kim Ann Zimmermann. History of computers: A brief timeline, Sep 2017.

[118] Uri Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM (JACM)*, 49(3):289–317, 2002.