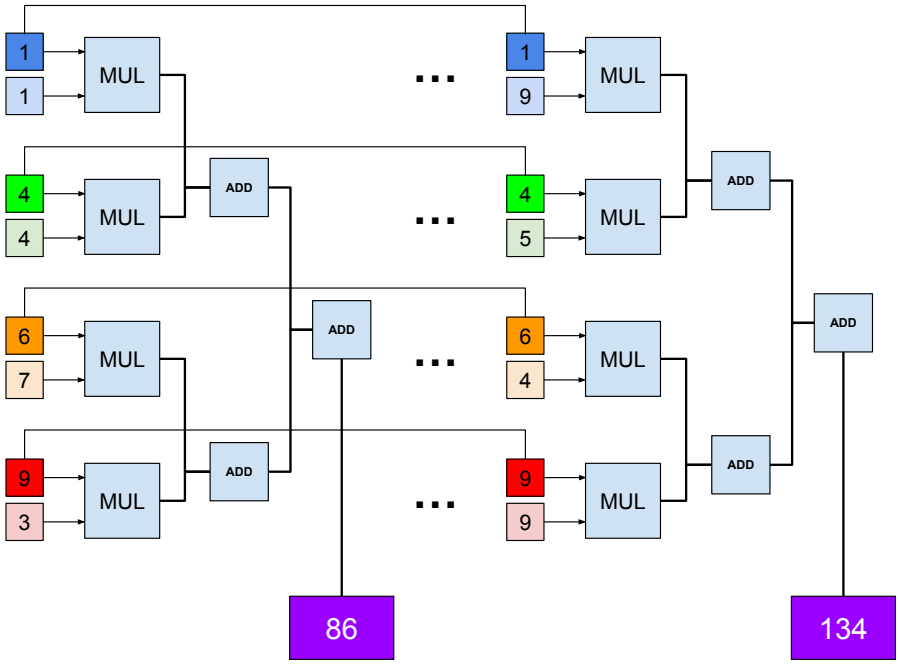
CHALMERS
UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG



QuadriSparse: RISC-V Sparse Matrix Accelerator and ISA Extension

A Tightly-Coupled Sparse-Dense Matrix Multiplication Accelerator for RISC-V

Master's thesis in Computer science and engineering

Nik Erlandsson
Oskar Swärd

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's thesis 2026

QuadriSparse: RISC-V Sparse Matrix Accelerator and ISA Extension

A Tightly-Coupled Sparse-Dense Matrix Multiplication Accelerator for RISC-V

Nik Erlandsson
Oskar Swärd

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

QuadriSparse: RISC-V Sparse Matrix Accelerator and ISA Extension
A Tightly-Coupled Sparse-Dense Matrix Multiplication Accelerator for RISC-V
Nik Erlandsson, Oskar Swärd

Supervisor: Mateo Vázquez Maceiras, Department of Computer Science and Engineering
Co-Supervisor: André Carvalho, Department of Computer Science and Engineering
Examiner: Pedro Petersen Moura Trancoso, Department of Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: The processing engine for sparse-dense matrix multiplication with Gustavson's algorithm.

Typeset in LaTeX
Gothenburg, Sweden 2026

QuadriSparse: RISC-V Sparse Matrix Accelerator and ISA Extension
A Tightly-Coupled Sparse-Dense Matrix Multiplication Accelerator for RISC-V
Nik Erlandsson, Oskar Swärd
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Sparse dense matrix multiplication (SpMM) is an important operation in many applications such as inference and training of pruned large language models, graph analytics and scientific computing. These applications often operate on data that is inherently sparse. Applying dense matrix multiplication (GEMM) to sparse data wastes computations on zero-valued elements, which SpMM avoids by skipping them. However, accelerators designed for dense matrix multiplication do not necessarily support sparse matrix multiplication efficiently. This motivates extensions that can exploit sparsity while retaining computation capabilities for dense workloads. This thesis explores the prospect of extending a small dense matrix multiplication accelerator with additional hardware for SpMM, evaluating the the performance benefits against the hardware overhead. We introduce QuadriSparse, a SpMM extension for unstructured sparsity that adds a partly new datapath to the small and efficient RISC-V accelerator Quadrilatero. The accelerator includes three new instructions to load a tile of the sparse matrix (`SPLD_W`), load a tile of the dense matrix (`DLD_W`), and multiply the two using Gustavson's algorithm (`SPMAC_W`). We evaluate the resulting accelerator in terms of execution time across different sparsity levels and matrix sizes and measure the FPGA resource utilization through synthesis. QuadriSparse achieves up to 6.6x lower execution time than dense execution on Quadrilatero at 99% sparsity and first outperforms the dense baseline at 95% sparsity. FPGA synthesis shows an increase in resource utilization of 4.6% in LUTs and 23% in DSP-blocks relative to the baseline design.

Keywords: SpMM, sparse matrix, matrix multiplication, RISC-V, computer architecture, accelerator.

Acknowledgements

We would like to thank our supervisors Mateo Vázquez Macerias and André Carvalho for their guidance throughout this work. With their support we navigated the many challenges encountered along the way. We also thank our examiner Pedro Petersen Moura Trancoso for always being supportive and giving us meaningful insights throughout this project. Without their help, the completion of this thesis would not have been possible.

Nik Erlandsson, Oskar Swärd, Gothenburg, 2026-09-09

List of Acronyms

Below is a list of acronyms that are used throughout the report.

AI	Artificial Intelligence
CNN	Convolutional Neural Network
COO	Coordinate Compressed format
CSC	Compressed Sparse Column
CSR	Compressed Sparse Row
FSM	Finite state machine
FU	Functional Unit
GEMM	General Matrix Multiplication or regular dense matrix multiplication
GNN	Graph Neural Network
HBM	High Bandwidth Memory
ISA	Instruction set architecture
LLM	Large Language Model
LSU	Load store unit
MAC	Multiply and Accumulate
MRF	Matrix Register File
NNZ	Number of non-zero elements
SA	Systolic Array
SPE	Sparse Processing Engine
SpGEMM	The multiplication of two sparse matrices
SpMM	The multiplication of a sparse and a dense matrix
SpMV	The multiplication of a sparse matrix and a dense vector

Contents

List of Figures

List of Tables

1

Introduction

Sparse matrix multiplication (SpMM) is an important primitive in many applications, including scientific calculations [1], artificial intelligence (AI) [2], linear algebra [3], [4], and graph analytics [5]. Many problems in the fields such as civil engineering, physics, or mathematics have a graph component. Graphs are typically represented using adjacency matrices that encode node connectivity. For many real-world graphs, the number of connections is substantially smaller than the number of possible connections, resulting in sparse adjacency matrices. Exploiting sparsity can improve the efficiency of linear algebra operations by avoiding unnecessary computations involving zero-valued elements.

Naturally occurring sparsity, arising from the structural constraints of a problem rather than from methods such as pruning, is common across many application domains, including computational fluid dynamics, electromagnetics, and circuit simulation. Sparsity levels in these domains can be extremely high (as cataloged in the SuiteSparse Matrix Collection [6]): for example, *FEM_3D_thermal2*, a finite element discretization, exhibits a sparsity level of 99.98%. Geometry and structural simulations such as *cage15* and *bone010* show similar sparsity levels, while large directed graph problems such as *web-Google* exceed this, reaching sparsity levels above 99.99%.

AI and deep learning applications rely heavily on massive matrix multiplications, which can be computationally expensive and energy-consuming. Sparsity can in many cases be introduced into these operations through pruning, the removal of low-impact neurons [7]. This can be exploited to reduce the amount of computation and memory accesses required. Graph neural networks (GNNs) operate on graph-structured data and provide an example where sparsity arises naturally from the graph connectivity [8].

Efficient execution of SpMM is therefore important for applications involving sparse matrix representations. However, exploiting sparsity efficiently is challenging due to irregular computation and memory access patterns inherent to sparse matrices. Hardware acceleration offer an opportunity to enhance SpMM performance by exploiting parallelism and using functional units that are purpose-built for the sparse matrix formats and computational patterns. This motivates the investigation of hardware extensions for accelerating SpMM.

This thesis addresses this by extending Quadrilatero [9], a small-scale RISC-V sys-

tolic array accelerator for dense matrix multiplication, with a tightly-coupled ISA extension for SpMM, resulting in an accelerator we call QuadriSparse. Three new instructions are introduced: `SPLD_W` and `DLD_W`, which load a tile of the sparse and dense matrix respectively, and `SPMAC_W`, which multiplies and accumulates them using Gustavson's algorithm. QuadriSparse targets unstructured sparsity and requires no offline pre-processing of the input matrices, while retaining the original dense instructions so that the same accelerator handles both GEMM and SpMM workloads. We evaluate QuadriSparse through cycle-accurate simulation, finding that it outperforms the dense baseline once sparsity exceeds roughly 95%, reaching a 6.6x speedup at 99% sparsity, at an area cost of only 4.6% more LUTs and 23% more DSP blocks. Table 1.1 places these characteristics alongside related accelerators from the literature.

1.1 Related Work

The acceleration of sparse matrix multiplication is a well studied field as many accelerators have been developed to support different constraints. This section is divided into two parts, tightly coupled followed by loosely coupled accelerators. Instruction level accelerators are tightly coupled and feature ISA extensions that are integrated into the standard CPU pipeline, while standalone accelerators are loosely coupled external or memory mapped. The former is closer to what is proposed by this thesis, but both categories are important in order to understand the current state of hardware acceleration of SpMM.

1.1.1 Instruction Level Accelerators

Zhang et al. [10] explores acceleration of matrix operations on RISC-V cores. It is built on the CV32E40P processor architecture and explores the optimization of sparse matrix multiplication and convolution. This implementation uses three custom instructions `K_LOAD`, `I_CAL` and `WB` (write back). `K_LOAD` focuses on loading the convolution kernel data and also perform sparsification and compression in column-major format. `I_CAL` handles the computation and `WB` writes back the results to memory. The final product achieved an efficiency of 675 GOPS/W. This shows that custom RISC-V ISA extensions for sparse matrix multiplication and convolution can achieve promising efficiency. Furthermore, the implementation is also verified on an FPGA which highlights the feasibility of sparse matrix optimization on a hardware-level.

IndexMAC, proposed by Titopoulos et al. [11] is a custom RISC-V instruction set to accelerate structured sparsity in a state-of-the art CNN setting. It introduces the `vindexmac` instruction, which combines the multiply accumulate and vector load operations. The proposed implementation shows promising results, reaching an average speedup of 1.95x for 1:4 sparsity and 1.88x for 2:4 sparsity. Only supporting semi-structured sparsity makes sense for AI applications but can be limiting for general purpose SpMM problems.

Another RISC-V implementation is the Matrix Sparsity Extension (MSE) architec-

ture by Ribas et al. [12]. Similarly to IndexMAC, this work assumes an $N : M$ structured sparsity format but only a sparsity ratio of 2:4. It defines the `mmkset`, `mmkshift` and `mmksp` instructions. The idea is the use mask register to encode the sparsity pattern, making the sparse tiles dense for computation. In the best case this implementation reached a speedup of 1.75x compared to its baseline.

Osterno et al. [13] introduce a custom RISC-V instruction to accelerate SpMV operations. For each non zero element in the sparse matrix, the `vconv` instruction matches its column index with an element from the dense vector. It then multiplies the two elements and stores the product in an output vector register. This effectively eliminates the inner loop of the SpMV multiplication resulting in a speedup of up to 5x.

RV-SpDNN developed by Jiang et al. [14] propose multiple custom instructions to accelerate sparse inference of deep neural networks. To achieve this, they have designed a hardware friendly dataflow called "super-im2col". Its purpose is to map sparse convolutional tasks into SpGEMM-tasks which helps alleviate the load imbalance that normally arises from sparse operations. Overall, the authors achieve a 4x speedup over their baseline.

1.1.2 Standalone Accelerators

Sextans developed by Song et al. [15] explores a streaming dataflow architecture for accelerating SpMM operations. To design a truly general hardware accelerator it partitions the sparse matrix A into multiple sub-matrices and the scheduled non-zero list of all the elements of A is stored linearly in memory. A pointer array Q is used to keep track of the starting index of the non-zero elements from each sub-matrix. Sextant uses an out-of-order non-zero scheduling approach which is an attempt to ensure that an arbitrary non-zero element is scheduled at the earliest cycle such that the row index of the non-zero does not have any read-after-write dependencies.

Azul introduced by Feldmann et al. [16] is designed specifically to accelerate iterative solving of sparse systems of linear equations. Azul is built from a grid of tiles with a large amount of SRAM distributed between the tiles. The main contribution of the paper is their novel data mapping algorithm, which greatly reduces communication between the tiles.

HiSpMM proposed by Sedigh Baroughi et al. [17] is a high performance, high bandwidth, accelerator for SpMM. The paper's main novelty is their dense row sharing strategy which identifies rows in the sparse matrix with disproportionally high number of non zero elements. These rows are segmented and split between the processing elements to enhance utilization. Another optimization HiSpMM makes is decoupling the memory channel allocation for the dense matrix from the processing elements. This allows the user to easily reallocate memory bandwidth between the sparse and the dense matrix depending on what the current task demands. Overall they observed a 5.8x speedup over the state of the art.

Flexagon, proposed by Hedge et al. [18] is a multi-flow sparse-sparse matrix multiplication accelerator (SpGEMM) that uses offline profiling of the given matrices

in order to decide what algorithm to use: inner product, outer product or Gustavson's algorithm. It is the first reconfigurable SpGEMM-accelerator, selecting the most optimal dataflow based on the input matrices. This distinguishes it from prior accelerators, which are often fixed for a single dataflow, making them lose performance when characteristics of the inputs changes. Flexagon was compared to state-of-the-art SIGMA-like accelerators where it achieved an average performance gain of 4.59x.

In contrast to Flexagon's focus on SpGEMM, Trapezoid, presented by Yang et al. [19], is a versatile accelerator that can handle multiple levels of sparsity efficiently: dense-, mildly sparse- (above 1% non-zeroes) and highly sparse matrices (less than 1% non-zeroes). It is able to switch between inner-product based dataflow and Gustavson based dataflow depending on these sparsity levels. The proposed accelerator was benchmarked against SIGMA as well as Flexagon with varying workloads and achieved a 4.3x and 2.9x geomean respectively.

1.1.3 Summary

Table 1.1 summarizes the state-of-the-art accelerators discussed above, alongside QuadriSparse, the accelerator presented in this work. While this thesis focused on SpMM (see Section 1.4.1), some of the works presented in this section focus a different operation, such as SpMV or SpGEMM. They remain relevant to this thesis as they address some of the same underlying challenges of unstructured sparsity, dataflow selection, and hardware-efficient sparse computation.

Table 1.1: State-of-the-Art summary

Work	Operation	Unstructured Sparsity?	Efficient at Low Sparsity?	Minimal Preproc.?	Small Scale?	Multimodal? (GEMM+SpMM)
Zhang et al. [10]	SpMM	✔	✘	✔	✔	✘
IndexMAC [11]	SpMM	✘	✔	✔	✔	✔
MSE [12]	SpMM	✘	✔	✔	✔	✔
Osterno et al. [13]	SpMV	✔	✘	✔	✔	✘
RV-SpDNN [14]	SpMM	✔	✘	✔	✔	✘
Sextans [15]	SpMM	✔	✘	✔	✘	✘
Azul [16]	SpMV	✔	✘	✘	✘	✘
HiSpMM [17]	SpMM	✔	✘	✘	✘	✘
Flexagon [18]	SpGEMM	✔	✘	✔	✘	✘
Trapezoid [19]	SpMM	✔	✔	✘	✘	✔
QuadriSparse	SpMM	✔	✘	✔	✔	✔

1.2 Research Questions

Sparse matrix multiplication can be performed on general-purpose hardware, but doing so is inefficient due to the computational cost of processing zero-valued elements. Accelerators for general matrix multiplication (GEMM) can provide parallelism for

matrix multiplication, but conventional GEMM execution does not inherently exploit the sparsity of the input matrix and can therefore still include computation involving zero-valued elements. Software implementations can instead use sparse matrix formats and algorithms to reduce these computations, but are limited by the available parallelism of the processor. Existing SpMM accelerators address these problems by introducing hardware support for sparse input matrices, involving techniques for skipping zero-valued elements. However, these designs are typically either large-scale standalone accelerators decoupled from a general-purpose core, or instruction-level extensions that target structured sparsity or lack native support for dense GEMM. It remains unclear whether a small-scale, tightly-coupled GEMM accelerator can be extended to also support unstructured SpMM efficiently, and what such an extension would cost in terms of complexity and hardware overhead.

This raises the following research questions:

1. Can a small-scale GEMM accelerator be extended to support SpMM, and at what sparsity level does it begin to outperform the original?
2. What architectural choices have to be made to achieve this, and how do traditional techniques for using and storing sparse matrices map to this small scale?
3. What are the performance and area overheads of the extended accelerator?

1.3 Goals and Contributions

The goal of this thesis was to add support for SpMM to an existing GEMM accelerator. Using an existing accelerator as a baseline allowed for faster development, as much of the infrastructure connecting to the RISC-V core could be reused, allowing focus to remain on developing the SpMM specific instructions. This also provided a convenient way to benchmark the SpMM accelerator as it could be directly compared to the dense accelerator.

The goal was for the resulting sparse accelerator to be tightly coupled to the host CPU via the CORE-V X-IF interface. Furthermore, the co-processor was designed to excel at unstructured sparsity while remaining small-scale and requiring minimal pre-processing, a combination that Section 1.1 shows to be underrepresented among existing instruction-level accelerators.

The main contributions of this thesis are:

- `QuadriSparse`[1]: An extension of the state-of-the-art systolic array coprocessor Quadrilatero to handle sparse workloads efficiently.
- Introduce an ISA extension to accelerate SpMM including three instructions `SPLD_W`, `SPLD_W`, `SPMAC_W`. These instructions load a tile of the sparse matrix, a tile of the dense matrix, and multiply the two using Gustavson's algorithm.

[1] QuadriSparse GitHub: https://github.com/nikerl/quadrisparse

- Verification and performance assessment of the extended accelerator through simulation as well as FPGA synthesis to estimate hardware cost. This found that the extended accelerator achieved up to 6.6x speedup over the baseline at 99% sparsity, with an area overhead of only 4.6% in LUTs and 23% in DSP-blocks.

1.4 Scope and Limitations

Below are the listed topics which were not considered to be within the scope of the project.

1.4.1 Focus on Sparse-Dense

The project did not aim to work with the generalized sparse-sparse matrix multiplication (SpGEMM) instead it focused specifically on the sparse-dense (SpMM & SpMV) case. SpGEMM adds another level of complexity as the accelerator has to deal with two sparse matrices with potentially very different levels of sparsity [20], which was beyond the scope of this thesis. In regards to the focus on sparse-dense, there are two distinct variations: mainly structured- and unstructured sparsity. Structured sparsity makes it easier to exploit parallelism but may restrict the use-case as it does not capture general workloads as well. Unstructured sparsity makes the hardware implementation more difficult due to its unpredictability. As a result, the benefits and trade-offs of sparsity-aware hardware optimizations can vary significantly depending on the sparsity pattern. Therefore, the thesis has focused on unstructured sparsity.

1.4.2 Baseline for Performance Evaluation

This thesis benchmarks QuadriSparse only against the baseline of the dense Quadrilatero instructions, and does not include a comparison against the accelerators discussed in Section 1.1. The standalone accelerators (Section 1.1.2) operate at much larger scale, use different memory hierarchies such as HBM, and are loosely coupled to the host rather than integrated at the instruction level. This makes them less comparable. The instruction-level accelerators (Section 1.1.1) are closer in scale and coupling and would in principle have made for a more directly comparable baseline. These were not included due to time constraints: several of these works are not open source or are poorly documented, and adapting the benchmarking setup to accommodate each one was judged infeasible considering the timeframe.

1.4.3 CPU and Compiler Integration

This thesis focused on the architectural design of the accelerator and did therefore not include compiler integrations and tests using a host CPU. As the goal was to explore the architecture of the accelerator, evaluation with a simple test bench was sufficient.

1.4.4 Energy Efficiency

Energy efficiency is not a metric that was measured and compared to the baseline. When hardware additions were made, thought was given to their effect on area and energy consumption but the design did not aim to optimize for energy efficiency. Therefore, it was not measured directly.

1.5 Ethical Considerations

This section deals with the potential societal, ethical and ecological aspects relevant to this thesis.

1.5.1 Societal Aspects

The thesis does not carry any direct societal implications as it focuses on low level hardware architecture and does not target specific applications, users, or societal domains. However, the project contributes to a relevant and active research area in the HPC-community. Advancement in SpMM computing efficiency can indirectly impact a wide range of applications by reducing computational cost and increasing energy efficiency. These are both relevant to the rise of AI as advancements in SpMM can enable lower power and hardware requirements in constrained environments such as edge-computing.

1.5.2 Ethical Aspects

The proposed accelerator is simply a computational tool and, as such, it is use-neutral. It does not directly introduce any ethical aspects related to specific use-cases. Potential aspects may arise when looking at how the work might be employed in higher-level applications. However, that is beyond the scope of the thesis.

1.5.3 Ecological Aspects

This thesis aims to design an accelerator that improves the performance of SpMM calculations. The ecological impact is indirect, and would show up as energy savings in applications that rely on sparse matrix multiplication. As machine learning workloads continue to grow, even small improvements at the hardware level can lead to lower overall power consumption and reduced environmental impact.

2

Theory

This chapter covers the theory required to understand this thesis. It explains what sparse matrices are, why they are needed, how they are stored, and how they are used. The chapter also provides information regarding some hardware concepts used in this thesis.

2.1 Computational Matrix Multiplication

Matrix multiplication is the process of multiplying two matrices, A and B, with each other. Every element in the result matrix is calculated by the dot product of a row from matrix A and a column from matrix B. When a matrix is said to be read in row-major order that means that it is read left to right then top to bottom. When a matrix is said to be read in column-major order that means it is read top to bottom then left to right. In traditional dense matrix multiplication the matrix A is read in row-major order and matrix B is read in column-major order. If the matrices are too large to be proceed in one step the multiplication can be tiled meaning that a smaller segments of the operand matrices processed in each step.

2.2 Sparsity

A sparse matrix is defined as a matrix in which a significant amount of the elements are zero.

2.2.1 Dense versus Sparse Matrix Multiplication

Dense-dense matrix multiplication refers to the multiplication of two dense matrices, resulting in a dense matrix. Sparse-dense matrix multiplication, on the other hand, involves multiplying a sparse matrix with a dense matrix. This typically produces a dense output matrix.

From a hardware perspective, the two are significantly different. Given two dense matrices of size $n \times n$, the number of written elements to the output matrix is always n^2. In contrast, sparse-dense matrix multiplication introduces irregular memory accesses in the input matrix and, depending on the representation, in the output matrix since the location of computed elements are not known statically. This makes the computation more difficult to parallelize, requiring specialized algorithms.

2.2.2 Structured Sparsity

The structure of a sparse matrix refers to the pattern, or lack thereof, in which the non-zero elements are distributed throughout the matrix. This can generally be divided into three groups: structured, semi-structured, and unstructured. The boundaries between these groups are not strict. The explanations below should be considered as representative examples of their respective groups rather than as strict definitions.

A matrix exhibiting structured sparsity will have the non-zero elements distributed in a predictable manner that can be exploited algorithmically. For example, a diagonally structured matrix will have its non zero elements concentrated around the main diagonal, see Figure 2.1(a). Row/column structured matrices, on the other hand, will have entire rows or columns that are zero, see Figure 2.1(b).

Semi-structured sparsity, also known as $N : M$ sparsity, is a subset of structured sparsity which display more fine grained structures in the distribution of its non-zero elements. Formally, every block of M elements should have N non zero elements. For example, if a matrix is said to be semi-structured with 1:4 sparsity then every block of four contiguous elements should contain one non zero element. Figure 2.1(c) shows a matrix with 2:4 sparsity.

A matrix with unstructured sparsity has the non zero elements distributed randomly throughout the matrix, see Figure 2.1(d). This category of sparsity is the hardest to accelerate due to its random distribution which makes load-balancing difficult and causes irregular memory accesses.

(a) Structured: Diagonal

8	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	5	0	0	0	0	0	0
0	0	2	5	0	0	0	0
0	0	0	0	7	2	0	0
0	0	4	0	0	1	0	0
0	0	0	0	0	8	1	0
0	0	0	0	0	0	0	4

(b) Structured: Row

0	0	0	0	0	0	0	0
1	2	8	5	1	6	1	2
1	9	3	3	8	2	7	4
0	0	0	0	0	0	0	0
9	3	5	2	8	3	8	1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
8	2	4	8	3	9	1	6

(c) Semi-Structured

0	2	6	0	0	2	0	9
1	0	0	6	0	0	7	6
3	0	4	0	8	0	1	0
0	5	0	8	0	5	0	3
0	2	0	1	6	2	0	0
3	0	0	6	0	0	2	6
1	0	8	0	0	3	4	0
0	5	0	9	0	4	0	8

(d) Unstructured

0	0	0	7	0	0	2	0
6	0	2	0	0	8	0	0
0	0	0	0	2	0	0	0
2	0	6	8	0	0	1	0
0	9	0	0	2	0	7	0
0	0	0	3	0	0	0	5
0	0	0	0	6	0	2	0
0	0	4	0	3	0	0	0

Figure 2.1: Visualization of four sparse matrices with differently structured sparsity

2.2.3 Applications

Zhu et al. [2] conducts a survey of Large Language Model (LLM) compression techniques. The section on model pruning is especially relevant to this thesis as it is the step that introduces sparsity into the weight matrix of the model. The survey finds that unstructured pruning, which removes individual parameters without regard to structure, produces the highest sparsity while retaining the best accuracy at the cost of needing more specialized accelerators. On the other hand, structured pruning removes entire neurons, attention heads, or layers. It typically results in lower sparsity and accuracy but maintains greater compatibility with general purpose hardware.

The paper shows that semi-structured pruning achieves better performance than structured pruning with higher compression ratios. It still performs worse than unstructured pruning, however it has several advantages, most notably improved load balancing due to its structured nature. This in turn simplifies the scheduling of parallel operations.

Computations involving graphs such as graph neural networks (GNN) and graph analytics often involve sparse operations. It is specifically the graph's adjacency matrix that often exhibits very high unstructured sparsities [5]. This is a result of a graph with a high number of nodes and low degree of connectivity. In the case of GNN inference the SpMM operation is the multiplication of the sparse adjacency matrix with the dense weight matrix. A concrete example of sparse matrix vector multiplication in graph analytics is Google's PageRank algorithm [21].

An important application for SpMM is in numerical linear algebra. For example the block Lanczos [4] and block Arnoldi [3] algorithms which are iterative approaches to finding eigenvalues and eigenvectors. Eigenvectors are vectors that, when multiplied with a transformation matrix, are only scaled and not rotated. The eigenvalue is the factor that the eigenvector is scaled by. A concrete example of when eigenvalues, and by extension SpMM, are useful is in civil engineering. The resonance frequencies of a structure like a bridge can often be found by calculating the eigenvalues of the system of equations that describe it. When the way the components in the bridge interact is modeled as a graph, sparsity arises naturally in the graphs adjacency matrix.

Computational fluid dynamics (CFD) is another application for SpMM, particularly simulations involving Flux Reconstruction (FR) as described by Wozniak et al. [1]. FR first divides the computational domain into smaller elements and a local flux is computed in each element. At the interfaces between neighboring elements a common "Riemann flux" is calculated to construct correction terms which ensure consistency between elements. The propagation of these correction terms can be expressed as a series of matrix multiplications, dense inside each element and sparse when propagating the correction to its neighbors. Once again the sparse matrix is similar to a graph's adjacency matrix.

2.3 Sparse Compressed Formats

Due to their high number of zero elements, sparse matrices are inefficient to store raw. Consequently, many different storage formats have been invented to represent them in more efficient ways. Note that the example figures presented below are scaled down for the sake of the explanation. At this scale they do not offer any meaningful space saving, but in practice they are much more efficient.

2.3.1 CSR Format

Compressed Sparse Row (CSR) consists of three arrays VAL, COL_IDX, and ROW_PTR [22]. The format stores the matrix in row major order. Given an $m \times n$ matrix

M with nnz non-zero elements, VAL is the length of nnz and stores the non-zero elements in order. COL_IDX stores the corresponding column indices of the non-zero values. ROW_PTR[i] points to the first element of row i and ROW_PTR[i+1] marks the end (see Figure 2.2). By compressing row-wise and using a row pointer, CSR enables each matrix vector multiplication to be performed as independent dot-products.

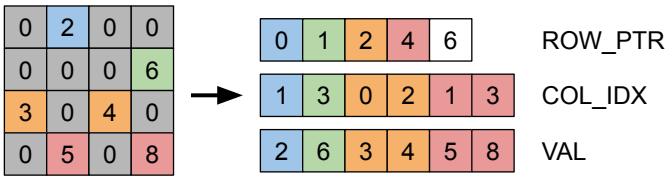


Figure 2.2: A sparse matrix and its CSR representation

2.3.2 CSC Format

Compressed Sparse Column (CSC) is the column-wise equivalent of CSR [20]. The matrix is read as column major and the three arrays are VAL, ROW_IDX, and COL_PTR (See Figure 2.3). VAL is the same as in CSR; a list of all non zero elements with ROW_IDX storing their row indices. COL_PTR stores the index in the VAL array to the first element of each column.

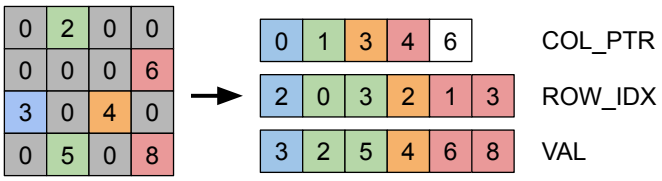


Figure 2.3: A sparse matrix and its CSC representation

2.3.3 COO Format

Coordinate format (COO) is similar to CSR in that it stores the value of each non-zero element in a VAL array and the column index of each non-zero in a COL_IDX array. Where the formats differ is in the third array, as CSR stores a pointer to each row while COO simply stores the row index of each non-zero value [23]. In this way COO is simpler to compress and decompress compared CSR but it takes up more space. For an $m \times n$ matrix with nnz non-zero values COO requires $3 * nnz$ elements compared to CSR which only requires $2 * nnz + m$ elements. See Figure 2.4 for an example of how a matrix is represented in the COO format.

2.3.4 DIA Format

The Diagonal format (DIA) is optimized for matrices where the non-zero elements are concentrated along the main diagonal. DIA consists of a DATA matrix and an OFFSET array. Each row of the DATA matrix represents a diagonal line from the

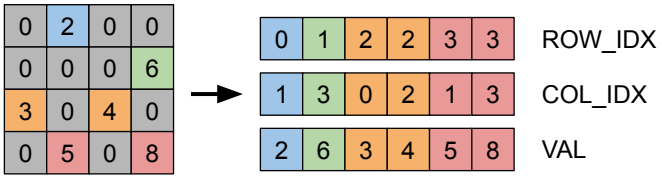


Figure 2.4: A sparse matrix and its COO representation

uncompressed matrix with at least one non-zero element. The values in the DATA matrix should be in the same column as they are in the uncompressed matrix. The OFFSET array contains the distance that the corresponding diagonal (row in the DATA matrix) is from the main diagonal. This means that some zeros will be encoded in the compressed format, as seen in Figure 2.5.

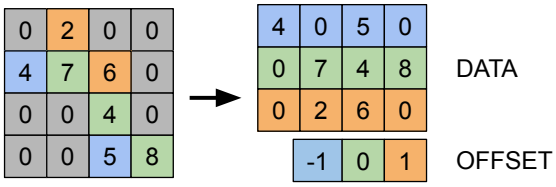


Figure 2.5: A sparse matrix and its DIA representation

2.3.5 SELL Format

ELLPACK is a well-known format for SpMV. For any given $M \times N$ matrix, the final representation is stored in an $M \times K$ array [24]. K represents the global max number of non-zero elements for the rows in the input matrix. Encoded rows that have less non-zeroes than K are zero-padded. The column indices are stored in a similarly structured array which also utilizes zero-padding.

The sliced ELLPACK format (SELL) is an extension of the ELLPACK compression format that performs the best where the number of non-zeroes per row is consistent [25]. In comparison to conventional ELLPACK, SELL partitions the sparse input matrix into slices S of consecutive rows [26]. Instead of keeping a global parameter K, each slice uses its own ELLPACK representation within itself. This means that the max number of non-zeroes K can vary across the different slices.

The final representation of the input matrix is stored in three separate arrays: COL_IDX, VAL and SLICE_START. VAL contains the non-zero entries and COL_IDX stores the corresponding column indices for the values in VAL. Additionally, SLICE_START holds the offset of the starting position of each slice within the COL_IDX and VAL arrays. With $S = 1$, the format is equivalent to the CSR format. SELL performs better than standard ELLPACK due to keeping a local ELLPACK encoding within each slice. This significantly reduces the padding overhead, since it does not encode according to the global max K for each individual row. Figure 2.6 shows a sparse matrix and its SELL representation with two slices, $S = 2$.

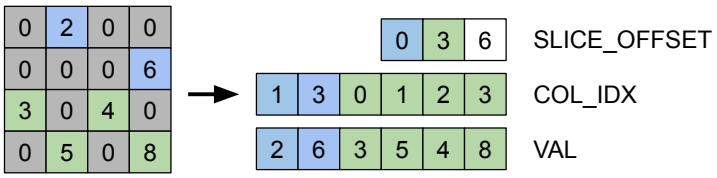


Figure 2.6: A sparse matrix and its SELL representation

2.3.6 Custom Compressed Formats

Several prior works have proposed novel formats or modified already well established ones to reduce memory bandwidth and increase the overall efficiency of sparse computation. Fowers et al. [27] focuses on making sparse matrix-vector multiplication (SpMV) more efficient by proposing a new architecture and a new encoding format that modifies the well known Compressed Sparse Row (CSR) format. It discusses the limitations of CSR, stating that its nature of storing rows sequentially makes parallelization difficult as each row has a different number of non-zero elements, which can cause load imbalance. Their solution Compressed Interleaved Sparse Row (CISR) divides the memory bus into fixed slots, one per processing element (PE) and use pre-arrangement so that each PE gets the correct data on arrival. This makes parallelization easier and simplifies hardware-complexity substantially.

Parravicini et al. [28] proposed a new BS-CSR format for FPGA accelerators equipped with High Bandwidth Memory (HBM) which has the small memory footprint of CSR and the streamability of the Coordinate Compressed Format (COO). It mentions that CSR is "unsuitable for fully-pipelined streaming FPGA designs", a limitation also identified by Fowers et al. [27]. The BS-CSR format fixes this issue by ajusting the way the matrix data is streamed. Their implementation changes the width to align with the 512-bit HBM packet width, making each packet contain its own values, indices and row pointer information. It can also pack more non-zero elements per packet as the bit-length encodings for the column index and row pointers are reduced. This format alongside a hardware architecture to exploit it is the main contribution. The HBM board proved to be 100x faster than a multi-threaded CPU and 2x faster than a GPU. Furthermore, the power-efficiency was 14.6x better than the GPU.

More recently, SPASM presented by Wu et al. [29] showed a promising result of 2.81x speedup when compared to state-of-the-art SpMV accelerators. It is a hardware-software framework and works by extracting information such as memory access patterns of the input matrices. More specifically, it tiles the matrix into 4x4 blocks and identifies the 16 most common access patterns across the blocks. Each individual block is encoded with its position in the matrix as well as its pattern ID which avoids storing every non-zero element. This process requires offline pre-processing. As mentioned, the speedup is promising, but a performance evaluation reveals that the pre-processing step comes with extensive overhead in the range of 10 000-20 000x of the execution time.

Another very recent publication presented Coruscant and was developed by Joo et

al. [30]. It uses a GPU Kernel and Sparse Tensor Core to advocate for unstructured sparsity in efficient LLM inference. It is GPU based and introduces a novel blocked format that uses a 64 bit long bitmap to encode the positions of the non zero elements in each block, along with an offset array that keeps the index of the fist non zero in each block. The paper modifies the tensor cores in the GPU to support operations on the bitmap compressed format without having to decompress it. Crouscant shows promising results with a 2.75x speedup over Nvidia's standard library cuBLAS.

2.4 SpMM Algorithms

Many algorithms for SpMM have been used throughout the literature. This section describes the three most common such algorithms. The examples below use the matrices A, B, and C where $C = A \times B$, A is sparse, and B is dense.

2.4.1 Gustavson's Algorithm

Gustavson's algorithm [31] is one of the most commonly used SpMM algorithms, and for good reasons. It is fast, highly parallelizable, and works well with compressed sparse formats, particularly CSR. Figure 2.7 provides a visual representation of how Gustavson's algorithm works.

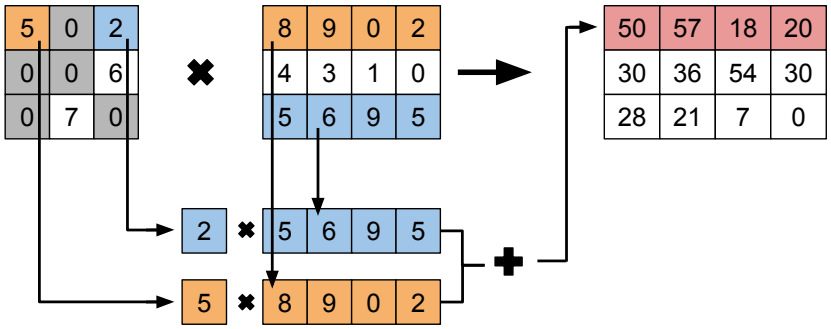


Figure 2.7: Gustavson's algorithm for SpMM

Gustavson calculates each row of the result matrix C_i as the sum of each non-zero element of the sparse row A_i multiplied by the row with index k of the dense matrix B_k. Which can be expressed as:

$$C_i = \sum_k A_{i,k} \cdot B_k \qquad \forall k \mid A_{i,k} \neq 0$$

Figure 2.8 shows a simplified pseudo-code for Gustavson's algorithm. It assumes A is a sparse matrix in CSR format. M is the number of rows in A, N is the number of columns in B and C, and nnz is the number of non-zero elements in A.

2.4.2 Inner Product

The inner product algorithm is similar to the regular matrix multiplication algorithm with the `ijk` loop order where each element of the result matrix is computed as the

```
// Sparse operand matrix A in CSR format
row_ptr[M]
col_idx[nnz]
val[nnz]

B[K][N] // Dense operand matrix B
C[M][N] // Dense result matrix C

// For each row of A
for i = 0 .. M-1:
    acc[0..N-1] = 0

    // For each nnz in row i
    for p = row_ptr[i] .. row_ptr[i+1]-1:
        k = col_idx[p]
        a = val[p]

        // For each column of B
        for j = 0 .. N-1:
            acc[j] += a * B[k][j]

    C[i][:] = acc
```

Figure 2.8: Pseudo-code for Gustavson's algorithm

dot-product of the corresponding row and column of the input matrices. To be efficient, inner product requires the sparse matrix A to be stored in the CSR format and the dense matrix B to be stored in column-major order (see Figure 2.9). Inner product has poor input matrix reuse as each element is read multiple times. On the other hand, output reuse is great as there are no partial results or accumulation steps each element of the result matrix is computed in one step.

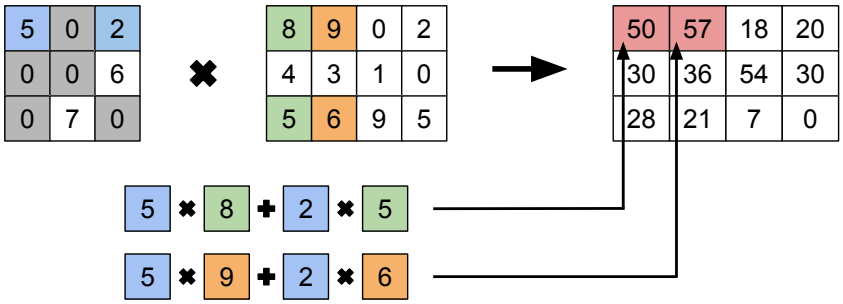


Figure 2.9: The inner product algorithm for SpMM

Inner product calculates each element of the result matrix $C_{i,j}$ as the sum the products of $A_{i,k} \cdot B_{k,j}$ for all non zero elements in A_i. This yields:

$$C_{i,j} = \sum_k A_{i,k} \cdot B_{k,j} \qquad \forall k \mid A_{i,k} \neq 0$$

Figure 2.10 contains pseudo-code for the inner product algorithm. It assumes that A is stored in CSR format and that B is stored in column-major order. M is the number of rows in A, N is the number of columns in B and C, and nnz is the number of non-zero elements in A.

```
1  // Sparse operand matrix A in CSR format
2  row_ptr[M]
3  col_idx[nnz]
4  val[nnz]
5
6  B[N][K] // Dense operand matrix B in column-major order
7  C[M][N] // Dense result matrix C
8
9  // For each row of A
10 for i = 0 .. M-1:
11     // For each column of B
12     for j = 0 .. N-1:
13         acc = 0
14         // For each nnz in row i
15         for p = row_ptr[i] .. row_ptr[i+1]-1:
16             k = col_idx[p]
17             acc += val[p] * B[j][k]
18         C[i][j] = acc
```

Figure 2.10: Pseudo-code for the inner product algorithm

2.4.3 Outer Product

The outer product algorithm for SpMM is similar to the standard GEMM algorithm with the `ikj` loop order. Every iteration of the outer loop multiplies a column of the sparse matrix with a row of the dense matrix producing a partial result matrix. Outer product expects the sparse matrix to be stored in the CSC format and the dense matrix to be read in row-major order. This means that, similarly to Gustavson, the computation consists of two steps: one multiplication step and one accumulation step. Zero-elements in the columns of the sparse matrix can be skipped easily. See Figure 2.11. Outer product has great input matrix reuse as each column of the sparse matrix and row of the dense matrix is only used once. However it is less space efficient as it has to store all the partial results.

Outer product calculates the result matrix C as the sum the cross product of $A_k \times B_k$ skipping the multiplications involving zero elements from A. Hence, we obtain:

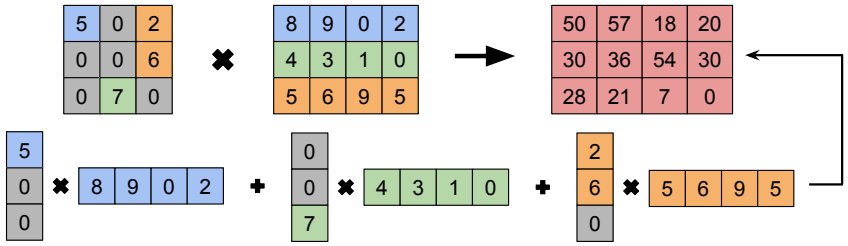


Figure 2.11: The outer product algorithm for SpMM

$$C = \sum_k A_k \times B_k \qquad \forall k \mid A_{i,k} \neq 0$$

Figure 2.12 contains pseudo-code for the outer product algorithm. It assumes that the sparse matrix A is stored in CSC format and that B is read in row-major order.

```
1  // Sparse operand matrix A in CSC format
2  col_ptr[K]
3  row_idx[nnz]
4  val[nnz]
5
6  B[K][N] // Dense operand matrix B in row-major order
7  C[M][N] // Dense result matrix C
8
9  // For each column of A
10 for i = 0 .. K-1:
11     Cp = [M][N] = 0 // Partial result
12     // For each non zero in column i
13     for p = col_prt[i] .. col_ptr[i+1]
14         // For each element of row p of B
15         for k = 0 .. K-1:
16             Cp[p][k] = val[p] * B[i][k]
17     C += Cp
```

Figure 2.12: Pseudo-code for the outer product algorithm

2.5 Hardware Concepts

This section contains an overview of hardware concepts relevant to the thesis.

2.5.1 Instruction Set Architecture

The Instruction Set Architecture (ISA) is the interface between hardware and software that specifies available CPU instructions and how they interact with the hard-

ware. Thus, by extending the ISA, new instruction encodings are added that represent application-specific operations. This makes the processor highly customizable as specific functionalities can be integrated. The custom instructions added in this thesis are covered in Chapter 3.

RISC-V is a free and open standard ISA which allows anyone to modify and develop processors for without permission or royalties. This has lead to RISC-V becoming popular among researchers and companies when developing accelerators or processors [32].

The design philosophy of a reduced instruction set architecture such as RISC-V is to simplify the processor. Only the most commonly used instructions are implemented and the data path is then optimized for this subset of instructions. If more complex functions are needed they can be accomplished by chaining together these simpler instructions [33]. This means that RISC-V features fewer, faster, and less complex instructions than Intel's x86 architecture, while requiring more instructions to accomplish complex tasks.

2.5.2 Systolic Array

A systolic array (SA) is a homogeneous and tightly coupled network of functional units (FU) usually laid out in a grid formation. Which rhythmically computes results and sends data throughout the network [34]. The data is consumed by an FU and the result passed to the next FU in the network. The functional units are typically simple arithmetic units often hard-coded to specific functions such as multiply accumulate. The flow direction in the network is configured to support specific calculations.

In the case of dense matrix multiplication the operand matrices are continuously fed through the SA [35]. Matrix A from the left and matrix B from the top. To ensure that the elements are in the right place at the right time the data is "skewed". The skewing step separates the rows of A and feeds them into the SA one each cycle. The same is done for the columns of B. See Figure 2.13 for a scaled down example of matrix multiplication in a systolic array.

2.6 Quadrilatero Accelerator

Quadrilatero [9] is a systolic array co-processor designed by the Embedded System Laboratory (ESL) at EPFL to be compatible with the CV32E40X RISC-V core by OpenHW Group [36]. Quadrilatero works with 4×4 tiles and includes instructions to load tiles, store tiles, multiply accumulate (MAC), and zero out a register. It has its own decoder which can dispatch instructions to three different execution units: Load Store Unit (LSU) for memory operations, Systolic Array (SA) for GEMM operations and Permutation Unit to reset matrix registers. Quadrilatero connects to the host core via the CORE-V-X-IF extension interface and accesses memory via the OBI protocol. Figure 2.14 displays a block diagram of the Quadrilatero accelerator. It contains four main blocks, controller, LSU, SA, and permutation

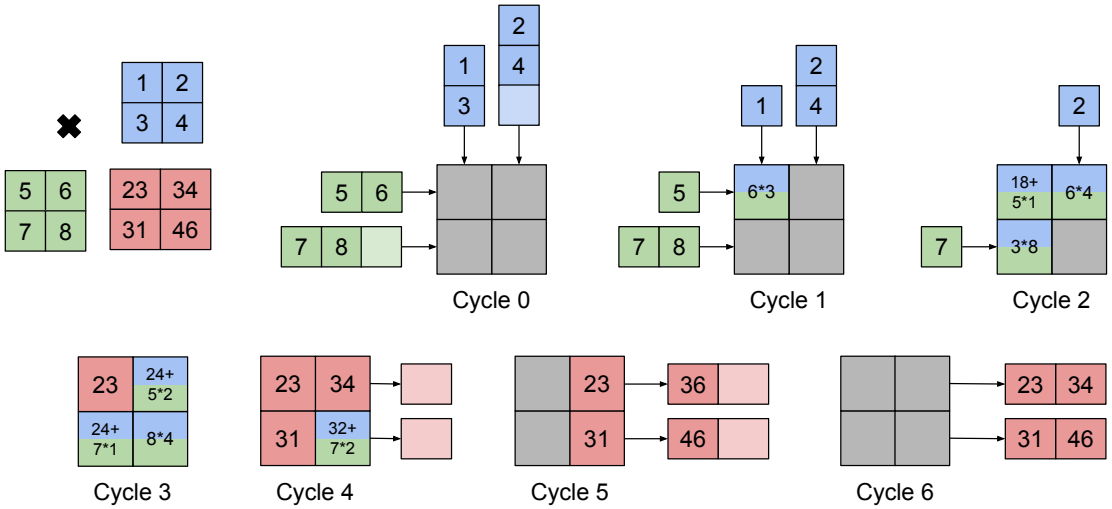


Figure 2.13: Matrix multiplication in a systolic array

unit. The LSU handles moves memory between the host and the matrix register file. The SA is composed of three stages where the data is loaded, then moved through the systolic array and finally stored back. The permutation unit simply zero fills a matrix register. Finally the controller takes care of communication with the host, decoding and dispatching instructions, as well as keeping track of memory access requests.

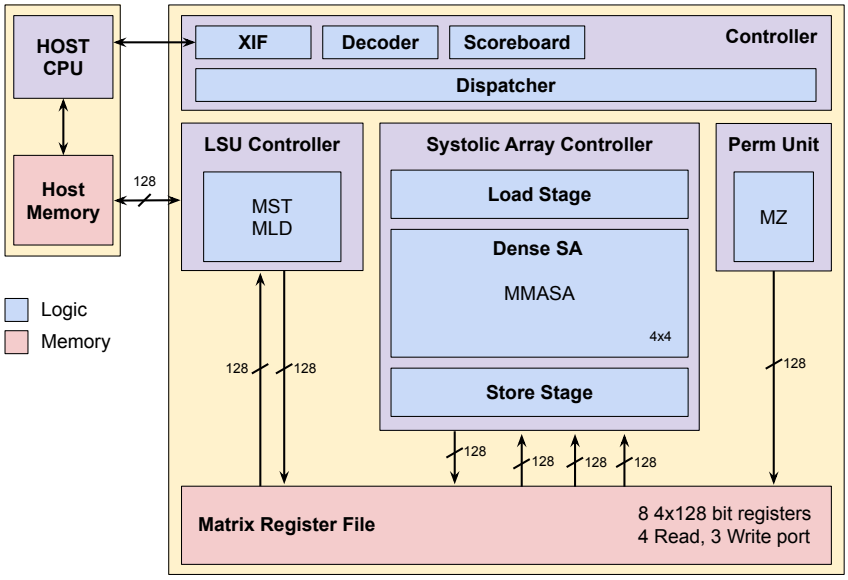


Figure 2.14: Hardware block diagram of Quadrilatero. Credit: Adapted from Quadrilatero [9]

2.6.1 Accelerator Interface

Quadrilatero is compatible with the CORE-V-X-IF accelerator interface by OpenHW Group [37] which simplifies the task of integrating custom instructions into the pro-

cessor. Any instructions that are not recognized by the host core will be sent over the extension interface to the accelerator. It is then up to the accelerator to filter out irrelevant instructions and to execute relevant ones. Through the interface, the accelerator gains access to the host core's register file and memory. In Quadrilatero, the matrix tiles are loaded from the host memory and the host's scalar registers are used to hold the relevant memory addresses. This interface will be leveraged to develop a simple test bench which will be used to benchmark the accelerator.

2.6.2 Memory Interface and Register File

Quadrilatero interacts with the host memory through the Open Bus Interface (OBI) protocol defined by OpenHW Group [38]. OBI is a simple, low overhead, request and response protocol. Quadrilatero uses four 32 bit memory ports operating in parallel for a total of 128 bits transferred per cycle. This is equal to one row of a matrix register. Internally, the memory interface is treated as a single 128 bit wide bus and the requests are only split when forwarded to the OBI interface. The OBI protocol consists of two phases: the address phase and the response phase. During the address phase the accelerator asserts `req` (request) high along with `addr` (address), `we` (write enable), `be` (byte enable), and `wdata` (for stores). The host asserts `gnt` (granted) when it accepts the request. After this, the accelerator is free to change these signals. During the response phase the host asserts `rvalid` for exactly one cycle. For reads, `rdata` is valid at that point.

The accelerator's Matrix Register File (MRF) holds eight 4×4 matrix registers. Given that each element is a 32 bit word each matrix register holds 512 bits, the entire MRF holds a total of 4 kb. The MRF provides three read ports and two write ports each 128 bits wide. A scoreboard is used for hazard prevention where each row of each register has an access queue. Only the instruction with the same id as the top of the queue is allowed to access that row. In practice most instructions lock all rows of a register.

2.6.3 Load Store Unit

Quadrilatero's Load Store Unit (LSU) is the accelerator's functional unit for its two memory instructions, `MLD_W` which loads a 4×4 tile of a matrix into one of the matrix registers, `MST_W` which stores the contents of a matrix register back into memory. Both instructions use scalar registers to hold the base address of the matrix in memory and the matrix's stride or its number of columns. The LSU is the bridge between the OBI memory interface and the MRF.

2.6.4 Systolic Array

The systolic array (SA) is pipelined in three stages Feed Forward (FF), Feed Skewed (FS), and Drain (DR). The FF stage reads one row per cycle from all three operand registers and sends the data to the skewer. The skewer sits between the FF and FS stages and is responsible for delaying each elements entry into the SA by the correct number of cycles. The FS stage is the main computation stage. Matrix A flows

horizontally through the SA, Matrix B and the accumulator results flow vertically. Finally the DR stage drains the SA and de-skews the result allowing it to be written back to the matrix register. Each stage takes four cycles and is fully pipelined meaning that three instructions can occupy the SA simultaneously, one per stage. When fully occupied, the SA achieves a peak throughput of 16 MACs/cycle [9].

3

Design

This chapter gives an in-depth view into the architectural design of QuadriSparse and presents the custom instruction set as well as the implementation details of the hardware extensions.

3.1 QuadriSparse Architecture

Taking into consideration the short timeline of the thesis it was not possible to build an accelerator from scratch. Therefore it was important to find a good GEMM accelerator to build on. Quadrilatero (see Section 2.6) provided a solid foundation for this project and most of the infrastructure could be reused with only small modifications. As a consequence, the new instructions listed below still operate on 4×4 tiles.

A benefit to starting with a GEMM accelerator and expanding it to also support sparse operations is that the resulting co-processor supports both sparse and dense operations. This makes for a far more flexible end product, especially considering that the SpMM operations produces a dense result matrix.

3.1.1 Design Goals

The design goals which govern the architecture laid out below are as follows:

- The accelerator should excel at unstructured sparsity
- The design should focus on high sparsity levels
- The architecture should not be too complex to implement

A more detailed discussion around each of these choices can be found in Section 6.4.

3.1.2 SpMM Algorithm

The first major design decision taken was which SpMM algorithm to chose. The three main contenders are Inner product, Outer product, and Gustavson's algorithm. Inner product and Outer product works similarly to regular matrix multiplication with different loop orders. Gustavson on the other hand does row-wise scalar vector multiplication, see Section 2.4.1, this is simple to implement and works well with the CSR format.

3.1.3 Dataflow / Execution Model

The planned execution model for QuadriSparse is as follows: a sparse tile load instruction which loads a tile of the sparse matrix containing four elements from the CSR values and column pointers arrays is loaded into a matrix register. The column pointers are used by a dense tile load instruction to load the relevant rows of the dense matrix into another register. Finally a sparse matrix multiply and accumulate instruction calculates the scalar vector product of each pair of non zero elements and dense rows and the results vectors are summed. This produces a partial result vector segment of shape 1×4. To produce a full result vector segment the algorithm loops over the sparse matrix row. To produce a complete result row it loops over the width of the dense matrix. Finally it loops over all the rows of the sparse matrix to produce the final result.

3.2 New Instructions

Considering the architecture outlined above, three new instructions were needed. A sparse tile load instruction called `SPLD_W`, a dense tile load instruction, `DLD_W`, and finally `SPMAC_W` sparse multiply and accumulate.

3.2.1 SPLD_W: Sparse Tile Load

The purpose of the `SPLD_W` instruction is to load a sparse tile from the CSR format into a destination matrix register. It does so by issuing two memory requests: one to the COL_IDX array and one to the VAL array, storing up to four elements from each in the first and second row of the register. The remaining rows are zero-padded. If the number of NNZ elements is less than four, any unfilled elements within the first two rows are also zero-padded. The instruction is defined as follows:

```
SPLD_W md, nnz, rs1, rs2
```

where `md` is the destination matrix register, `rs1` and `rs2` are scalar register holding the base address of VAL and COL_IDX respectively, and `nnz` is a 3-bit immediate used for specifying the number of non-zeroes to load. This immediate is only necessary for the occasional zero-padding of the top rows, as mentioned earlier.

The instruction issues two sequential 128-bit memory requests over the OBI-bus. It is handled by the register LSU, which manages memory requests and writes results directly into the register file. The zero-padding of the bottom rows are overlapped with the memory requests which hides the memory latency. Once both responses arrive, data is written to row 0 and 1 respectively. An overview of the `SPLD_W` data flow is shown below in Figure 3.1

3.2.2 DLD_W: Dense Tile Load

The `DLD_W` instruction loads four elements of each of the relevant rows of the dense matrix B into a matrix register to be consumed by the `SPMAC_W` instruction. According to Gustavson's algorithm the rows of the dense matrix B are selected by the

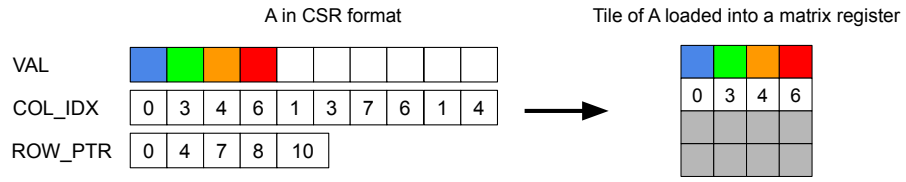


Figure 3.1: SPLD Instruction flow

column indices from the tile of the sparse matrix produced by the `SPLD_W` instruction.

This instruction is designed to feature a small finite state machine (FSM). It begins by loading the index row of the sparse tile into a local buffer. Then it proceeds to use the indices to load the a row from the dense matrix in memory and store it in a matrix register. This repeats for all four rows.

The instruction is defined as such:

```
DLD_W md, msp, rs1, rs2
```

Where `md` is the destination register, `msp` is the register containing the sparse tile produced by the `SPLD_W` instruction, `rs1` and `rs2` are scalar registers that contain the base address of the dense matrix and its row stride, respectively. See Figure 3.2 for a visual representation of how this instruction works.

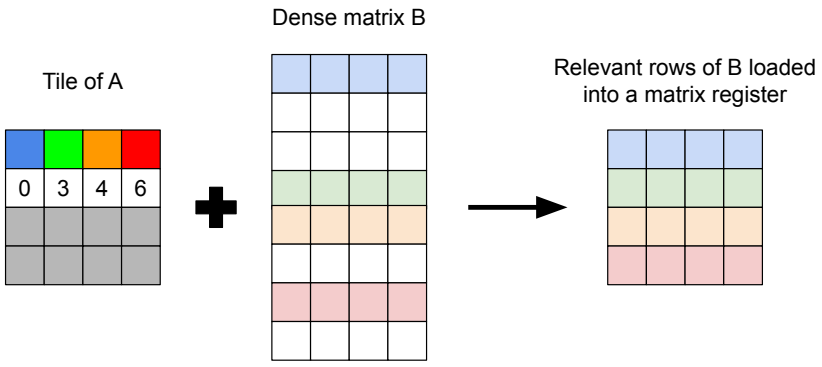


Figure 3.2: DLD instruction logic

3.2.3 SPMAC_W: Sparse Multiply Accumulate

The `SPMAC_W` combines the result of the previous two custom instructions in order to compute one output row of a sparse-dense matrix multiplication. It is defined as:

```
SPMAC_W md, ms, mw
```

where `md` is the accumulator register, `ms` is the sparse register loaded by `SPLD_W`, and `mw` is a dense register loaded by the `DLD_W` instruction. For each sparse value in `ms`, the instruction broadcasts that value across four parallel multipliers, each multiplying the value with the corresponding dense row in `mw`. The results are

computed in one single cycle and then added to the value of `md` and written back. This produces one output row for the sparse dense product. The dataflow as well as the accumulators can be seen in Figure 3.3.

From a row-wise perspective this operation is four parallel scalar vector multiplications and a final accumulation. However it might be clearer to visualize it from a column-wise perspective where the operation becomes the dot product of the sparse row segment and the dense tile column.

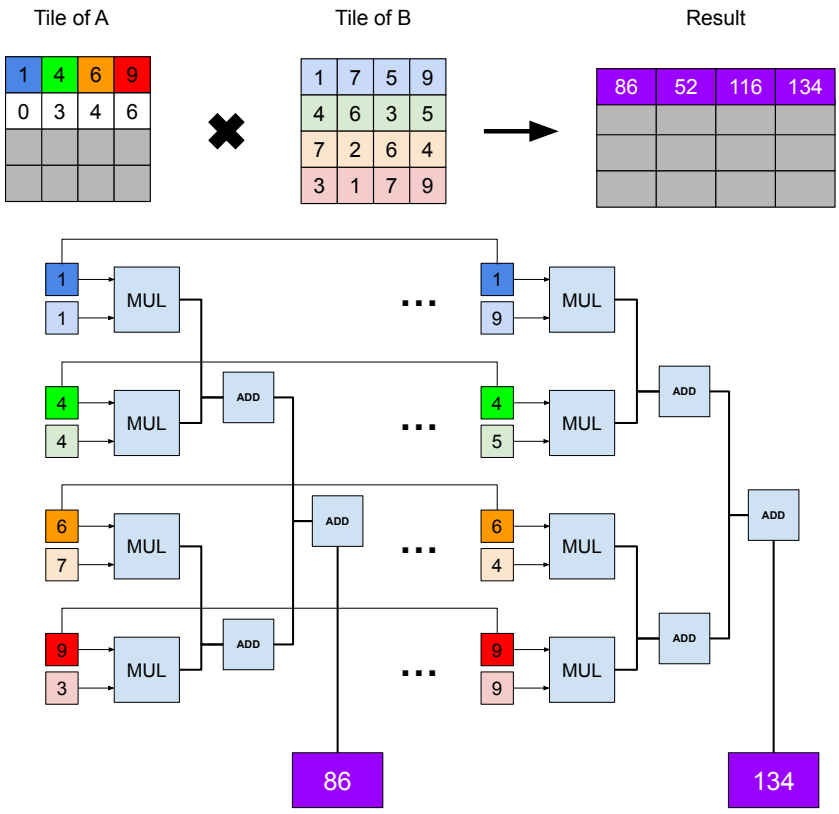


Figure 3.3: SPMAC Instruction flow with the 16 multipliers and 4 adder trees

3.2.4 Instruction Encodings

The RISC-V encoding of the new instructions is shown in Table 3.1. Bits 6-0 represent the `opcode` (CUSTOM 1), bits 31-25 are `func7`, finally bits 14-12 are `func3`. `nnz` is a 3-bit immediate that holds the number of non zero values to load from the sparse matrix. `ms1`, `ms2`, and `md` are matrix registers.

3.3 Hardware Extensions

New hardware was added to Quadrilatero in order to support the new custom instructions. Figure 3.4 shows a block diagram of the QuadriSparse accelerator, adapted from the Quadrilatero paper [9]. Note that the areas of the modules are not to

Table 3.1: Binary encoding for the QuadriSparse instructions

mnemonic	func7					func3			opcode
	31-25	24	23-21	20-18	17-15	14-12	11-10	9-7	6-0
`SPLD_W`	0010000	0	000	000	nnz	000	10	md	0101011
`DLD_W`	0001000	0	000	000	ms1	000	10	md	0101011
`SPMAC_W`	1111000	0	ms1	ms2	md	000	10	000	0101011

scale. The green modules represent the hardware added by this thesis, the remaining modules and structures are reused from Quadrilatero, in some cases with minor modifications. Section 5.1 estimates the area overhead of the added instructions using FPGA synthesis.

The simple and lightweight `SPLD_W` instruction was implemented inside Quadrilatero's LSU module. The `DLD_W` instruction on the other hand was more complex, it needed to first fetch the index row from the sparse tile and then start fetching non-contiguous rows from memory. It was decided to break out this instruction into a separate module for the sake of readability and code quality. The new module still uses Quadrilatero's LSU controller, meaning that the area overhead of this decision is negligible.

The `SPMAC_W` instruction is implemented with dedicated multiplier and adder units within the systolic array module, we call this the Sparse Processing Engine (SPE). This decision was made as a tradeoff between area cost and latency. It allowed the `SPMAC_W` instruction to bypass the FS stage and do all of the multiplications and accumulations in one cycle. By implementing the SPE within the systolic array module, it can reuse the same read ports and control signals without the need to add additional hardware interface. The addition of 16 multipliers and four adder trees introduced a small area cost but simplified the implementation. SpMM with Gustavson's algorithm is a very different operation compared to matrix multiplication. Implementing efficient support for both in the same systolic array would have been time consuming and difficult, with a risk of slowing down both the sparse and the dense operation.

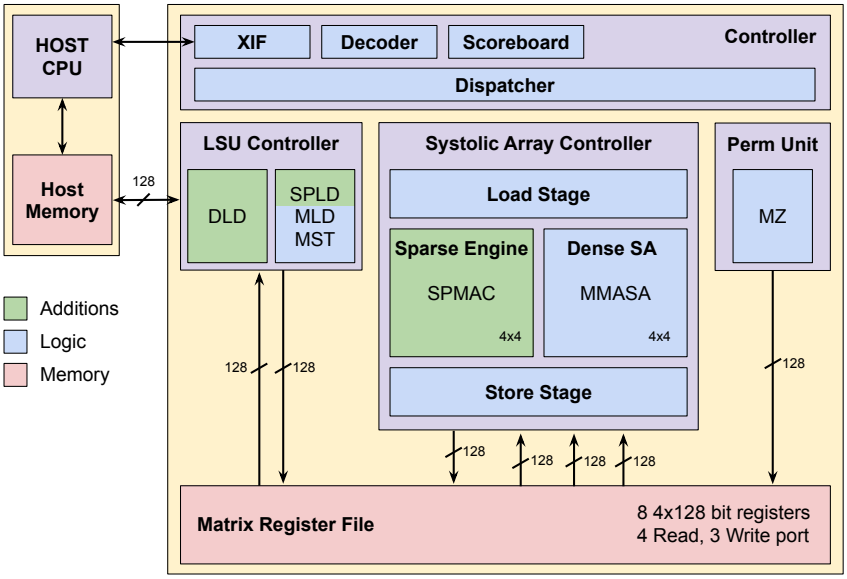


Figure 3.4: Hardware block diagram of QuadriSparse, note: areas are not to scale. Credit: Adapted from Quadrilatero [9]

4

Evaluation Methodology

This chapter details the methodology used to evaluate the accelerator both in terms of performance through simulation and in terms of area overhead through functional synthesis. The accelerator does not work standalone. It requires a host which can issue instructions and handle system memory. In a production environment this host would be a real CPU core, such as the CV32E40X core by OpenHW Group. In a testing and development environment, however, a simple "test bench" is preferable. The test bench only needs to implement the functions required by the accelerator and some performance counters for evaluation; it can thus be much more basic than a real CPU.

4.1 Test Bench

To facilitate testing of the accelerator during development a simple test bench was written in non-synthesizable SystemVerilog. The source code for this test bench as well as detailed instruction for how to run simulations is available in the project's GitHub repository[1]. The test bench interfaces directly with QuadriSparse's extension interface, see Section 2.6.1. The test bench loads five data files into memory which contain the different CSR arrays of the sparse matrix, the dense matrix, as well as the result of the multiplication to use as reference.

The simulated memory is flat array of 128 bit rows each row containing four packed 32 bit words, this is done to match the OBI interface that is used by the accelerator, see Section 2.6.2. The simulated memory provides access with a 1 cycle latency.

The test bench can be switched between sparse and dense operations via a runtime argument to facilitate simple benchmarking. The sparse multiplication is done via Gustavson's algorithm, and the dense multiplication is done as described in the Quadrilatero paper [9]. After the multiplication is complete the result is verified against the reference matrix that was provided together with the operand matrices.

The dense matrix multiplication flow implemented in this test bench was derived from Quadrilatero [9]. The kernel computes an 8×8 output tile of the matrix product using four accumulator registers, one per 4Œ4 sub-block. For each step along the K dimension, two 4×4 blocks of A and two of B (stored transposed) are loaded and multiply-accumulated into the four sub-tiles, reusing each loaded block

[1] QuadriSparse GitHub: https://github.com/nikerl/quadrisparse

```
for row = 0 .. len(ROW_PRT):
    if row is empty: continue

    for col_tile_start = 0 .. N_COLS/4, col_tile_start+4:
        tiles_in_group = min(4, N_COLS/4 - col_tile_start)
        // Zero accumulators
        mzero acc[0..tiles_in_group]
        // Walk sparse row
        for val_ptr = row_start .. row_end:
            chunk = min(nnz_remaining, alignment_limit, 4)
            spld.w sr0, &VAL[val_ptr], &COL_IDX[val_ptr]   // load sparse chunk
            val_ptr += chunk
            // For each tile in the group load and mac
            for tile = 0 .. tiles_in_group:
                col_tile_idx = col_tile_start + tile
                dld.w  dr[tile%2], &B[col_tile_idx * 16]  // load dense tile
                spmac  sr0, dr[tile%2], acc[tile]         // multiply accumulate
        // Store output tile group
        for tile = 0 .. tiles_in_group:
            col_tile_idx = col_tile_start + tile
            mst.w acc[tile], &C[row * N_COLS + col_tile_idx * 16]
```

Figure 4.1: Sparse multiplication

across two MAC operations. Once K is exhausted, the four sub-tiles are written back to the corresponding quadrant of C.

The listing in Figure 4.1 contains pseudo-code for the sparse multiplication flow. This implementation of Gustavson's algorithm will produce one complete result vector of size 1×4 per iteration of the second loop. This is not the most efficient implementation of Gustavson however there are two constraints that force this implementations. Firstly, there are only eight matrix registers which have to be split between the sparse tile, the dense tiles, and the accumulator registers. Secondly, the store instructions, `MST_W`, are reused from Quadrilatero and they are not accumulative meaning that the entire result segment needs to be done in one iteration.

4.2 Benchmarking Setup

The benchmarking of the accelerator was done via a Bash script which loops through different combinations of sparsities and matrix sizes. Early testing found that sparsities under 70% were too slow to be relevant thus the sparsities tested were: 70%, 80%, 90%, 95%, 97%, and 99%. Eight matrix sizes between 16×16 and 2048×2048, doubling at each level, were tested. No matrices larger than than 2048×2048 were tested as simulation times became prohibitively long. All simulations were done

using Verilator v5.032, this is a state-of-the-art SystemVerilog simulator.

Each combination of the above was tested 5 times with different matrices and the arithmetic mean of the results was calculated. The dense matrix multiplication was executed once for each matrix size to use as a baseline for comparisons, one run was enough for the dense case as it is not effected by the randomness of unstructured sparse data.

The data used to verify the accelerator's correctness and to benchmark it was synthetic. Matrices were generated by a Python script to the desired size and sparsity. The generation was done as follows: First the appropriate number of non-zero elements were generated, for example if the matrix was specified to have 100 elements and 80% sparsity then 20 non-zero elements were generated. This ensured that the sparsity level was always correct. If the required number of non-zero elements was not an integer it was rounded down to the nearest integer. These non-zero elements were then placed randomly within the matrix by generating random column and row indices. If a generated position was already taken another position would be generated. The dense matrix was generated similarly but with all cells filled, a result matrix used to verify the correctness of the operations was generated by simple software matrix multiplication.

These matrices were saved as flat text files where each line contained one element of the matrix encoded as a 32-bit hexadecimal number. This format was chosen for the sake of simplicity when testing. The ability to visually look through the data and to easily modify it was desired. In actual operation of the hardware accelerator the data would be loaded into memory as 32-bit integers.

4.3 Synthesis Setup

Synthesis of both QuadriSparse and Quadrilatero was done in Vivado 2025.2 by AMD Xilinx to estimate the area overhead introduced by the new instructions. The synthesis target was the Kintex-7 160T FPGA, specifically the "xc7k160tfbg484-3" model. The dependency management tool "Bender" was utilized to summarize the project into a tcl script which Vivado needs for to link dependencies during synthesis.

5

Results

This chapter outlines resource utilization results gathered from functional synthesis as well as the performance results from benchmarking QuadriSparse. The performance evaluation was done at different matrix sizes and sparsities and compared against the dense matrix multiplication instructions from Quadrilatero. A discussion of these results is presented in the next chapter. Two versions of the sparse accelerator were evaluated: an initial, unoptimized version (Section 5.3), and an optimized version with reduced cycle counts per instruction (Section 5.4). The complete data from the evaluation can be found in Appendix A.

As the non-zero elements are placed randomly throughout the matrix there may be some variations in the number of instructions executed from run to run. A five run arithmetic mean was used for all data points presented in this chapter to minimize these variations.

Throughout this chapter the performance of the accelerator is presented relative to a baseline "Dense". This refers to the performance of GEMM operations with the original Quadrilatero instructions. This baseline was chosen because it directly addresses the core research questions in the thesis (RQ 1): whether extending a small scale GEMM accelerator to support SpMM is worthwhile compared to using the existing dense instructions. While it would have been interesting to compare QuadriSparse against other instruction level accelerators, this was not possible due to time constraints. See section 1.4.2 for a more detailed explanation.

5.1 Synthesis Results

Both the original Quadrilatero and the extended QuadriSparse were synthesized in using AMD Xilinx 2025.2 to assess QuadriSparse's area overhead. The accelerators were synthesized for the Kintex-7 160T FPGA. The post-synthesis resource utilization for Quadrilatero and QuadriSparse are shown in Table 5.1 and Table 5.2 respectively. The most important metric is the number LUTs and DSP blocks used. Quadrilatero uses 63 132 LUTs while QuadriSparse needs 66 051, resulting in 4.6% overhead. The overhead for the number of registers as flip flops is 12%. For DSP-blocks the overhead is 23%, with Quadrilatero using 256 DSP block and QuadriSparse using 316 blocks. See Appendix B for the full resource utilization numbers.

Table 5.1: Post-synthesis resource utilization for *Quadrilatero* (Vivado 2025.2, device xc7k160tfbg484-3).

Resource	Used	Available	Utilization
Slice LUTs	63 132	101 400	62.26%
Registers (FFs)	13 683	202 800	6.75%
DSP blocks	256	600	42.67%
BUFG clocks	1	32	3.13%

Table 5.2: Post-synthesis resource utilization for *QuadriSparse* (Vivado 2025.2, device xc7k160tfbg484-3).

Resource	Used	Available	Utilization
Slice LUTs	66 051	101 400	65.14%
Registers (FFs)	15 351	202 800	7.57%
DSP blocks	316	600	52.67%
BUFG clocks	1	32	3.13%

5.2 Test Bench Verification

To verify that the test bench produces representative results and correctly emulates a real CPU core without adding latency the dense multiplication flow was compared against the data provided in the Quadrilatero paper [9]. The paper offered one relevant data point, that being the measured cycle count for completing the multiplication of two 64×64 matrices. Quadrilatero measured 17.7k cycles in this test. The test bench developed for this project measured 17.9k cycles using the same algorithm and problem size as Quadrilatero. This difference is within the margin of error of the sampling method used. Thus it has been verified that the test bench is comparable to the accelerator being attached to a real CPU core.

5.3 First Version

This section contains the results from the performance evaluation of the first, unoptimized, version of the accelerator. Figure 5.1 shows how the performance, relative to dense baseline, scales with matrix size and sparsity. The performance scales well with increased sparsity, especially at 99% which achieves up to 3x the performance of the baseline. The performance also scales with increased matrix sizes, however all but the 99% line has plateaued at the largest tested matrix size of 2048×2048. Matrices with 97% sparsity begins to outperform the baseline at sizes above 1024×1024 and achieves at best an 11% performance uplift with 2048×2048 matrices. The lower sparsity levels are slower than the baseline with the 95% level, for example, achieving only 0.68x the performance of the dense instructions.

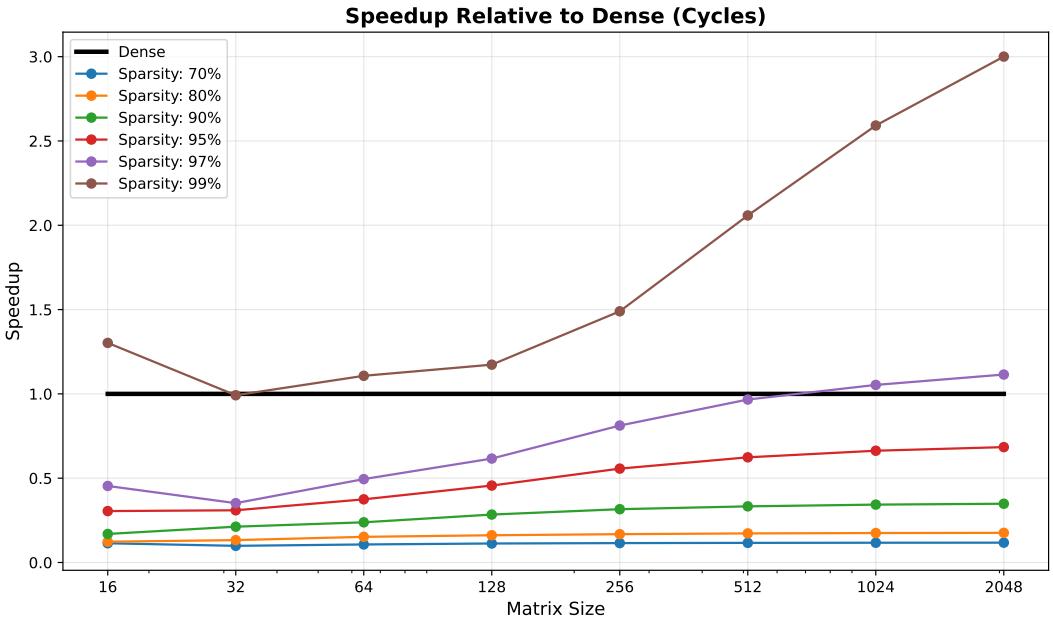


Figure 5.1: Speedup in number of cycles relative to dense, First version

5.4 Optimized Version

This section details the performance of the optimized accelerator. This version uses the same architecture as the first version, but with `SPLD_W` and `DLD_W` streamlined to reduce their per-instruction cycle count. For example through better overlap with memory latency and pipelined register writes. A detailed discussion of these optimizations can be found in Section 6.3.

The line graph in Figure 5.2 shows similar scaling both in terms of matrix size and sparsity as the first version, but with the performance shifted upwards in all cases. The peak performance increase measured was 6.60x at 99% sparsity and matrices of size 2048×2048. Both the 97% and 95% sparsities now comfortably outperform the dense baseline at matrix size above 128×128. They achieve a 2.5x and 1.5x performance increase at 2048×2048, respectively. The performance at 90% sparsity is close to the baseline achieving at most 0.78x the performance of Quadrilatero.

Figure 5.3 includes a gantt chart of the instructions issued during one iteration of the inner loop of Gustavson's algorithm, that is the instructions needed to process one row of the sparse matrix. For the sake of illustration the matrices in this example are 16×16 with 60% sparsity. This resulted in enough non zero elements in the first row to fill two `SPLD_W` instructions, each being followed by four `DLD_W` and `SPMAC_W` instructions respectively. The calculation starts with four `MZ` instructions to reset the accumulator registers, and ended with four `MST_W` instructions to store the result.

The `SPLD_W`, `DLD_W` and `SPMAC_W` instructions require 5, 9, and 10 cycles respectively. None of these instructions are pipelined, however there is still concurrency as the sparse processing engine and the load store unit can operate in parallel. This shows

that the DLD_W and SPMAC_W instructions dominate the execution time. A stall can be observed between each instruction, 1 cycle after each SPLD_W and SPMAC_W instruction and 2 cycles after each DLD_W instruction.

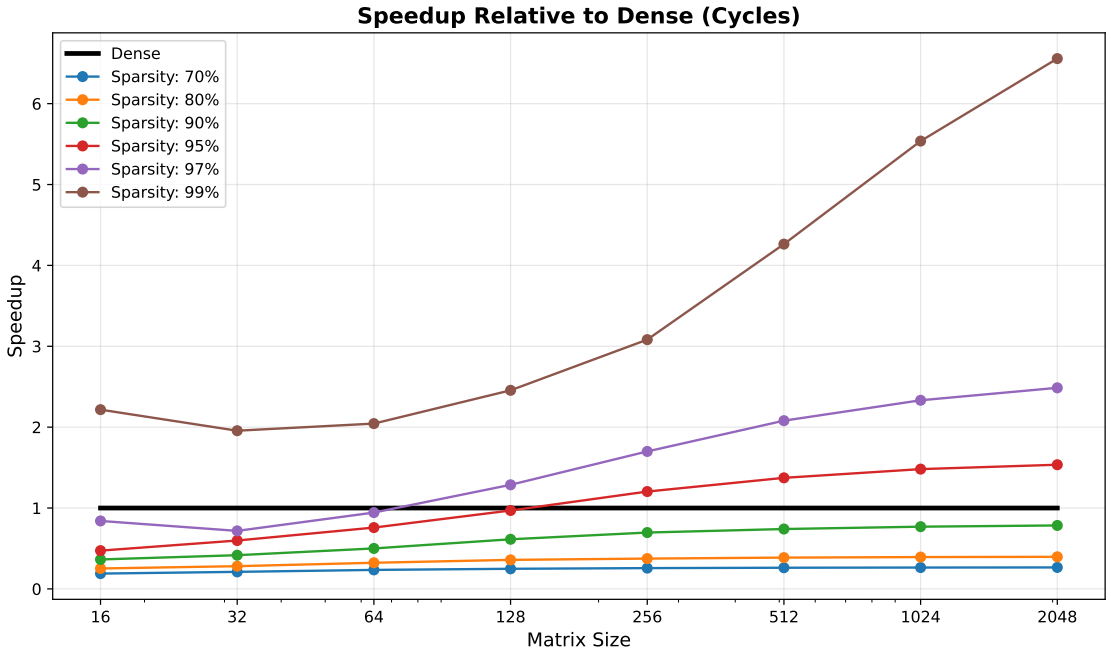


Figure 5.2: Speedup in number of cycles relative to dense, Optimized

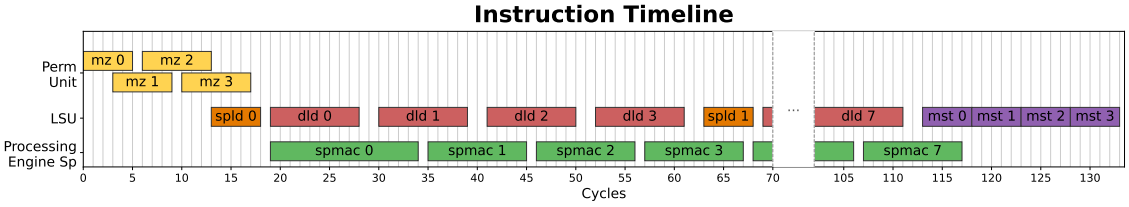


Figure 5.3: One iteration of Gustavson's inner loop, 16 × 16 matrix

5.5 Architecture

The architecture of the accelerator, see Chapter 3, determines what each instruction does and how they work together to complete the SpMM computation. This effectively determines the number of instructions needed to complete any one operation. Figure 5.4 shows the decrease in number of instructions needed to complete a multiplication of two matrices of varying sizes and sparsities as compared to dense operations on the original Quadrilatero. As shown by the chart, the number of instructions scale inversely with increased sparsity, with 99% sparsity requiring only 6% of the instructions compared to dense. The results also scale by matrix size to a lesser extent. This scaling appears to plateau with the threshold depending on the sparsity. Note that the X-axis is log scaled, meaning that plateauing is less visible compared to a normally scaled chart.

As seen in the graph all data points except two appear to follow the same curve. These outliers are the 97% and 99% sparsity levels at the workload size of 16×16 and are a result of the very low number of non-zero elements in these runs.

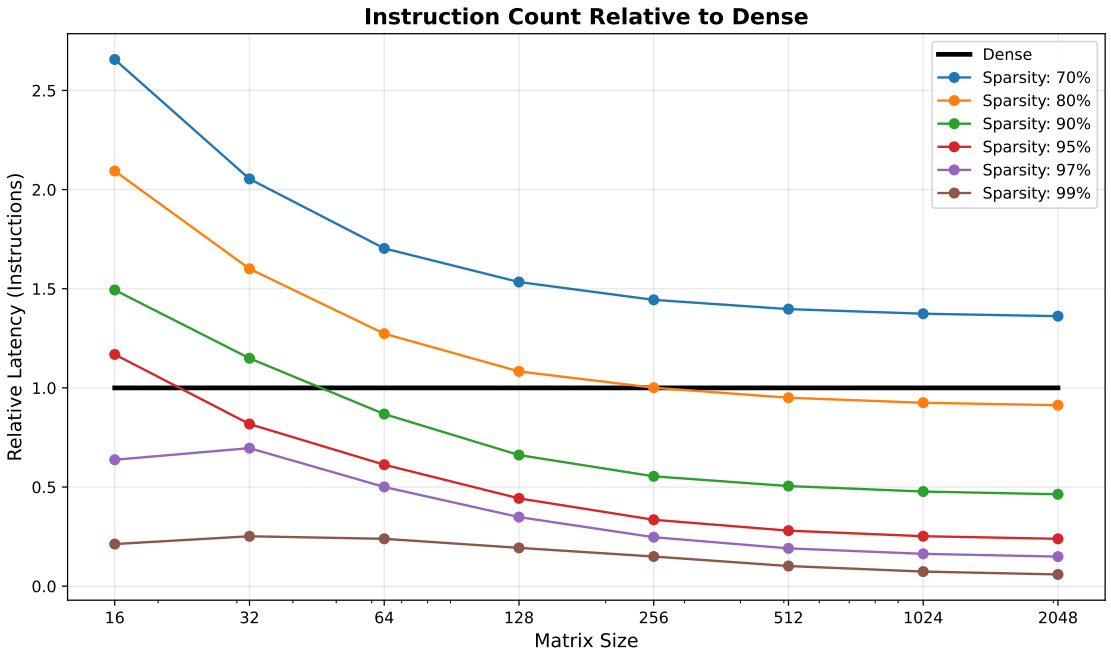


Figure 5.4: Number of instructions relative to dense, lower is better

6

Discussion

This chapter discusses the result presented in Chapter 5 and analyzes the performance of the proposed implementation. Furthermore, a breakdown of the design choices, architectural trade-offs, performance trends and limitations of the design will be presented. Lastly, this chapter propose improvements of the implementation and areas of future work.

6.1 Summary of Findings

This thesis sought to integrate support for sparse inputs into the systolic array co-processor Quadrilatero. This extended version is what we call QuadriSparse and it retains full support for the Quadrilatero instructions. The results show that the proposed implementation outperforms Quadrilatero at sparsity levels exceeding 95% and scales well with increased sparsity, reaching a peak speedup of 6.6x at 99% sparsity. This was achieved with an area overhead of only 4.6% in LUTs and 23% in DSP-blocks. Taking into account the goals presented in Section 1.3, this thesis was a success.

6.2 Performance Interpretation

As mentioned, the optimized version reaches a peak speedup of 6.6x and outperforms Quadrilatero at any sparsity levels from 95% and onward. The use of Gustavson's algorithm and CSR changes the dataflow for SpMM in QuadriSparse and it differs from the standard dataflow in Quadrilatero. As a result the number of instructions increased significantly at very low sparsities (see Figure 5.4). The relative instruction count does improve with larger matrices but the high number of instructions at low sparsities are expected. When there are many non-zero elements, QuadriSparse doesn't benefit enough from the CSR format and Gustavson's algorithm as the number of skipped instructions is minimal. Sparsity levels above 95% is where we can observe that the reduction in computation ensures that the Gustavson algorithm becomes more efficient than using regular matrix multiplication.

Looking at the instructions and computation of SpMM in QuadriSparse, the `SPLD_W`, `DLD_W` and `SPMAC_W` instructions are what replaces the `MLD_W` and `MMASA_W` instructions in Quadrilatero. The `MST_W` and `MZ` instructions from Quadrilatero are reused. One key difference between the two is that the SpMM operation in QuadriSparse

bypasses the systolic array and the computed result is a 1×4 row segment instead of a full 4×4 matrix tile. As a consequence, the sparse dataflow requires more fine-grained operations in order to perform the dynamic lookups, sparse traversal and partial computations. In contrast, the dense dataflow in Quadrilatero benefits from a dense structure in the input matrices, which allows the data to flow through the systolic array efficiently. As opposed to Quadrilatero, which packs the sparse matrix in CSR format on the software side. This explains the observed increase of instructions in QuadriSparse for low sparsity levels.

The massive performance gain at 99% is expected. As the sparsity increases for the input matrix, fewer elements will be packed into VAL and COL_IDX for the CSR. Consequently, fewer elements will be streamed through the accelerator, efficiently skipping all non-zero elements which is a vast majority at these levels. Quadrilatero on the other hand, will stream all elements through the systolic array, resulting in multiplications containing zero-valued operands which do not contribute to the final result. As said, it is at around 95% that QuadriSparse's zero-skipping starts to outperform Quadrilatero.

When computing the product of a sparse matrix A, $m \times k$, in CSR format with a sparsity level Sp, and a dense matrix B, $k \times n$, the flow is as follows: Four result segments of shape 1×4 are computed concurrently, reusing the sparse tile between them. The algorithm iterates over the current sparse row, loading tiles using the `SPLD_W` instruction. For each sparse tile, four dense tiles are loaded and multiply accumulated using the `DLD_W` and `SPMAC_W` instructions respectively. This repeats for all columns of the dense matrix and all rows of the sparse matrix. The complexity of this implementations is then:

$$\mathcal{O}(k * (1 - Sp) * n * m)$$

Formally the sparsity level Sp is a constant meaning that the complexity is the same as for regular matrix multiplication. This can be seen in Figure 5.4 where all lines appear to be asymptotically parallel to the baseline. However, in practice the sparsity level directly scales the number of instructions needed and is therefore responsible for the achieved speedup. Thus it is warranted to include the sparsity in the complexity analysis.

The speedup trends seen in all results (for example Figure 5.2) reveal an overhead in the sparse operations that decreases with the workload size. This results in the performance relative to dense increasing when the matrix sizes are increased. Furthermore the trend lines become asymptotically parallel with the baseline at large matrix sizes which shows that the overhead becomes negligible for these workloads. We believe that this overhead is caused by under-utilization of the last `SPLD_W` instruction of a sparse row, and by extension all the `SPMAC_W` instructions involving that sparse tile. This happens if the number of non-zero elements in a row of the sparse matrix is not divisible by four, and its effect is the greatest at low matrix sizes.

6.3 Optimizations

Extending hardware is not a simple task. Therefore, the early focus of the project was to get the instructions implemented and running. However, it became clear that the result was not satisfactory. This version gained a peak performance gain of 3x at 99% sparsity and started outperforming Quadrilatero at around 97% sparsity (see Figure 5.1). Since the instruction sequence is largely determined by the chosen SpMM algorithm, subsequent performance optimizations focused on reducing the cycle count for the individual instructions.

6.3.1 SPLD_W

The first version of `SPLD_W` required 7 cycles. This was unexpectedly high, as the previous load `MLD_W` in Quadrilatero was pipelined to 4 cycles. The original implementation of `SPLD_W` loaded the sparse data before performing zero-fills. It first waited for the memory responses for row 0 and row 1. After receiving the data, zeros were written to rows 2 and 3. This meant that zero-fill writes occurred entirely after the memory latency had elapsed. The optimization reordered the register writes so that zero-filling begins immediately after instruction start. To achieve this, the instruction's register writes were changed to start at row 2 and then wrap around to row 0. The load data was only consumed from the FIFO when the counter had wrapped around, effectively overlapping the zero-fills with the memory-latency.

6.3.2 DLD_W

The original version of the `DLD_W` instruction was highly unoptimized. It required 21 cycles, which was brought down to 9 cycles. Originally the register file controller would force instructions to read all the rows of a register, this was legacy behavior from Quadrilatero. The RF controller was modified to allow selective register reads, meaning that an instruction could now read only one row of a matrix register if needed. This was utilized in the `DLD_W` instruction by allowing it to read only the index row of the sparse tile saving 3 cycles. Another large cycle saving came from pipelining the writing of the dense rows to the matrix register. The next request is issued in the same cycle as the previous one is being processed saving another 3 cycles. The rest of the improvement came from streamlining the instruction logic and its integration with the memory and register file interfaces. For example, loading the first dense row one cycle earlier than before and asserting the finish signal on the same cycle as the operation finishes.

6.4 Design Choices

This section discusses the most important design choices taken to arrive at the architecture described in Chapter 3.

6.4.1 The Focus on Unstructured Sparsity

Looking at the state-of-the-art implementations of sparse-dense accelerators, a number of works instead target structured or semi-structured sparsity, whether on GPU, CPU, or FPGA, for example [11], [39] and [12]. Because of the predictable structure of $N : M$ sparsity, these accelerators can be parallelized to a higher degree without having to worry about load balancing, as all rows or tiles will have the same number of non-zero elements. As the decision to use Quadrilatero as a starting point for this project was taken early, we knew that our instructions would be working on small 4×4 tiles. This limited parallelism but it also made load balancing a non-issue, removing one of the main incentives for restricting ourselves to structured sparsity. Additionally, unstructured sparsity enables higher sparsity levels and increases the number of applications that can be accelerated efficiently.

6.4.2 Sparsity Level

Sparse matrix accelerators benefit greatly from increased sparsity and there are many applications which naturally produce matrices with very high sparsity. For example unstructured pruning of LLMs and other neural networks can enable high sparsity levels above 90% [7]. This is a result of the fact that more elements can be pruned when no consideration is given to where the pruned elements are located. Another example of an application which produces high sparsities is graph analytics and graph neural networks (GNN). The graphs adjacency matrix often exhibits extremely high sparsities well above 99% [5]. The relevant SpMM calculation here is the multiplication of this sparse adjacency matrix and the neural network's dense weight matrix.

6.4.3 Choosing the SpMM Algorithm

As described in Section 2.4 the literature is dominated by three major SpMM algorithms. Those being inner product, outer product, and Gustavson's algorithm. There were three main considerations when choosing the algorithm to model the QuadriSparse instructions after: How well the algorithms could be integrated with the existing infrastructure in Quadrilatero, how well they can be tiled, and the complexity of implementing them.

When constrained to the 4×4 matrix registers in Quadrilatero Gustavson's algorithm became the optimal choice. The inner product algorithm works on the same data and produces the same result as Gustavson but traverses the data less optimally. The outer product algorithm conducts the same operations as Gustavson but distributes the work among multiple output rows making it less focused. Once this was discovered the choice to use Gustavson's algorithm became clear. Under these constraints, all three algorithms behave similarly but Gustavson's traversal and work distribution is more optimal.

With the selection of Gustavson's for the SpMM algorithm the number of relevant sparse formats was cut down to just two: CSR and COO. As Gustavson's algorithm processes one row of the sparse matrix at a time it allows the accelerator to offload

the handling of the outer loop to the software side. Since the accelerator never has to touch the row indices, the higher compression of CSR is favorable compared to the simpler row index decoding of COO.

6.4.4 Not Reusing the Systolic Array

The initial plan was to reuse the entire hardware of Quadrilatero, consequently streaming the data through the systolic array. However, it was later determined that this was not an ideal solution for our project as it introduces three major problems: implementation complexity, hardware under-utilization and dataflow mismatch.

As mentioned, the systolic array in Quadrilatero is made for 4×4 dense multiplication and expects skewed input where each row of A arrives one cycle late so that each partial product can be aligned and computed correctly over each PE. Our intended flow with Gustavson's algorithm that uses `SPMAC_W` operates on a single sparse row at a time with a 4×4 sub-matrix will leave 3/4 of the mesh idle. To fully utilize the array, each `SPMAC_W` instruction would need to be dispatched together. However, each sparse row needs an independent 4×4 submatrix which makes this approach highly impractical. Furthermore, `SPMAC_W` can't start execution before `DLD_W` has fully written all four dense rows to the register file (see Figure 4.1). This eliminates any opportunity to interleave `DLD_W` and `SPMAC_W` execution.

Finally, it was noticed that the FS stage, responsible for skewing the input, is completely redundant for `SPMAC_W` since we do not need to skew the input. This stage alone costs 4 cycles and is not needed for the instruction since the data arrives pre-aligned.

Given these limitations, a decision was made to implement a separate data path for the sparse computation while maximizing the reuse of existing hardware. As a result, SpMM computation is routed through the sparse processing engine via the same read ports as the systolic array with the output registers and DR (writeback stage) also shared.

6.5 Design Limitations

QuadriSparse has one small functional limitation. The dense matrix's number of columns need to be divisible by four or zero-padded to a number divisible by four. This is a limitation in the current state of the `DLD_W` instruction as it will always load four elements of each selected row without regard for the end of the matrix. We do however believe that this can be fixed fairly easily. The instruction needs to be extended to take a mask or a number of elements to load similar to how the `SPLD_W` instruction works.

While the accelerator achieves good performance compared to the baseline, all instructions added are completely un-pipelined. This leaves an opportunity for future work to further increase the performance by pipelining the instructions. The `DLD_W` and the `SPMAC_W` instructions will benefit the most from pipelining as they are the slowest and by far the most used instructions. The Quadrilatero instructions achieve

a throughput of roughly one instruction every 4 cycles. If the QuadriSparse instructions can be pipelined to the same degree, with both the LSU and sparse processing engine achieving high utilization, the performance gain at 99% sparsity could reach as high as 17x. This is based on the difference in the number of instructions executed by QuadriSparse being roughly 17x less than Quadrilatero.

The sparse instructions added by this thesis were all built for 32-bit integer values. For the sake of time it was only possible to support one data type. 32-bit integers were chosen as it is the most common and versatile data type. Future works may want to extend the accelerator to support other data types. For example, high precision floating point units may be useful for scientific applications while low precision integers or floating points may be useful for AI applications.

7

Conclusion and Future Work

This project has successfully extended the RISC-V matrix multiplication accelerator Quadrilatero with a novel architecture to support sparse-dense matrix multiplication. The results demonstrate that an SpMM accelerator can be efficient at the small scale of a 4×4 instruction, addressing the first half of RQ 1. QuadriSparse defines and implements three novel instructions: `SPLD_W` to load a tile of the sparse matrix, `DLD_W` to load a tile of the dense matrix, and `SPMAC_W` to multiply and accumulate the two. The SpMM algorithm used is Gustavson's with the sparse matrix stored in the CSR format. The thesis shows that SpMM can be efficient at the small scales of 4×4 tiles, provided the sparsity is above 90%. This covers RQ 2.

The custom sparse instructions begin to outperform Quadrilatero when the level of sparsity exceeds 95% and achieves a 6.6x speedup at 99% sparsity and 2048×2048 matrices, satisfying the second half of RQ 1. The design was synthesized for the Xilinx Kintex-7 160T FPGA to estimate the area overhead of the new instructions. The increase in resource utilization was found to be a modest 4.6% for LUTs and 23% for DSP-blocks, demonstrating the area overhead of the extended accelerator and addressing RQ 3. This thesis was a success seeing as it has answered the research questions and accomplished the goals set out in Sections 1.2 and 1.3.

Future works may continue development of QuadriSparse to add support for more data types or to introduce pipelining. If the QuadriSparse instructions can be pipelined to the same degree as Quadrilatero, with both the LSU and sparse processing engine achieving high utilization, the performance gain at 99% sparsity could reach as high as 17x. The next steps to deploy QuadriSparse is to integrate the new instructions in a compiler toolchain and to start performing integration tests with a real core, such as CV32E40X. At this point, testing with realistic memory latencies and cache hierarchies should be conducted to ascertain how the accelerator would perform in a production environment. Another promising area for future research is developing novel algorithms that can take full advantage of an accelerator that supports hardware acceleration for both sparse and dense operations in a single package. This opens up an opportunity for future work to explore algorithms that intelligently select between the two modes of operation based on the sparsity of different parts of a matrix.

Bibliography

[1] B. D. Wozniak, F. D. Witherden, F. P. Russell, P. E. Vincent, and P. H. Kelly, "GiMMiK-Generating bespoke matrix multiplication kernels for accelerators: Application to high-order computational fluid dynamics," *Computer Physics Communications*, vol. 202, pp. 12–22, 2016, ISSN: 0010-4655. DOI: 10.1016/j.cpc.2015.12.012.

[2] X. Zhu, J. Li, Y. Liu, C. Ma, and W. Wang, "A survey on model compression for large language models," *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 1556–1577, Nov. 2024, ISSN: 2307-387X. DOI: 10.1162/tacl_a_00704.

[3] M. Sadkane, "Block-arnoldi and davidson methods for unsymmetric large eigenvalue problems," *Numerische Mathematik*, vol. 64, no. 1, pp. 195–211, Dec. 1, 1993, ISSN: 0945-3245. DOI: 10.1007/BF01388687.

[4] R. G. Grimes, J. G. Lewis, and H. D. Simon, "A shifted block lanczos algorithm for solving sparse symmetric generalized eigenproblems," *SIAM Journal on Matrix Analysis and Applications*, vol. 15, no. 1, pp. 228–272, 1994. DOI: 10.1137/S0895479888151111.

[5] W. Tang and P. Zhang, "GPGCN: A general-purpose graph convolution neural network accelerator based on RISC-V ISA extension," *Electronics*, vol. 11, no. 22, 2022, ISSN: 2079-9292. DOI: 10.3390/electronics11223833.

[6] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, 2011. DOI: 10.1145/2049662.2049663.

[7] C. Schulte, S. Wagner, A. Runge, D. Bariamis, and B. Hammer, "Best of both, structured and unstructured sparsity in neural networks," in *Proceedings of the 3rd Workshop on Machine Learning and Systems*, ser. EuroMLSys '23, Rome, Italy: Association for Computing Machinery, 2023, pp. 104–108, ISBN: 9798400700842. DOI: 10.1145/3578356.3592583.

[8] U. Mukhopadhyay, A. Tripathy, O. Selvitopi, K. Yelick, and A. Buluc, "Sparsity-Aware Communication for Distributed Graph Neural Network Training," in *Proceedings of the 53rd International Conference on Parallel Processing*, Gotland Sweden: ACM, Aug. 2024, pp. 117–126, ISBN: 979-8-4007-1793-2. DOI: 10.1145/3673038.3673152.

[9] D. Cammarata, M. Perotti, M. Bertuletti, *et al.*, "Quadrilatero: A RISC-V programmable matrix coprocessor for low-power edge applications," in *Proceedings of the 22nd ACM International Conference on Computing Frontiers: Workshops and Special Sessions*, ser. CF '25 Companion, Association for Com-

puting Machinery, 2025, pp. 66–69, ISBN: 9798400713934. DOI: 10.1145/3706594.3726978.

[10] H. Zhang, J. Ding, B. Jiang, T. Lu, W. Xu, and Z. Chai, "Optimizing sparse matrix convolution on RISC-V core: Custom instructions for embedded system," *ACM Trans. Embed. Comput. Syst.*, vol. 24, no. 6, Oct. 2025, ISSN: 1539-9087. DOI: 10.1145/3756322.

[11] V. Titopoulos, K. Alexandridis, C. Peltekis, C. Nicopoulos, and G. Dimitrakopoulos, "IndexMAC: A custom RISC-V vector instruction to accelerate structured-sparse matrix multiplications," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, ISSN: 1558-1101, Mar. 2024, pp. 1–6. DOI: 10.23919/DATE58400.2024.10546747.

[12] L. Ribas, I. Aquino, I. Felzmann, L. Wanner, and G. Araújo, "MSE: A matrix sparsity extension for RISC-V," in *Anais do XXVI Simpósio em Sistemas Computacionais de Alto Desempenho*, Bonito/MS: SBC, 2025, pp. 254–265. DOI: 10.5753/sscad.2025.16697.

[13] M. Osterno, C. Marcon, J. Silveira, and J. Silveira, "A lightweight RISC-V vector merge instruction to boost SpMxDV algorithm," in *2025 XV Symposium on Computing Systems Engineering (SBESC)*, 2025, pp. 1–6. DOI: 10.1109/SBESC68008.2025.11288899.

[14] B. Jiang, J. Ding, H. Zhang, T. Lu, and Z. Chai, "An efficient RISC-V processor with customized instruction set for sparse dnn acceleration on embedded system," *Journal of Systems Architecture*, vol. 171, p. 103 649, 2026, ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2025.103649.

[15] L. Song, Y. Chi, A. Sohrabizadeh, Y.-k. Choi, J. Lau, and J. Cong, "Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '22, Virtual Event, USA: Association for Computing Machinery, 2022, pp. 65–77, ISBN: 9781450391498. DOI: 10.1145/3490422.3502357.

[16] A. Feldmann, C. Golden, Y. Yang, J. S. Emer, and D. Sanchez, "Azul: An accelerator for sparse iterative solvers leveraging distributed on-chip memory," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024, pp. 643–656. DOI: 10.1109/MICRO61859.2024.00054.

[17] A. Sedigh Baroughi, M. Bheemasandra Rajashekar, A. Raj Baranwal, and Z. Fang, "HiSpMM: High performance high bandwidth sparse-dense matrix multiplication on hbm-equipped fpgas," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 19, no. 1, Mar. 2026, ISSN: 1936-7406. DOI: 10.1145/3774327.

[18] F. Muñoz-Martínez, R. Garg, M. Pellauer, J. L. Abellán, M. E. Acacio, and T. Krishna, "Flexagon: A Multi-dataflow Sparse-Sparse Matrix Multiplication Accelerator for Efficient DNN Processing," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023, New York, NY, USA: Association for Computing Machinery, Mar. 2023, pp. 252–265, ISBN: 978-1-4503-9918-0. DOI: 10.1145/3582016.3582069.

[19] Y. Yang, J. S. Emer, and D. Sanchez, "Trapezoid: A Versatile Accelerator for Dense and Sparse Matrix Multiplications," in *2024 ACM/IEEE 51st An-*

nual International Symposium on Computer Architecture (ISCA), Jun. 2024, pp. 931–945. DOI: 10.1109/ISCA59077.2024.00072.

[20] G. Xiao, C. Yin, T. Zhou, X. Li, Y. Chen, and K. Li, "A survey of accelerating parallel sparse linear algebra," *ACM Comput. Surv.*, vol. 56, no. 1, Aug. 2023, ISSN: 0360-0300. DOI: 10.1145/3604606.

[21] W. Kim, J. Lee, S. Kim, and J.-H. Kim, "Multi-mode transprecision sparse matrix-vector multiplication engine for pagerank," in *2022 International Conference on Electronics, Information, and Communication (ICEIC)*, 2022, pp. 1–3. DOI: 10.1109/ICEIC54506.2022.9748316.

[22] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA: SIAM, 2003, ISBN: 978-0-898715-34-7.

[23] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct methods for sparse matrices*. USA: Oxford University Press, Inc., 1986, ISBN: 0198534086.

[24] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on cuda," Nvidia Technical Report NVR-2008-004, Nvidia Corporation, Tech. Rep., 2008.

[25] NVIDIA Corporation, *NVPL Sparse Storage Formats*, Accessed: 2026-05-21, 2026. [Online]. Available: https://docs.nvidia.com/nvpl/latest/sparse/storage_format/sparse_matrix.html.

[26] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures," in *High Performance Embedded Architectures and Compilers*, Y. N. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, and X. Martorell, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 111–125, ISBN: 978-3-642-11515-8.

[27] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, "A High Memory Bandwidth FPGA Accelerator for Sparse Matrix-Vector Multiplication," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2014, pp. 36–43. DOI: 10.1109/FCCM.2014.23.

[28] A. Parravicini, L. G. Cellamare, M. Siracusa, and M. D. Santambrogio, "Scaling up HBM Efficiency of Top-K SpMV for Approximate Embedding Similarity on FPGAs," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, ISSN: 0738-100X, Dec. 2021, pp. 799–804. DOI: 10.1109/DAC18074.2021.9586203.

[29] Z. Wu, M. Wang, and H. K.-H. So, "A Hardware-Software Design Framework for SpMV Acceleration with Flexible Access Pattern Portfolio," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, ISSN: 2378-203X, Mar. 2025, pp. 836–848. DOI: 10.1109/HPCA61900.2025.00068.

[30] D. Joo, H. Hosseini, R. Hadidi, and B. Asgari, "Coruscant: Co-Designing GPU Kernel and Sparse Tensor Core to Advocate Unstructured Sparsity in Efficient LLM Inference," in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '25, New York, NY, USA: Association for Computing Machinery, Oct. 2025, pp. 232–245, ISBN: 979-8-4007-1573-0. DOI: 10.1145/3725843.3756065.

[31] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Trans. Math. Softw.*, vol. 4, no. 3, pp. 250–269, Sep. 1978, ISSN: 0098-3500. DOI: 10.1145/355791.355796.

[32] E. Cui, T. Li, and Q. Wei, "RISC-V instruction set architecture extensions: A survey," *IEEE Access*, vol. 11, pp. 24 696–24 711, 2023. DOI: 10.1109/ACCESS.2023.3246491.

[33] S. O. Aletan, "An overview of RISC architecture," in *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied computing: technological challenges of the 1990s*, ser. SAC92, ACM, Apr. 1992, pp. 11–20. DOI: 10.1145/143559.143570.

[34] H. T. Kung, C. E. Leiserson, *et al.*, "Systolic arrays (for vlsi)," in *Sparse Matrix Proceedings 1978*, SIAM Philadelphia, PA, USA, vol. 1, 1979, pp. 256–282.

[35] N. P. Jouppi, C. Young, N. Patil, *et al.*, "In-datacenter performance analysis of a tensor processing unit," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 1–12, Jun. 2017, ISSN: 0163-5964. DOI: 10.1145/3140659.3080246.

[36] OpenHW Group, *Openhwgroup/cv32e40x: Openhw group core-v cv32e40x RISC-V ip*, Oct. 2023. [Online]. Available: https://github.com/openhwgroup/cv32e40x.

[37] OpenHW Group, *Openhwgroup/core-v-xif: Openhw group specification: Core-v extension interface (cv-x-if)*, May 2024. [Online]. Available: https://github.com/openhwgroup/core-v-xif.

[38] OpenHW Group, *Openhwgroup/OBI: The OpenBus Interface spec*, Apr. 2023. [Online]. Available: https://github.com/openhwgroup/obi.

[39] E. Taka, N.-C. Huang, C.-C. Chang, K.-C. Wu, A. Arora, and D. Marculescu, "Systolic sparse tensor slices: Fpga building blocks for sparse and dense ai acceleration," in *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '25, Monterey, CA, USA: Association for Computing Machinery, 2025, pp. 159–171, ISBN: 9798400713965. DOI: 10.1145/3706628.3708867.

A

Benchmarking Data

This appendix contains the complete data from the benchmarking of the accelerator.

A.1 First Version

Table A.1: Full results from the benchmarking of the first version

Mode	#Runs	Matrix Size	Sparsity	Avg #Instructions	Avg #Cycles
Sparse	5	16	0.7	362	3325
Sparse	5	16	0.8	336	3076
Sparse	5	16	0.9	257	2239
Sparse	5	16	0.95	153	1243
Sparse	5	16	0.97	102	835
Sparse	5	16	0.99	34	291
Dense	1	16	N/A	160	379
Sparse	5	32	0.7	2366	24 769
Sparse	5	32	0.8	1862	18 441
Sparse	5	32	0.9	1274	11 523
Sparse	5	32	0.95	924	7903
Sparse	5	32	0.97	836	6953
Sparse	5	32	0.99	306	2467
Dense	1	32	N/A	1152	2447
Sparse	5	64	0.7	14 900	167 271
Sparse	5	64	0.8	10 940	117 551
Sparse	5	64	0.9	7556	75 063
Sparse	5	64	0.95	5332	47 759
Sparse	5	64	0.97	4088	36 167
Sparse	5	64	0.99	1976	16 155
Dense	1	64	N/A	8704	17 887

Sparse	5	128	0.7	103 160	1 215 947
Sparse	5	128	0.8	73 856	848 019
Sparse	5	128	0.9	44 696	481 899
Sparse	5	128	0.95	30 224	300 195
Sparse	5	128	0.97	23 968	222 267
Sparse	5	128	0.99	13 936	116 755
Dense	1	128	N/A	67 584	136 991
Sparse	5	256	0.7	767 600	9 320 451
Sparse	5	256	0.8	533 600	6 382 451
Sparse	5	256	0.9	295 424	3 392 019
Sparse	5	256	0.95	178 640	1 925 731
Sparse	5	256	0.97	130 400	1 320 051
Sparse	5	256	0.99	81 104	719 699
Dense	1	256	N/A	532 480	1 072 159
Sparse	5	512	0.7	5 906 624	72 892 115
Sparse	5	512	0.8	4 017 632	49 174 771
Sparse	5	512	0.9	2 130 368	25 479 123
Sparse	5	512	0.95	1 184 288	13 600 563
Sparse	5	512	0.97	800 096	8 776 819
Sparse	5	512	0.99	428 928	4 121 555
Dense	1	512	N/A	4 227 072	8 482 847
Sparse	5	1024	0.7	46 269 056	575 858 323
Sparse	5	1024	0.8	31 173 248	386 322 067
Sparse	5	1024	0.9	16 064 768	196 626 707
Sparse	5	1024	0.95	8 512 832	101 807 955
Sparse	5	1024	0.97	5 507 840	64 078 611
Sparse	5	1024	0.99	2 478 080	26 038 291
Dense	1	1024	N/A	33 685 504	67 485 727
Sparse	5	2048	0.7	366 268 544	4 578 403 475
Sparse	5	2048	0.8	245 462 912	3 061 621 651
Sparse	5	2048	0.9	124 660 736	1 544 883 219
Sparse	5	2048	0.95	64 282 112	786 796 051
Sparse	5	2048	0.97	40 083 200	482 965 267
Sparse	5	2048	0.99	15 908 480	179 438 227
Dense	1	2048	N/A	268 959 744	538 378 271

A.2 Optimized Version

Table A.2: Full results from the benchmarking of the optimized version

Mode	#Runs	Matrix Size	Sparsity	Avg #Instructions	Avg #Cycles
Sparse	5	16	0.7	425	2001
Sparse	5	16	0.8	335	1501
Sparse	5	16	0.9	239	1041
Sparse	5	16	0.95	187	801
Sparse	5	16	0.97	102	451
Sparse	5	16	0.99	34	171
Dense	1	16	N/A	160	379
Sparse	5	32	0.7	2366	11 611
Sparse	5	32	0.8	1844	8711
Sparse	5	32	0.9	1324	5871
Sparse	5	32	0.95	942	4091
Sparse	5	32	0.97	802	3411
Sparse	5	32	0.99	290	1251
Dense	1	32	N/A	1152	2447
Sparse	5	64	0.7	14 828	76 151
Sparse	5	64	0.8	11 084	55 351
Sparse	5	64	0.9	7556	35 751
Sparse	5	64	0.95	5332	23 591
Sparse	5	64	0.97	4356	18 951
Sparse	5	64	0.99	2080	8751
Dense	1	64	N/A	8704	17 887
Sparse	5	128	0.7	103 664	550 911
Sparse	5	128	0.8	73 208	381 711
Sparse	5	128	0.9	44 696	223 311
Sparse	5	128	0.95	29 936	141 311
Sparse	5	128	0.97	23 552	106 431
Sparse	5	128	0.99	13 064	55 791
Dense	1	128	N/A	67 584	136 991
Sparse	5	256	0.7	768 896	4 171 551
Sparse	5	256	0.8	532 880	2 860 351
Sparse	5	256	0.9	294 992	1 538 751

Sparse	5	256	0.95	178 352	890 751
Sparse	5	256	0.97	131 552	630 751
Sparse	5	256	0.99	79 648	347 871
Dense	1	256	N/A	532 480	1 072 159
Sparse	5	512	0.7	5 906 336	32 412 511
Sparse	5	512	0.8	4 017 632	21 919 711
Sparse	5	512	0.9	2 134 400	11 457 311
Sparse	5	512	0.95	1 184 000	6 177 311
Sparse	5	512	0.97	806 144	4 078 111
Sparse	5	512	0.99	429 824	1 989 791
Dense	1	512	N/A	4 227 072	8 482 847
Sparse	5	1024	0.7	46 285 760	255 541 151
Sparse	5	1024	0.8	31 158 848	171 502 751
Sparse	5	1024	0.9	16 077 440	87 717 151
Sparse	5	1024	0.95	8 490 368	45 566 751
Sparse	5	1024	0.97	5 496 896	28 936 351
Sparse	5	1024	0.99	2 482 112	12 187 551
Dense	1	1024	N/A	33 685 504	67 485 727
Sparse	5	2048	0.7	366 253 568	2 028 334 111
Sparse	5	2048	0.8	245 468 672	1 357 306 911
Sparse	5	2048	0.9	124 671 104	686 209 311
Sparse	5	2048	0.95	64 272 896	350 663 711
Sparse	5	2048	0.97	40 131 584	216 545 311
Sparse	5	2048	0.99	15 932 672	82 106 911
Dense	1	2048	N/A	268 959 744	538 378 271

B

Synthesis Data

This appendix contains the complete data from the synthesis of QuadriSparse and the baseline Quadrilatero.

B.1 Quadrilatero

Table B.1 contains the post-synthesis resource utilization by functional type. Table B.2 contains the resource utilization by primitive.

Table B.1: Full post-synthesis resource utilization for *Quadrilatero* (Vivado 2025.2, device xc7k160tfbg484-3).

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs	63 132	0	0	101 400	62.26
LUT as Logic	63 132	0	0	101 400	62.26
LUT as Memory	0	0	0	35 000	0.00
Slice Registers	13 683	0	0	202 800	6.75
Register as Flip Flop	13 632	0	0	202 800	6.72
Register as Latch	51	0	0	202 800	0.03
F7 Muxes	1056	0	0	50 700	2.08
F8 Muxes	0	0	0	25 350	0.00
Unique Control Sets	221	-	0	25 350	0.87
DSP Blocks (DSP48E1)	256	0	0	600	42.67
BUFGCTRL Clock	1	0	0	32	3.13

Table B.2: Number of elements used by primitive for *Quadrilatero*

Ref Name	# Used	Type
LUT6	31171	LUT
FDCE	13564	Flop & Latch
LUT5	12733	LUT
LUT3	9433	LUT
LUT4	8486	LUT
LUT2	7106	LUT
CARRY4	3310	CarryLogic
LUT1	3184	LUT
MUXF7	1056	MuxFx
OBUF	468	IO
DSP48E1	256	Block Arithmetic
IBUF	246	IO
FDPE	68	Flop & Latch
LDCE	51	Flop & Latch
BUFG	1	Clock

B.2 QuadriSparse

Table B.3 contains the post-synthesis resource utilization by functional type. Table B.4 contains the resource utilization by primitive.

Table B.3: Full post-synthesis resource utilization for *QuadrilSparse* (Vivado 2025.2, device xc7k160tfbg484-3).

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	66 051	0	0	101 400	65.14
LUT as Logic	66 051	0	0	101 400	65.14
LUT as Memory	0	0	0	35 000	0.00
Slice Registers	15 351	0	0	202 800	7.57
Register as Flip Flop	15 300	0	0	202 800	7.54
Register as Latch	51	0	0	202 800	0.03
F7 Muxes	1183	0	0	50 700	2.33
F8 Muxes	0	0	0	25 350	0.00
Unique Control Sets	295	-	0	25 350	1.16
DSP Blocks (DSP48E1)	316	0	0	600	52.67
BUFGCTRL Clock	1	0	0	32	3.13

Table B.4: Number of elements used by primitive for *Quadrisparse*

Ref Name	# Used	Type
LUT6	31796	LUT
FDCE	15230	Flop & Latch
LUT5	13059	LUT
LUT3	9607	LUT
LUT4	9278	LUT
LUT2	8125	LUT
CARRY4	3506	CarryLogic
LUT1	3193	LUT
MUXF7	1183	MuxFx
OBUF	468	IO
DSP48E1	316	Block Arithmetic
IBUF	246	IO
FDPE	70	Flop & Latch
LDCE	51	Flop & Latch
BUFG	1	Clock

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY