Reconfigurable High-Performance Computing of Sparse Linear Algebra

by

Erfan Bank Tavakoli

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved April 2024 by the
Graduate Supervisory Committee:

Fengbo Ren, Chair
Aviral Shrivastava
Deliang Fan
Hassan Ghasemzadeh

ARIZONA STATE UNIVERSITY

May 2024

ABSTRACT

This thesis presents novel software/hardware co-design methodologies aimed at accelerating sparse linear algebra applications within the realm of High-Performance Computing (HPC). The motivation stems from the limitations of conventional CPU- and GPU-based solutions for sparse linear algebra, which are hindered by fixed hardware architecture and memory hierarchy, frequent off-chip memory access, and high energy consumption. In response, this work explores the deployment of Field-Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs) to overcome these challenges through their customized nature, offering performance and energy efficiency gains.

The scope of the thesis is divided into three main parts: firstly, it introduces a framework that combines an FPGA computational kernel with a novel scheduling algorithm running on a host processor for accelerating the supernodal multifrontal algorithm for sparse Cholesky factorization. This approach minimizes off-chip memory access and on-chip memory requirements by efficiently managing data dependencies and enhancing data locality. Secondly, it presents FSpGEMM, an OpenCL-based framework for accelerating general sparse matrix-matrix multiplication on FPGAs. FSpGEMM exploits a new compressed sparse vector format (CSV) and a custom buffering scheme tailored to Gustavson's algorithm, significantly improving computational performance by optimizing memory access patterns. Additionally, a row reordering technique is utilized to increase the data reuse enabled by the CSV format. Lastly, the thesis proposes an ASIC design for Sparse Tensor Core, which utilizes a Hardware Merge Sorter to increase parallelism in processing units without compromising operating frequency, offering a high-speed solution for sparse linear algebra operations.

In summary, the thesis addresses the challenges of implementing sparse linear al-

gebra algorithms on FPGAs and ASICs, such as the complexity of data dependencies and the need for efficient memory management. By proposing solutions that enhance computational performance, reduce energy consumption, and improve the usability of FPGAs and ASICs in HPC infrastructures, this work contributes to computational science, offering a pathway toward more efficient and sustainable computing for complex, data-intensive applications.

DEDICATION

To my mother, whose love and strength are the foundation of all my achievements.

ACKNOWLEDGMENTS

I want to thank my supervisor, Dr. Fengbo Ren, for all his guidance and help throughout my PhD and beyond. I'm also grateful for the unwavering support from my family, without whom this wouldn't have been possible.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 Motivation

High-Performance Computing, or HPC, is the use of powerful computer systems and parallel processing techniques to solve complex computational problems that require vast amounts of data and processing power. HPC is used in a variety of fields, including scientific research, engineering, and finance, to simulate and model complex systems, analyze large datasets, and solve optimization problems. HPC is a rapidly evolving field, with continued advances in hardware and software enabling increasingly complex and data-intensive applications [32]. Sparse linear algebra algorithms are computational techniques for linear algebra problems that are optimized to handle sparse data, where most of the values in a dataset are zero or empty. These algorithms are designed to perform computations only on the non-zero values, thereby reducing the computational load and memory requirements, which makes them faster and more efficient than traditional algorithms [24, 46].

Nonetheless, the existing HPC solutions to sparse linear algebra applications based on CPUs and GPUs suffer from very limited performance due to two primary reasons. First, sparse algorithms (*e.g.*, the multifrontal algorithms [55]) are recursive and have complex data dependencies for sequentially updating the intermediate results from previous iterations. For the sake of data locality and computational performance, an algorithm-tailored buffering scheme for efficiently storing the intermediate results must be employed for computing a sparse algorithm. Unfortunately, the deep memory hierarchy and fixed hardware architecture of CPUs and GPUs can hardly be adapted

to efficiently implement such an algorithm-tailored buffering scheme. Consequently, CPU- and GPU-based solutions suffer from poor cache locality and often require frequent off-chip memory access for computing sparse algorithms, greatly limiting their performance. Second, sparse algorithms involve complex operations (*e.g.*, inverse square root) that are often computed using approximation algorithms (*e.g.*, the Newton-Raphson method [70]) that are also iterative and have strong loop-carried data dependency. Unfortunately, the legacy hardware architectures of CPUs and GPUs, while being able to exploit massive spatial parallelism, lack the capability to exploit the temporal/pipeline parallelism that is critical to resolving such loop-carried data dependency, which results in long loop initiation intervals causing further reduced performance. In addition to the limited performance issue of CPU- and GPU-based sparse algorithms solutions for HPC applications, these solutions suffer from very high energy consumption due to high runtime (*i.e.*, low performance) and power consumption of CPUs and GPUs (*e.g.*, 135 W thermal design power for Intel Xeon Processor E5-2637 v3 and 250 W for NVIDIA V100 TENSOR CORE GPU). The high energy consumption of CPUs and GPUs in HPC data centers has received significant attention due to its high economic, environmental, and performance costs [92].

As FPGAs are being deployed as an emerging accelerator in data centers [28, 71], FPGA computing offers an alternative solution to accelerating sparse algorithms for HPC applications. An FPGA is a farm of configurable hardware resources whose functionality and interconnection can be redefined at run-time by programming its configuration memory. A state-of-the-art FPGA carries an enormous amount of fine- and coarse-grained logic, computation, memory, and I/O resources. Upon the reconfiguration of these resources, an FPGA can implement any custom hardware architecture to accelerate algorithms with both performance and energy efficiency

gains [72, 29, 15]. Specifically, the fine-grained logic resources and the abundant on-chip memory and register resources on FPGA devices can be used to implement the customized buffering scheme tailored to a given sparse algorithm to allow efficient storage of intermediate results and data movement among and within processing elements (PEs) with no or reduced off-chip memory access [83, 88]. Furthermore, the hardware flexibility of an FPGA allows its reconfigurable resources to compose not only spatial but also temporal/pipeline parallelism both at a fine granularity and on a massive scale to best resolve the complex loop-carried data dependency that exists in sparse algorithms to minimize loop initiation intervals for improved performance [57]. Furthermore, while providing higher performance for executing high-dependency algorithms, FPGAs generally have lower power consumption than CPUs and GPUs, which directly translates into lower energy consumption per task (higher energy efficiency).

1.2 Scope of Work

The goal of this proposal is to present software/hardware co-design methodologies for accelerating sparse linear algebra applications, specifically identifying performance bottlenecks in scientific computing using conventional HPC methods, addressing them using Application-Specific Integrated Circuits (ASIC) and FPGA designs, and enabling easy integration and adoption of these designs into existing HPC infrastructures to improve their usability.

The first part of this proposal discusses a framework consisting of an FPGA kernel implementing a throughput-optimized hardware architecture for accelerating the supernodal multifrontal algorithm for sparse Cholesky factorization and a host program implementing a novel scheduling algorithm for finding the optimal execution order of supernodes computations for an elimination tree on the FPGA to eliminate

the need for off-chip memory access for storing intermediate results. Moreover, the proposed scheduling algorithm minimizes on-chip memory requirements for buffering intermediate results by resolving the dependency of parent nodes in an elimination tree through temporal parallelism. In the second part, a new compressed sparse vector (CSV) format for representing sparse matrices and FSpGEMM, an OpenCL-based HPC framework for accelerating general sparse matrix-matrix multiplication on FPGAs, are presented. The proposed FSpGEMM framework includes an FPGA kernel implementing a throughput-optimized hardware architecture based on Gustavson's algorithm and a host program implementing pre-processing functions for converting input matrices to the CSV format tailored for the proposed architecture. FSpGEMM utilizes a new buffering scheme tailored to Gustavson's algorithm. In the third part, an ASIC design of a co-processor, namely Sparse Tensor Core, for accelerating sparse linear algebra is proposed. Sparse Tensor Core includes throughput-optimized processing units. The processing units adopt highly parallel Hardware Merge Sorters as their main computation logic, enabling high-throughput computations.

1.3 Accelerating Sparse Cholesky Factorization

So far, there has been limited work for accelerating sparse Cholesky factorization on FPGAs [82, 78]. The limitations of the existing work are three-fold. First, the existing work adopts either the left-looking [78] or the multifrontal algorithm [82] in their implementations. These algorithms are less optimized in terms of the memory access and computational complexity than the supernodal multifrontal algorithm for sparse Cholesky factorization [82, 55, 62]. Second, the existing work [78] based on the multifrontal algorithm fails to provide a scheduling algorithm for ordering and assigning the computation of different nodes in an elimination tree. The lack of a scheduling algorithm ignores the dependency among different nodes in an elimination

tree, which inevitably demands frequent off-chip memory access and increases the size of on-chip memory required to load and store intermediate results. Third, the FPGA accelerator architecture proposed in [82] does not allow on-chip communication among different PEs, which enforces a large amount of off-chip memory access to occur for transferring intermediate results among PEs.

This chapter discusses FSCHOL, an OpenCL-based HPC framework for accelerating sparse Cholesky factorization on FPGAs. The proposed FSCHOL framework consists of an FPGA kernel implementing an energy-efficient and throughput-optimized hardware architecture and a host program implementing a novel scheduling algorithm. I adopt the supernodal multifrontal algorithm [82, 55, 62] that requires much less memory access and features lower computational complexity than the left-looking and the multifrontal algorithm used in the existing work, which is critical to more efficient hardware mapping and improved performance.

Moreover, I propose a memory-optimized scheduling algorithm for the host program for provisioning the execution of the supernodal multifrontal Cholesky factorization, and potentially all elimination-tree-based multifrontal methods, on an FPGA device. The scheduling algorithm identifies the dependency among computation nodes in an elimination tree and correspondingly arrange their computation order on the FPGA device to avoid off-chip memory access as well as to minimize the on-chip memory requirements for storing intermediate results. This is the key to enabling data locality, thereby improving both computational performance and energy efficiency. Finally, the proposed OpenCL-based FPGA kernel architecture enables pipelined on-chip transfers of intermediate results among PEs by utilizing FIFO channels and eliminates undesired off-chip memory accesses by working in coordination with the scheduling algorithm running on the host side.

1.4 Accelerating Sparse General Matrix-Matrix Multiplication

There has been limited research on enhancing SpGEMM performance on FPGAs [53, 41, 33, 49]. Most of the existing works [53, 41, 33] adopt the inner product algorithm [81] in their implementations. The inner product algorithm attempts to compute all zero and nonzero output values. In the case of sparse matrices, there are a considerable amount of computations that result in a zero output, consuming clock cycles that can be spared with domain knowledge embedded into the architecture. Additionally, the dot product operation between a row and a column of the input matrices requires index matching, which further contributes to the overhead of the SpGEMM algorithm. Therefore, the inner product algorithm is not suitable for SpGEMM. Gustavson's algorithm presents an alternative SpGEMM data flow, resolving the mentioned issues yet introducing irregular memory access patterns. The existing implementation of Gustavson's algorithm [49] relies on cache-based hardware architecture to reuse the input matrix. However, when a row of the first input matrix (referred to as matrix A for the rest of the dissertation) has many nonzeros in large matrices, compute units require many rows of the second input matrix (matrix B), resulting in thrashing the cache and incurring significant performance penalties.

Different from prior work, I adopt Gustavson's algorithm to avoid zero output computation and reduce the synchronization overhead of computing partial products. Benefiting from the hardware flexibility of FPGAs, I propose a custom buffering scheme tailored to Gustavson's algorithm to improve the reuse of input matrices, thus largely reducing the amount of memory access. The custom buffering scheme is enabled by the proposed new Compressed Sparse Vector (CSV) format that transforms the memory access pattern of input matrices from irregular to regular, which improves memory bandwidth utilization and eliminates unnecessary memory stalls.

Also, a row reordering technique is proposed to rearrange rows of the input matrix to increase data locality for more efficient buffering, leading to increased temporal hardware resource utilization. Finally, I propose FSpGEMM, an OpenCL-based framework for accelerating SpGEMM on FPGAs. The proposed FSpGEMM framework consists of an FPGA OpenCL kernel implementing a throughput-optimized hardware architecture and a host program implementing preprocessing functions for reordering input matrices and converting them to the CSV format tailored to the proposed architecture. Overall, such synergies between the buffering scheme and Gustavson's algorithm, as well as between the CSV format and FSpGEMM as a result of the co-design methodology, significantly improve the computational performance.

1.5 Sparse Tensor Core

Considerable research has been conducted on developing efficient hardware accelerators for sparse linear algebra computations [90, 51, 75, 79]. The main performance bottleneck within processing elements of these accelerators arises from the *merge* (*i.e.*, addition) of sparse vectors (*e.g.*, rows of a sparse matrix), with a computation complexity of $\mathcal{O}(n^2)$. This complexity is due to comparing all indices of one operand (*i.e.*, sparse vector) against all indices of the other operand. Recent studies have introduced n-way (radix-n) merging, whereby n sparse vectors are added in parallel [90, 75]. However, the processing elements experience long stalls until all n operands have been retrieved from the main memory, a process reliant on the memory system. As a result, the discrepancy between the memory system's architecture and that of the processing elements leads to low resource utilization and, ultimately, diminished overall computational throughput.

Contrary to previous efforts, I propose the adoption of an E-record *Hardware Merge Sorter* (HMS) [76, 77] within the processing elements of sparse linear algebra

hardware architectures. An E-record HMS needs the minimum number (*i.e.*, 2) of operands while providing E parallel outputs per clock cycle. This approach facilitates the advantages of parallel merging without necessitating E parallel memory channels, thus achieving high utilization. Furthermore, I introduce the Sparse Tensor Core, an ASIC design for accelerating SpGEMM operations within sparse linear algebra. This co-processor is capable of executing highly parallel SpGEMM computations and has the potential to be applied to other sparse linear algebra algorithms (*e.g.*, sparse matrix-vector multiplication). Overall, the synergy between the memory system and processing elements, alongside the co-design of hardware architecture with the SpGEMM algorithm, markedly enhances computational performance.

Chapter 2

RELATED WORK

Three categories of works are seen in the previous papers: one related to the computation of the Cholesky factorization, one targeting the general matrix-matrix multiplication, and one discussing Algebraic Multigrid methods. In each category, I discussed and categorized CPU-, GPU-, and FPGA-based designs for dense (if applicable) and sparse algorithms.

2.1 Cholesky Factorization

2.1.1 Sparse Cholesky Factorization on CPUs and GPUs

cuSPARSE [2] is a popular CUDA sparse matrix library that can be used to approximate the sparse Cholesky factorization on Nvida GPUs. Since such an approximation algorithm is intrinsically different from the Supernodal Multifrontal algorithm adopted in this work that exactly computes the sparse Cholesky factorization, cuSPARSE is not considered as a reference method for comparison in this proposal.

CHOLMOD [19] is a set of routines for factorizing sparse symmetric positive definite matrices for CPUs and GPUs [74] using multifrontal and supernodal multifrontal sparse Cholesky factorization methods. Its supernodal Cholesky factorization provides highly optimized implementations relying on LAPACK and the Level-3 BLAS.

Recursive behavior of sparse Cholesky factorization algorithms and complex data dependencies among nodes in an elimination tree result in frequent off-chip memory access and poor cache locality. Moreover, complex iterative operations (*e.g.*, inverse square root) with strong loop-carried data dependency in sparse Cholesky factorization algorithms lead to low temporal/pipeline parallelism. Therefore, running these

algorithms on CPUs and GPUs suffers low performance.

2.1.2 Accelerating Sparse Cholesky Factorization on FPGAs

The work in [78] and [82] are based on the left-looking and multifrontal Cholesky factorization method. There is a major drawback in both of these works. The left-looking and multifrontal algorithms need more memory access and have larger computational complexity than the supernodal multifrontal algorithm [55]. Additionally, the work in [82] requires a scheduling algorithm for assigning the computation of nodes in an elimination tree to the FPGA accelerator. However, [82] did not provide any scheduling algorithm, resulting in suboptimal ordering of different nodes and, consequently frequent off-chip memory access. Moreover, their proposed hardware architecture introduces a long access latency and high overhead to store and read intermediate results to/from the off-chip memory. Since the work in [78] did not provide the runtime of their design, I compare the performance of FSCHOL with [82]. Neither of these two works provided power or energy consumption.

2.2 General Sparse Matrix-Matrix Multiplication

2.2.1 Sparse Matrix Formats

Figure 4.2 shows a compressed representation of a sparse matrix using Compressed Sparse Row (CSR) format [16]. In this format, nonzero values of the sparse matrix are laid out in the row-major orientation in the off-chip memory. The CSR format stores a sparse matrix using three arrays V, COL_INDEX, and ROW_PTR representing nonzero values and column index of the nonzero elements, and the pointer to the first nonzero element in the first two arrays, respectively.

In Compressed Sparse Column (CSC) [16] format, the nonzero elements are stored

in the column-major orientation using three arrays V, ROW_INDEX, and COL_PTR for nonzero values, row index, and the pointer to the start of each column.

These data formats are not tailored to a SpGEMM algorithm or hardware architecture. Hence, they are not efficient for specific methods or hardware designs, which leads to a huge performance loss. Unlike the CSR or CSC formats, the new CSV format makes the input data access in the proposed buffering scheme regular.

2.2.2 SpGEMM Algorithms

There are three main methods for computing SpGEMM: inner product, outer product, and Gustavson's method. The differences among the three methods are twofold. First, these methods require different data formats for the input matrices to acquire contiguous access to off-chip memory. Second, some methods avoid wasted computations by calculating the nonzero elements of the output matrix.

The inner product algorithm [81] computes all the elements of the output matrix, including zero elements. In SpGEMM, most of the output elements are nonzeros. Additionally, computing each element involves a dot product operation, including index matching and multiply-accumulate (MAC). Jamro et al. [41] identifies index matching as a hardware-expensive and the most time-consuming operation of the inner product method. Therefore, the inner product algorithm inevitably causes its implementations to summer from both performance and energy consumption overheads.

The outer product algorithm [81] performs an outer product operation between a column (row) of the first input matrix and row (column) of the second input matrix. The result of each outer product operation is a large partial sum matrix with the same dimensions as the input matrices. The number of partial sums produced is equal to the number of rows of the input matrices that are often large. Therefore, buffering and

accessing partial sums requires off-chip memory access that incurs long access latency and consumes high energy. Moreover, the addition of large partial sums suffers from synchronization overhead. Consequently, the outer product algorithm suffers from undesired performance and energy consumption overhead.

Figure 2.1 illustrates the row-wise Gustavson's method for multiplying two sparse matrices ($A \times B$). In this method, each non-zero element in a row of the first input matrix (*e.g.*, $A(i, j)$ where i and j are the row and column indices, respectively) is multiplied by all non-zero elements of the corresponding row of the second input matrix (*e.g.*, $B(j, :)$ where : is the slice operation), resulting in a intermediate row of sparse partial products (*e.g.*, $C_{j,temp}(i, :)$). The addition of the sparse partial products from the multiplication of all nonzero elements in a row (*e.g.*, the ith row) of the first input matrix with the corresponding rows of the second input matrix results in a final row of the output matrix (*e.g.*, $C(i, :) = \sum_j C_{j,temp}(i, :)$).

The addition of sparse partial products consists of two operations: sort and merge. Each sparse row is represented by a vector of pairs (VAL, COL_IND) representing the actual value and the column index of the corresponding nonzero elements. The rows are already sorted by COL_IND. To add two sorted sparse vectors, the two vectors are first sorted into a single vector based on COL_IND. Then, VALs of consecutive elements are merged (*i.e.*, added) with the same COL_IND into a single element.

The column-wise Gustavson's method is similar to the row-wise one but with rows and columns switched. Gustavson's method does not require the hardware-expensive index-matching operation among the elements of input matrices. Also, in Gustavson's method, the addition operation of sparse partial products has low on-chip memory requirement and synchronization overhead. In contrast, since an entire intermediate matrix is dealt with using off-chip memory in the outer product

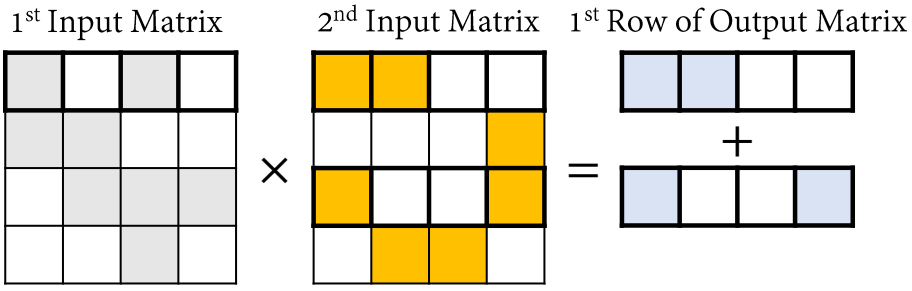


Figure 2.1: An Example For Calculating The First Row Of The Output Matrix For SpGEMM Using The Row-Wise Gustavson's Method. The Colored Elements Represent Nonzero Values, And The Thick Borders Highlight The Rows Involved In The Computation.

method, utilizing Gustavson's method results in both improved performance and lower energy consumption. Therefore, I choose the row-wise Gustavson's method to develop FSpGEMM upon.

However, Gustavson's method maintains sub-par reuse of the second input matrix's data due to the irregular access pattern of rows that are read from off-chip memory. Such an access pattern is dependent on the order of non-zero column indices in the rows of the first input matrix, making the caching policy or buffering scheme for the second input matrix too complex and costly to implement.

2.2.3 Accelerating Dense GEMM on FPGAs

Parametrized FPGA implementations of dot-product and matrix-vector multiplication kernels are presented in [44]. The work in [44] also compared the proposed kernels with CPU and GPU implementations in terms of performance and energy efficiency. FBLAS [26] proposes scalable, modular, and OpenCL-based implementations of the Basic Linear Algebra Subprograms (BLAS) library to improve the reusability of the FPGA kernels. This work on accelerating the BLAS library targets dense vectors

and matrices thus are not suitable nor efficient for accelerating SpGEMM.

2.2.4 Accelerating SpGEMM on FPGAs

The work in [53] and [41] proposes the design and implementation of the inner product algorithm for SpGEMM. They study the performance and energy consumption trade-off of the design by tuning the architectural parameters (*i.e.*, the number of PEs and the block size in blocking decomposition). Changing architectural parameters results in different FPGA resource utilization. FP-AMG [33] implemented a SpGEMM kernel for the acceleration of Algebraic Multigrid Solvers on FPGAs. All the existing FPGA-based works adopt the inner product method for SpGEMM, thus suffering from undesired performance and energy consumption overheads due to the costly index matching operation. Differently, I adopt Gustavson's algorithm to eliminate expensive index matching operations, zero output computations, and storage of large intermediate values resulting in significant performance and consequently, energy efficiency improvements.

2.2.5 Accelerating SpGEMM on Application-specific Integrated Circuits (ASICs)

There has been some recent work on accelerating SpGEMM on ASICs using either the outer product or Gustavson's method. OuterSPACE [64], SpArch [91], and the work in [66] utilize the outer product method. SpArch reduces the number of partial output matrices by matrix condensing to mitigate the overheads of synchronization and off-chip memory access. However, all these works still require off-chip memory access to store and retrieve intermediate results (*i.e.*, partial output matrices). MatRaptor [81] and GAMMA [90] use Gustavson's method to accelerate SpGEMM. Nonetheless, MatRaptor suffers from poor data reuse of the input matrices, leading to unnecessary off-chip memory access for loading input matrices repetitively with

a large performance penalty. GAMMA suggests utilizing a cache-based structure to reuse input matrix rows to simultaneously retrieve multiple rows from matrix B for parallel merging. This process demands significant memory bandwidth to fetch data and results in prolonged PE stalls in the event of a cache miss.

2.2.6 Accelerating SpGEMM on FPGAs

The work in [53] and [41] propose the design and implementation of the inner product algorithm on FPGAs. They study the performance and energy consumption trade-off by tuning the architectural parameters (*i.e.*, the number of processing elements, and the block size in the blocking decomposition). FP-AMG [33] implemented a SpGEMM kernel for the acceleration of Algebraic Multigrid Solvers on FPGAs. However, all these works adopt the inner product method for SpGEMM, thus suffering from undesired performance and energy consumption overheads due to the costly index matching operation. Differently, I adopt Gustavson's algorithm to eliminate expensive index-matching operations and zero output computations, resulting in significant performance and, consequently, energy efficiency improvements.

The work in [49] implements Gustavson's algorithm on a cache-based architecture for reducing bank conflicts on embedded FPGA devices, limiting the scalability and the scope of their design to FPGAs with traditional DDR-based memory systems. Additionally, even though the matrices are sparse, there are many non-zero elements per each row of matrix A, and the sparsity pattern of the adjacent rows might be very disjoint. In these cases, caching a large number of rows from matrix B is impossible and leads to long-latency data fetch due to cache misses. However, I propose a scalable hardware architecture compatible with an arbitrary memory system and buffering scheme enabled by a new format and a row reordering technique to ensure the locality of data.

2.2.7 GPU Implementations of SpGEMM

There have been many research approaches for the GPU acceleration of SpGEMM over the past few years, including cuSPARSE [61], Cusp [22], nsparse [60], RMerge [30], AC-SpGEMM [87], bhSparse [56], spECK [65], and TileSpGEMM [63]. TileSpGEMM [63] proposed an algorithm for introducing sparsity to the tiled dense GEMM and storing non-empty tiles in a sparse form. TileSpGEMM outperforms other GPU implementations and is used as the reference GPU solution for comparison in this study.

2.2.8 SpGEMM in DNNs

Matrix-matrix multiplication is a crucial operation in deep neural networks (DNNs). Sparsity can reduce memory and computation needs by leveraging data compression techniques such as pruning in both weights and activations. The systolic array-based designs proposed in [14] and [47] process convolutions by first transforming feature maps into a matrix using the im2col technique [84], where the input feature map matrix is dense, and the filter matrix is sparse, and the core operation is sparse matrix-dense matrix multiplication. Differently, in our design, I assumed both input matrices are sparse, which happens to be the case when input feature maps (or activation values) are zero or close to zero.

SparseTIR [89] proposed a compilation abstraction for optimizing the GPU performance of deep learning workloads, achieved through the construction of a composable search space of formats and transformations. The work in [52] proposed a sparse convolution flow in TVM to enhance the im2col plus GEMM implementation of convolution for optimization coverage with different characteristics.

2.2.9 Reordering Algorithms

Previous research addressed memory access problems, including irregular row accesses, in SpGEMM and SpMV (sparse matrix-vector multiplication) algorithms, using preprocessing techniques such as tiling and reordering [42, 36, 69, 90]. An optimal solution to these problems is NP-complete [69]. Therefore, these works adopted different heuristics to find a feasible solution. Most approaches [42, 90, 69] are not runtime efficient. The work in [48] adopted the ELL format for matrix representation, which suffers from redundant calculations, and attempted to alleviate this issue by reorganizing the data positions. The study conducted in [67] tested several reordering methods, such as Approximate Minimum Degree [12], Distance function [68], Reverse Cuthill–McKee [21], and METIS [43], to observe their impact on GPU performance.

Differently, our approach uses a novel sparse format without incurring any redundant computations, and our proposed row reordering algorithm has a better runtime complexity by using a more efficient heuristic and only considering the non-empty rows.

Chapter 3

ACCELERATING SPARSE CHOLESKY FACTORIZATION

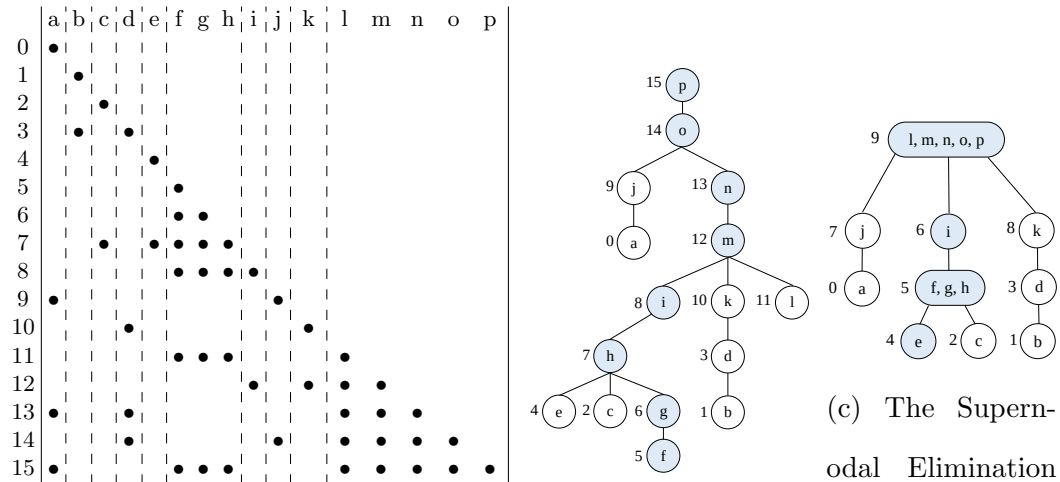
This chapter discusses *FSCHOL* which is an OpenCL-based HPC framework proposed for accelerating sparse Cholesky factorization on FPGAs.

3.1 Multifrontal Cholesky Factorization

Cholesky factorization is an efficient method for decomposing a symmetric positive-definite matrix (A) into the product of a lower triangular matrix and its transpose ($L \times L^T$). In many real-life science and engineering applications, matrix A is sparse [54]. One of the significant methods for sparse matrix factorization was introducing the multifrontal Cholesky factorization [27].

The multifrontal method reorganizes the overall factorization of a sparse matrix into a sequence of partial factorizations of smaller dense matrices [55]. The main feature of the multifrontal method is that the update contributions from a factor column i ($L(:\, i)$) to the remaining submatrix are computed and accumulated with contributions from other factor columns before the updates are performed. Therefore, this method reduces the number of memory accesses and operations comparing to left or right-looking algorithms [82].

The main concepts in the multifrontal method are the elimination tree, frontal and update matrices (denoted by F and U, respectively), and the extended-add operation. The elimination tree of the matrix L is defined as a tree structure with n nodes such that node P is the parent of node C if and only if the first subdiagonal nonzero element at column C is located at row P. Figure 3.1a and 3.1b shows the nonzero pattern of an example matrix L and its corresponding elimination tree, respectively.



(a) The Nonzero (Circles) Pattern Of Matrix L.

(b) The Elimination Tree.

(c) The Supernodal Elimination Tree.

Figure 3.1: The Factor Matrix Sparsity Pattern And Its Corresponding Elimination Trees.

If I define the critical path as the longest path of nodes from the first to the top level of the elimination tree (*e.g.*, colored nodes in Figure 3.1b), the number of nodes in the critical path determines the maximum amount of dependency among nodes to be resolved.

A practical improvement to the multifrontal method is the use of supenodes [55]. A supernode is a group of columns (*i.e.*, nodes in the elimination tree) if they can be treated as one computational unit in the course of sparse Cholesky factorization. If I define the sparsity structure (*i.e.*, nonzero patterns) of column j as $Struct(L(:,j))$, the set of contiguous columns $\{j, j+1, \cdots, j+t\}$ constitutes a supernode if $Struct(L(:,k)) = Struct(L(:,k+1)) \cup \{k\}$ [62]. One can refer to [82] and [27] for the detailed comparison on different sparse Cholesky factorization algorithms.

Figure 3.1c shows the supernodal elimination tree of Figure 3.1b. Algorithm 1

describes the sparse Cholesky factorization using the supernodal multifrontal method. In Algorithm 1, notion $nonzeros(V)$ is equivalent to the dense form (*i.e.*, nonzero elements) of sparse vector V. Also, notion $\oplus$ represents the extend-add operation that adds two matrices with different dimensions by extending the smaller matrix with zeros. Comparing the critical path of the two methods, the node dependency is reduced. Moreover, Algorithm 1 introduces more parallelism in each outer loop iteration. Additionally, since the update matrix is generated per supernode rather than a node, the number of operations and memory accesses is reduced.

3.2 Framework Design

The FSCHOL framework consists of two parts: The FPGA kernel and the host program running on the CPU. The kernel code implemented on the FPGA accelerator performs computationally intensive tasks. On the host side, the OpenCL API supports efficient management and scheduling of tasks running on the FPGA.

3.2.1 Hardware Architecture of the FPGA Kernel

Architectural Overview

Figure 5.3 shows a high-level block diagram of FSCHOL's hardware architecture, including five modules: two processing elements (PEs), one load, and two store modules. The PEs are responsible for computations, while load and store modules read and write input and output data to/from off-chip memory, respectively. All modules process data in a pipelined and vectorized fashion. PEs are connected to load and store modules via FIFO channels. Also, PEs utilize FIFOs to send and receive intermediate results to/from each other. Separating load/store modules from PEs and connecting them using a FIFO helps compensate for the difference between the off-chip memory bandwidth and the data processing throughput.

Algorithm 1: The Supernodal Multifrontal Cholesky Factorization [55]

for *each supernode S in increasing order of first column subscript* **do**

Let $S = \{j, j+1, \cdots, j+t\}$;

Let $j+t, i_1, \cdots, i_r$ be the locations of nonzero elements in $L(:, j+t)$;

$$F_S = \begin{bmatrix} a_{j,j} & a_{j,j+1} & \cdots & a_{j,j+t} & a_{j,i_1} & \cdots & a_{j,i_r} \\ a_{j+1,j} & a_{j+1,j+1} & \cdots & a_{j+1,j+t} & a_{j+1,i_1} & \cdots & a_{j+1,i_r} \\ \vdots & \vdots & \cdots & \vdots & \vdots & \cdots & \vdots \\ a_{j+t,j} & a_{j+t,j+1} & \cdots & a_{j+t,j+t} & a_{j+t,i_1} & \cdots & a_{j+t,i_r} \\ a_{i_1,j} & a_{i_1,j+1} & \cdots & a_{i_1,j+t} & 0 & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots & \vdots & \cdots & \vdots \\ a_{i_r,j} & a_{i_r,j+1} & \cdots & a_{i_r,j+t} & 0 & \cdots & 0 \end{bmatrix};$$

nchildren := no. of children of S in supernodal elimination tree;

for *C from 1 to nchildren* **do**

$F_S = F_S \oplus U$; `// update(S,C)`

end

`/* start of factorize(S)` `*/`

for *i from 0 to t* **do**

$nonzeros(L(:, j+i)) = F_S(i:, i)/\sqrt{F_S(i,i)}$;

end

$$U_S = F_S(t+2:, t+2:) - \begin{bmatrix} l_{i_1,j+t} \\ \vdots \\ l_{i_r,j+t} \end{bmatrix} \begin{bmatrix} l_{i_1,j+t} & \cdots & l_{i_r,j+c} \end{bmatrix};$$

`/* end of factorize(S)` `*/`

end

When the data processing throughput does not match the available off-chip memory bandwidth, and loading and storing primary input and output data happen in the same module that the data are being processed, the load and store operations would be stalled for the computation units. When the depth of the FIFO channels is optimized by the offline compiler, the load and store modules are able to continuously read and write data from/to the off-chip memory and write and read them to/from the channels, respectively. For each supernode, in addition to several consecutive columns of input matrix A depending on the size of the supernode, a PE needs configuration information on how to process the assigned supernode (job). The job information is set by the scheduling algorithm in the host program. Therefore, channels Q_A and Q_{job} are used to send the elements of matrix A and the job information, respectively. Channel Q_P is used to send the Boolean elements of the pattern matrix to PEs for the extend-add operation which the details are discussed in Section 3.2.1. Channel Q_L sends the consecutive columns of factor matrix L of the corresponding processed supernode from a PE to the store module. Channels Q_F transmit the intermediate value of matrix F among PEs.

Load/Store Modules

Each load module iterates over the number of jobs that are assigned by the scheduling algorithm. First, the load module sends a data structure containing the configuration bits of the assigned job described in Table 3.1. When a supernode S is assigned to a PE, S needs to be updated by all of its children according to lines 6-8 of Algorithm 1. Bit *job.up* determines whether S is updated by any of its children. If not, the load module reads input data from off-chip memory and write them to Q_A to be consumed by a PE. Store modules read the output data from PEs and send them to the off-chip memory as soon as a vector of factor matrix L is ready.

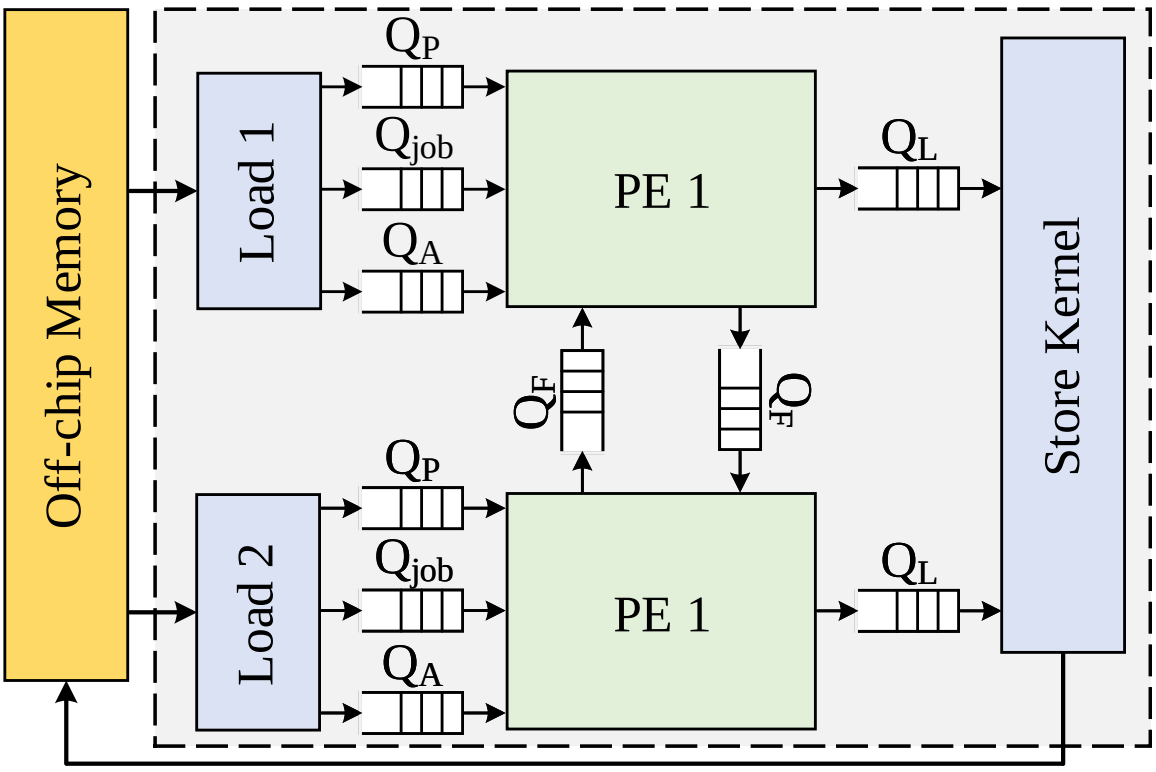


Figure 3.2: The High-Level Block Diagram Of The Hardware Architecture.

Processing Element (PE)

Based on Algorithm 1, I define two major operations in each outer loop iteration: *update*, and $factorize$. Operation $update(S, C)$ (line 7 of Algorithm 1) updates the frontal matrix of supernode S (F_S) using the update matrix of its child supernode C (U_C). Operation $factorize(S)$ (lines 9-12 of Algorithm 1) produces $t + 1$ columns of factor matrix L and the update matrix of supernode node S (U_S). Figure 4.6 shows the high-level block diagram of each PE. Modules Vector Addition and Matrix Extension are responsible for operation *update* and modules Sqrt, Vector Division, Outer Product, and Vector Subtraction perform operation $factorize$. For each scheduled job, a PE performs one *update* and one $factorize$ operation (depending on the update status of F_S) in a pipelined and vectorized fashion.

Table 3.1: Configuration Bits Of A Job.

Attribute	Description	Type
job.up	supernode is partially updated	Boolean
job.last_c	supernode is going to be updated by its last child	
job.F_rd	intermediate matrix F should be read from inter-module FIFO	
job.F_wr	intermediate matrix F should be written to inter-module FIFO	
job.U_rd	update matrix U should be read from FIFO	

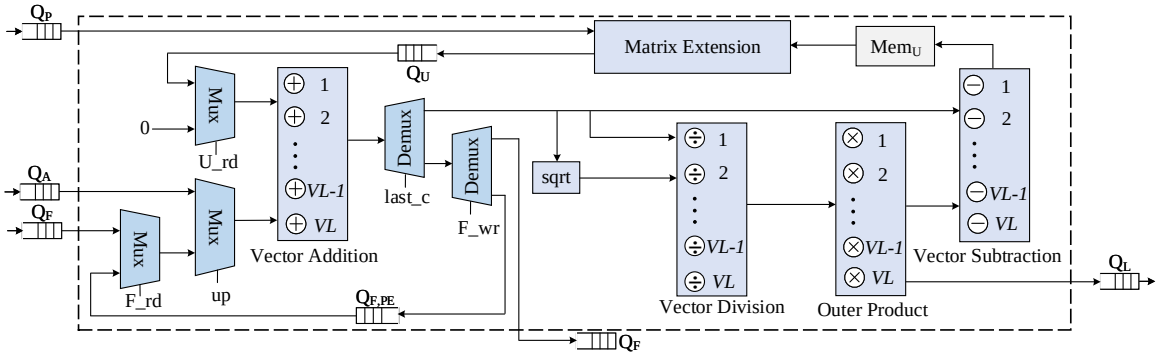


Figure 3.3: The High-Level Block Diagram Of A PE.

When there is no available update matrix to update the frontal matrix of supernode S, intermediate (*i.e.*, partially updated) values of F_S are stored to complete the update process (the *for* loop at lines 9-11 of Algorithm 1) later. Storage units

for storing intermediate results inside each PE include RAM Mem_U for the update matrix, FIFO channel Q_U for the extended matrix update matrix, and FIFO channel $Q_{F,PE}$ for the partially updated frontal matrix. Two inter-module FIFO channels are used to send and receive the intermediate matrix F (Q_F) among PEs whenever necessary.

For each job, PE starts by reading the configuration bits to control the multiplexers. Vectors V_U and V_F are defined to represent a vector with size VL of matrices F and U, respectively. If *job.up* is not set, node S is not updated by any of its children, and its factor vector V_F was not initialized with the values of matrix A as described in line 4 of Algorithm 1. Therefore, vector V_F is initialized by input data from Q_A. If node S is already updated, depending on the value of $job.F\_rd$, V_F reads its value from Q_F or $Q_{F,PE}$. According to Table 3.1, if $job.F\_rd$ is set, it shows that the intermediate values of matrix F as the result of a previous update of supernode S by one of its other children are stored at the inter-module FIFO $Q_{F,PE}$; otherwise they are stored at intra-module FIFO Q_F. If node S does not have any child, there is no update matrix to update node S. Therefore, U_{rd} determines whether vector V_U should be initialized by zeros (when there is no child) or by the values stored at Q_U. After loading vectors V_U and V_F with the corresponding values, they are added in parallel, and the output is stored at V_F.

After adding vectors V_U and V_F, a PE decides whether to store the results or feed V_F to the pipeline for the $factorize(S)$ operation. As described in Table 3.1, bit $job.last\_c$ defines whether node S is ready to be factorized. If it is not set, PE stores vector V_F in FIFO Q_F or $Q_{F,PE}$ depending on the value of $job.F\_wr$. If it is set, the PE takes the square root of the first $t+1$ diagonal elements of F_S $(f_{1,1}, f_{2,2}, \cdots, f_{t+1,t+1})$ and divides the superdiagonal elements of F_S by the square root value. The PE uses elements $l_{i_1,j+t}, l_{i_1,j+t}, \cdots, l_{i_r,j+c}$ for the consequent outer

$$F = \begin{bmatrix} f_{1,1} & f_{1,2} & f_{1,3} \\ f_{2,1} & f_{2,2} & f_{2,3} \\ f_{3,1} & f_{3,2} & f_{3,3} \end{bmatrix}, U = \begin{bmatrix} u_{1,1} & u_{1,2} \\ u_{2,1} & u_{2,2} \end{bmatrix}, P = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

$$\Rightarrow F \oplus U = \begin{bmatrix} f_{1,1}+u_{1,1} & f_{1,2} & f_{1,3}+u_{1,2} \\ f_{2,1} & f_{2,2} & f_{2,3} \\ f_{3,1}+u_{2,1} & f_{3,2} & f_{3,3}+u_{2,2} \end{bmatrix}$$

Figure 3.4: An Example Of Extend-Add Operation.

product operation. Moreover, update matrix U_S is computed as describe at line 10 of Algorithm 1 and stored in Mem_U.

Figure 3.4 shows an example of the extend operation at line 7 of Algorithm 1 and how update matrix U is extended and stored on Q_U. The PE initializes an index counter for Mem_U and a vector of elements to zeros, and loops over all elements of a Boolean pattern matrix (P) received from FIFO channel Q_P. If an element of matrix P is true, an element of the update matrix in Mem_U is stored to the vector location indexed by the counter, and the index counter is increased by 1. Otherwise, the corresponding element of the vector is skipped and kept as zero. Once VL iterations have been passed, the vector is stored on Q_U.

OpenCL Implementation

FSCHOL takes advantage of the loop unrolling technique provided by OpenCL for parallel implementation of vectorized operations. The PEs are implemented as *autorun* modules. An *autorun* module starts executing automatically and does not need to be launched by a host. Therefore, the Intel OpenCL offline compiler for FPGA does not need to generate the communication logic between the host and PEs, which

Table 3.2: The Required Size (Bits) For Each Storage Unit.

Unit	Size
$Q_U(2\times)$	$[(N+M)\times(N+M)\times VL]\times VL\times WL$
$Q_F(2\times)$	
$Q_{F,PE}(2\times)$	
$Mem_U(2\times)$	$[N\times VL]\times[N\times VL]\times WL$

reduces logic utilization and enables additional performance optimizations [40]. However, load and store modules are launched explicitly by the host since they need to access off-chip memory.

Most storage units are implemented as FIFO channels rather than RAMs, where random access is not required. However, for Mem_U there is no choice other than using RAMs. Although using shift registers would be more efficient in terms of hardware complexity, the number of elements to be stored is unknown at the compile time and depends on dimensions of F_S. Additionally, no intermediate data is written and read to/from off-chip memory.

As mentioned before, the datapath has VL single-precision floating-point numbers, and the arithmetic submodules perform VL operations in parallel. Table 3.2 shows the size of storage units for each PE. Dimensions of FIFOs and RAM blocks are represtend as $Depth \times Width$ and $Rows \times Columns$, respetively. Parameter WL is the word length of the data format (*e.g.*, 32 bits for single-precision floating-point).

There are three tunable parameters: VL, N, and M. Parameter M and N are used to scale the design to support arbitrary sizes for frontal and update matrices based on the maximum supernode size (*i.e.*, the maximum number of consecutive columns with the same sparsity pattern) and the maximum number of nonzero elements among all

columns of the factor matrix, respectively. The total number of required DSPs for the FPGA kernel with two PEs is equal to $7 \times VL + 5$. According to Table 3.2, total required size for the storage units is $(8N^2 + 6M^2 + 12M \times N) \times VL^2 \times WL$ bits.

Performance-optimized Model for Determining Design Parameters

It is important to provide a guideline to derive design parameters (VL, N, and M) to minimize the runtime subject to the available on-chip resources and the characteristics of the input matrices.

I formulate the runtime (R) as $R = \frac{N}{F \times P \times U}$, where N, F, P, and U are the number of floating-point operations (FLOPs) to factorize a matrix using Algorithm 1, the clock frequency, the computational parallelism, and the spatial-temporal utilization factor, respectively. U is a statistical metric that measures the average occupation (in both space and time) of the available computation resources in a computing device for performing the FLOPs of a given algorithm. Therefore, U is the average ratio of effective computational parallelism per clock cycle ranging from 0 to 1. N is an algorithmic parameter and depends on dimensions and the sparsity structure of the input matrix. P is determined by the total number of utilized DSP units calculated as $7 \times VL + 5$. Therefore, R can also be expressed as

$$R = \frac{N_{Ops}}{F \times (7 \times VL + 5) \times U} = \frac{\alpha}{7 \times VL + 5}, \tag{3.1}$$

where $\alpha = \frac{N_{Ops}}{F \times U}$ is a empirical term that lumps the algorithm- and implementation-specific terms N_{Ops}, F, and U. Note that α is introduced mainly to simply the formulation and can be treated as a constant when determining the optimal architectural parameters.

I define the constrained optimization problem as

$$\text{minimize} \quad R(VL) = \frac{\alpha}{7 \times VL + 5}$$

$$\begin{aligned} \text{subject to} \quad & f_1(VL) \leq C_1, f_2(VL, M, N) \leq C_2, \\ & M = \lceil \frac{C_3}{VL} \rceil, N = \lceil \frac{C_4}{VL} \rceil, \end{aligned} \tag{3.2}$$

where $f_1(VL) = 7 \times VL + 5$ and C_1 are the total number of required and available DSP blocks, respectively, $f_2(VL, M, N) = (8N^2 + 6M^2 + 12M \times N) \times VL^2 \times WL$ and C_2 are the total required and available RAM size, respectively, C_3 is the maximum supernode size (*i.e.*, maximum number of consecutive columns of the factor matrix with the same sparsity pattern), and C_4 is the maximum number of nonzero elements among all columns of the factor matrix. For the total available RAM size (C_2) one must consider a margin from the values reported in the FPGA device datasheet to consider for RAM resources used for the glue and control logics.

Based on $R(VL) = \frac{\alpha}{7 \times VL + 5}$, to minimize the runtime, VL should be maximized. The constrained optimization problem defined as Equation 4.7 has an analytical solution as following.

1. Derive the first constraint for VL from $f_1(VL) \leq C_1$.

2. Derive the second constraint for VL by using $M = \lceil \frac{C_3}{VL} \rceil$ and $N = \lceil \frac{C_4}{VL} \rceil$ in $f_2(VL, M, N) \leq C_2$.

3. Now I have two ranges of values for VL. The maximum value of VL is determined from the tighter constraint.

4. Using calculated VL, the values of N and M are found from $M = \lceil \frac{C_3}{VL} \rceil$ and $N = \lceil \frac{C_4}{VL} \rceil$.

3.2.2 Scheduling Algorithm

Concepts

The main challenges involved in co-designing the scheduling algorithm with the the proposed hardware architecture for the FPGA kernel stem from two perspectives. On the one hand, as the objective of the scheduling algorithm is to minimize the amount of off-chip memory access on the FPGA device, the impact of workload scheduling on the amount of off-chip memory access must be accurately modeled based on the specific architecture of the FPGA kernel. On the other hand, the scheduling algorithm needs to generic enough to be able to adapt to the FPGA kernel implementations in various computational parallelism on different FPGA devices.

I propose and implement a scheduling algorithm that intelligently assigns the update and factorization operations (*update* and $factorize$) to PEs to minimize on-chip memory requirements with no off-chip memory access for the storage of intermediate results. The main goal of the algorithm is to pipeline the dependency among a child supernode C and its parent supernode S to avoid storing the update matrix of supernode C. Therefore, the algorithm assigns operation $update(S, C)$ immediately after operation $factorize(C)$. In this order, when supernode C is factorized and its update matrix is produced, the update matrix is consumed by supernode S at the same PE to reduce the on-chip storage requirements and reduce communications among different PEs. I provide the pseudo-code only for PE1 as it is the same for PE2. Note that the algorithm works for both supernodal and potentially multifrontal Cholesky factorization.

Details

In Algorithm 2, list nc is used to keep track of the number of children that each supernode is already updated by. Line 3-6 works on supernodes with no child. Since they do not have any dependency, it is only needed to initialize the frontal matrix F with elements of the matrix A and immediately factorize the supernode. Once a supernode with no dependency is factorized in a PE, its update matrix is ready to update its parent supernode. Therefore, the supernode ID is pushed to PE's queue. Then, the supernode ID is popped from the queue, and the corresponding parent supernode is updated. Suppose a supernode is updated by all of its children. In that case, it is ready to be factorized, and the resulting update matrix is used for updating the parent supernode in the consequent job assigned to this PE. Otherwise, the queue would be empty, and the algorithm starts with lines 3-6. During this process, if a parent supernode needs to be updated in a different PE than it was updated, the configuration bits of the job assigned to the other PE is updated to let the PE know that it has to send the intermediate frontal matrix to the inter-module FIFO (Q_F) as shown by crossed-out functions in Figure 3.5.

The scheduling algorithm is implemented as a part of the host code. As an example, I apply the algorithm to the elimination tree illustrated in Figure 3.1c. The ordered list of supernodes is $T = [0, 4, 1, 2]$. Lists P and NC are $[7, 3, 5, 8, 5, 6, 9, 9, 9, -1]$ and $[0, 0, 0, 1, 0, 2, 1, 1, 1, 3]$, respectively. Value -1 in list P shows that the last node is reached. The output of this example is shown in Figure 3.5.

update(9,8)	factorize(9)		
update(8,3)	factorize(8)	update(9,6)	send(F_9)
update(3,1)	factorize(3)	update(6,5)	factorize(6)
update(1,-)	factorize(1)	update(5,2)	factorize(5)
update(9,7)	store(F_9)	update(2,-)	factorize(2)
update(7,0)	factorize(7)	update(5,4)	store(F_5)
update(0,-)	factorize(0)	update(4,-)	factorize(4)
PE1		**PE2**	

Figure 3.5: The Output Of Scheduling Algorithm For The Supernodal Elimination Tree In Figure 3.1c. The Time Increases From Bottom To Top.

3.3 Evaluation

3.3.1 Setup

I evaluate the performance and energy efficiency of FSCHOL in terms of runtime (s) and energy consumption (J). To evaluate the design, I select a set of matrices from the publicly available SuiteSparse Matrix Collection [25] (formerly known as the University of Florida Sparse Matrix Collection), a set of sparse matrices in real applications. The characteristics of the matrices are summarized in Table 5.1 including matrix dimensions (the number of rows and columns are equal), the density percentage calculated from $\frac{No.\ of\ Nonzero\ Elements}{Matrix\ Size} \times 100$, and the number of supernodes.

The data format in our design is single-precision floating-point (32-bit). I develop and implement FSCHOL using Intel FPGA SDK for OpenCL with Quartus Prime Pro 20.1. I compare FSCHOL with CPU and GPU versions of CHOLMOD [19].

As mentioned before, the supernodal multifrontal method decomposes the sparse

Cholesky factorization into a series of dense factorizations. These dense factorizations rely on dense BLAS and LAPACK libraries. Therefore, to improve the performance of the CHOLMOD library on CPU, I use Intel Math Kernel Library (Intel MKL) [86] instead of single-threaded BLAS and LAPACK routines. Intel MKL is a set of highly optimized, threaded, and vectorized math functions that maximize the performance of Intel's processors.

I measure the performance of the CPU implementation of CHOLMOD on a dual-socket Intel Xeon E5-2637 v3 CPU [6] with an effective bandwidth of 51 GB/s per socket, and the GPU implementation of CHOLMOD on an NVIDIA Tesla V100 GPU, one of the most powerful data center GPUs for accelerating HPC [9]. Our work is evaluated on an Intel Stratix 10 GX FPGA Development Board [5]. Table 3.4 summarizes the specifications of the CPU system, the GPU device, and the FPGA board used in the evaluation.

3.3.2 Experiment Results

Performance Comparison with CPU and GPU Implementations

The architectural parameters for implementing the FPGA kernel are $VL = 128$, $N = 4$, and $M = 2$. The FPGA kernel runs at 236 MHz. Table 3.5 shows the resource utilization for the implemented FPGA kernel.

Table 3.6 shows the performance comparison of FSCHOL in terms of runtime in seconds with the GPU version of CHOLMOD and the CPU version of CHOLMOD enhanced with Intel MKL library for implementing the supernodal multifrontal Cholesky factorization algorithm. The lower runtime shows higher performance. According to Table 3.6, the CPU implementation outperforms the GPU version of CHOLMOD since algorithms and applications with low arithmetic computation and complex mem-

ory handling are more efficient to be mapped on CPUs than on GPUs [28]. Also, the GPU runs with the error correction code (ECC) turned on at the base clock speed. One can further boost the GPU performance using a boost clock speed with ECC turned off [1] at the cost of much higher power consumption and random bit errors. I choose the base clock speed with ECC to evaluate the sustainable performance achievable in a scientific computing environment. The GPU can not work at a boost clock speed permanently or for a long time. Moreover, turning off ECC compromise the results probabilistically where accurate results are necessary for scientific computing.

Figure 5.6 shows the performance improvement of FSCHOL over the CPU and GPU implementations for corresponding the matrix. FSCHOL improves the performance of CPU and GPU versions of CHOLMOD by 0.8×-29.4× and 3.7×-33.7×, respectively. FSCHOL improves the performance on average by 5.5× and 9.7× over CPU and GPU implementations, respectively. For matrices where FSCHOL is less performant than the CPU implementation of CHOLMOD, the sparsity pattern is less structured and there are too many supernodes that cause performance degradation, while for matrices with a very structured sparsity pattern and a small number of nonzero supernodes compared to the matrix dimension, FSCHOL significantly improves the performance as the performance scales with the number of supernodes and the matrix size.

Performance Comparison with the State-of-the-art FPGA Implementation

The work in [82] implemented the multifrontal Cholesky factorization algorithm on a Xilinx Virtex-7FPGA VC709 evaluation board with a 28nm FPGA device [10]. Differently, FSCHOL is implemented on an Intel Stratix 10 GX FPGA development board with a 14nm FPGA device [4]. For a fair comparison, I normalize the performance numbers of [82] to the 14nm technology node using the scaling factor of

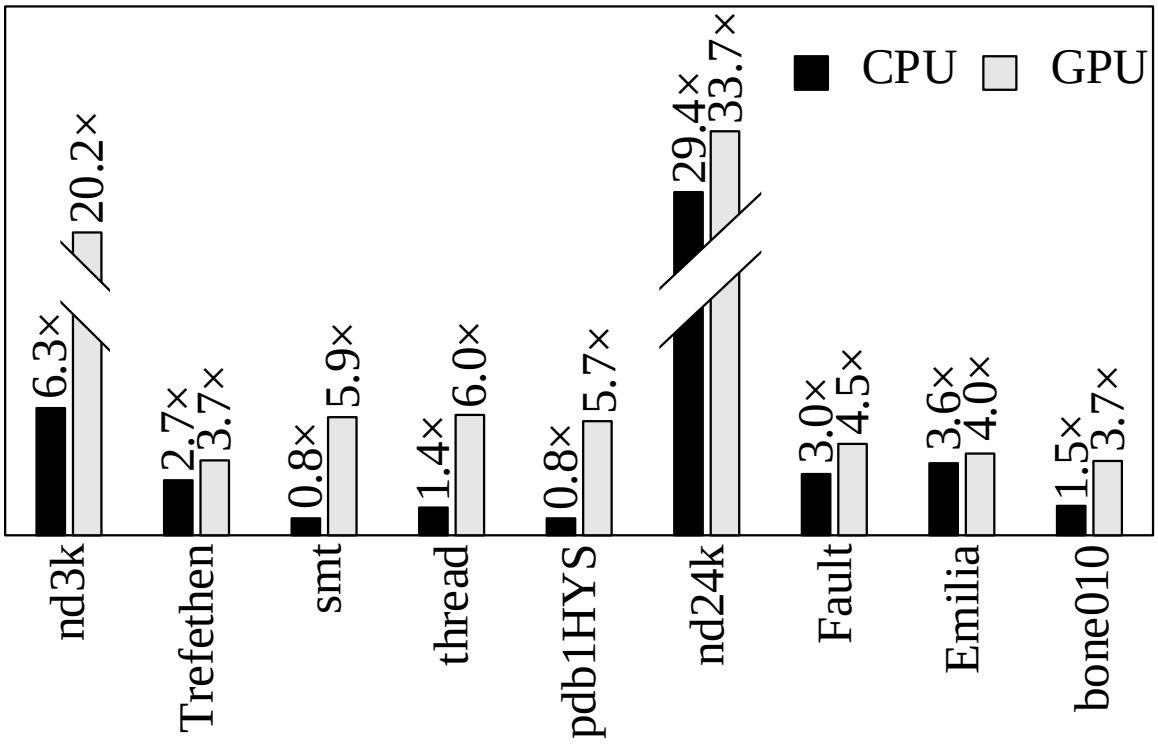


Figure 3.6: Runtime Speedup Of The FPGA Implementation Of FSCHOL Over The CPU And GPU Implementations Of CHOLMOD. The GPU Runs With ECC Turned On At The Base Clock Speed.

$delay = 1/S$, where $S = 28/14nm$, commonly used in the existing literature [73]. In addition, the performance of the FSCHOL solution is evaluated based on the same matrices used in the reference work [82].

Table 3.7 shows the performance comparison of FSCHOL in terms of normalized runtime in seconds with the work in [82] for implementing the multifrontal Cholesky factorization algorithm. The original performance results of [82] are also shown in Table 3.7. Similarly, the lower runtime shows the higher performance, and the numbers in parentheses show the performance improvement of FSCHOL with respect to the corresponding matrix. The experimental results show that the proposed FSCHOL solution outperforms the reference design in terms of performance on average by 11.7×

across the three different benchmarks.

The performance improvement of FSCHOL over the work in [82] is primarily due to the elimination of off-chip memory access for buffering intermediate results as a result of our software-hardware co-design methodology. Specifically, the scheduling algorithm implemented in the host program is tailored to the proposed hardware architecture of the FPGA kernel to offload the computational workloads of different supernodes in an optimized order that maximizes the data reuse of intermediate results, thus avoids unnecessary off-chip memory access.

Energy Efficiency Comparison with CPU and GPU Implementations

I measure the average power consumption of the Intel Stratix 10 GX development board using the Power Monitor tool in the Board Test System (BTS) application provided by Intel [39] during FPGA kernel execution. BTS measures the supply voltage and the drawn current of the entire FPGA board by reads values from on-board sensors.

For the power measurement of the CPU, I utilize likwid-powermeter tool from Likwid [31] to access the Running Average Power Limit (RAPL) counters on the Intel CPU. The RAPL interface is controlled through MSR registers [38]. For the power measurement of the GPU, I utilize the POWER query option [7] of NVIDIA System Management Interface (*nvidia-smi*) [8] tool.

Table 3.8 summarizes the energy consumption (J) of different implementations calculated from multiplying the runtime (s) and the power consumption (W). Figure 3.7 show the energy consumption reduction factor of FSCHOL over the CPU and GPU implementations for corresponding the matrix. FSCHOL reduces the energy consumption of the CPU and GPU implementations on by 1.6×-54.7× and 8.5×-92.1×, respectively. FSCHOL reduces the energy consumption on average by 10.3×

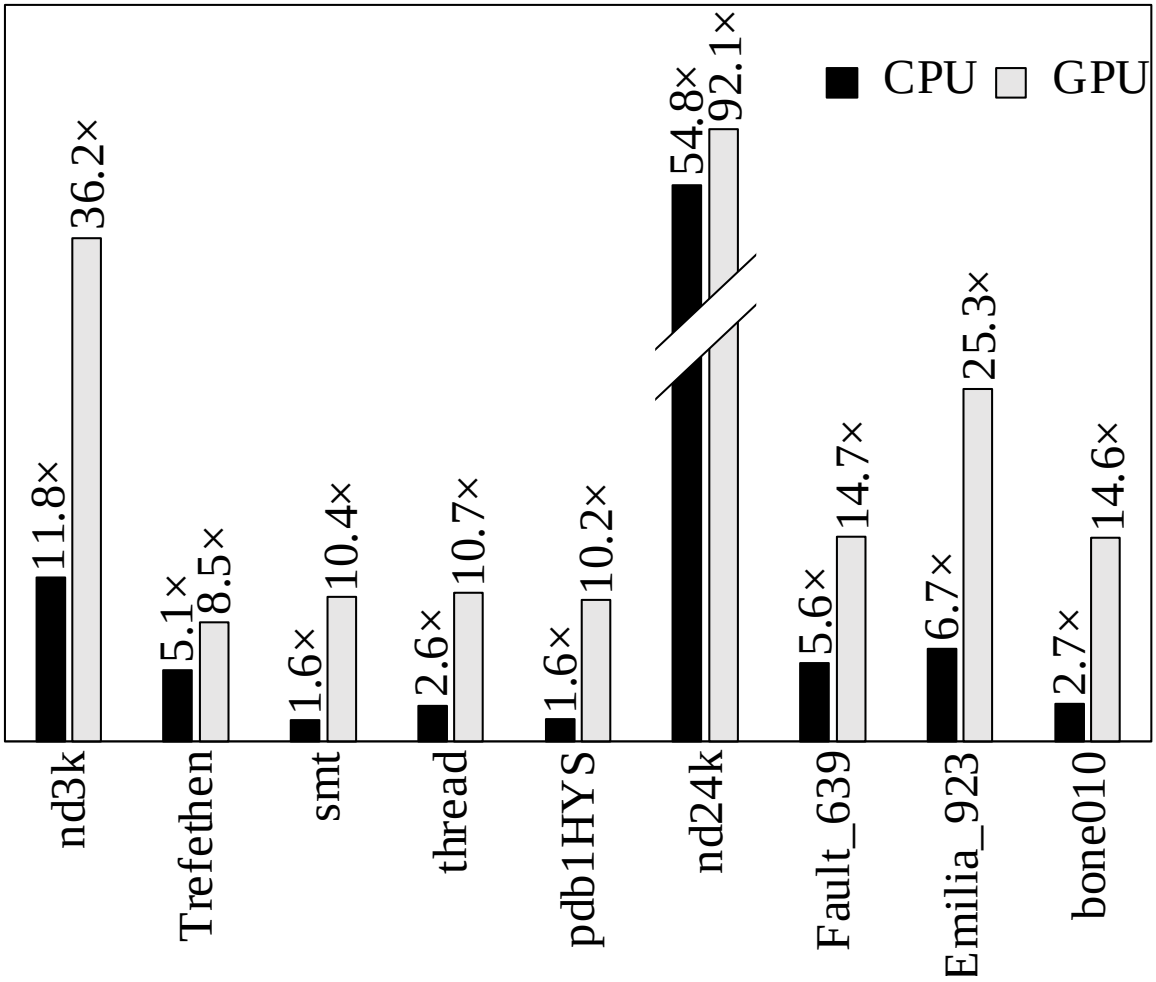


Figure 3.7: Energy Consumption Reduction Of The FPGA Implementation Of FSCHOL Compared To The CPU And GPU Implementations Of CHOLMOD. The GPU Runs With ECC Turned On At The Base Clock Speed.

and 24.7× over CPU and GPU implementations, respectively. Since the work in [82] did not provide any result on power or energy consumption, I can not compare FSCHOL in terms of energy consumption with [82].

3.4 Conclusion

In this chapter, I present FSCHOL, an OpenCL-based HPC framework for FPGA acceleration of sparse Cholesky factorization. FSCHOL includes a deeply pipelined and scalable FPGA kernel that accelerates supernodal multifrontal Cholesky factorization algorithm and a scheduling algorithm for efficient assignment of computational nodes for potentially all elimination-tree-based multifrontal methods.

I propose a performance-optimized model to derive architectural parameters for the FPGA kernel subject to the available on-chip resources (DSPs and RAMs) and input matrix characteristics (the maximum supernode size and the maximum number of nonzero elements among all columns of the factor matrix) to map the design into suitable data-center grade FPGAs.

The experimental results based on the Intel Stratix 10 GX FPGA development board for accelerating the Cholesky factorization of a set of sparse matrices from SuiteSparse Matrix Collection show on average one order of magnitude higher performance and lower energy consumption compared to the state-of-the-art implementations of sparse Cholesky factorization on CPU, GPU, and the other FPGA work [82].

Algorithm 2: The Scheduling Algorithm

Input: The list of (super)nodes with no child (T), the list of parents for each (super)node (P), and the list of the no. of children of each (super)node (NC).

Output: The assignment of *update* and $factorize$ operations to each PE.

```
1  initialize list nc to zeros to keep track of the no. children that updates each (super)node;
   initialize Q_1 and Q_2 as empty to store the ready (super)node to be updated or factorized;
2  while the top (super)node is not factorized do
3      if Q_1 is empty then
4          S = T[0];
5          assign update(S, −) to PE_1;
6          remove T[0] from T;
7          p = P[S];
8          push(p, Q_1);
9      else
10         S = pop(Q_1);
11         indicate (super)node S is being updated;
12         indicate vector V_U should be read from Q_U;
13         if nc[S] is equal to zero then
14             if (super)node S was updated in this PE then
15                 indicate vector V_F should be read from Q_{F,PE};
16             else
17                 indicate vector V_F should be read from Q_F;
18                 update job bits of the other PE to write the intermediate vector V_F to Q_F;
19             end
20         end
21         nc[S] + +;
22         if nc[S] is equal to NC[S] then
23             assign(p[S]) to PE_1;
24             push(S, Q_1);
25         end
26     end
       /* similar approach for PE_2                                          */
27 end
```

Table 3.3: The Specification Of Matrices Chosen From The SuiteSparse Matrix Collection.

Matrix	#Supernodes	#Rows	Density (%)
nd3k	87	9,000	4.049
Trefethen_20000b	3,678	19,999	0.139
smt	856	25,710	0.567
thread	923	29,736	0.503
pdb1HYS	1,149	36,417	0.328
nd24k	625	72,000	0.554
Fault_639	30,305	638,802	0.007
Emilia_923	43,270	923,136	0.005
bone010	44,319	986,703	0.005

Table 3.4: Pecifications Of The CPU System, The GPU Device, And The FPGA Board Used In The Evaluation.

Hardware Platform	Specification
Intel Xeon E5-2637 v3 CPU	15M Cache, 3.50 GHz Clock Frequency, 4 Cores, 68 GB/s Memory Bandwidth
NVIDIA Tesla V100 GPU	16 GB HBM2, 640 Tensor Cores, 5120 CUDA Cores, 1245-1380 MHz Clock Frequency, 900 GB/s Memory Bandwidth
Intel Stratix 10 GX FPGA Board	5760 DSPs, 229 Mb M20K, 15 Mb MLAB, 15 GB/s Memory Bandwidth

Table 3.5: Resource Utilization On Intel Stratix 10 GX With $VL = 128$, $N = 4$, And $M = 2$.

Resource	ALUTs	FFs	RAMs	DSPs
Utilization	315,858	634,846	6,844	901
	17%	17%	58%	16 %

Table 3.6: Runtime (*Second*) Comparison Between The FPGA Implementation Of FSCHOL And The CPU And GPU Implementations Of CHOLMOD.

Matrix	CHOLMOD		FSCHOL
	CPU	GPU[#]	
nd3k	0.31	0.99	0.05
Trefethen_20000b	3.56	4.82	1.30
smt	0.26	1.84	0.31
thread	0.47	2.04	0.34
pdb1HYS	0.36	2.40	0.42
nd24k	10.49	12.02	0.36
Fault_639	33.00	49.26	10.89
Emilia_923	55.57	62.90	15.57
bone010	23.28	58.48	15.88

[#] ECC on and base clock speed.

Table 3.7: Runtime (*Second*) Comparison Between The FPGA Implementation Of FSCHOL And The Reference Work In [82].

Matrix	[82]	[82]#	FSCHOL (Speedup)
nd3k	1.96	0.98	0.05 (19.6×)
Trefethen_20000b	3.94	1.97	1.30 (1.5×)
nd24k	10.12	5.06	0.36 (14.1×)

Technology scaling to 14nm: $delay = 1/S$ where $S = L/14nm$.

Table 3.8: Energy Consumption (J) Comparison Between The FPGA Implementation Of FSCHOL And The CPU And GPU Implementations Of CHOLMOD.

Matrix	CHOLMOD		FSCHOL
	CPU	GPU#	
nd3k	13	39	1
Trefethen_20000b	146	244	29
smt	11	72	7
thread	19	80	8
pdb1HYS	15	95	9
nd24k	430	723	8
Fault_639	1353	3523	240
Emilia_923	2279	8665	342
bone010	954	5108	349

ECC on and base clock speed.

Row-major Order

A		C	
B			D
	F	G	
E		H	

Read Order		A	C	B	D	F	G	E	H
CSR Format	VAL	A	C	B	D	F	G	E	H
	COL_IND	0	2	0	3	1	2	0	2
	ROW_PTR	0	2	4	6	8			

Vector-major Order

A		C	
B			D
	F	G	
E		H	

Read Order		A	B	C	D	E	F	G	H
CSV Format	VAL	A	B	C	D	E	F	G	H
	COL_IND	0	0	2	3	0	1	2	2
	ROW_IND	0	1	0	1	3	2	2	3

Figure 4.1: The Sparse Matrix Representation Using The CSR And CSV Formats. Each CSV Vector Is In The Length Of The Number Of Computing Units (2 Assumed In This Example). Transparent Lines Show The Storage Order For Each Format.

Chapter 4

ACCELERATING GENERAL SPARSE MATRIX-MATRIX MULTIPLICATION

This chapter explains the proposed approach for an OpenCL-based HPC framework for accelerating general sparse matrix-matrix multiplication on FPGAs and a new compressed sparse vector (CSV) format for representing sparse matrices.

4.1 Compressed Sparse Vector (CSV) Format

As GEMM and SpGEMM are primarily computed with spatial parallelism in CPU, GPU, and most custom hardware accelerators, each row of input matrix A is assigned to a different computation unit. Using Gustavson's method, I read a vector of nonzeros with the same column index from matrix A and read the corresponding

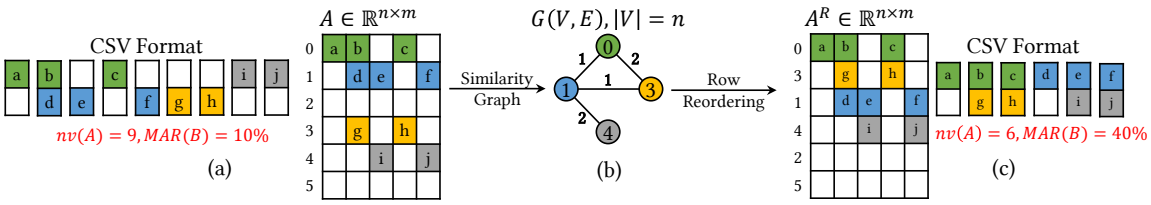


Figure 4.2: (A) The Sparse Matrix A Is Represented Using The CSV Format. Each Sparse Vector Is In The Length Of The Number Of Computing Units (2 In This Example). The Order Of Sparse Vectors Shows The Storage Layout. (B) The Similarity Graph For Rows Of Matrix A Is Constructed Based On The Sparsity Pattern Along Columns. (C) The Reordered Matrix Using The Row-Reordering Algorithm. Note The Empty Rows Are Moved To The End Of The Matrix. The CSV Representation Indicates Reduced Memory Access.

row of matrix B for reuse among computation units (see Fig. 4.5a). Reading the nonzero elements from multiple rows with the same column index A based on the CSR format in a vector fashion (see Fig. 4.1) leads to a non-continuous memory access pattern. The same problem exists for the column-wise Gustavson's method when reading multiple columns of the input matrix using the CSC format [16]. A non-continuous memory access pattern results in a large performance penalty. Additionally, using the CSC and CSR formats for the row- and column-wise Gustavson's algorithms, respectively, requires having a very large lookup table (in the size of the input matrix dimension) to keep track of the locations of the last nonzero entry read from the input matrix, which is too costly to implement for large matrices. Different formats have been used in literature for data compression and improving resource utilization [35], such as Doubly Compressed Sparse Row (DCSR) [17], compressed sparse blocks (CSB) [11], SELL-P [13], and ELLPACK-R [85]. However, they suffer from the same issue mentioned above for the standard CSR format.

To address these issues, I propose the CSV format tailored to Gustavson's algorithm for storing input matrix A. The CSV format uses three attributes for representing each nonzero element: the value VAL, the row index ROW_INDEX, and the column index COL_INDEX. Therefore, the location of the last nonzero entry read from the input matrix is always clearly indicated with a pair of row and column indices without needing a look-up table for storage. In addition, the nonzero elements in the CSV format are stored in a new vector-major order (see Fig. 4.1), which is a key difference from Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) formats that adopt a row- or column-major order. Storing nonzero elements in a vector-major order (that matches the number of computational units) is the key to assuring a continuous memory access pattern, which improves both the memory bandwidth utilization and computational performance.

4.2 Framework Design

The FSpGEMM framework consists of an FPGA kernel written in OpenCL and a host program running on an FPGA accelerator and a host CPU, respectively, as shown in Fig. 4.3. The kernel code implemented on the FPGA accelerator performs the SpGEMM computation based on Gustavson's algorithm. On the host side, I provide the utility functions for preprocessing and storing raw matrix files in the CSV format and the OpenCL API for provisioning tasks running on the FPGA accelerator. FSpGEMM utilizes a two-fold method for reducing memory access to the global memory: 1) a new buffering scheme enabled by the proposed CSV format and 2) a row reordering technique as a preprocessing step.

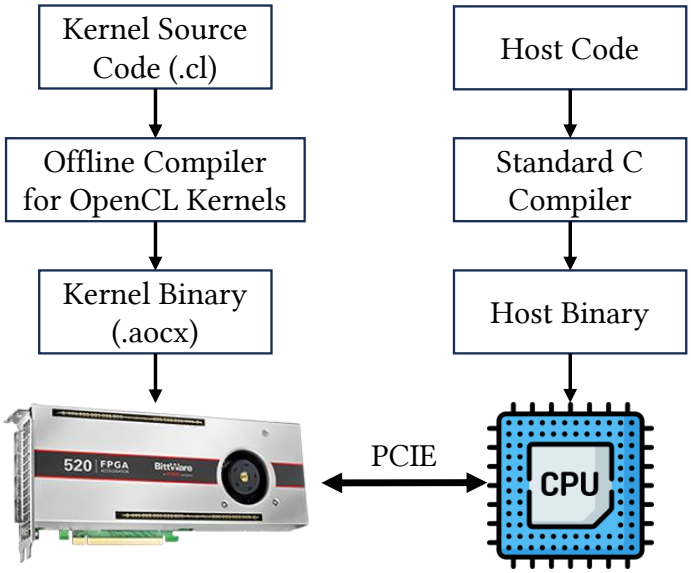


Figure 4.3: The Programming Flow Of OpenCL For FPGA.

4.2.1 Data Buffering Scheme

I consider the row-wise Gustavson's method as the baseline algorithm. To improve the data reuse of matrix B and reduce the amount of memory access, I propose to process multiple rows of matrix A in parallel using multiple computing units (*i.e.*, one row per computing unit) while sharing a row of matrix B among all computing units (CUs). To this end, a CSV vector of nonzero values of matrix A is read. Based on the column index of the vector, the corresponding row from matrix B is then read and buffered in the on-chip memory. As a result, the access to the buffered row of matrix B is reused for multiple rows of matrix A.

The example in Fig. 4.5a shows how the proposed data buffering scheme avoids one round of memory access to the fifth (since the circled CSV vector is in the fifth column of matrix A) row of matrix B by reusing this buffered row for all computing units.

I define $nv(A)$ and $nnz(A)$ as the number of sparse vectors and nonzero elements in

matrix A, respectively (*e.g.*, $nnz(A) = 8$ and $nv(A) = 6$ in Fig. 4.1). In Gustavson's algorithm, each nonzero element of matrix A is multiplied by a row of matrix B. Since nonzero elements in a row of matrix B are stored consecutively in the global memory, a row of matrix B is read in the burst mode in a single memory access. Therefore, I define the total number of memory access to matrix B as the total number of nonzero elements in matrix A (*i.e.*, $nnz(A)$). Using our proposed data buffering scheme, for each sparse vector in matrix A, one row (corresponding to one memory access) of matrix B is read and reused. Therefore, there are $nv(A)/nnz(A)$ memory accesses to matrix B using our proposed data buffering scheme. Consequently, I define the memory access reduction percentage for reading matrix B from the global memory as

$$MAR(B) = (1 - \frac{nv(A)}{nnz(A)}) \times 100, \tag{4.1}$$

The proposed data buffering scheme can be implemented on all customizable hardware (*e.g.*, FPGA and ASIC devices). Also, it can potentially be applied to any fixed-architecture device with a compatible memory hierarchy.

4.2.2 Row Reordering Technique

Despite Gustavson being a more efficient dataflow compared to inner- and outer-product methods, it can suffer from high data traffic. To address this issue, the proposed CSV format enables the reuse of rows of B when multiple nonzeros in A share the same column index within adjacent rows of A. However, due to the lack of structure in some nearby rows of A, which contain mostly disjoint sets of column indices, the reuse of rows of B might be decreased. I propose to take advantage of a tailored preprocessing algorithm to further increase data reuse. I developed a row reordering technique that aims to position similar rows in terms of the sparsity

pattern next to each other.

Problem Formulation

I define the similarity score between sparse rows i and j (*i.e.*, $A[i]$ and $A[j]$) as:

$$score(S_i, S_j) = |S_i \cap S_j|, \tag{4.2}$$

where S_i and S_j are the sets of column indices representing the sparse rows. I aim to find an optimal row ordering for the $n \times m$ matrix $A = (a_{ij})$ to maximize the total similarity scores between adjacent pairs of rows, *i.e.*,

$$\text{maximize } \sum_{i=2}^{n} score(S_{i-1}^R, S_i^R), \tag{4.3}$$

where S_i^R is the set of column indices representing the sparse rows in the reordered matrix A^R.

Challenges

Computing the similarity between every pair of rows in a matrix becomes impractical when dealing with a large number of rows and can surpass the memory capacity of a regular computer and demand an extensive amount of computing cycles. Moreover, the problem is NP-Complete [69], further emphasizing the need to rely on heuristics for a feasible solution.

Solution

To address the mentioned memory and computational issues, I first construct a sparse weighted graph $G(V, E)$ where each vertex represents a row of the matrix and the weight between two vertex is equal to the similarity score between corresponding rows. Then, I perform a heuristic to find the longest path in the graph that visits each vertex

exactly once. Unlike [90, 42, 69], which considered all rows of the input matrix in their approach, our approach only considers the non-empty rows. By defining r as the number of non-empty rows ($r \leq n$), I tackle the problem with a reduced set of variables, leading to a more efficient outcome. It is crucial to observe that the majority of the weights are zero because there is no similarity between most rows due to the sparsity of the original matrix. The graph's sparseness allows for an efficient and practical solution. I employ a vertex insertion heuristic. The weight of an edge connecting two vertices is determined by the number of columns where both corresponding rows have a non-zero value, *i.e.*,

$$w(u, v) = |\{i \in [1, m] | A[u][i] \neq 0 \text{ and } A[v][i] \neq 0\}| . \tag{4.4}$$

Fig. 4.2b illustrates an example of graph construction. The matrix has 6 rows, while the graph has only 4 vertices representing non-empty rows of the matrix. The weight of edge (0, 3) is equal to 2 because columns 1 and 3 share nonzero elements among rows 0 and 3. The heuristic starts from an arbitrary vertex and repeatedly selects the farthest (*i.e.*, largest weight) unvisited vertex as the next destination until all vertices have been visited. I mark the visited nodes along the path to avoid cycles. The procedure is described in Algorithm 3, where the output is the reordered matrix (A^R) and function $maxPath$ is defined as:

$$\begin{aligned} maxPath(u, visited) = & \arg\max_{v \in V} w(u, v) \\ & \text{s.t. } (u, v) \in E \text{ and } visited[v] \neq 1. \end{aligned} \tag{4.5}$$

Fig. 5.2 depicts Algorithm 3 executing on the example graph illustrated in Fig. 4.2b. Fig. 4.2(c) displays the memory layout after applying the reordering algorithm. Rows with similar sparsity patterns (*i.e.*, rows 0 and 3, or rows 1 and 4) are placed together. By comparing the number and density of sparse vectors in two cases, the number of memory accesses is decreased from 9 to 6 . The time complexity

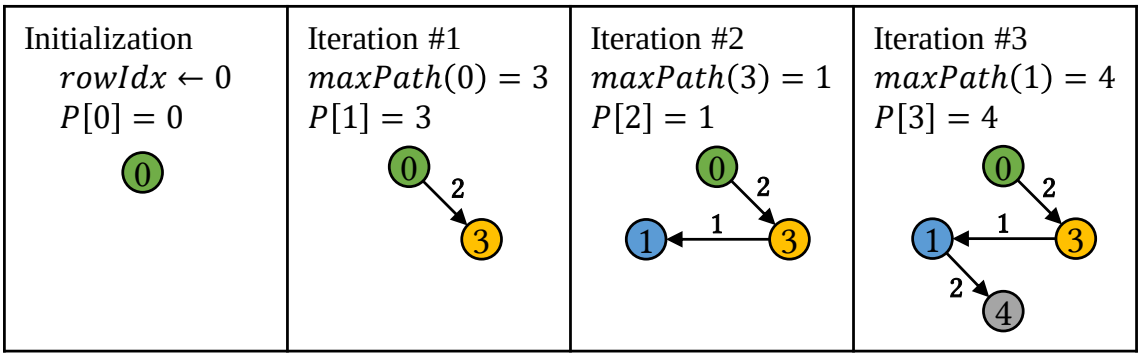
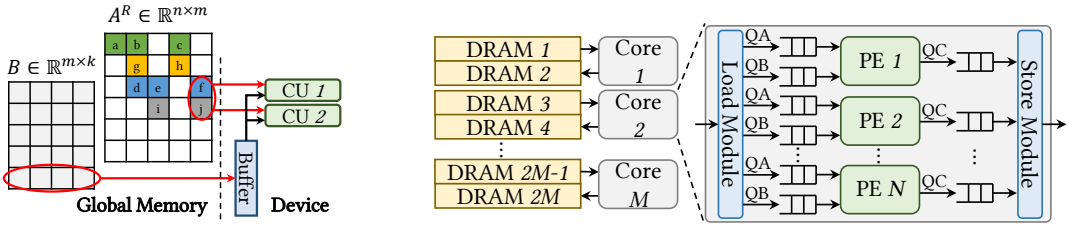


Figure 4.4: Running Example Of Algorithm 3 For The Matrix In Fig. 4.2a. The Path Generated At Each Step Denotes The Order Of Rows In The Reordered Matrix A^R.

of Algorithm 3 is $\mathcal{O}(r \times q^2)$, where q is the average number of nonzeros per row and column. Therefore, the runtime complexity increases linearly and quadratically with the matrix dimension and density, respectively. The time complexities of the reordering algorithms in [90] and [42] are $\mathcal{O}(n \times \log(n) \times q^2)$ and $\mathcal{O}(n \times \log(n))$, respectively. Since $n \gg q$ and $n > r$ in sparse matrices, our algorithm significantly outperforms them. Fig. 4.2a and Fig. 4.2c show $MAR(B)$ calculated from Equation 4.1 before and after applying the row reordering technique, respectively.

Fig. 5.3 shows a high-level block diagram of the hardware architecture of the FPGA design, including multiple processing cores. Each core is responsible for processing a partition of matrix A represented in the CSV format and computes a partition of the output matrix. Each core includes three types of modules: Processing Element (PE), Load, and Store. The PEs are responsible for computing the nonzero elements of the output matrix based on Gustavson's algorithm. The Load and Store modules inside each core connect to two memory channels and read and write input and output data from and to the memory, as well as feed and receive data to and from the PEs via FIFO channels, respectively. Separating the Load and Store modules from PEs and connecting them using FIFO channels facilitates the data flow



(a) (b)

Figure 4.5: The Corresponding Elements In (A) And (B) Are Color-Coded. (A) An Example Of The Proposed Data Buffering Scheme. Circled Elements Show The CSV Vector And The Corresponding Row Of Matrix B Being Processed. Each CSV Vector Is In The Length Of The Number Of Computing Units (2 CUs Assumed In This Example). (B) The High-Level Block Diagram Of The Scalable Hardware Architecture. Cores Are Compatible With High-Bandwidth Memory (HBM) Or Traditional DDR-Based Memory Banks.

control for balancing the memory bandwidth and the data processing throughput by PEs. The depth of the FIFO channels is optimized by the offline compiler such that the Load and Store modules are able to continuously read and write data from and to the memory, respectively, while pushing data throughout the module pipeline.

Each PE is responsible for calculating one row of the output matrix at a time. Multiple PEs process multiple rows independently in parallel. Each PE receives a nonzero scalar value of matrix A along with additional scheduling information through a QA channel. Also, each PE gets nonzero values from the corresponding row of matrix B from a QB channel. The output results are streamed to the Store module via a QC channel.

The FPGA kernel in FSpGEMM includes two types of OpenCL kernels: 1) *autorun* kernel, which starts executing automatically and does not need to be managed by a host program, and 2) host-driven kernel. As the offline compiler provided by Intel

FPGA SDK for OpenCL does not need to implement any glue logic for communicating with the host program, using *autorun* kernels reduces logic utilization and allows for the mapping of more computation resources for improved performance[40]. The PEs are *autorun* kernels, and the Load and Store modules are host-driven kernels since they access the memory that must be allocated first by a host program. Overall, the FPGA kernel is parameterized with two architectural parameters: N (number of PEs) and M (number of cores), which allows users to balance the trade-off between computational parallelism and FPGA resource usage.

Load and Store Modules

In FSpGEMM, matrix A and matrix B are represented by the CSV and the CSR format, respectively. The Load module iterates over the total number of nonzero elements in matrix A. In each iteration, the Load module first sends a data structure ($aType$ in Table 4.1) containing the value of a nonzero element of matrix A along with the other attribute data using a QA channel. Specifically, *eor* indicates whether the received nonzero element is the last one in the current row of matrix A. If so, $eor = True$ signals the PEs to finish the computations and reset to their initial states.

The Load module takes advantage of the CSV format and identifies the nonzero elements of matrix A in the same sparse vector by comparing the column indices of two consecutive nonzero elements. If the column index of the next nonzero element is different from the column index of the current nonzero element (*i.e.*, indicating it is the last element of the sparse vector), the Load module buffers the nonzero element of the corresponding row of matrix B and sends it to the PEs. The Load module also calculates the number of nonzero elements in a row of matrix B ($bType.size$) that the PEs need to expect from the Load module. Additionally, the row index of the nonzero element from matrix A ($aType.rowIdx$) is sent to the PEs to determine the

row index of the output matrix.

Since the Load module reads a row of matrix B at a time, the data are read from memory using the CSR format to enable contiguous and regular access. Each array of matrix B is represented by the data structure $bType$ (see Table 4.1) in which its array members $bType.val$ and $bType.colIdx$ contain the values and the column indices of the nonzero elements in a row of matrix B.

The Store module reads the computation results of the output matrix from the PEs, including the nonzero value and its row and column indices. Since the valid output values of different PEs are not necessarily produced at the same time, the FIFO channels QC are used to facilitate the data flow control and assure data integrity.

Processing Elements (PEs)

Fig. 5.3 shows the high-level block diagram of a PE. Each PE is responsible for two major operations for producing each row of the output matrix: 1) the element-wise multiplication between a nonzero value of matrix A and a row of matrix B resulting in a sparse partial row and 2) the addition of partial rows. These functionalities are implemented by two main units: a merge unit for sorting and accumulating (SA) partial rows and a memory unit implementing a double buffering scheme for efficient storage of intermediate results.

Algorithm 4 describes the conventional (*i.e.*, unoptimized) software-like implementation of the merge operation. Inputs are the scalar nonzero element from matrix A (a), the array of nonzero elements from matrix B ($bArr$), the array of the current partial row from the selected buffer (buf), and the number of nonzero elements in the current partial row ($bufCntr$). In Loop 1, the unoptimized unit performs a scalar-array multiplication and stores the new partial products in array $cArr$. Then, in Loop 2, the unoptimized unit compares the column indices of two sparse rows stored

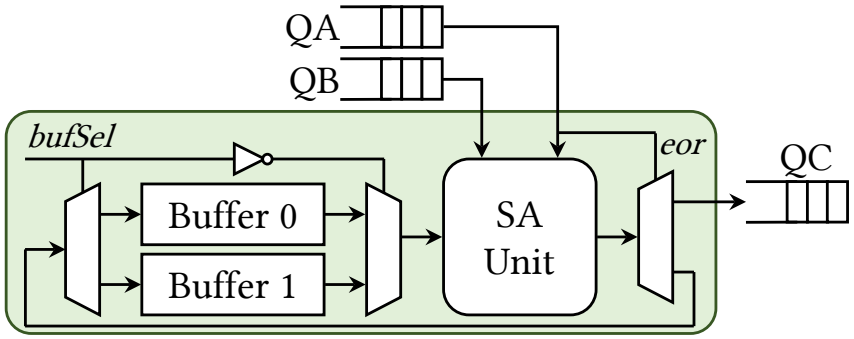


Figure 4.6: The High-Level Block Diagram Of A PE.

in $bArr.colIdx$ and $buf.colIdx$. Since the two arrays are already sorted, the smaller column index is the smallest index in both arrays. The pointer for the array with the smaller index will be increased by one. In the case that the indices from two arrays are equal, the accumulation occurs where the current values from $cArr.val$ and $buf.val$ are added. In Loops 3 and 4, the unit checks if there are any remaining elements in the selected buffer and the new partial products array, respectively. Finally, Loop 5 outputs the new partial row.

In Algorithm 4, there are five loops for the conventional implementation of the merge operation. Loop control structures have a significant overhead, so I propose to combine the bodies of five loops into one fused loop to reduce the amount of control structures needed to one, which saves the FPGA area while maintaining unit throughput and reduces latency. Algorithm 5 describes the optimized implementation of the proposed sort and accumulate (SA) unit.

I utilize a double-buffering technique to facilitate the fully pipelined addition of $cArr$ with buf. At any time, while buf is being read from one of the on-chip buffers and added with $cArr$ in the SA unit, the results of the addition operation are stored in the other buffer. The data structure $cType$ is used to store the temporary computed output elements.

Listing 4.1: Snapshot of the primary sections of the host code, including OpenCL APIs and preprocessing functions.

```
context = clCreateContext(...);
binary_file = getBoardBinaryFile(...);
clBuildProgram(...);
queue = clCreateCommandQueue(...);
load_kernel = clCreateKernel(...);
store_kernel = clCreateKernel(...);
matrix = readMatrix(...);
reordered_matrix = reorder(matrix);
mat_buf = clCreateBuffer(...);
clEnqueueWriteBuffer(..., mat_buf, ...);
clSetKernelArg(...)
clEnqueueTask(queue, load_kernel, ...);
clEnqueueTask(queue, store_kernel, ...);
clFinish;
clEnqueueReadBuffer(...);
```

At the end of Algorithm 5, the double buffer selector signal ($bufSel$) is toggled between 0 and 1. Whenever finished processing on the vector of matrix B, the PE decides whether to store $cType$ (see Table 4.1) in the other buffer or send the final value to a QC channel based on the value of $aType.eor$ (see Fig. 4.6).

Performance Optimization for Determining Architectural Parameters

I provide a guideline to derive the optimal architectural parameters (the number of cores M and N PEs per core) for minimizing the runtime subject to the available resources on a given FPGA device. I formulate the runtime (R) as $R = \frac{N_{Ops}}{F \times P \times U}$, where N_{Ops}, F, P, and U are the number of floating-point operations to produce an output matrix, the clock frequency, the computational parallelism, and the spatial-temporal utilization factor, respectively. U is a statistical metric that measures, on average, how busy the available computation resources are in a computing device (in both space and time) for performing useful operations of a given algorithm. Thus, U is defined as the ratio of effective computational parallelism per clock cycle on average and ranges from 0 to 1 and can be calculated as $SU \times TU$, where SU and TU denote spatial and temporal utilization, respectively. N_{Ops} is an algorithmic parameter and depends on the dimensions and the sparsity pattern of input matrices. Computational parallelism P is defined as the total available number of floating-point operations per clock cycle by the FPGA device. Therefore, the total number of utilized DSP units calculated as $N \times M$ is equal to $P \times SU$. Finally, R can also be expressed as

$$R = \frac{N_{Ops}}{F \times P \times SU \times TU} = \frac{N_{Ops}}{F \times N \times M \times TU} = \frac{\alpha}{N \times M}, \tag{4.6}$$

where N_{Ops}, F, and TU are lumped as α. I define the constrained optimization problem as

$$\begin{aligned} &\text{minimize} \quad R(N, M) = \frac{\alpha}{N \times M} \\ &\text{subject to } C(N, M) \leq c, \end{aligned} \tag{4.7}$$

where function $C(N, M) = \beta \times N \times M$ and constant c are the total required and available logic resources, respectively, where β is a linear fitting parameter that captures the proportionality of the actual logic resource usage over computational parallelism

that can be derived based on the logic resource usage reported by the offline compiler given a target FPGA device.

Based on $R(N, M) = \frac{\alpha}{N \times M}$, the runtime R is minimized when N and M are maximized. Thus, the constrained optimization problem defined in Equation 4.7 has an analytical solution that can be derived as follows.

1. Set N equal to the maximum sparse vector length where there is a linear memory access reduction based on the MAR analysis introduced in section 4.2.1.

2. Choose $M = 1$ and run the offline compiler to derive the value of $\beta \times N$ from $C(N, 1)$.

3. Derive M from $M = \lfloor \frac{c}{\beta \times N} \rfloor$.

While the provided guideline is introduced to extract reasonable values for N and M, one needs to note that scaling the design and architectural parameters at a certain point might cause implementation failure due to routing congestion or significantly reduce the clock frequency.

4.2.3 Host Program

The host program running on the CPU includes two major parts: preprocessing utilities and OpenCL API functions. The utility functions read matrix A to reorder and store it in the CSV format prior to the computation by the FPGA kernel. Note that the preprocessing step only needs to be performed once. I have utilized OpenCL API functions to create memory buffers, enqueue kernels, and read the results back from the FPGA device, as summarized in Listing 4.1.

4.3 Evaluation

4.3.1 Experiment Setup

I evaluate the performance of FSpGEMM in terms of runtime (s) and the number of execution clock cycles per SpGEMM computation. To evaluate the FSpGEMM framework, I select two benchmarks of sparse matrices from the publicly-available SuiteSparse Matrix Collection [25] (formerly known as the University of Florida Sparse Matrix Collection), a collection of sparse matrices in real applications. One benchmark includes the commonly-used matrices for evaluating SpGEMM research in the literature [90], and the extended benchmark is used in the SOTA FPGA work [49]. The specifications of the matrices are summarized in Table 5.1, including the number of rows, the average number of nonzeros per row, and the density calculated as $\frac{No.\ of\ Nonzeros}{Matrix\ Size}$. Our design adopts the single-precision floating-point (32-bit) data format. I develop and implement FSpGEMM on the Bittware 520N-MX FPGA board using Intel FPGA SDK for OpenCL with Quartus Prime Pro 20.1. I compare FSpGEMM with SOTA FPGA and GPU implementations [49, 50, 63]. I measure the runtime of the GPU implementation of TileSpGEMM [63] on an NVIDIA GeForce RTX 3090 GPU as the same device is used in [63].

4.3.2 MAR Evaluation

As mentioned in Section 4.1, a naive implementation of Gustavson's algorithm suffers from poor data reuse when reading a random row of matrix B from global memory. To address this issue, while processing multiple rows of matrix A in parallel (*i.e.*, each row in one PE), a row of matrix B is reused among all PEs. I calculate the memory access reduction percentage (MAR %) using Equation 4.1 for a set of matrices summarized in Table 5.1. Fig. 4.7 shows the MAR percentage that can be

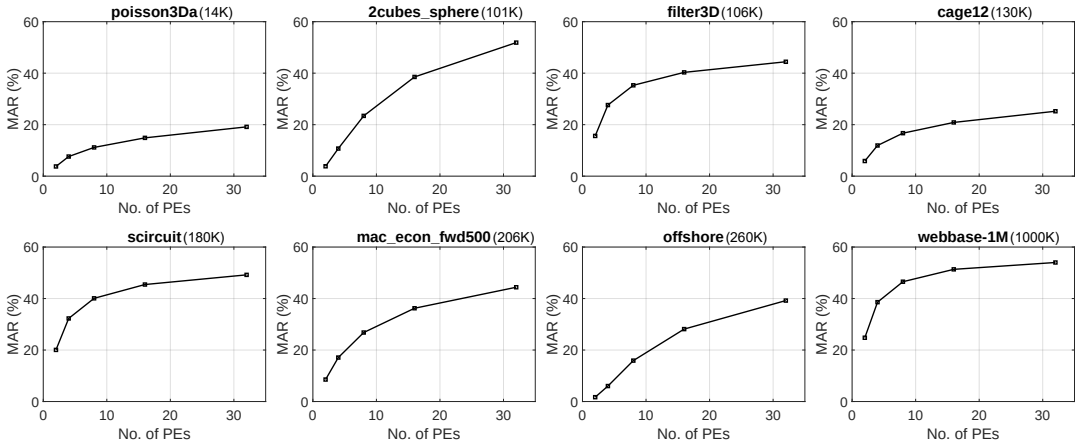


Figure 4.7: Memory Access Reduction (MAR) Percentage With Respect To Different Input Matrices And The Number Of PEs. The Difference In Sparsity Patterns Of The Input Matrices Results In The Difference In MAR Percentage. The Dimensions Of Matrices Are Noted In Parenthesis.

achieved by the proposed data buffering scheme with respect to different matrices and numbers of PEs. The results show that the amount of MAR that can be archived by the proposed data buffering scheme monotonically improves as the number of PEs increases. 1.7%-24.8%, 6.0%-38.6%, 15.9%-46.5%, 28.1%-51.3%, and 39.2%-54.0% MAR can be achieved at the PE number of 2, 4, 8, 16, and 32, respectively, across the selected sparse matrices. The improvement with increasing the number of PEs is because a row of matrix B is buffered and shared for processing more rows of matrix A. Thus, the amount of access to the row of matrix B is decreased.

4.3.3 Experimental Results

Using the proposed guideline for determining architectural parameters provided in Section 22, I derive parameters N and M as follows:

- Derive maximum *No. of PEs* with linear MAR slope from Fig. 4.7 $\rightarrow N = 16$.

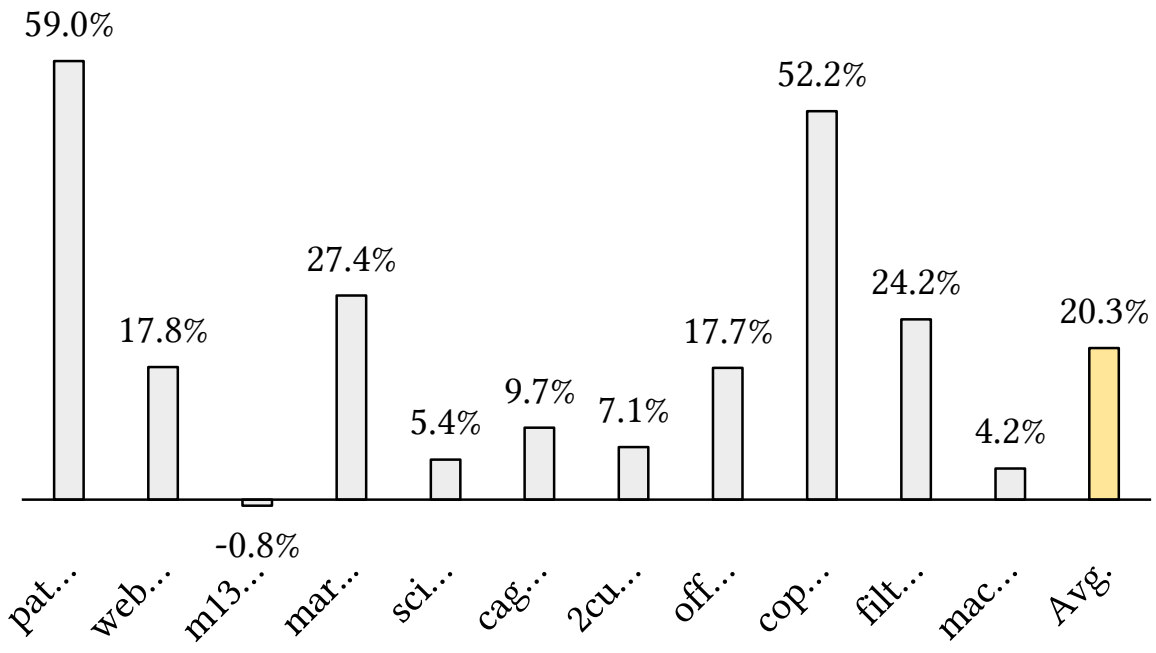
- Since RAMs have the highest utilization among all resources, $C(N, M)$ and $c = 5,279$ (based on the FPGA specification) are the total required and available RAMs for *Kernel System* (as opposed to *Static Partition*), respectively.
- Choose $N = 16$ and $M = 1 \rightarrow C(16, 1) = \beta \times N = 561$ based on the utilization report.
- $M = \lfloor \frac{c}{\beta \times N} \rfloor = \lfloor \frac{5,279}{561} \rfloor = 9$.
- $N = 16$ and $M = 6$ are the starting points of the design space exploration. The offline compiler fails at $M = 9$. Therefore, I decrease M until I get the highest performance at $M = 6$. I refer to this implementation as *FSpGEMM-m6* throughout this section. *FSpGEMM-m6* runs at 269 MHz.

Finally, the ALUTs, FFs, RAMs, MLABs, and DSPs utilization (usage) for *FSpGEMM-m6* are 46% (646,160), 47% (1,317,004), 72% (4,946), 4% (3,060), and 25% (978), respectively.

Row Reordering Technique Evaluation

The row reordering technique jointly increases the reuse of matrix B and the hardware resource utilization, resulting in performance improvement. Fig. 4.8 illustrates the performance improvement achieved by the preprocessing step using the row reordering technique for the commonly-used benchmark running on *FSpGEMM-m6*. The preprocessing technique improves the overall performance on average by 20.3%. According to Fig. 4.8, the preprocessing technique works better for matrices with large dimensions, which is common in real-world applications. Typically, preprocessing a matrix takes more time than performing SpGEMM on the same matrix. This means that preprocessing is only worthwhile if matrix A is used frequently, which is true for matrices found in the SuiteSparse Matrix Collection.

Figure 4.8: Performance Speedup Achieved By The Row Reordering Technique For The Commonly-Used Benchmark On *FSpGEMM-M6*.



Performance Comparison with SOTA FPGA Implementations

The SOTA FPGA implementation in [49] (*GSCSp-om*) uses four memory channels. For a fair comparison, I utilize the same number of HBM channels (*i.e.*, $M = 2$ with two channels per core and $N = 16$), and I refer to this implementation as *FSpGEMM-m2*. Table 4.3 details the comparison of FSpGEMM in terms of the number of execution clock cycles with the SOTA FPGA implementation for performing SpGEMM computation. The number of execution cycles for *FSpGEMM-m2* is calculated from $R \times F$, where R and F are the runtime and clock frequency, respectively.

Table 4.3 shows the runtime speedup achieved by FSpGEMM over SOTA FPGA works [49, 50] with respect to the number of execution clock cycles for different sparse matrices. The speedup over the highest-performance SOTA implementation [49] is up

Figure 4.9: Normalized Performance Comparison Of *FSpGEMM-M6* Compared To The SOTA GPU Implementation For The Complete Benchmark In Terms Of Effective Runtime.

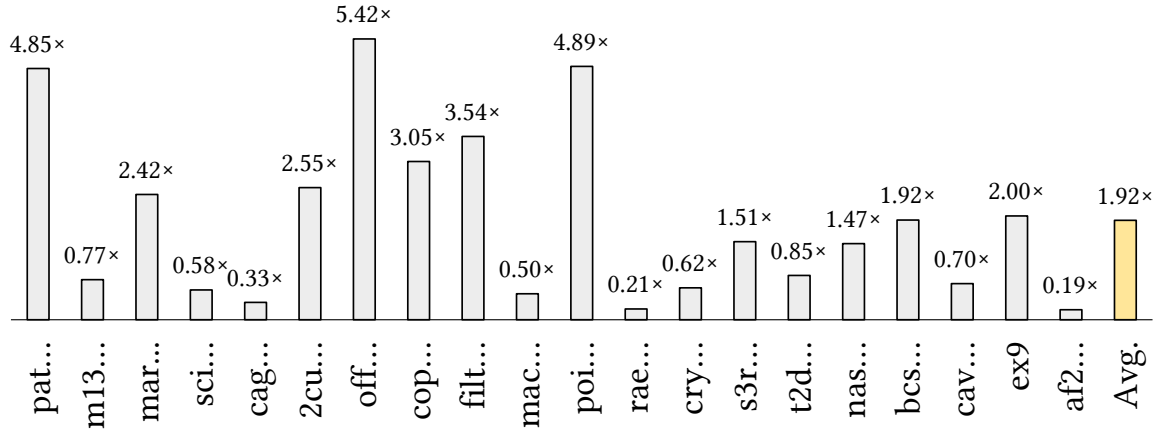
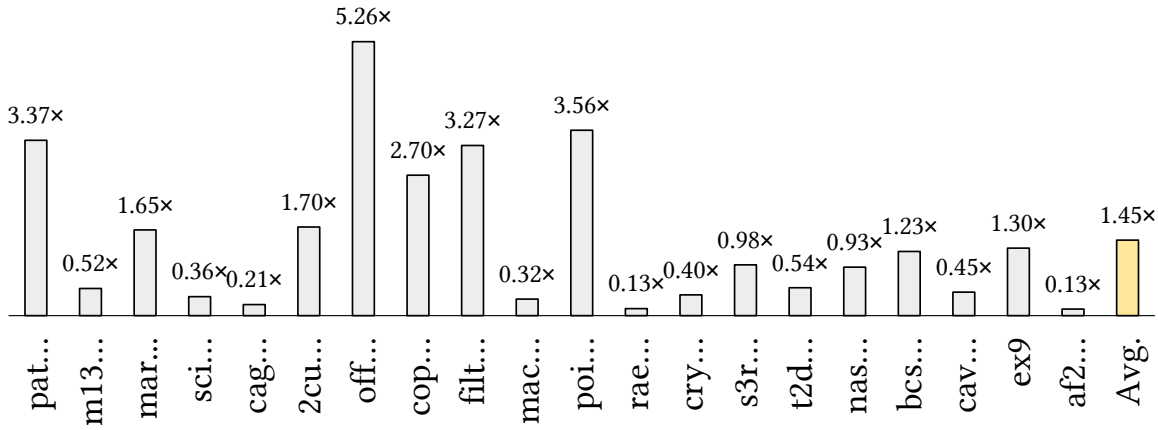


Figure 4.10: Energy Consumption Reduction Achieved Using *FSpGEMM-M6* Compared To The SOTA GPU Implementation For The Complete Benchmark.



to 3.49× for matrix *nasa2910*. On average, FSpGEMM achieves 2.23× higher performance than the SOTA FPGA counterpart. It should be noted that such speedup ratios are achieved without considering the actual runtime. Note that *FSpGEMM-*

m2 runs at 273 MHZ while the clock frequency reported at [49] is below 100 MHz, leading to a much lower performance. The speedup mainly stems from the memory access reduction due to the customized data buffer scheme tailored to Gustavson's algorithm, the improved memory bandwidth utilization as a result of the architectural co-design with the proposed CSV format and row reordering preprocessing, and the throughput-optimized hardware architecture tailored to the Gustavson's algorithm, all of which improves the overall utilization of the computation resources on the FPGA at run time.

Note that similar to related work [90, 42], the row reordering algorithm is a preprocessing step on the matrices. It is not part of the SpGEMM computation on the FPGA device and should not be considered in the SpGEMM runtime evaluation. The performance evaluation of the proposed row reordering algorithm and its comparison to related work in terms of the asymptotic runtime complexity is provided in Section 4.2.2.

Performance Comparison with the SOTA GPU Implementation

Since the performance of SpGEMM is memory-bounded, a direct comparison of the raw performance of the FPGA implementation on Bittware 520N-MX with 410 GB/s memory bandwidth with a GPU implementation on an NVIDIA GeForce RTX 3090 GPU with 936 GB/s memory bandwidth can be misleading from an architectural point of view. For a fair comparison, I compare the performance in the metrics of effective runtime. Effective runtime is defined as the runtime normalized to the memory bandwidth, which measures the effective performance of different computing devices with respect to the same memory bandwidth (the GPU bandwidth in this case).

Fig. 4.9 shows the effective runtime speedup achieved by FSpGEMM over the

GPU implementation with respect to different sparse matrices. On average, FSpGEMM achieves 1.92× higher effective runtime than the GPU counterpart. It should be noted that such speedup ratios are achieved at the condition that the FPGA implementation runs at a 5-15× lower clock frequency than the CPU and GPU implementations. The speedup mainly stems from the off-chip memory access reduction due to the customized data buffer scheme tailored to Gustavson's algorithm, the improved memory bandwidth utilization as a result of the architectural co-design with the proposed CSV format, the temporal/pipeline parallelism offered by FPGAs for resolving the strong data dependency in the Gustavson's algorithm, and the throughput-optimized hardware architecture tailored to the Gustavson's algorithm, all of which improves the overall spatial and temporal utilization of the computation resources on the FPGA at run time.

Energy Efficiency Comparison with the SOTA GPU Implementation

I measure the average power consumption of the FPGA board using the monitor tool in the Board Management Controller (BMC). BMC measures the supply voltage and the drawn current of the entire FPGA board by reading values from onboard sensors. For the power measurement of the GPU, I utilize the POWER query option of the NVIDIA System Management Interface (*nvidia-smi*) tool.

The energy consumption (J) per SpGEMM computation of different implementations is calculated as the product of runtime (s) and average power consumption (W). Fig. 4.10 shows the corresponding energy consumption reduction achieved by the FPGA implementation. On average, FSpGEMM achieves 1.45× higher energy efficiency than its GPU counterpart.

4.4 Conclusion

In this paper, I present FSpGEMM, an OpenCL-based framework for FPGA acceleration of SpGEMM. FSpGEMM includes a deeply pipelined and scalable FPGA kernel that accelerates the SpGEMM algorithm and a set of utility functions for the preprocessing step and storing the input matrices with the proposed CSV format.

I introduce a performance-optimized model to derive architectural parameters for the FPGA kernel subject to the available on-chip resources (DSPs and memory blocks) to map the design into the arbitrary FPGAs. The experiment results based on the Bittware 520N-MX FPGA board for accelerating SpGEMM of a set of sparse matrices from the SuiteSparse Matrix Collection show higher performance compared to SOTA implementation of SpGEMM on FPGA.

Algorithm 3: Similarity-Based Row Reordering Algorithm

Input: A, n, and r.

Output: A^R.

```
1  initialize A^R with n empty rows;
2  initialize G(V, E) with r vertices and no edges;
3  initialize hash table visited[][] for r vertices;
4  initialize permutation array P[] with size r;
5  for each non-empty row of A do
6      for u ∈ column indices of the row do
7          for v ∈ rows indices of column u and u ≠ v do
8              if (u, v) ∉ E then
9                  E ← E ∪ (u, v);
10                 w(u, v) ← 0;
11             end
12             w(u, v) ← w(u, v) + 1;
13         end
14     end
15 end
16 rowIdx ← index of an arbitrary non-empty row of A;
17 visited[rowIdx] ← 1;
18 P[0] ← rowIdx;
19 A^R[0] ← A[rowIdx];
20 for i ← 1 to r − 1 do
21     rowIdx ← maxPath(P[i − 1], visited);
22     visited[rowIdx] ← 1;
23     P[i] ← rowIdx;
24     A^R[i] ← A[rowIdx];
25 end
```

Table 4.1: Descriptions Of Data Structures $aType$, $bType$, And $cType$ For Data Transfer Via Channels QA, QB, And QC, Respectively.

Field	Description	Type
	aType	
val	nonzero value in A	float
rowIdx	row index of nonzero element in A	uint
eor	end of A's row signal	bool
	bType	
size	no. of nonzeros in row of B	uint
val	array of nonzero values in row of B	float[]
colIdx	array of the column indices in row of B	uint[]
	cType	
val	nonzero value of C	float
rowIdx	row index of nonzero element in C	uint
colIdx	column index of nonzero element in C	uint

Algorithm 4: Pseudo-Code For The Conventional Merge Operation.

Input: a, $bArr[]$, $buf[]$, and $bufCntr$.

Output: c and $cCntr$.

```
1  initialize bufPtr, bPtr, cCntr, pArr[], and cArr[] with zeros;
2  Loop 1: for i ← 0 to b.size do
3      pArr[i] = a.val × bArr.val[i];
4  end
5  Loop 2: while bufPtr < bufCntr and bPtr < b.size do
6      if buf.colIdx[bufPtr] < bArr.colIdx[bPtr] then
7          cArr[cCntr].val ← buf.val[bufPtr];
8          cArr[cCntr].colIdx ← buf.colIdx[bufPtr];
9          bufPtr ← bufPtr + 1;
10     else if buf.colIdx[bufPtr] > bArr.colIdx[bPtr] then
11         cArr[cCntr].val ← cArr[bPtr];
12         cArr[cCntr].colIdx ← bArr.colIdx[bPtr];
13         bPtr ← bPtr + 1;
14     else
           /* similar to lines 7-9 and 11-13                          */
15     end
16     cCntr ← cCntr + 1;
17 end
18 Loop 3: for i ← bufPtr to bufCntr do
       /* similar to lines 7-9                                        */
19 end
20 Loop 4: for i ← bPtr to b.size do
       /* similar to lines 11-13                                      */
21 end
22 Loop 5: for i ← 0 to cCntr do
23     c.val ← cArr[i];
24     c.rowIdx = a.rowIdx;
25     write c to its output register;
26 end
27 write cCntr to its output register;
```

Algorithm 5: Pesudo-Code For The Proposed Optimized SA Unit.

Input: a, $bArr[]$ and $bufCntr$.

Output: c, $cCntr$.

```
1  initialize bufPtr, bPtr, cCntr, and cTemp with zeros;
2  Fused Loop: while bufPtr < bufCntr or bPtr < b.size do
3      cTemp = a.val × bArr.val[bPtr];
4      if bufPtr < bufCntr and bPtr < b.size then
5          if buf.colIdx[bufPtr] < bArr.colIdx[bPtr] then
6              c.val ← buf.val[bufPtr];
7              c.colIdx ← buf.colIdx[bufPtr];
8              bufPtr ← bufPtr + 1;
9          else if buf.colIdx[bufPtr] > bArr.colIdx[bPtr] then
10             c.val ← cTemp;
11             c.colIdx ← bArr.colIdx[bPtr];
12             bPtr ← bPtr + 1;
13         else
               /* similar to lines 6-8 and 10-12                    */
14         end
15     else if bufPtr < bufCntr then
           /* similar to lines 6-8                                  */
16     else
           /* similar to lines 10-12                                */
17     end
18     c.rowIdx = a.rowIdx;
19     write c to its output register;
20     cCntr ← cCntr + 1;
21 end
22 write cCntr to its output register;
```

Table 4.2: The Specification Of The Commonly-Used Benchmark [90] And The Extended Benchmark Used In The SOTA FPGA Work [49, 50].

Matrix	No. of Rows	NNZ/Row	Density
Commonly-used Benchmark			
patents_main	240,547	2	0.0010%
webbase-1M	1,000,005	3	0.0003%
m133-b3	200,200	4	0.0020%
mario002	389,874	5	0.0014%
scircuit	170,998	6	0.0033%
cage12	130,228	16	0.0120%
2cubes_sphere	101,492	16	0.0160%
offshore	259,789	16	0.0063%
cop20k_A	121,192	22	0.0179%
filter3D	106,437	25	0.0239%
mac_econ_fwd500	206,500	6	0.0030%
Extended Benchmark			
poisson3Da	13,514	26	0.19%
raefsky1	3,242	91	2.80%
crystk01	4,875	65	1.33%
s3rmt3m3	5,357	39	0.72%
t2dah_a	11,445	15	0.13%
nasa2910	2,910	60	2.06%
bcsstk24	3,562	45	1.26%
cavity26	4,562	30	0.66%
ex9	3,363	30	0.88%
af23560	23,560	21	0.09%

Table 4.3: *FSpGEMM-M2* Speedup Over SOTA FPGA Implementations In Terms Of Execution Cycles.

Matrix	No. of Execution Cycles		
	FSpGEMM-m2	*[50]* (Speedup×)	*[49]* (Speedup×)
poisson3Da	16,645,660	19,878,983 (1.19)	16,638,816 (1.00)
raefsky1	21,288,307	27,458,639 (1.29)	25,658,832 (1.21)
crystk01	6,100,726	21,885,313 (3.59)	20,202,582 (3.31)
s3rmt3m3	2,831,289	8,628,745 (3.05)	8,411,161 (2.97)
t2dah_a	2,304,309	5,582,450 (2.42)	5,354,111 (2.32)
nasa2910	3,359,772	12,797,230 (3.81)	11,731,798 (3.49)
bcsstk24	2,241,033	7,588,259 (3.39)	7,354,144 (3.28)
cavity26	5,491,883	6,632,677 (1.21)	6,012,791 (1.09)
ex9	1,368,700	4,011,290 (2.93)	3,790,519 (2.77)
af23560	16,661,103	15,408,694 (0.92)	13,979,280 (0.84)
Average		2.38×	2.23×

Chapter 5

SPARSE TENSOR CORE

This chapter presents the ASIC design of the Sparse Tensor Core co-processor for accelerating sparse linear algebra.

5.1 Hardware Merge Sorter

Sorting is a key process in many applications, and researchers have worked on making sorting faster for CPUs [37, 20] and GPUs [59, 23]. Recently, there's also been a focus on speeding up sorting using FPGAs [45, 18, 80, 58, 76, 77], which can sort data faster than the optimized implementations on CPUs and GPUs. For example, one FPGA-based sorter could sort data at 76.8 Gb/s [18], which is faster than the 30 Gb/s or 50 Gb/s achieved by a multi-core processor [37, 20] or a GPU [59, 23], respectively.

Most hardware sorters use a merge logic to combine two sorted lists into one, and they are known as Hardware Merge Sorters (HMS). The simplest merge logic can output one element at a time [45], and some designs can output two or more elements per cycle, called E-record merge logic (where E is 2 or more). Each record is divided into a key and data part, and sorting is based on comparing the keys.

There are two types of existing E-record merge logics, as shown in Figures 5.1a and 5.1b. In these figures, R represents the bit width of the keys. The inputs are two ordered lists $A = \{a_0, a_1, ..., a_{n-1} | a_0 \leq a_1 \leq ... \leq a_{n-1}\}$ and $B = \{b_0, b_1, ..., b_{m-1} | b_0 \leq b_1 \leq ... \leq b_{m-1}\}$. The model in Figure 5.1a uses two groups of FIFOs, two input controllers, and a merge logic. The two sorted lists A and B are stored in these FIFOs, with the elements arranged as shown in Figure 5.1a. The merge logic combines two

lists and outputs the smallest E elements in order. For example, assuming certain conditions for the elements, E specific entries marked in gray are removed from their FIFOs. After this, the next E elements at the front of each FIFO group are not in order and must be sorted again before they can be merged. Each input manager rearranges records using an $E \times E$ crossbar and the gate levels go up as E gets bigger, leading to a drop in frequency. Since the performance of an E-record merge logic depends on the product of E and its operating frequency, this model makes it hard to create an efficient HMS.

Previous studies [18, 58] have used the model in Figure 5.1b, which includes two FIFOs, a selector logic, and a merge logic. The FIFOs keep sorted lists. To merge, the selector checks the first record in each FIFO and picks the smaller one. Then, E records from that FIFO go into the merge logic, which sorts and outputs them at the rate of E records each cycle.

Figure 5.2 shows an example of the 4-record merge logic in the second model. The two sorted input lists in this example are $\{0, 2, 4, 6, 8, 10, 12, 14\}$ and $\{1, 3, 5, 7\}$. I propose to append input lists with the largest possible value M. This enables the consumption of the remaining elements in the last step without compromising functionality since these values are easily discarded. In Figure 5.2, all feedback registers are set to the lowest value. In the beginning, it compares the first element of two FIFOs, and since $0 \leq 1$, the elements $\{2, 4, 6, 8\}$ in the corresponding FIFOs are selected. The logic sorts out eight elements, combines the four it was given with three from the feedback registers, picks the four smallest ones, which should be $\{0, 0, 0, 0\}$, and then stores the bigger three ($\{2, 4, 6\}$) in the registers. In the next step, the first elements from two FIFOs, 8 and 1, are compared, and the selector chooses which FIFO to use. Therefore, four elements $\{1, 3, 5, 7\}$ from the lower FIFO go into the merge logic. Because in the previous step, elements $\{2, 4, 6\}$ were fed back, the merge

logic gives out the smallest four elements $\{1, 2, 3, 4\}$ and keeps the larger three elements $\{5, 6, 7\}$ for later. In Step 3, a similar data flow repeats, but the selector picks the first FIFO since the second one is empty. This merge logic path has $1 + \log_2 E$ groups of a multiplexer and a comparator.

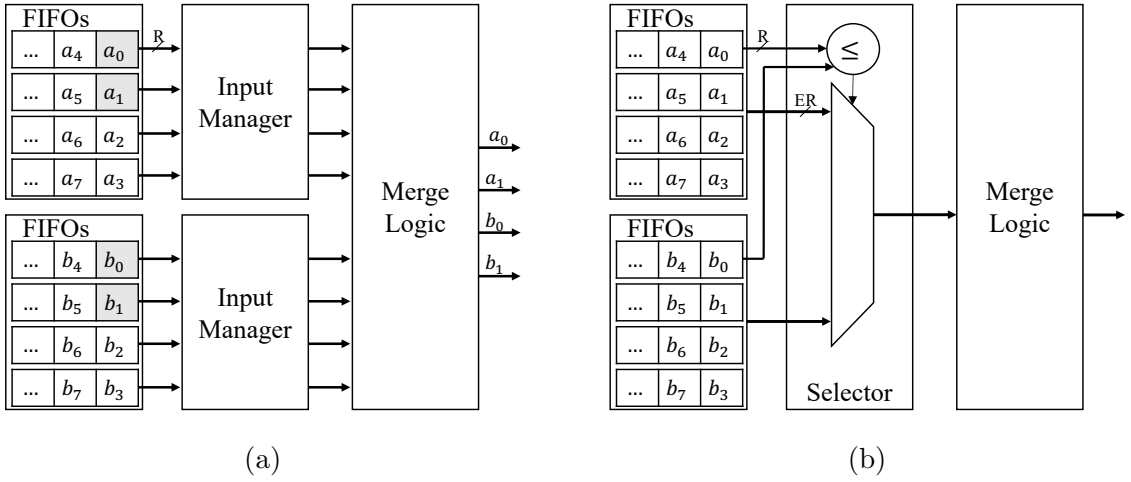


Figure 5.1: Two Models Of E-Record Merge Logics With $E = 4$.

5.2 Hardware Architecture

Figure 5.3 shows the high-level block diagram of the hardware architecture of Sparse Tensor Core, including Multiply and Marge Units (MMUs), Addition Units (AUs), a high-bandwidth memory (HBM), and a banked cache. Each MMU and AU is responsible for processing a partition of a sparse matrix and computing the corresponding partition of the output matrix. HBM has N channels where each channel is directly connected to a set-associate bank in the cache, and each bank is connected to an MMU and an AU.

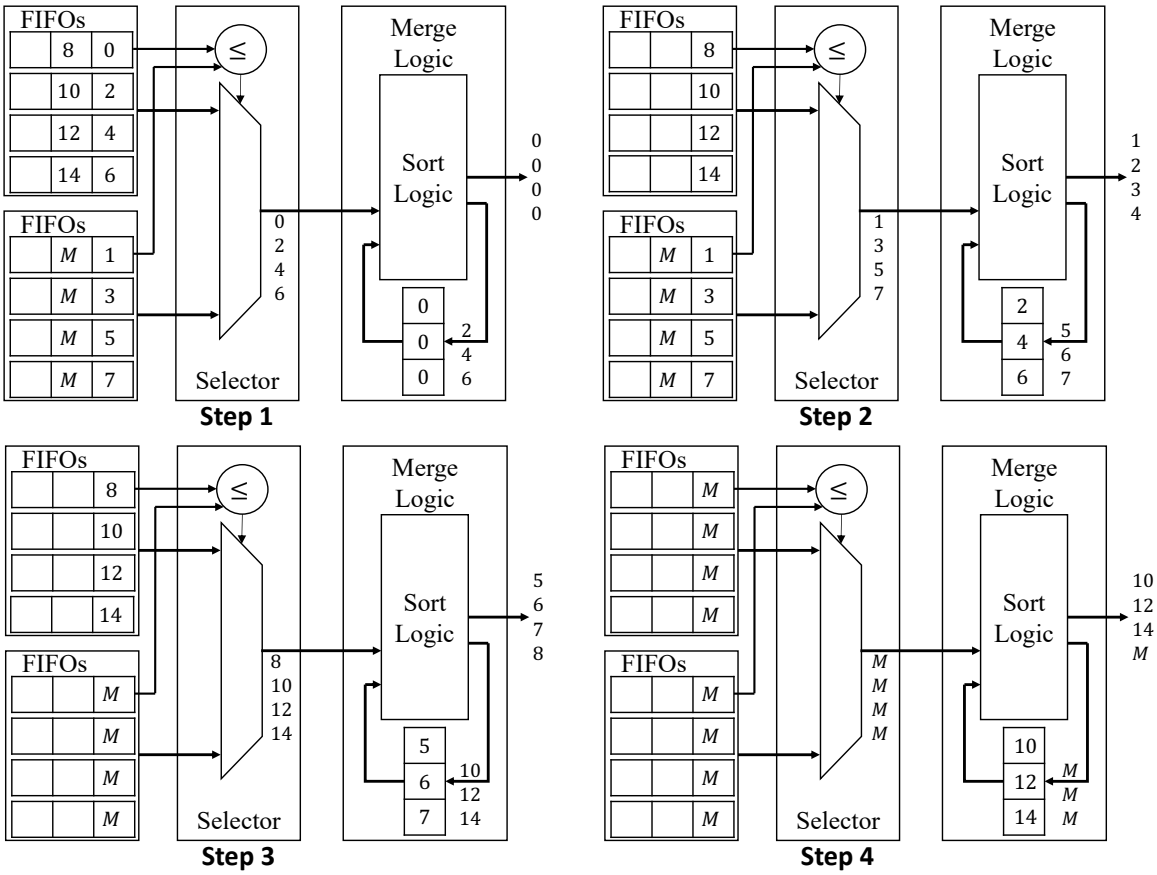


Figure 5.2: This Example Shows How To Merge Two Sorted Lists $\{0, 2, 4, 6, 8, 10, 12, 14\}$ And $\{1, 3, 5, 7\}$, By Using A Method That Merges 4 Elements And Feeds Backs Three Elements At Each Step.

5.2.1 Multiply and Merge Unit

Figure 5.4a shows the block diagram of the hardware architecture of an MMU consisting of an array of 8 parallel multipliers and an 8-record HMS. Each element is a data pair with a value and an index field (*i.e.*, (val, idx)). The incoming streams are scalar elements x from the one input matrix, a vector of elements y_i $(0 \leq i \leq 7)$ from the other input matrix, and a vector of elements z_i $(0 \leq i \leq 7)$ from partial (intermediate) results. The output of the multiplier array is a vector of elements $\hat{y}_i$

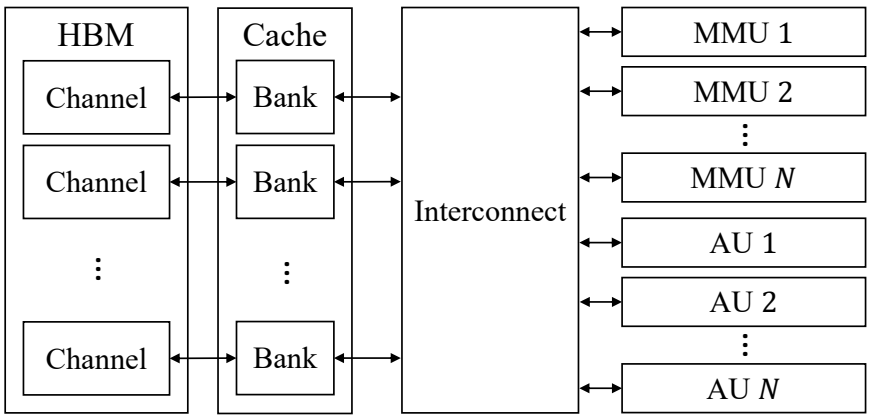
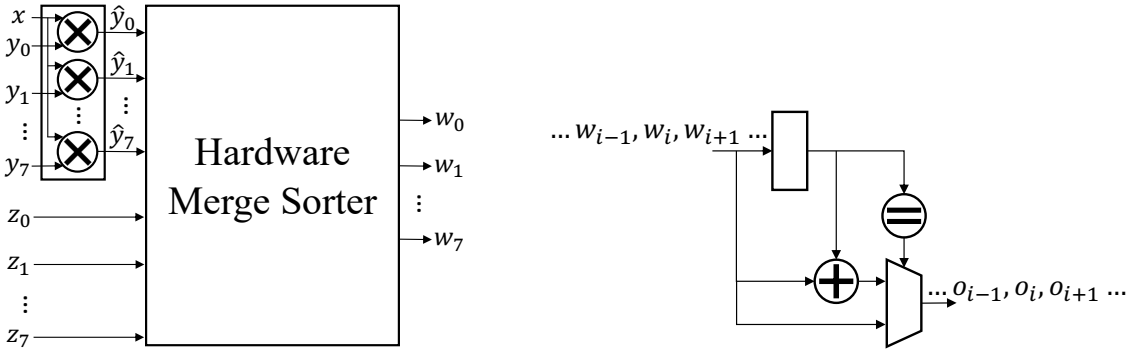


Figure 5.3: High-Level Block Diagram Of The Sparse Tensor Core Hardware Architecture.



(a) (b)

Figure 5.4: The Block Diagrams Of (A) A Multiply And Merge Unit (MMU) And (B) An Addition Unit (AU).

$(0 \leq i \leq 7)$ such that $\begin{cases} \hat{y}_i(val) = x(val) \times y_i(val) \\ \hat{y}_i(idx) = y_i(idx) \end{cases}$. The HMS takes vectors $\hat{y}_i$ and z_i and merges them based on their indices into a vector of 8 elements (w_i) in parallel.

5.2.2 Addition Unit

Figure 5.4b shows the block diagram of the hardware architecture of an AU consisting of a comparator and an adder. AUs are responsible for the summation of

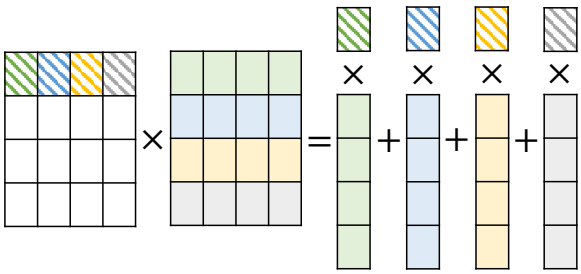


Figure 5.5: The Data Flow For Sparse Matrix-Matrix Multiplications In SpGEMM.

elements with the same index. Since the elements are already sorted, a pairwise index comparison of neighboring elements is sufficient for combining all elements with the same index.

5.3 Data Flow

Figure 5.5 shows the data flow for calculating one output row in SpGEMM. Mapping SpGEMM operations to Sparse Tensor Core is decomposed into two phases. In phase one, the first input matrix is partitioned among N MMUs to generate N partial sub-matrices. In phase two, each sub-matrix is passed through one AU to generate rows of the output matrix.

5.4 Evaluation

5.4.1 Experiment Setup

I evaluate the performance of Sparse Tensor Core in terms of Giga Operations Per Second ($GOPS$). I select a benchmark of sparse matrices from the publicly-available SuiteSparse Matrix Collection [25] (formerly known as the University of Florida Sparse Matrix Collection), a collection of sparse matrices in real applications. The benchmark includes the commonly used matrices for evaluating SpGEMM research in the literature [51]. The specifications of the matrices are summarized in Table 5.1, in-

Table 5.1: The Specification Of The Commonly-Used Benchmark Used In The SOTA SpGEMM Work [51].

Matrix	#Rows	#Columns	Density
hugetrace-0010	12,057,441	12,057,441	2.48e-07
kkt_power	2,063,494	2,063,494	3.00e-06
Hardesty2	929,901	303,645	1.42e-05
poisson-3Da	13,514	13,514	1.93e-03
raefsky3	21,200	21,200	3.31e-03
nemsemm1	3,945	75,352	3.54e-03
cari	400	1200	3.18e-01
lp_fit2d	25	10,524	4.90e-01

cluding the number of rows and columns and the density calculated as $\frac{No.\ of\ Nonzeros}{Matrix\ Size}$. Our design adopts the double-precision floating-point (64-bit) data format. The RTL for the MMUs and AUs was written and synthesized on the 45nm FreePDK45 standard cell library [3] using Synopsys Design Compiler, aiming for a target frequency of 1GHz at 1.25V. I implemented a cycle-accurate simulator in Python to measure the performance of Sparse Tensor Core, which meticulously simulates the interactions among hardware components. The simulator utilizes *pycachesim* [34] to model the cache and is multi-threaded to reduce the simulation time. I compare our design with the SOTA ASIC design of SpGEMM (*i.e.*, Spada [51]) with similar hardware configurations for a fair comparison as described in Table 5.1.

MMU	16 MMUs @ 1GHz
AU	16 AUs @ 1GHz
Cache	2 MB: 16 banks, 128 sets, 16-ways, 64B line, LRU replacement policy
Main Memory	128 GB/s, 16 64-bit HBM channels, 8 GB/s/channel

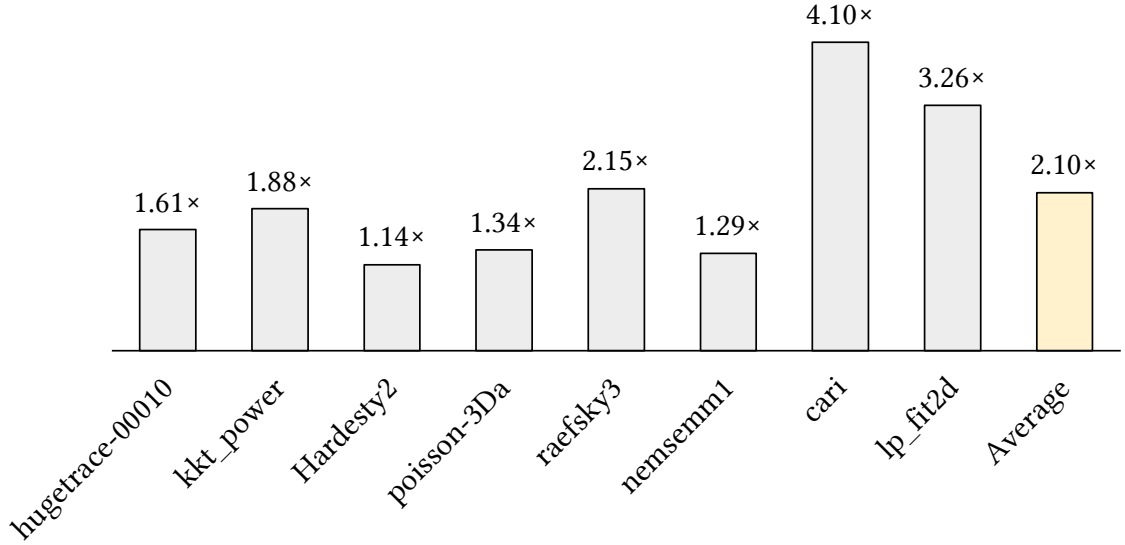
5.4.2 Performance Comparison with the SOTA ASIC Design

Table 5.2 summarizes the performance of Sparse Tensor Core per SpGEMM operation in terms of GOPS. Figure 5.6 shows the runtime speedup achieved by Sparse Tensor Core over Spada [51] for different sparse matrices. The speedup for SpGEMM is up to 4.10× for matrix *cari*. On average, Sparse Tensor Core achieves 2.10× higher performance than the SOTA SpGEMM counterpart. The speedup mainly stems from the parallel processing feature of the proposed MMUs, the improved memory bandwidth as a result of fully utilizing the memory channel data width (*8B*) per memory access and the throughput-optimized hardware architecture tailored to the SpGEMM algorithm, all of which improves the overall utilization of the computation resources at run time.

Table 5.2: The Performance (GOPS) Of Sparse Tensor Core Achieved On Computing $A \times A^T$ For The Commonly-Used Benchmarks.

Matrix	GOPS
hugetrace-0010	6.43
kkt_power	20.73
Hardesty2	34.68
poisson-3Da	22.74
raefsky3	71.63
nemsemm1	30.98
cari	111.89
lp_fit2d	35.9

Figure 5.6: Performance Speedup Over Spada [51] For The Commonly-Used Benchmark.



Publications

- Bank Tavakoli, E., Riera, M., Quraishi, M. H., & Ren, F. (2021, October). FSCHOL: An OpenCL-based HPC Framework for Accelerating Sparse Cholesky Factorization on FPGAs. In 2021 IEEE 33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD) (pp. 209-220). IEEE.

- Bank Tavakoli, E., Riera, M., Quraishi, M. H., & Ren, F. (2024). FSpGEMM: A Framework for Accelerating Sparse General Matrix–Matrix Multiplication Using Gustavson's Algorithm on FPGAs. IEEE Transactions on Very Large Scale Integration (VLSI) Systems.

- Quraishi, M. H., Bank Tavakoli, E., & Ren, F. (2021). A Survey of System Architectures and Techniques for FPGA Virtualization. IEEE Transactions on Parallel and Distributed Systems, 32(9), 2216-2230.

- Bank Tavakoli, E., Beygi, A., & Yao, X. (2022). RPkNN: An OpenCL-Based FPGA Implementation of the Dimensionality-Reduced kNN Algorithm Using Random Projection. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, doi: 10.1109/TVLSI.2022.3147743.

- Huang, J., Sergin, N., Dua, A., Bank Tavakoli, E., Yan, H., Ren, F., & Ju, F. (2020, September). Edge Computing Accelerated Defect Classification Based

on Deep Convolutional Neural Network With Application in Rolling Image Inspection. In International Manufacturing Science and Engineering Conference (Vol. 84263, p. V002T07A037). American Society of Mechanical Engineers.

- Yu, Z., Trindade, B. M., Green, M., Zhang, Z., Sneha, P., Bank Tavakoli, E., ... & Ren, F. (2022, October). A Data-Driven Approach for Automated Integrated Circuit Segmentation of Scan Electron Microscopy Images. In 2022 IEEE International Conference on Image Processing (ICIP) (pp. 2851-2855). IEEE.

- Riera, M., Quraishi, M. H., Bank Tavakoli, E., & Ren, F. FLASH 1.0: A Software Framework for Rapid Parallel Deployment and Enhancing Host Code Portability in Heterogeneous Computing. Under Review.

- Riera, M., Bank Tavakoli, E., Quraishi, M. H., & Ren, F. HALO 1.0: A Hardware-agnostic Accelerator Orchestration Framework for Enabling Hardware-agnostic Programming with True Performance Portability for Heterogeneous HPC. Under Review.

REFERENCES

[1] Cholmod, 2024.

[2] cusparse :: Cuda toolkit documentation, 2024.

[3] Freepdk45, 2024.

[4] Intel stratix 10 fpga features, 2024.

[5] Intel stratix 10 gx/sx product table, 2024.

[6] Intel xeon processor e5-2637 v3 83358, 2024.

[7] nvidia-smi documentation, 2024.

[8] Nvidia system management interface, 2024.

[9] Nvidia v100 tensor core gpu, 2024.

[10] Virtex-7 fpga family, 2024.

[11] Hasan Metin Aktulga, Aydin Buluç, Samuel Williams, and Chao Yang. Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1213–1222. IEEE, 2014.

[12] Patrick R Amestoy, Timothy A Davis, and Iain S Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, 1996.

[13] Hartwig Anzt, Stanimire Tomov, and Jack J Dongarra. Accelerating the lobpcg method on gpus using a blocked sparse matrix vector product. In *SpringSim (HPS)*, pages 75–82, 2015.

[14] Bahar Asgari, Ramyad Hadidi, Hyesoon Kim, and Sudhakar Yalamanchili. Eridanus: Efficiently running inference of dnns using systolic arrays. *Ieee micro*, 39(5):46–54, 2019.

[15] Erfan Bank-Tavakoli, Seyed Abolfazl Ghasemzadeh, Mehdi Kamal, Ali Afzali-Kusha, and Massoud Pedram. Polar: A pipelined/overlapped fpga-based lstm accelerator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(3):838–842, 2019.

[16] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 233–244, 2009.

[17] Aydin Buluc and John R Gilbert. On the representation and multiplication of hypersparse matrices. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–11. IEEE, 2008.

[18] Jared Casper and Kunle Olukotun. Hardware acceleration of database operations. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pages 151–160, 2014.

[19] Yanqing Chen, Timothy A Davis, William W Hager, and Sivasankaran Rajamanickam. Algorithm 887: Cholmod, supernodal sparse cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software (TOMS)*, 35(3):1–14, 2008.

[20] Minsik Cho, Daniel Brand, Rajesh Bordawekar, Ulrich Finkler, Vincent Kulandaisamy, and Ruchir Puri. Paradis: an efficient parallel algorithm for in-place radix sort. *Proceedings of the VLDB Endowment*, 8(12):1518–1529, 2015.

[21] Elizabeth Cuthill. Several strategies for reducing the bandwidth of matrices. In *Sparse Matrices and their Applications: Proceedings of a Symposium on Sparse Matrices and Their Applications, held September 9–10, 1971, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, the National Science Foundation, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department.*, pages 157–166. Springer, 1972.

[22] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2014. Version 0.5.0.

[23] Andrew Davidson, David Tarjan, Michael Garland, and John D Owens. *Efficient parallel merge sort for fixed and variable length keys*. IEEE, 2012.

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

[25] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.

[26] Tiziano De Matteis, Johannes de Fine Licht, and Torsten Hoefler. Fblas: streaming linear algebra on fpga. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13. IEEE, 2020.

[27] Iain S Duff and John K Reid. The multifrontal solution of indefinite sparse symmetric linear. *ACM Transactions on Mathematical Software (TOMS)*, 9(3):302–325, 1983.

[28] Fernando A Escobar, Xin Chang, and Carlos Valderrama. Suitability analysis of fpgas for heterogeneous platforms in hpc. *IEEE Transactions on Parallel and Distributed Systems*, 27(2):600–612, 2015.

[29] Mohammad Farhadi, Mehdi Ghasemi, and Yezhou Yang. A novel design of adaptive and hierarchical convolutional neural networks using partial reconfiguration on fpga. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2019.

[30] Felix Gremse, Andreas Hofter, Lars Ole Schwen, Fabian Kiessling, and Uwe Naumann. Gpu-accelerated sparse matrix-matrix multiplication by iterative row merging. *SIAM Journal on Scientific Computing*, 37(1):C54–C71, 2015.

[31] Thomas Gruber, Jan Eitzinger, Georg Hager, and Gerhard Wellein. Likwid 5: Lightweight performance tools, 2019.

[32] Georg Hager and Gerhard Wellein. *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010.

[33] Pouya Haghi, Tong Geng, Anqi Guo, Tianqi Wang, and Martin Herbordt. Fpamg: Fpga-based acceleration framework for algebraic multigrid solvers. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 148–156. IEEE, 2020.

[34] Julian Hammer. pycachesim, 2015.

[35] Changwan Hong, Aravind Sukumaran-Rajam, Bortik Bandyopadhyay, Jinsung Kim, Süreyya Emre Kurt, Israt Nisa, Shivani Sabhlok, Ümit V Çatalyürek, Srinivasan Parthasarathy, and P Sadayappan. Efficient sparse-matrix multi-vector product on gpus. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 66–79, 2018.

[36] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 300–314, 2019.

[37] Hiroshi Inoue and Kenjiro Taura. Simd-and cache-friendly algorithm for sorting an array of structures. *Proceedings of the VLDB Endowment*, 8(11):1274–1285, 2015.

[38] Intel. Intel 64 and ia-32 architectures software developer's manual: Volume 3, 2016.

[39] Intel. Intel stratix 10 gx fpga development kit user guide, 2020.

[40] Intel. Intel fpga sdk for opencl pro edition: Programming guide, 2022.

[41] Ernest Jamro, Kazimierz Wiatr, et al. The algorithms for fpga implementation of sparse matrices multiplication. *Computing and Informatics*, 33(3):667–684, 2014.

[42] Peng Jiang, Changwan Hong, and Gagan Agrawal. A novel data transformation and execution strategy for accelerating sparse matrix multiplication on gpus. In *Proceedings of the 25th ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 376–388, 2020.

[43] George Karypis and Vipin Kumar. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices, 1997.

[44] Srinidhi Kestur, John D Davis, and Oliver Williams. Blas comparison on fpga, cpu and gpu. In *2010 IEEE computer society annual symposium on VLSI*, pages 288–293. IEEE, 2010.

[45] Dirk Koch and Jim Torresen. Fpgasort: A high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 45–54, 2011.

[46] Moritz Kreutzer, Jonas Thies, Melven Röhrig-Zöllner, Andreas Pieper, Faisal Shahzad, Martin Galgon, Achim Basermann, Holger Fehske, Georg Hager, and Gerhard Wellein. Ghost: building blocks for high performance sparse linear algebra on heterogeneous systems. *International Journal of Parallel Programming*, 45:1046–1072, 2017.

[47] HT Kung, Bradley McDanel, and Sai Qian Zhang. Adaptive tiling: Applying fixed-size systolic arrays to sparse convolutional neural networks. In *2018 24th International Conference on Pattern Recognition (ICPR)*, pages 1006–1011. IEEE, 2018.

[48] Chao-Lin Lee, Chen-Ting Chao, Wei-Hsu Chu, Ming-Yu Hung, and Jenq-Kuen Lee. Accelerating ai applications with sparse matrix compression in halide. *Journal of Signal Processing Systems*, 95(5):609–622, 2023.

[49] Shiqing Li, Shuo Huai, and Weichen Liu. An efficient gustavson-based sparse matrix-matrix multiplication accelerator on embedded fpgas. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

[50] Shiqing Li and Weichen Liu. Accelerating gustavson-based spmm on embedded fpgas with element-wise parallelism and access pattern-aware caches. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2023.

[51] Zhiyao Li, Jiaxiang Li, Taijie Chen, Dimin Niu, Hongzhong Zheng, Yuan Xie, and Mingyu Gao. Spada: Accelerating sparse matrix multiplication with adaptive dataflow. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 747–761, 2023.

[52] Hui-Hsin Liao, Chao-Lin Lee, Jenq-Kuen Lee, Wei-Chih Lai, Ming-Yu Hung, and Chung-Wen Huang. Support convolution of cnn with compression sparse matrix multiplication flow in tvm. In *50th International Conference on Parallel Processing Workshop*, pages 1–7, 2021.

[53] Colin Yu Lin, Ngai Wong, and Hayden Kwok-Hay So. Design space exploration for sparse matrix-matrix multiplication on fpgas. *International Journal of Circuit Theory and Applications*, 41(2):205–219, 2013.

[54] George Lindfield and John Penny. Chapter 2 - linear equations and eigensystems. In George Lindfield and John Penny, editors, *Numerical Methods (Fourth Edition)*, pages 73 – 156. Academic Press, fourth edition edition, 2019.

[55] Joseph WH Liu. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM review*, 34(1):82–109, 1992.

[56] Weifeng Liu and Brian Vinter. An efficient gpu general sparse matrix-matrix multiplication for irregular data. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 370–381. IEEE, 2014.

[57] Dejan Marković and Robert W Brodersen. *DSP architecture design essentials*. Springer Science & Business Media, 2012.

[58] Susumu Mashimo, Thiem Van Chu, and Kenji Kise. High-performance hardware merge sorter. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–8. IEEE, 2017.

[59] Duane Merrill and Andrew Grimshaw. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for gpu computing. *Parallel Processing Letters*, 21(02):245–272, 2011.

[60] 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.

[61] Maxim Naumov, L Chien, Philippe Vandermersch, and Ujval Kapasi. Cusparse library. In *GPU Technology Conference*, 2010.

[62] Esmond G Ng and Barry W Peyton. Block sparse cholesky algorithms on advanced uniprocessor computers. *SIAM Journal on Scientific Computing*, 14(5):1034–1056, 1993.

[63] Yuyao Niu, Zhengyang Lu, Haonan Ji, Shuhui Song, Zhou Jin, and Weifeng Liu. Tilespgemm: A tiled algorithm for parallel sparse general matrix-matrix multiplication on gpus. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 90–106, 2022.

[64] 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.

[65] Mathias Parger, Martin Winter, Daniel Mlakar, and Markus Steinberger. speck: accelerating gpu sparse matrix-matrix multiplication through lightweight analysis. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 362–375, 2020.

[66] Dong-Hyeon Park, Subhankar Pal, Siying Feng, Paul Gao, Jielun Tan, Austin Rovinski, Shaolin Xie, Chun Zhao, Aporva Amarnath, Timothy Wesley, et al. A 7.3 m output non-zeros/j, 11.7 m output non-zeros/gb reconfigurable sparse matrix–matrix multiplication accelerator. *IEEE Journal of Solid-State Circuits*, 55(4):933–944, 2020.

[67] Juan C Pichel, Francisco F Rivera, Marcos Fernández, and Aurelio Rodríguez. Optimization of sparse matrix–vector multiplication using reordering techniques on gpus. *Microprocessors and Microsystems*, 36(2):65–77, 2012.

[68] Juan C Pichel, David E Singh, and Jesús Carretero. Reordering algorithms for increasing locality on multicore processors. In *2008 10th IEEE International Conference on High Performance Computing and Communications*, pages 123–130. IEEE, 2008.

[69] Ali Pinar and Michael T Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, pages 30–es, 1999.

[70] J-A Pineiro and Javier D Bruguera. High-speed double-precision computation of reciprocal, division, square root, and inverse square root. *IEEE Transactions on Computers*, 51(12):1377–1388, 2002.

[71] Masudul Hassan Quraishi, Erfan Bank Tavakoli, and Fengbo Ren. A survey of system architectures and techniques for fpga virtualization. *arXiv preprint arXiv:2011.09073*, 2020.

[72] Abid Rafique, George A Constantinides, and Nachiket Kapre. Communication optimization of iterative sparse matrix-vector multiply on gpus and fpgas. *IEEE Transactions on Parallel and Distributed Systems*, 26(1):24–34, 2014.

[73] Fengbo Ren and Dejan Marković. 18.5 a configurable 12-to-237ks/s 12.8 mw sparse-approximation engine for mobile exg data aggregation. In *2015 IEEE International Solid-State Circuits Conference-(ISSCC) Digest of Technical Papers*, pages 1–3. IEEE, 2015.

[74] SC Rennich, TA Davis, and P Vandermersch. Gpu acceleration of sparse matrix factorization in cholmod. In *GPU Technology Conference*, 2014.

[75] Fazle Sadi, Joe Sweeney, Tze Meng Low, James C Hoe, Larry Pileggi, and Franz Franchetti. Efficient spmv operation for large and highly sparse matrices using scalable multi-way merge parallelization. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 347–358, 2019.

[76] Makoto Saitoh, Elsayed A Elsayed, Thiem Van Chu, Susumu Mashimo, and Kenji Kise. A high-performance and cost-effective hardware merge sorter without feedback datapath. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 197–204. IEEE, 2018.

[77] Makoto Saitoh and Kenji Kise. Very massive hardware merge sorter. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 86–93. IEEE, 2018.

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

[79] Linghao Song, Yuze Chi, Licheng Guo, and Jason Cong. Serpens: A high bandwidth memory based accelerator for general-purpose sparse matrix-vector multiplication. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 211–216, 2022.

[80] Wei Song, Dirk Koch, Mikel Luján, and Jim Garside. Parallel hardware merge sorter. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 95–102. IEEE, 2016.

[81] 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.

[82] Yichun Sun, Hengzhu Liu, and Tong Zhou. Sparse cholesky factorization on fpga using parameterized model. *Mathematical Problems in Engineering*, 2017, 2017.

[83] Russell Tessier and Wayne Burleson. Reconfigurable computing for digital signal processing: A survey. *Journal of VLSI signal processing systems for signal, image and video technology*, 28(1-2):7–27, 2001.

[84] 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.

[85] Francisco Vazquez, G Ortega, José-Jesús Fernández, and Ester M Garzón. Improving the performance of the sparse matrix vector product with gpus. In *2010 10th IEEE International Conference on Computer and Information Technology*, pages 1146–1151. IEEE, 2010.

[86] 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.

[87] Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. Adaptive sparse matrix-matrix multiplication on the gpu. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 68–81, 2019.

[88] Roger Woods, John McAllister, Gaye Lightbody, and Ying Yi. *FPGA-based implementation of signal processing systems*. Wiley Online Library, 2017.

[89] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. Sparsetir: Composable abstractions for sparse compilation in deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 660–678, 2023.

[90] Guowei Zhang, Nithya Attaluri, Joel S Emer, and Daniel Sanchez. Gamma: Leveraging gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 687–701, 2021.

[91] 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.

[92] Zhou Zhou, Jemal H Abawajy, Fangmin Li, Zhigang Hu, Morshed U Chowdhury, Abdulhameed Alelaiwi, and Keqin Li. Fine-grained energy consumption model of servers based on task characteristics in cloud data center. *IEEE access*, 6:27080–27090, 2017.