

CHALMERS
UNIVERSITY OF TECHNOLOGY

Design of Complex Multiply-Accumulate RISC-V extension for Accelerating Quantum Simulations

Master's thesis in Embedded Electronic System Design

Stas Akopau

Department of Microtechnology and Nanoscience
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026

Master's thesis 2026

Design of Complex Multiply-Accumulate RISC-V extension for Accelerating Quantum Simulations

Stas Akopau



Department of Microtechnology and Nanoscience
Chalmers University of Technology
Gothenburg, Sweden 2026

Design of Complex Multiply-Accumulate RISC-V extension for Accelerating Quantum Simulations
Stas Akopau

© Stas Akopau, 2026.

Supervisors:
Pedro Petersen Moura Trancoso, Computer Science and Engineering
Alessio Cicero, Computer Science and Engineering

Examiner: Lena Peterson, Microtechnology and Nanoscience

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

Typeset in LaTeX
Gothenburg, Sweden 2026

Design of Complex Multiply-Accumulate RISC-V extension for Accelerating Quantum Simulations
Stas Akopau
Department of Microtechnology and Nanoscience
Chalmers University of Technology

Abstract

Quantum circuit simulation is an important tool for developing and testing quantum algorithms, but its computational and memory requirements grow exponentially with the number of qubits. A significant part of the computational workload consists of complex-valued matrix–vector operations. Depending on the circuit, the corresponding unitary matrices may contain many zero-valued coefficients that do not contribute to the result.

This thesis investigates whether such sparsity can be exploited using specialized hardware. A preliminary analysis of representative quantum circuits was performed to examine the sparsity of their unitary matrices. Based on this motivation, a complex sparse matrix-vector multiplication (SpMV) accelerator was developed using the compressed sparse row (CSR) representation. The architecture consists of vector permutation, parallel complex multiplication, and row accumulation stages. It is intended for use as a coprocessor within an X-HEEP RISC-V system, although the present evaluation considers the standalone accelerator.

The accelerator was verified through RTL simulation and synthesized using Cadence Genus with the ASAP7 technology library. Synthetic 8×8 complex matrix tiles with densities between 9.4% and 100% were used for evaluation. Timing closure was demonstrated at up to 440 MHz. Reducing the number of non-zero coefficients from 64 to 6 decreased the execution time from 129 to 21 clock cycles, corresponding to a speedup of approximately 6.14× relative to a 100%-dense tile processed by the same accelerator. At 440 MHz, the estimated energy per tile-level operation was reduced by approximately 74%. These results demonstrate that the proposed architecture can exploit matrix sparsity to reduce tile-level execution latency and estimated energy. Complete processor integration and end-to-end quantum-simulation performance remain subjects for future evaluation.

Keywords: quantum circuit simulation, statevector simulation, complex multiply-accumulate, CMAC, RISC-V, hardware accelerator, instruction-set extension

Acknowledgements

I would like to express my sincere gratitude to my supervisors, Alessio Cicero and Pedro Petersen Moura Trancoso, for giving me the opportunity to work in the field of quantum computing and for their valuable advice, feedback, and support throughout this master's thesis.
I would also like to thank my examiner, Lena Peterson, for reviewing this report and for showing interest in this work.

Stas Akopau, Gothenburg, November 2026

Contents

List of Abbreviations **1**

1 Introduction **3**
1.1 Related Work . . . 3
1.2 Purpose and Goal . . . 5
1.3 Thesis Outline . . . 7

2 Technical Background **9**
2.1 Introduction to Quantum Computing . . . 9
2.2 Simulating Quantum Circuits . . . 11
2.3 Sparse and Compressed Matrix Representations . . . 12

3 Method **15**
3.1 Research Approach . . . 15
3.2 Circuit and Sparsity Analysis . . . 15
3.3 Evaluation . . . 16
3.4 Functional Verification . . . 16
3.5 Synthesis . . . 17
3.6 Simulation and Measurement . . . 17

4 Design **19**
4.1 Design Overview . . . 19
4.2 CSR Representation and Dataflow . . . 19
4.3 Proposed Architecture . . . 22
4.3.1 Vector Permutation and Multiplication . . . 24
4.3.2 Row Accumulation . . . 26
4.4 Sparsity-Dependent Execution . . . 28
4.5 Intended Coprocessor Integration . . . 28

5 Results **30**
5.1 Initial analysis of quantum circuits . . . 30
5.2 Design evaluation . . . 31
5.2.1 Synthesis characterization . . . 32
5.2.2 Sparsity evaluation . . . 36

6 Discussion **41**
6.1 Interpretation of the design evaluation . . . 41

6.1.1 Cost of Implementation 41
6.1.2 Effect of matrix sparsity 42
6.1.3 Trade-off between Energy and Performance 42
6.2 Limitations of the evaluation 43
6.2.1 Limited sampling 43
6.2.2 Synthesis-level characterization 43
6.2.3 Matrix size and application-level performance 44
6.2.4 X-HEEP coprocessor integration 44
6.2.5 Absence of a dedicated dense baseline 45
6.3 Future work 45
6.3.1 Row accumulator optimization 45
6.3.2 System-level X-HEEP evaluation 46
6.3.3 Quantum-simulation evaluation 46
6.3.4 Comparison with a dense implementation 46

7 Conclusion **47**

Bibliography **49**

List of Abbreviations

ASIC	Application-specific integrated circuit
CMAC	Complex multiply-accumulate
CPU	Central processing unit
CSR	Compressed sparse row
CSV	Comma-separated values
CV-X-IF	CORE-V eXtension Interface
FP64	64-bit floating-point
FSM	Finite-state machine
GHZ	Greenberger–Horne–Zeilinger
GPU	Graphics processing unit
MQT	Munich Quantum Toolkit
QAOA	Quantum approximate optimization algorithm
QCS	Quantum circuit simulation
QFT	Quantum Fourier transform
QMDD	Quantum multiple-valued decision diagram
RISC-V	Reduced instruction set computer V
RTL	Register-transfer level
SpMV	Sparse matrix–vector multiplication
VCD	Value change dump
WNS	Worst negative slack
X-HEEP	eXtensible Heterogeneous Energy-Efficient Platform

1

Introduction

The development and analysis of quantum algorithms depends on the ability to study their behavior without relying on costly quantum hardware. As a result, classical computing systems remain an essential tool for quantum circuit simulation research (QCS). Through simulation, researchers can verify circuit functionality, evaluate algorithmic behavior, and explore new designs before deploying them on quantum hardware.

Despite its importance, quantum circuit simulation is computationally expensive. Simulating an n-qubit system requires maintaining a state vector containing 2^n complex amplitudes. This exponential growth leads to substantial demands on both memory and execution time. State-vector simulation involves repeated matrix-vector operations that require complex-valued data to be moved, stored, and processed. These operations can produce considerable memory and arithmetic overhead on classical hardware.

This challenge has motivated a broad range of optimization strategies, including algorithmic methods, efficient data representations, and specialized hardware support. Quantum state updates involve complex multiplication and accumulation, making complex multiply-accumulate (CMAC) operations an important computational kernel. However, when the matrix representing an operation contains zero-valued coefficients, a dense matrix–vector multiplication also performs CMAC operations that do not contribute to the result.

Sparse matrix–vector multiplication (SpMV) avoids these unnecessary operations by storing and processing only non-zero matrix coefficients. Many quantum gates act on only one or two qubits, and their matrices can be sparse when represented as operations on a larger state vector. This motivates investigating a complex SpMV accelerator whose execution time can reflect the number of non-zero coefficients. Sparsity depends on the operation, however: the combined unitary matrix of an entire circuit may be considerably denser than the matrices of its individual gates. This thesis therefore examines sparsity in selected circuit matrices and evaluates the accelerator across controlled matrix densities.

1.1 Related Work

A significant body of work on quantum circuit simulation focuses on hardware-aware optimization, particularly through GPU acceleration and collaborative CPU-GPU

execution. A prominent example is Qiskit Aer [1], a high-performance quantum circuit simulator that supports GPU-accelerated execution. In this simulator, the CPU manages orchestration and execution control, while the GPU performs parallel updates of the quantum state vector.

This heterogeneous CPU-GPU simulation can, however, face several bottlenecks. These include limited GPU memory, low GPU utilization, and frequent host-device transfers that introduce data-movement and synchronization overheads. Kernel-launch and processor-coordination overheads can also reduce performance, particularly when the state vector cannot be stored entirely in GPU memory. Several recent works address these limitations through compression, dynamic data placement, and overlapping computation with data transfer [2], [3], [4].

BMQSim is one approach to address these limitations [2]. It proposes a CPU-GPU co-designed for full-state quantum circuit simulation even under severe memory constraints. Compressed state-vector blocks are stored in CPU memory, while state updates are offloaded to the GPU. A transfer-concealed pipeline overlaps data transfer, decompression, simulation, and recompression, while circuit partitioning reduces how often these operations are required. BMQSim also uses GPU-based lossy compression and a two-level memory-management scheme with GPUDirect Storage as a fallback [5]. Its main contribution is therefore the optimization of memory usage and data movement rather than the acceleration of the underlying arithmetic operations.

Q-GPU takes a related approach to improving full-state quantum circuit simulation on GPUs [3]. It addresses the poor GPU utilisation and excessive host–device communication that occur when the state vector cannot fit entirely in GPU memory. Q-GPU combines dynamic amplitude allocation with proactive data exchange, zero-amplitude pruning, dependency-aware gate reordering, and lossless compression. Across nine evaluated circuits, the framework achieved an average speedup of 3.55× over its baseline GPU implementation. Like BMQSim, Q-GPU primarily optimises data placement and communication rather than the underlying arithmetic hardware.

BQSim proposes a GPU-oriented framework for batch quantum circuit simulation [4]. Unlike simulators that focus on accelerating a single input state, BQSim combines decision-diagram-based circuit optimization with a GPU-friendly sparse representation and a task-graph execution model. This reduces kernel-launch overhead and allows execution to overlap with data transfer. The reported evaluation shows performance improvements over cuQuantum, Qiskit Aer, and FlatDD across representative workloads. Although BQSim uses sparse data representations, it targets GPU-based batch execution rather than a dedicated sparse matrix-vector accelerator.

In contrast to these system-level approaches, this work targets the arithmetic operations used during quantum state updates. State-vector simulation repeatedly applies complex-valued linear transformations, whose execution requires complex multiplication and accumulation. This motivates the development of a complex sparse matrix-vector multiplication (SpMV) accelerator that processes only non-zero matrix coefficients. Rather than focusing on scheduling, memory placement,

or communication between processing devices, the proposed design aims to reduce unnecessary arithmetic operations and accelerate the processing of sparse quantum transformation matrices.

1.2 Purpose and Goal

This thesis investigates whether matrix sparsity can be exploited to reduce the computational workload of quantum circuit simulation. The work combines an analysis of unitary matrices generated from selected quantum circuits with the development of a complex sparse matrix–vector multiplication (SpMV) accelerator intended for integration as a coprocessor within the X-HEEP RISC-V platform.

The proposed accelerator processes complex-valued matrix tiles stored in Compressed Sparse Row (CSR) format together with dense state-vector segments. By processing only non-zero matrix coefficients, the design avoids arithmetic operations involving explicit zeros and allows its execution time to depend on matrix sparsity. The architecture uses vector permutation, parallel complex multiplication, and sparsity-dependent row accumulation.

This thesis addresses three research questions:

- What sparsity is present in the unitary matrices generated from the selected quantum circuits?
- How does matrix density affect the execution latency and estimated energy of the proposed CSR-based complex SpMV accelerator?
- What timing, area, and power trade-offs arise when the accelerator is synthesized at different target clock frequencies?

Based on this motivation, the following goals are defined:

- Analyze the sparsity of unitary matrices generated from selected quantum circuits and determine whether sparse processing could reduce storage and arithmetic workload.
- Design and implement a complex CSR-based SpMV accelerator that uses vector permutation, parallel complex multiplication, and row accumulation.
- Functionally verify the accelerator and characterize its post-synthesis timing, area, and power using an ASIC-oriented design flow.
- Evaluate how matrix density affects accelerator execution latency, speedup, and estimated energy using synthetic 8×8 matrix tiles with controlled density levels.
- Implement an RTL interface module for connecting the accelerator to an X-HEEP processor through the CORE-V eXtension Interface (CV-X-IF).

The contribution of this thesis is the design and experimental characterization of a CSR-based complex SpMV accelerator intended for RISC-V coprocessor integration. The evaluation characterizes how matrix density affects latency and estimated energy for the evaluated workloads, and how clock-frequency constraints affect its

implementation cost. It also identifies two architectural limitations: the row accumulator limits timing at higher frequencies, while control and row-processing overhead limit the benefit of sparsity at low densities.

The experimental evaluation focuses on the standalone accelerator datapath. CPU-controlled matrix partitioning, data transfers, cross-tile accumulation, and end-to-end quantum circuit simulation are outside the scope of this thesis.

1.3 Thesis Outline

Chapter 2 introduces the technical background required for this thesis. It presents the basic principles of quantum computing, state-vector simulation, matrix–vector multiplication, matrix partitioning, and sparse matrix representations.

Chapter 3 describes the methodology used to analyze circuit-matrix sparsity, generate synthetic evaluation workloads, verify the accelerator, and obtain latency, timing, area, power, and energy results.

Chapter 4 presents the proposed CSR-based complex SpMV accelerator. It explains the CSR dataflow, the vector permutation, complex multiplication, and row accumulation stages, and the intended integration of the accelerator with the X-HEEP RISC-V platform.

Chapter 5 presents the results of the circuit sparsity analysis, post-synthesis characterization, and sparsity-dependent accelerator evaluation. Chapter 6 discusses the results, implementation trade-offs, limitations, and possible directions for future work. Finally, Chapter 7 summarizes the main findings and conclusions of the thesis.

2

Technical Background

2.1 Introduction to Quantum Computing

A quantum circuit is a computational model that represents a sequence of quantum operations, called quantum gates, applied to qubits.

A qubit is the basic unit of information in quantum computing. In a classical computer, a bit can take one of two values, namely 0 or 1. A qubit is also associated with two basis states, denoted by $|0\rangle$ and $|1\rangle$. However, unlike a classical bit, a qubit can exist in a superposition of these two basis states. A single-qubit state can be written as

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

where $|0\rangle$ and $|1\rangle$ represent the basis states, while α and β are complex amplitudes. The probabilities of measuring the qubit in state 0 or 1 are given by $|\alpha|^2$ and $|\beta|^2$, respectively. These probabilities must satisfy the normalization condition

$$|\alpha|^2 + |\beta|^2 = 1.$$

While a single qubit is described by two amplitudes, a system of multiple qubits must describe all possible basis states of the system. For example, a two-qubit system has four basis states, namely $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$. Its quantum state can therefore be written as

$$|\psi\rangle = \alpha_0|00\rangle + \alpha_1|01\rangle + \alpha_2|10\rangle + \alpha_3|11\rangle$$

where each α_i is a complex amplitude.

More generally, a system of n qubits is described by 2^n complex amplitudes, one for each possible basis state. These amplitudes are usually stored in the form of a column vector called the state vector:

$$|\psi\rangle = \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_{2^n-1} \end{bmatrix}$$

The state vector is one of the most common ways to represent a quantum state in classical simulation. Each entry corresponds to one basis state of the full quantum system. As in the single-qubit case, the probabilities of measurement are obtained from the squared magnitudes of the amplitudes, and the state must satisfy the normalization condition

$$\sum_{i=0}^{2^n-1} |\alpha_i|^2 = 1.$$

Similarly to how logic gates are the basic building blocks of classical circuits, quantum gates are the basic building blocks of quantum circuits. Quantum gates are represented by unitary matrices and are therefore reversible. In general, different quantum gates are described by matrices of different sizes depending on the number of qubits on which they act.

For example, the Hadamard gate is a single-qubit gate that creates an equal superposition when applied to the basis state $|0\rangle$. Its matrix form is

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

A quantum circuit is built by arranging such gates in sequence. In a circuit diagram, each horizontal line represents a qubit, while gates placed on these lines represent operations applied from left to right. For example, a 3-qubit circuit used in Grover's algorithm can be represented as follows:

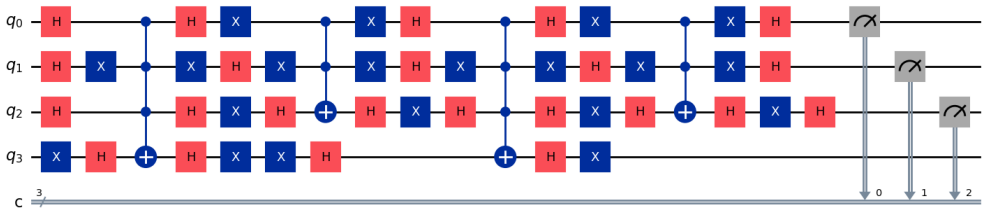


Figure 2.1: A 3-qubit Grover circuit generated by Qiskit.

A quantum circuit can also be represented mathematically by a single unitary matrix that describes the combined effect of all gates in the circuit. For an n-qubit circuit, this matrix has size $2^n \times 2^n$. Therefore, for a 3-qubit circuit, the corresponding unitary matrix has size 8×8. An example of such a matrix representation is shown below:

$$U \approx \begin{bmatrix} -0.088 + 0j & 0.442 + 0j & \cdots & 0 + 0j & 0 + 0j \\ -0.088 + 0j & -0.265 + 0j & \cdots & 0 + 0j & 0 + 0j \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 + 0j & 0 + 0j & \cdots & 0.354 + 0j & 0.354 + 0j \\ 0 + 0j & 0 + 0j & \cdots & 0.354 + 0j & -0.354 + 0j \end{bmatrix}$$

This matrix-based view is useful for analysis, although in practice quantum circuit simulation often works directly on the state vector rather than constructing the full unitary matrix explicitly.

2.2 Simulating Quantum Circuits

Quantum circuit simulation consists of applying an ordered sequence of quantum gates to a quantum state. For a system of n qubits, the state is commonly represented by a state vector containing 2^n complex amplitudes. Since most circuits consist mainly of one-qubit and two-qubit gates, simulation can be viewed either as the sequential application of individual gates or as the application of one combined $2^n \times 2^n$ unitary matrix.

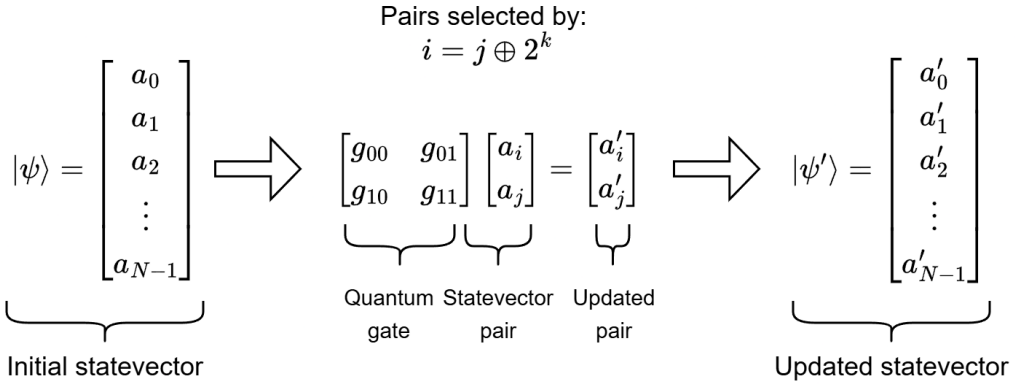


Figure 2.2: state vector update produced by a one-qubit gate acting on amplitudes a_i and a_j.

In standard state vector simulation, gates are applied sequentially to the state vector. Figure 2.2 illustrates the application of a one-qubit gate acting on qubit k. The operation updates pairs of amplitudes whose indices differ only in the k-th bit. These pairs can be written as $(i, i \oplus 2^k)$, where the k-th bit of i is assumed to be zero. For example, applying a gate to q_0 in a two-qubit system produces the pairs $(0, 1)$ and $(2, 3)$. Each pair is multiplied by the same 2×2 gate matrix.

Alternatively, the complete circuit can be represented as one unitary matrix that is multiplied by the state vector. However, this matrix has dimensions $2^n \times 2^n$ and therefore becomes impractically large as the number of qubits increases.

Each element of a matrix–vector result is obtained from a sum of products:

$$y_r = \sum_k U_{r,k} x_k, \tag{2.1}$$

where both the matrix coefficients $U_{r,k}$ and vector elements x_k may be complex. The repeated multiplication and addition of complex values is commonly described as a complex multiply-accumulate (CMAC) operation.

$$\begin{bmatrix} g_{00} & g_{01} & \cdots & g_{0,2^n-1} \\ g_{10} & g_{11} & \cdots & g_{1,2^n-1} \\ \vdots & \vdots & \ddots & \vdots \\ g_{2^n-1,0} & g_{2^n-1,1} & \cdots & g_{2^n-1,2^n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{2^n-1} \end{bmatrix} = \begin{bmatrix} a'_0 \\ a'_1 \\ \vdots \\ a'_{2^n-1} \end{bmatrix}$$

Figure 2.3: Multiplication of a $2^n \times 2^n$ matrix with a state vector.

The proposed accelerator implements this computation using three processing stages. First, the vector permutation stage aligns the required state vector elements with the non-zero matrix coefficients stored in CSR order. Parallel complex multipliers then calculate the corresponding products, after which the row accumulator combines the products belonging to each matrix row. The detailed implementation of these stages is presented in Sections 4.3.1 and 4.3.2.

The multiplication of a large unitary matrix by a state vector can be divided into smaller operations. The matrix is partitioned into tiles U_{ij}, while the state vector is divided into corresponding segments x_j. Each tile produces a partial output vector

$$\mathbf{p}_{ij} = \mathbf{U}_{ij}\mathbf{x}_j. \tag{2.2}$$

The complete output segment is obtained by accumulating all partial results that contribute to the same matrix rows:

$$\mathbf{y}_i = \sum_j \mathbf{p}_{ij}. \tag{2.3}$$

In the intended execution model, the CPU divides a larger unitary matrix into accelerator-sized tiles and divides the state vector into corresponding segments. In the evaluated configuration, each accelerator invocation processes one 8×8 matrix tile and one 8×1 state vector segment. The CPU supplies the matrix tile together with the corresponding vector segment, receives the resulting partial output, and accumulates partial results when multiple tiles contribute to the same output segment. CPU-controlled partitioning, repeated tile scheduling, data transfers, and cross-tile accumulation are outside the scope of this thesis.

Partitioning makes the computation compatible with a fixed-size accelerator, but it does not by itself reduce the number of arithmetic operations. Further reductions can be achieved by exploiting structure within the matrix, particularly the presence of zero-valued coefficients and repeated patterns.

2.3 Sparse and Compressed Matrix Representations

The density of a matrix is defined as the fraction of its coefficients that are non-zero:

$$D = \frac{N_{\text{nz}}}{N_{\text{rows}} N_{\text{cols}}}, \tag{2.4}$$

where N_{nz} is the number of non-zero coefficients. Matrix sparsity is correspondingly defined as $S = 1 - D$. For a complex coefficient, the value is considered non-zero when either its real or imaginary component is non-zero.

A dense representation stores every matrix coefficient, including explicit zeros. In contrast, the compressed sparse row (CSR) format stores only the non-zero coefficients in a `values` array. Their column positions are stored in `col_idx`, while `row_ptr` identifies which coefficients belong to each matrix row. CSR was selected because this organization maps directly to the accelerator's row-wise multiplication and accumulation, while avoiding the processing of explicit zeros.

In the proposed execution model, each matrix tile is supplied to the accelerator in CSR format, while the corresponding state vector segment remains dense. The detailed CSR dataflow implemented by the accelerator is presented in Section 4.2.

Another form of compression is the quantum multiple-valued decision diagram, which represents matrices and state vectors using a graph structure. Repeated or equivalent substructures can share the same graph nodes, reducing the amount of information that must be stored. An example of this representation is shown in Figure 2.4.

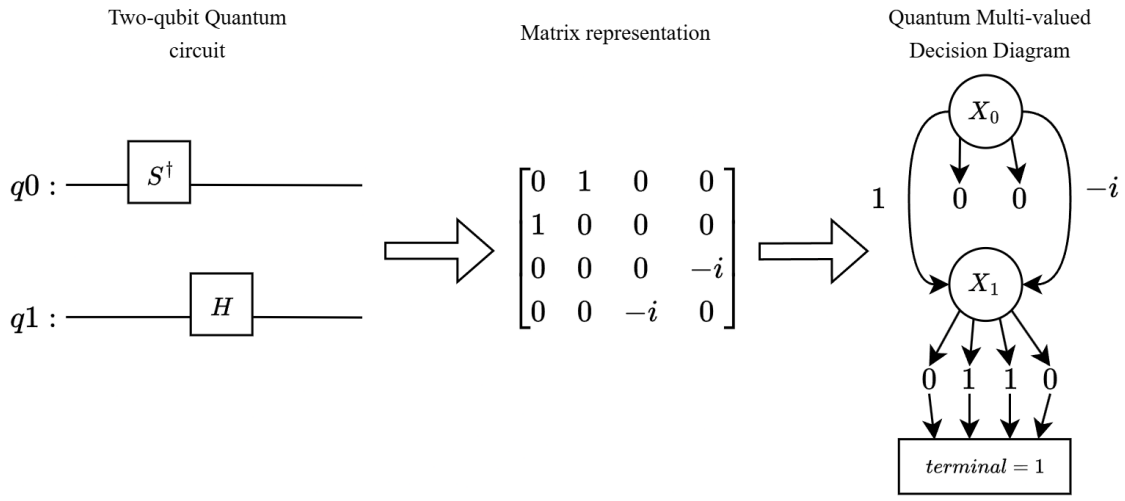


Figure 2.4: Example of a QMDD representation.

QMDDs can significantly reduce storage and computation when a matrix contains repeated or equivalent substructures. Their benefit decreases when the matrix structure becomes irregular and provides fewer opportunities for node sharing. CSR exploits a different property: it removes explicit zero-valued coefficients while preserving direct row-wise access to the remaining values. This makes CSR well suited to the SpMV dataflow used by the proposed accelerator.

The concepts presented in this chapter provide the technical basis for the proposed accelerator. Partitioned matrix–vector multiplication explains how a larger unitary

transformation can be divided into accelerator-sized operations, while sparsity and CSR provide a way to avoid storing and processing explicit zero-valued coefficients. The following chapters describe the methodology used to analyse circuit sparsity and generate synthetic evaluation workloads, the implementation of the CSR-based accelerator, and the evaluation of its timing, area, power, latency, and energy characteristics.

3
Method

3.1 Research Approach

The methodology of this work was divided into three main stages: sparsity analysis, hardware implementation, and evaluation. First, quantum circuits and their corresponding transformation matrices were analysed to determine whether their sparsity could be used to reduce computational workload. Based on this analysis, a CSR-based complex SpMV accelerator was designed and functionally verified. Finally, the accelerator was evaluated in terms of execution latency and hardware characteristics, with particular attention to the effect of matrix sparsity.

3.2 Circuit and Sparsity Analysis

The first stage of the work examined the sparsity of unitary matrices generated from different quantum circuits. The evaluated benchmarks included QFT, Grover, QAOA, random, W-state, and GHZ circuits, with configurations of up to eight qubits.

The benchmark circuits were obtained at the algorithmic level using a quantum circuit benchmark, MQT Bench [6]. Final measurement operations were removed before the corresponding unitary matrices were generated using the Qiskit `Operator` class. For a circuit containing n qubits, this produced a complex unitary matrix with dimensions $2^n \times 2^n$.

Because floating-point calculations can produce small residual values instead of exact zeros, a tolerance of 10^{-12} was used to accommodate accumulated rounding errors; a matrix coefficient was therefore classified as non-zero when its magnitude exceeded this threshold:

$$|U_{r,k}| > 10^{-12}. \tag{3.1}$$

For each generated matrix, the number of non-zero coefficients was counted and used to calculate its density and sparsity according to (2.4). The positions of the non-zero coefficients were also visualized to examine how they were distributed within the matrix.

The purpose of this analysis was to determine whether unitary matrices generated from different circuit structures exhibit sufficiently different sparsity levels to motivate sparse processing. The matrices obtained from this analysis were not supplied

directly to the RTL accelerator. Instead, the accelerator was evaluated separately using synthetic 8×8 matrices, allowing matrix density to be controlled independently of a particular quantum circuit.

3.3 Evaluation

The fully dense 8×8 matrix case was used as the baseline for evaluating sparsity-dependent performance. All sparse and dense matrices were processed by the same accelerator architecture using the same predefined input vector. This isolates the effect of matrix density without introducing differences between hardware implementations.

The speedup at density D was calculated relative to the fully dense case as

$$\text{Speedup}(D) = \frac{T_{\text{dense}}}{T(D)}, \tag{3.2}$$

where T_{dense} is the execution latency at 100% density and $T(D)$ is the latency at the evaluated density. This comparison measures the latency reduction achieved by processing fewer non-zero coefficients; it does not compare the proposed accelerator with a separate dense hardware architecture.

To isolate the effect of matrix density on the proposed accelerator, synthetic 8×8 complex matrices were generated using a Python script. The target density was increased approximately linearly from 10% to 100%, covering cases from highly sparse to fully dense matrices. Since each matrix contained 64 coefficients, the realised density was determined by the integer number of non-zero coefficients according to Equation 2.4.

For each density level, the script randomly selected the positions of the non-zero coefficients and assigned randomly generated complex values to them. All remaining coefficients were set to zero. The resulting matrix was then converted to CSR format before being supplied to the accelerator.

One random matrix configuration was evaluated for each density level. The results therefore show the observed relationship between matrix density and accelerator latency, rather than an average across multiple random sparsity patterns.

3.4 Functional Verification

A fixed predefined dense input vector was used for all matrix-density cases. Keeping the vector unchanged ensured that differences between the experiments were caused by the generated matrices rather than by changes in the input vector.

For each workload, the testbench loaded the matrix coefficients from a CSV file and converted the matrix into CSR format before supplying it to the accelerator. A reference output was calculated separately inside the testbench using dense complex matrix–vector multiplication and SystemVerilog `real` arithmetic.

After the accelerator asserted the done signal, every element of the output vector was compared with the corresponding reference result. The 64-bit representations of the real and imaginary components were compared exactly, without applying an additional numerical tolerance. Any mismatch caused the simulation to terminate with an error. The testbench also verified that the CSR-invalid signal remained deasserted and that all eight output rows were checked.

A test case was considered successful only when every output element matched the reference result and no CSR-format error was reported.

3.5 Synthesis

The accelerator was implemented in SystemVerilog to facilitate integration with X-HEEP, which uses the same hardware description language. The design was synthesized using Cadence Genus with the ASAP7 standard-cell library [7]. The same library was used throughout the evaluation to provide a consistent technology reference for comparing timing, cell area, and estimated power. A separate synthesis run was performed for each target clock frequency, allowing the tool to optimize the implementation under the corresponding timing constraint.

An initial synthesis sweep was performed at target frequencies of 100, 200, 300, 400, and 500 MHz. Additional synthesis points at 420, 440, and 460 MHz were subsequently evaluated to provide a finer characterization near the timing limit of the design.

Timing closure at each synthesis point was determined from the worst negative slack (WNS) reported by Genus. A synthesis point was considered timing-clean when the reported WNS was non-negative. The frequency sweep was used to identify the highest evaluated target frequency satisfying the timing constraint and to bound the timing limit between adjacent passing and failing synthesis points.

The implementation area and total power estimates were obtained from the corresponding post-synthesis Genus reports. These values were used to examine how increasingly strict timing constraints affected the hardware cost of the accelerator. The workload-specific power estimates used for the energy evaluation were obtained separately using VCD switching activity, as described in Section 3.6.

The reported synthesis results characterize the accelerator datapath before physical implementation. They do not include placement, routing, clock-tree synthesis, or post-layout interconnect parasitics.

3.6 Simulation and Measurement

The accelerator was simulated at both RTL and gate level. Verilator was used for RTL simulation to verify the functional behavior of the SystemVerilog implementation. Cadence Xcelium was used for gate-level simulation of the synthesized netlist with the ASAP7 standard-cell models, allowing the implementation to be checked after synthesis.

Execution latency was measured from the assertion of the accelerator start signal until the assertion of the done signal. This interval includes vector permutation, complex multiplication, and row accumulation for one 8×8 matrix tile.

The number of execution cycles was converted to time using the clock frequency corresponding to each timing-clean post-synthesis implementation. For a measured cycle count N_{cycles} and clock frequency f_{clk}, the execution latency was calculated as

$$T_{\text{exec}} = \frac{N_{\text{cycles}}}{f_{\text{clk}}}. \tag{3.3}$$

The measurement does not include CPU-controlled matrix partitioning, transfer of tiles between memory and the accelerator, or accumulation of partial results across multiple tiles. These system-level operations were outside the scope of the evaluation.

For the workload-specific energy evaluation, a separate Value Change Dump (VCD) file was generated during RTL simulation for each matrix-density workload. The VCD files were supplied to Cadence Genus to annotate the switching activity of the synthesized design and obtain a workload-specific total power estimate. The estimated energy required for one tile-level SpMV operation was then calculated as

$$E_{\text{SpMV}} = P_{\text{total}} T_{\text{exec}}, \tag{3.4}$$

where P_{total} is the workload-specific power estimate reported by Genus and T_{exec} is the measured execution time from the assertion of the start signal to the assertion of the done signal.

4

Design

4.1 Design Overview

The proposed architecture implements SpMV for complex-valued data, where SpMV denotes sparse matrix–vector multiplication. Its purpose is to accelerate the matrix–vector operations used in quantum circuit simulation by exploiting the sparsity of the corresponding transformation matrices. The matrix is stored in compressed sparse row (CSR) format, while the state vector is stored in dense form. Both the real and imaginary components of every value are represented using double-precision floating-point numbers.

For one accelerator invocation, a tile of the unitary matrix is applied to the corresponding segment of the state vector. This produces a partial output vector:

$$\mathbf{p}_{ij} = \mathbf{U}_{ij}\mathbf{x}_j, \tag{4.1}$$

where $\mathbf{U}_{ij}$ is a matrix tile, $\mathbf{x} * j$ is the corresponding state-vector segment, and $\mathbf{p} * ij$ is the partial result. The evaluated accelerator configuration processes 8×8 matrix tiles and eight-element vector segments. Multiple partial results may contribute to the same output segment and are assumed to be accumulated by the CPU.

The proposed design avoids this unnecessary work by storing and processing only the non-zero matrix coefficients. This choice is motivated by the structure of many operators used in quantum circuit simulation, whose matrix representations contain a large proportion of zero-valued elements. The resulting computational workload depends mainly on the number and distribution of non-zero coefficients rather than on the total number of matrix elements.

The remainder of this chapter first explains the CSR dataflow through a worked example. It then describes the top-level accelerator and its vector permutation, complex multiplication, and row accumulation stages. Finally, it discusses the relationship between sparsity and execution latency, followed by the intended integration with the X-HEEP RISC-V platform [8].

4.2 CSR Representation and Dataflow

In CSR format, the non-zero coefficients of a matrix are stored in the `values` array. The corresponding column positions are stored in `col_idx`, while `row_ptr` identi-

fies the range of stored coefficients belonging to each matrix row. For row r, the corresponding output element is calculated as

$$y_r = \sum_{k=\text{row_ptr}[r]}^{\text{row_ptr}[r+1]-1} \text{values}[k]\, x[\text{col_idx}[k]]\,. \tag{4.2}$$

The equation shows that only the vector elements referenced by stored matrix coefficients are required. The dense matrix therefore does not need to be reconstructed before the multiplication is performed.

To illustrate this dataflow, consider the following 4×4 complex matrix:

$$A = \begin{bmatrix} 1+j & 0 & 2 & 0 \\ 0 & 0 & 3-j & 0 \\ 4 & 0 & 0 & 2j \\ 0 & 0 & 0 & 0 \end{bmatrix}, \tag{4.3}$$

and the input vector

$$x = \begin{bmatrix} 1 \\ 2-j \\ 1+j \\ 2 \end{bmatrix}. \tag{4.4}$$

Instead of storing all 16 elements of A, the CSR representation stores only its five non-zero coefficients:

$$\text{values} = [1+j,\ 2,\ 3-j,\ 4,\ 2j], \tag{4.5}$$
$$\text{col_idx} = [0,\ 2,\ 2,\ 0,\ 3], \tag{4.6}$$
$$\text{row_ptr} = [0,\ 2,\ 3,\ 5,\ 5]. \tag{4.7}$$

Table 4.1 shows how each stored coefficient is associated with an element of the input vector.

Table 4.1: Relationship between the CSR entries and vector elements in the example matrix.

CSR index k	values[k]	col_idx[k]	Selected vector element
0	$1+j$	0	x_0
1	2	2	x_2
2	$3-j$	2	x_2
3	4	0	x_0
4	$2j$	3	x_3

Using `col_idx`, the required vector elements can be arranged in the same order as the stored coefficients:

$$x_{\text{perm}} = [x_0,\ x_2,\ x_2,\ x_0,\ x_3] = [1,\ 1+j,\ 1+j,\ 1,\ 2]. \tag{4.8}$$

Each selected vector element is then multiplied by the corresponding matrix coefficient:

$$p_0 = (1+j)(1) = 1+j, \tag{4.9}$$
$$p_1 = 2(1+j) = 2+2j, \tag{4.10}$$
$$p_2 = (3-j)(1+j) = 4+2j, \tag{4.11}$$
$$p_3 = 4(1) = 4, \tag{4.12}$$
$$p_4 = (2j)(2) = 4j. \tag{4.13}$$

The `row_ptr` array determines how these products are grouped. For the first row, row_ptr[0] = 0 and row_ptr[1] = 2, so p_0 and p_1 are accumulated:

$$y_0 = p_0 + p_1 = 3+3j. \tag{4.14}$$

The second row contains only p_2 and therefore requires no complex addition:

$$y_1 = p_2 = 4+2j. \tag{4.15}$$

The third row contains p_3 and p_4:

$$y_2 = p_3 + p_4 = 4+4j. \tag{4.16}$$

Finally, row_ptr[3] = row_ptr[4] = 5 indicates that the fourth row contains no stored coefficients, giving

$$y_3 = 0. \tag{4.17}$$

The complete output vector is consequently

$$y = \begin{bmatrix} 3+3j \\ 4+2j \\ 4+4j \\ 0 \end{bmatrix}. \tag{4.18}$$

This example illustrates the three operations implemented by the accelerator: selecting vector elements using the column indices, multiplying the selected elements by the stored coefficients, and accumulating the products according to the row pointers. It also shows that different rows can require different amounts of reduction work. In this example, the first and third rows each require one complex addition, whereas the second and fourth rows require none.

4.3 Proposed Architecture

The top-level structure of the proposed accelerator is shown in Figure 4.1. It implements the CSR dataflow using three main processing stages: vector permutation, parallel complex multiplication, and row accumulation. The top level also contains transaction registers and a finite-state machine (FSM) that controls the execution sequence.

The accelerator receives four input data structures. The CSR matrix is provided through `csr_values_i`, `col_idx_i`, and `row_ptr_i`, while `vector_i` contains the dense input vector. At the beginning of a transaction, these inputs are captured by internal registers. This prevents the operands from changing during the multi-cycle computation. The registered values are denoted by `csr_values_q`, `col_idx_q`, `row_ptr_q`, and `vector_q`.

After the transaction has been registered, the control FSM activates the processing stages in sequence. The vector permutator first selects and orders the required vector elements according to `col_idx_q`. The vector multiplier then computes the products between these elements and the corresponding coefficients in `csr_values_q`. Finally, the row accumulator uses `row_ptr_q` to combine the products and construct the output vector.

Valid/ready handshake signals coordinate data transfer between the processing blocks. These signals allow each multi-cycle stage to indicate when its input has been accepted and when its output is available. The design is therefore a controlled sequence of processing stages; the term *pipeline* refers to the staged datapath rather than to fully overlapped execution of multiple SpMV transactions.

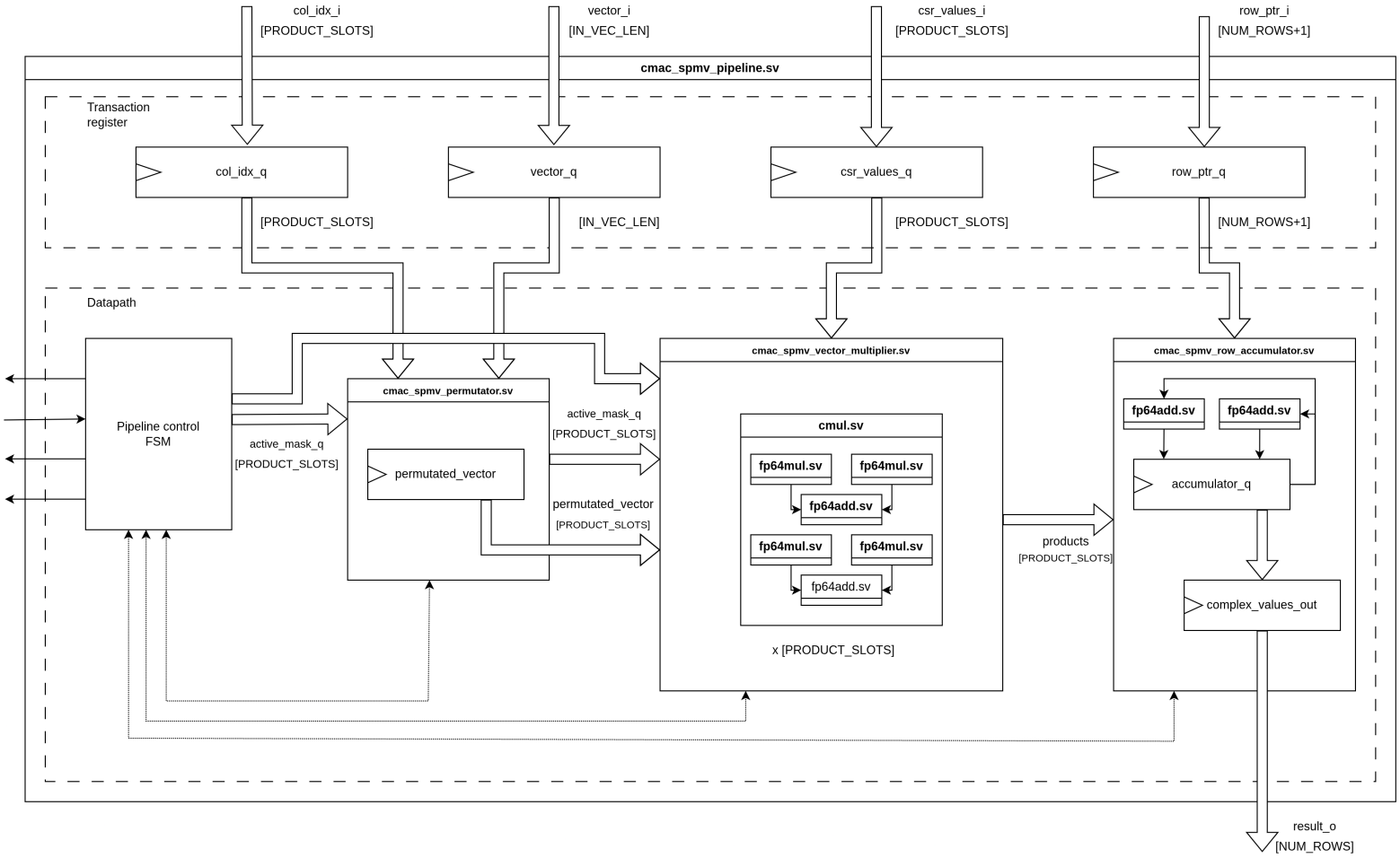


Figure 4.1: Top-level architecture of the proposed CSR-based complex SpMV accelerator.

4.3.1 Vector Permutation and Multiplication

The first datapath operation is vector permutation. For every stored matrix coefficient k, the corresponding vector element is selected using its CSR column index:

$$x_{\text{perm}}[k] = x[\text{col_idx}[k]]\,. \tag{4.19}$$

This operation aligns each non-zero matrix coefficient with the vector element required for its multiplication. An active mask identifies the product slots that contain valid CSR entries. Inactive slots represent unused capacity and do not contribute to the result.

The permuted vector and the non-zero matrix coefficients are passed to the vector multiplier. This block contains multiple complex multiplication units operating in parallel. For each active product slot k, it evaluates

$$p_k = \text{values}[k]\, x_{\text{perm}}[k]. \tag{4.20}$$

Replicating the complex multiplication units provides spatial parallelism across the stored coefficients. The resulting products are forwarded to the row accumulator after all active multiplication operations have completed.

Complex Multiplication Unit

Figure 4.2 shows one complex multiplication unit used as a processing element within the vector multiplier.

For two complex input values

$$a = a_{\text{real}} + j a_{\text{imag}}, \qquad b = b_{\text{real}} + j b_{\text{imag}}, \tag{4.21}$$

their product is

$$ab = (a_{\text{real}} b_{\text{real}} - a_{\text{imag}} b_{\text{imag}}) + j\,(a_{\text{real}} b_{\text{imag}} + a_{\text{imag}} b_{\text{real}})\,. \tag{4.22}$$

The complex multiplication unit therefore performs four FP64 multiplications and two FP64 addition or subtraction operations. The four partial products are computed in parallel. One arithmetic unit then subtracts the products that form the real component, while the other adds the products that form the imaginary component.

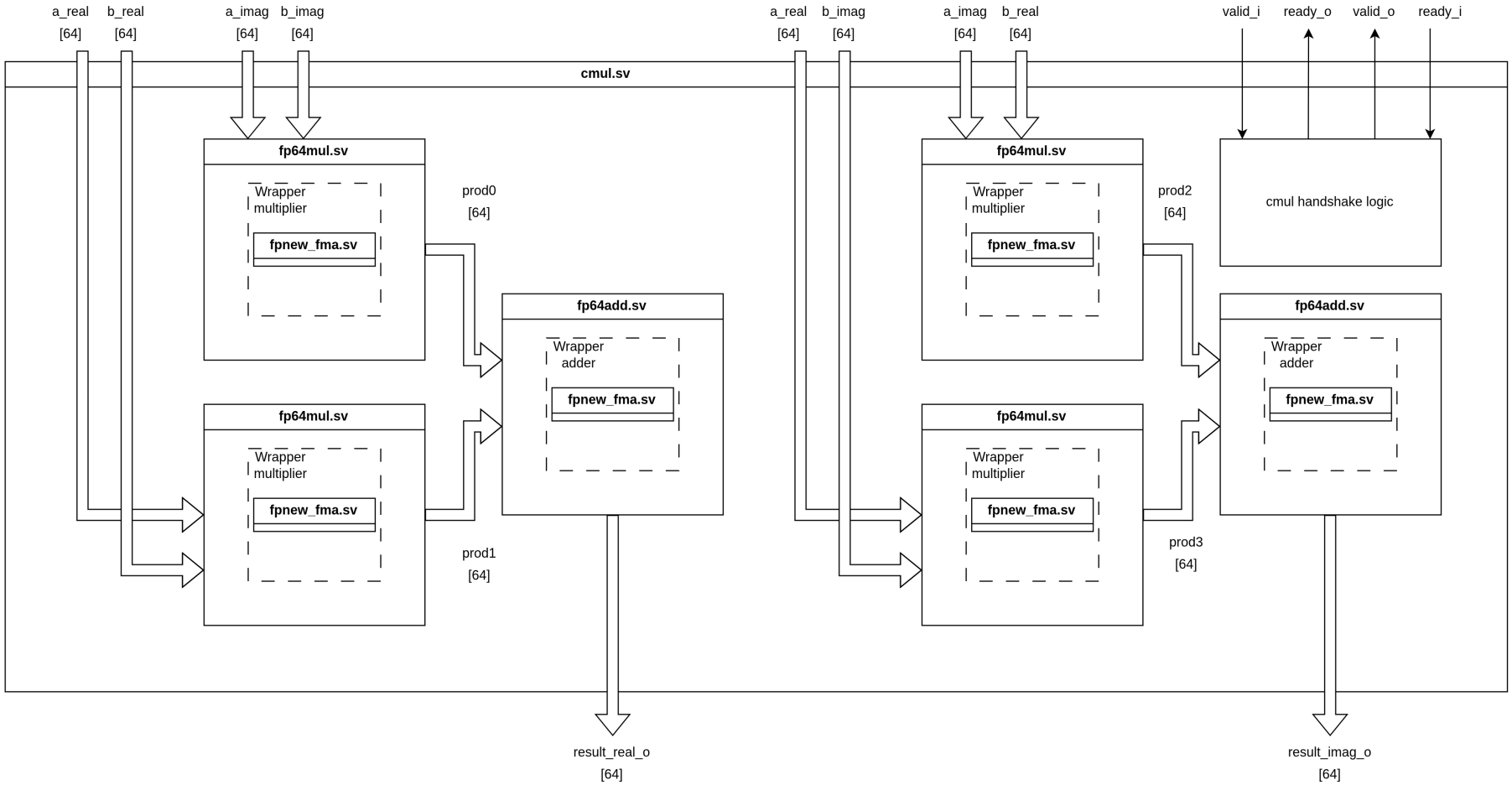


Figure 4.2: Architecture of a complex multiplication processing element used by the parallel vector multiplier.

The floating-point datapath is implemented using wrappers around FPnew arithmetic units [9]. Valid/ready handshake logic allows these multi-cycle units to communicate with the surrounding stages without assuming a fixed single-cycle latency. Multiple instances of the complex multiplication unit are instantiated in the vector multiplier so that several non-zero coefficients can be processed concurrently.

4.3.2 Row Accumulation

The vector multiplier produces a set of independent complex products. The row accumulator, shown in Figure 4.3, combines these products according to the rows of the original sparse matrix.

The product selector FSM uses `row_ptr` to identify the products that belong to the current row. For row r, it selects the interval

$$\text{row_ptr}[r] \leq k < \text{row_ptr}[r+1]. \tag{4.23}$$

Because each product is complex, its real and imaginary components are accumulated independently. Two FP64 adders operate in parallel: one adds the real components and the other adds the imaginary components. Their outputs are stored in the complex accumulator register `accumulator_q`.

After all products belonging to a row have been processed, the completed complex value is written to the corresponding position in the output vector. The accumulator then advances to the next row and repeats the operation until all output elements have been generated. An empty row produces zero, while a row containing one stored coefficient can use its product directly.

Unlike the multiplication stage, row accumulation is sequential. This choice reduces the number of floating-point adders required, but makes the accumulation latency dependent on the number of products in each row, hence type of execution that is sparsity-dependent.

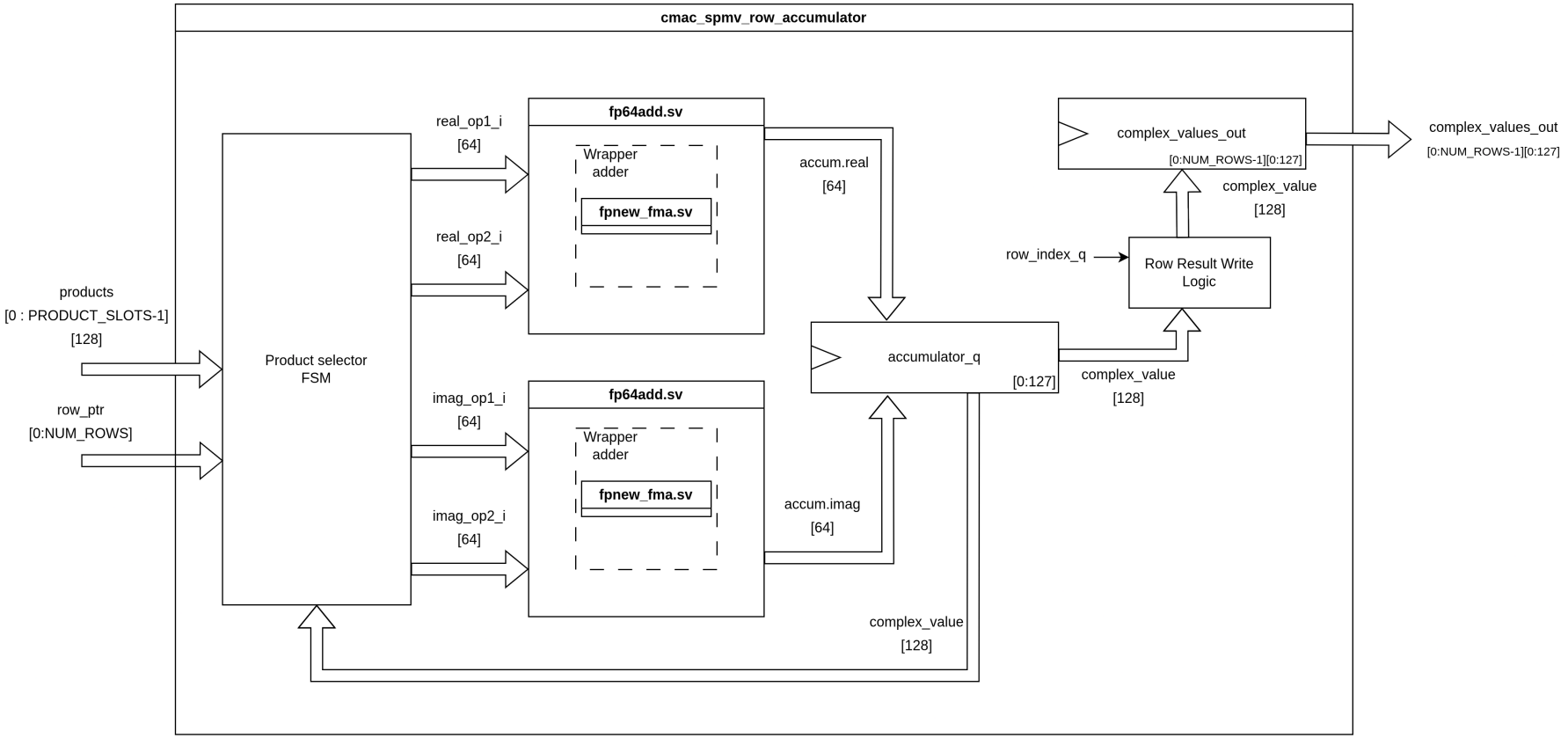


Figure 4.3: Architecture of the CSR row accumulator.

4.4 Sparsity-Dependent Execution

CSR removes explicit zero-valued coefficients from the input representation. Consequently, such coefficients require neither storage in the accelerator input nor complex multiplication. Sparsity also affects the number of operations performed by the sequential row accumulator.

For a row containing n_r non-zero coefficients, and with the first product used to initialise the row result, the number of required complex additions is

$$N_{\text{add},r} = \max(n_r - 1, 0). \tag{4.24}$$

A row with no stored coefficients therefore produces zero without an addition, while a row with one coefficient uses its product directly. Sequential additions are required only when a row contains two or more non-zero coefficients.

For the complete matrix, the total number of complex additions is

$$N_{\text{add}} = \sum_r \max(n_r - 1, 0). \tag{4.25}$$

If N_{nz} is the total number of non-zero matrix coefficients and R_{active} is the number of non-empty rows, this expression can be written as

$$N_{\text{add}} = N_{\text{nz}} - R_{\text{active}}. \tag{4.26}$$

The main source of sparsity-dependent latency is therefore the row accumulation stage. The multiplication stage processes the available product slots in parallel, whereas the accumulator performs a variable number of sequential reductions determined by the distribution of non-zero coefficients.

This behaviour is relevant to quantum circuit simulation because different operators can have substantially different sparsity patterns. A permutation-like operator may contain only one non-zero coefficient per row and thus require no reduction after multiplication. An operator with several non-zero coefficients per row requires additional accumulation cycles. Consequently, the same hardware can execute SpMV operations with different latencies depending on the structure of the represented operator.

The architecture therefore combines parallel complex multiplication with sparsity-dependent sequential accumulation. This provides a compromise between arithmetic parallelism and hardware resource usage, while allowing execution time to reflect the amount of useful work in the sparse matrix.

4.5 Intended Coprocessor Integration

The accelerator was designed to operate as a coprocessor for the X-HEEP RISC-V platform [8]. In addition to the SpMV datapath described above, an interface

module was developed to connect the accelerator to the processor and provide access to external memory.

In the intended execution model, the complete unitary matrix and state vector are divided into accelerator-sized tiles and vector segments. The CPU supplies one CSR-encoded matrix tile together with the corresponding dense state-vector segment and initiates the operation through a custom instruction. The accelerator then computes the partial matrix–vector product and returns the resulting vector to the CPU.

When several matrix tiles contribute to the same output segment, the CPU is assumed to accumulate their partial results before storing the completed segment. CPU-controlled partitioning, CSR-data preparation, scheduling of repeated accelerator operations, and accumulation across matrix tiles were not implemented or evaluated in this work.

An RTL interface module was implemented to connect the accelerator to an X-HEEP processor through the CORE-V eXtension Interface (CV-X-IF) [10, 11]. However, the complete processor-coprocessor workflow was not evaluated. The synthesis and performance results reported in Section 5.2 characterize the standalone accelerator processing one 8×8 matrix tile. They do not include instruction-issue, memory-access, or other system-level overheads, and they do not establish the performance of a complete quantum circuit simulation.

5

Results

5.1 Initial analysis of quantum circuits

An initial circuit analysis was performed to examine the sparsity of unitary matrices generated from different quantum circuits. The analysis included QFT, Grover, QAOA, W-state, GHZ, and randomly generated circuits with configurations of up to eight qubits. To provide a direct comparison between circuits of equal size, Figure 5.1 and Table 5.1 present the results for the six-qubit configurations, whose unitary matrices contain 64×64 coefficients. A coefficient was classified as non-zero when its magnitude exceeded the numerical tolerance of 10^{-12}. In the figure, black cells represent non-zero coefficients and white cells represent zeros.

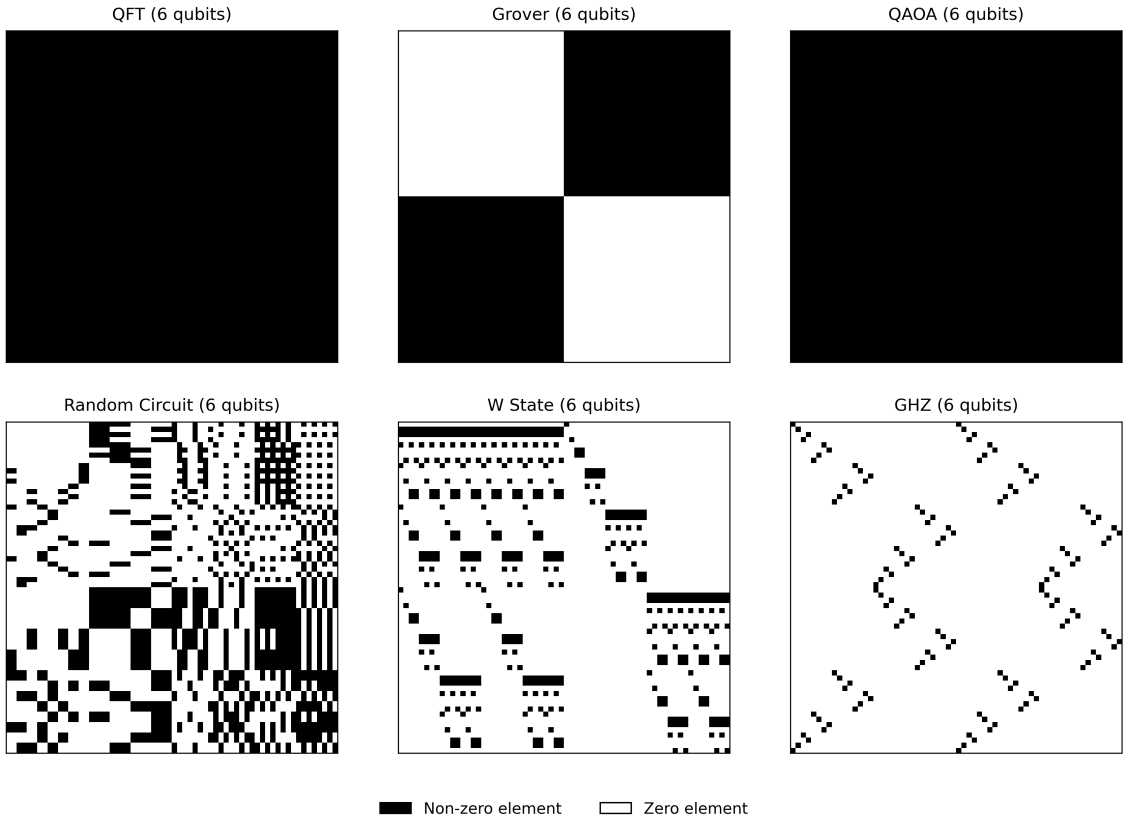


Figure 5.1: Representative sparsity patterns obtained from quantum-circuit unitary matrices.

Table 5.1: Sparsity characteristics of the unitary matrices for the evaluated six-qubit quantum circuits. Each matrix has dimensions 64×64 and contains 4096 elements.

Benchmark	Non-zero elements	Density (%)	Sparsity (%)
QFT	4096	100.00	0.00
Grover	2048	50.00	50.00
QAOA	4096	100.00	0.00
Random Circuit	1392	33.98	66.02
W State	486	11.87	88.13
GHZ	128	3.13	96.88

The six-qubit results show that matrix sparsity depends strongly on circuit structure. The QFT and QAOA matrices are fully dense, whereas the Grover matrix has a sparsity of 50.00%. The random circuit, W-state, and GHZ matrices exhibit progressively higher sparsities of 66.02%, 88.13%, and 96.88%, respectively. These results demonstrate that sparsity-based acceleration is strongly workload dependent: some circuit matrices provide substantial opportunities to skip zero-valued coefficients, while others provide none. This variation motivates evaluating the proposed accelerator across a broad range of matrix densities. The accelerator evaluation uses separately generated synthetic matrices so that density can be controlled independently of a particular quantum circuit.

5.2 Design evaluation

The proposed accelerator was evaluated in two stages. First, the design was synthesized using Cadence Genus under a range of clock-frequency constraints to determine its achievable operating frequency and the associated area and power trade-offs. Second, the RTL design was simulated using clock periods corresponding to the timing-clean post-synthesis frequency points. Matrices of different densities were used to evaluate how sparsity affects execution latency, speedup, and estimated energy per SpMV.

For the RTL evaluation, the architecture was configured for an 8×8 complex matrix and an 8×1 complex input-vector segment. Synthetic matrices containing randomly positioned non-zero coefficients were used, while the input vector contained fixed predefined values. The implementation was dimensioned to support the fully dense case, in which all 64 matrix elements must be represented by the CSR datapath. Each measurement therefore represents one 8×8 tile-level SpMV operation.

The results in this section characterize the standalone accelerator datapath. In its intended CPU-assisted use, a larger unitary matrix would be divided into tiles, and the CPU would supply each tile together with the corresponding state-vector segment. The CPU is assumed to store and accumulate the resulting partial outputs. Although interface logic was developed for the intended use of the accelerator as a coprocessor for the X-HEEP RISC-V platform, the complete processor–coprocessor system was not evaluated. Processor interaction, data-transfer overhead, cross-tile

accumulation, and custom-instruction overhead are therefore not included in the reported results.

5.2.1 Synthesis characterization

An initial synthesis sweep was performed between 100 and 500 MHz. After the upper timing limit of the design became apparent, additional synthesis points were evaluated at 420, 440, and 460 MHz to obtain a finer characterization close to the maximum achievable clock frequency.

Table 5.2 summarizes the synthesis results. The area values correspond to the same area metric used in Figure 5.3, while the power values are the total power estimates reported for the frequency sweep.

Table 5.2: Summary of synthesis results for different target frequencies.

Frequency [MHz]	WNS [ns]	Area [μm^2]	Power [W]	Timing met?
100	0.000	582181	0.546	Yes
200	0.000	583872	0.899	Yes
300	0.000	617592	1.184	Yes
400	0.000	998690	2.144	Yes
420	0.000	1038945	2.277	Yes
440	0.000	1073949	2.401	Yes
460	-0.0143	1096294	2.502	No
500	-0.1723	1171582	2.812	No

Timing

Figure 5.2 shows the worst negative slack (WNS) obtained for the different target clock frequencies.

Timing closure was achieved for all evaluated frequencies up to and including 440 MHz. At 460 MHz, the first timing violation was observed, with a WNS of -0.0143 ns. Increasing the target frequency to 500 MHz resulted in a significantly larger violation of -0.1723 ns.

The results therefore place the timing limit of the synthesized design between 440 and 460 MHz. Among the evaluated synthesis points, 440 MHz is the highest frequency for which timing closure was demonstrated, while the implementation constrained to 460 MHz exhibits negative slack. Since no intermediate frequencies were evaluated, the exact maximum achievable frequency within this range was not determined.

In addition to the WNS, the number of violating paths increased substantially once timing closure was lost. At 460 MHz, 301 paths violated the timing constraint, whereas at 500 MHz this increased to 6193 paths. The corresponding total negative slack increased from -2.4422 ns to -66.2088 ns. This indicates that the 500 MHz constraint is not limited by only one marginal critical path, but exceeds the timing capability of a considerably larger portion of the design.

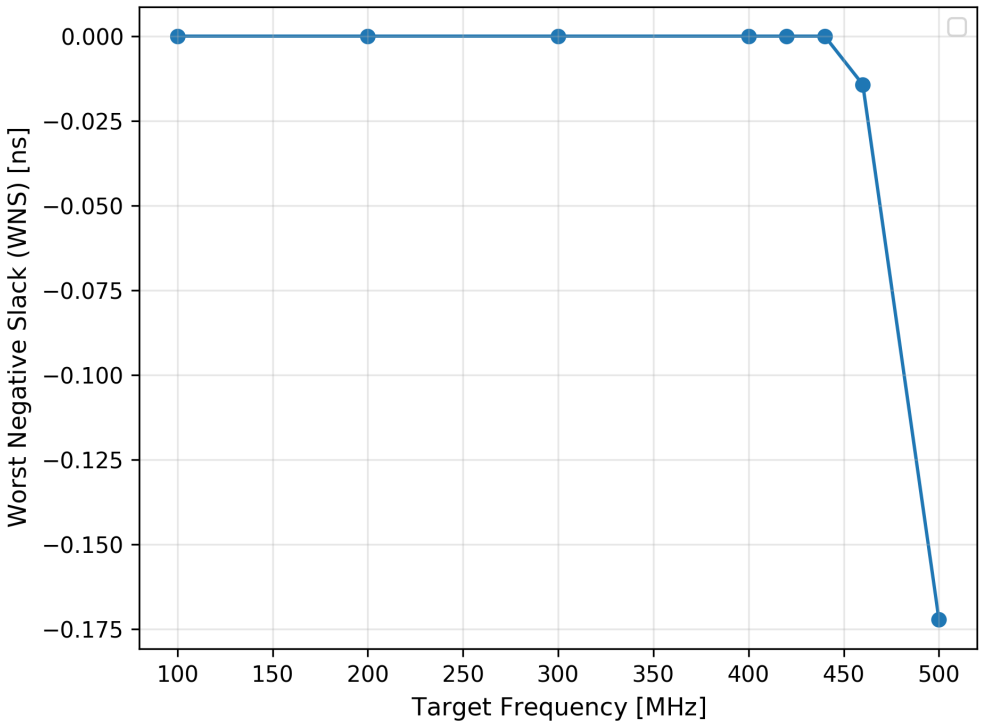


Figure 5.2: Worst negative slack as a function of the synthesis target frequency.

The location of the reported critical path also changes with the target frequency. For synthesis targets between 100 and 300 MHz, the worst path is located within the parallel complex multiplication stage. From 400 MHz onward, the reported worst path instead passes through the floating-point addition logic of the row accumulator. Among the evaluated synthesis points, this identifies the row accumulation stage as the primary timing-limiting block at higher operating frequencies.

Area

Figure 5.3 shows the synthesized implementation area as a function of the target frequency.

At the lower synthesis frequencies, the area remains relatively stable. The implementation occupies approximately 5.82×10^5 μm^2 at 100 MHz and changes only marginally at 200 MHz. At 300 MHz, the area increases to approximately 6.18×10^5 μm^2.

A considerably larger increase occurs between 300 and 400 MHz. At 400 MHz, the synthesized area is approximately 9.99×10^5 μm^2, corresponding to an increase of approximately 62% relative to the 300 MHz implementation. The area subsequently increases more gradually, reaching approximately 1.07×10^6 μm^2 at 440 MHz.

At the highest evaluated target of 500 MHz, the implementation occupies approximately 1.17×10^6 μm^2, approximately twice the area of the 100 MHz implementation. However, this configuration does not satisfy the requested timing constraint.

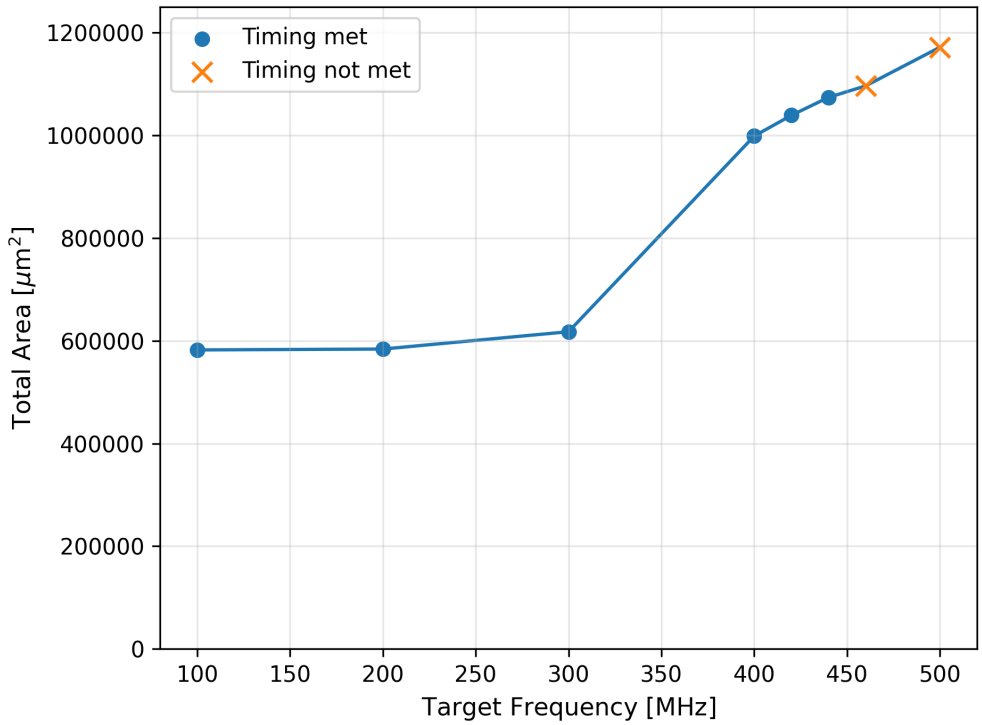


Figure 5.3: Total synthesized area as a function of the target clock frequency.

The increase in area under tighter clock constraints indicates that the synthesis tool increasingly trades implementation resources for timing improvement, for example through cell selection, buffering, and logic restructuring. The pronounced increase between 300 and 400 MHz therefore marks a clear rise in implementation cost as the target frequency is increased.

Power

The corresponding total power estimates are shown in Figure 5.4.

The estimated total power increases from approximately 0.55 W at 100 MHz to 0.90 W at 200 MHz and 1.18 W at 300 MHz. A larger increase is again observed between 300 and 400 MHz, where the power rises to approximately 2.14 W.

At 440 MHz, which is the highest evaluated target for which timing closure was achieved, the estimated total power is approximately 2.40 W. This corresponds to approximately 4.4 times the power of the 100 MHz implementation. At 460 MHz and 500 MHz, the estimated power continues to increase to approximately 2.50 W and 2.81 W, respectively, although these implementations no longer satisfy timing.

The leakage component remains small compared with the dynamic power over the entire frequency range. Consequently, the increase in total power is dominated by dynamic power and is associated both with the higher operating frequency and with the larger implementation produced under tighter timing constraints.

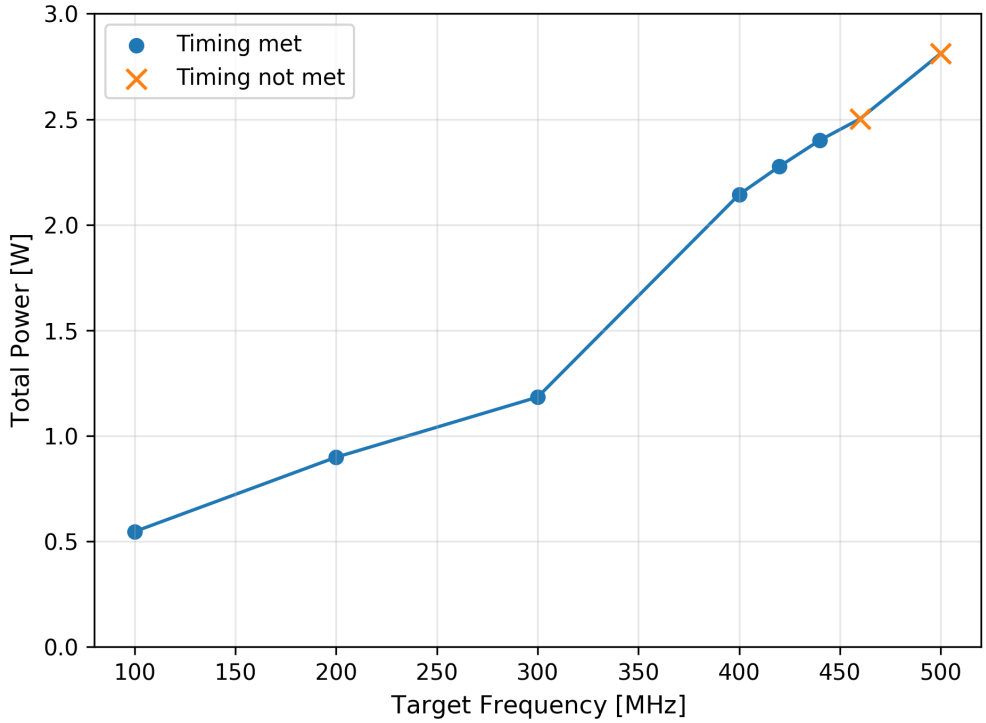


Figure 5.4: Estimated total power as a function of the synthesis target frequency.

Area–power–frequency trade-off

Figure 5.5 presents the area and power results normalized to the 100 MHz implementation.

The normalized results highlight that increasing the target frequency has a substantial implementation cost. Between 100 and 440 MHz, the synthesized area increases by approximately 84%, whereas the estimated total power increases by approximately 339%. The 440 MHz implementation therefore requires approximately 1.84× the area and 4.39× the power of the 100 MHz implementation.

The results also reveal a noticeable implementation-cost transition between 300 and 400 MHz. A 33% increase in target frequency over this interval results in approximately a 62% increase in area and an 81% increase in estimated power. Beyond 400 MHz, the area and power continue to increase while the available timing margin approaches zero.

Among the evaluated synthesis points, 440 MHz represents the highest frequency for which timing closure was demonstrated. The 460 MHz result is close to timing closure but exhibits negative slack, while the 500 MHz constraint results in substantially larger timing violations. Overall, the synthesis results demonstrate a clear trade-off between clock frequency, silicon area, and power consumption, with the row accumulator ultimately limiting further frequency scaling in the current architecture.

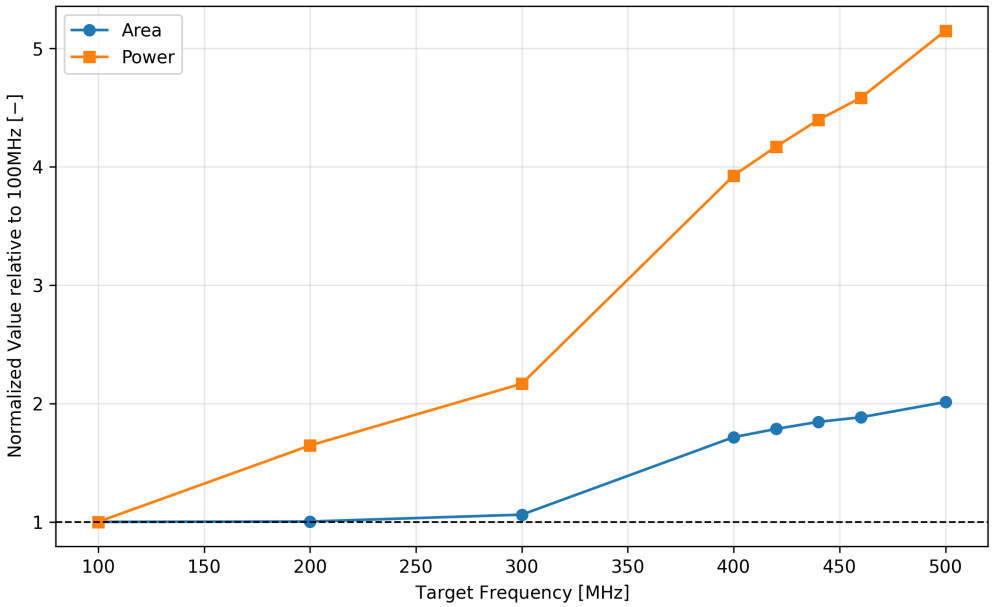


Figure 5.5: Area and power normalized to the 100 MHz synthesis result.

5.2.2 Sparsity evaluation

Following the synthesis characterization, the effect of matrix sparsity on the execution behaviour of the accelerator was evaluated. Only implementations for which timing closure was achieved were included, corresponding to target frequencies of 100, 200, 300, 400, 420, and 440 MHz.

For each implementation, an 8×8 matrix was evaluated at requested densities from 10% to 100%. Because the matrix contains only 64 elements, the requested densities cannot in general be represented exactly. The test matrices therefore contain between 6 and 64 non-zero coefficients, with actual densities ranging from 9.4% to 100%.

Each density point was evaluated using a single pseudorandom sparsity pattern generated with a fixed random seed. Multiple seeds were initially considered, but the total time required to simulate every density and frequency configuration made averaging impractical. The results should therefore be interpreted as representative observations for the evaluated matrix patterns rather than as statistical averages over all possible matrices at each density. All simulations included in the results completed successfully.

Execution latency

Figure 5.6 shows the execution latency as a function of matrix density for each timing-clean implementation.

The execution latency decreases consistently as the number of non-zero matrix coefficients is reduced. This behaviour follows from the sparsity-dependent execution of the accelerator. The multiplication stage processes active coefficients in parallel,

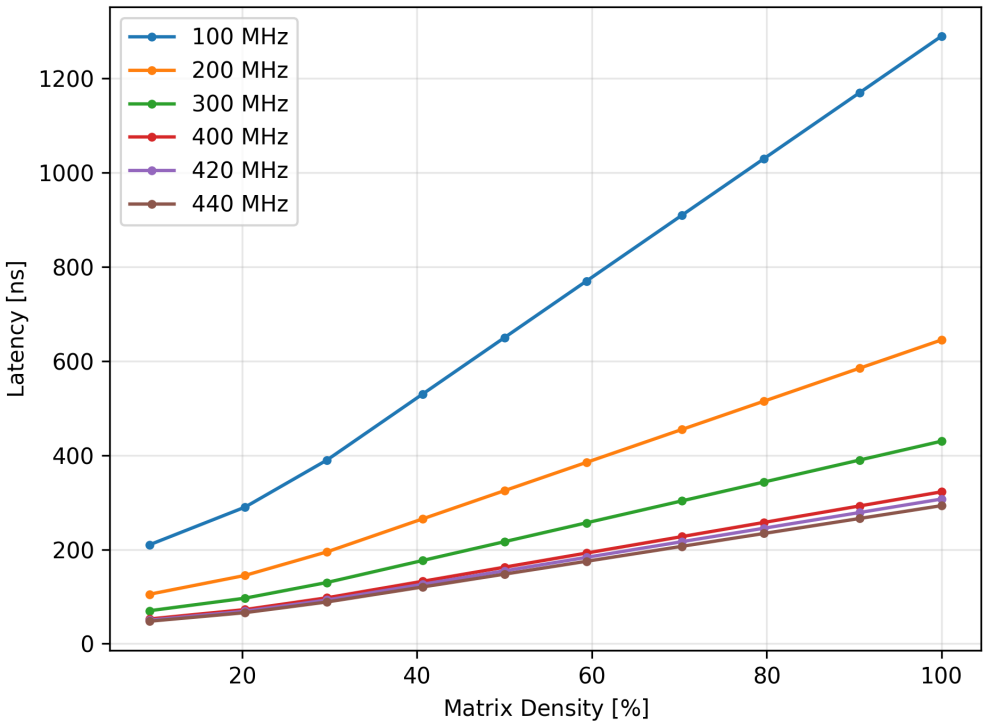


Figure 5.6: Execution latency of one tile-level SpMV operation as a function of matrix density and operating frequency.

while the row accumulator performs a variable number of sequential reduction operations. Reducing the number of stored matrix coefficients therefore reduces the total number of cycles required for one SpMV operation.

For the fully dense matrix containing 64 non-zero coefficients, the operation requires 129 clock cycles. At the lowest evaluated density of 9.375%, corresponding to six non-zero coefficients, the execution time is reduced to 21 cycles. At 440 MHz, this corresponds to a latency reduction from approximately 293.2 ns to 47.7 ns.

The same cycle count is observed for a given sparsity pattern across all frequency points. Changing the operating frequency therefore scales the absolute execution time but does not change the number of cycles required by the algorithm. As a consequence, the relative speedup obtained from sparsity is independent of clock frequency for the evaluated implementations.

Speedup from sparsity

The speedup relative to the fully dense case is defined as

$$S(d) = \frac{T_{\text{dense}}}{T_d}, \tag{5.1}$$

where T_{dense} is the execution time for the 100%-dense matrix and T_d is the execution time at density d. Since both measurements at a given frequency point use the same

clock frequency, the speedup can equivalently be calculated from the execution cycle counts as

$$S(d) = \frac{C_{\text{dense}}}{C_d}. \tag{5.2}$$

Figure 5.7 compares the measured speedup with an ideal reference obtained by assuming execution time is directly proportional to the number of non-zero matrix coefficients.

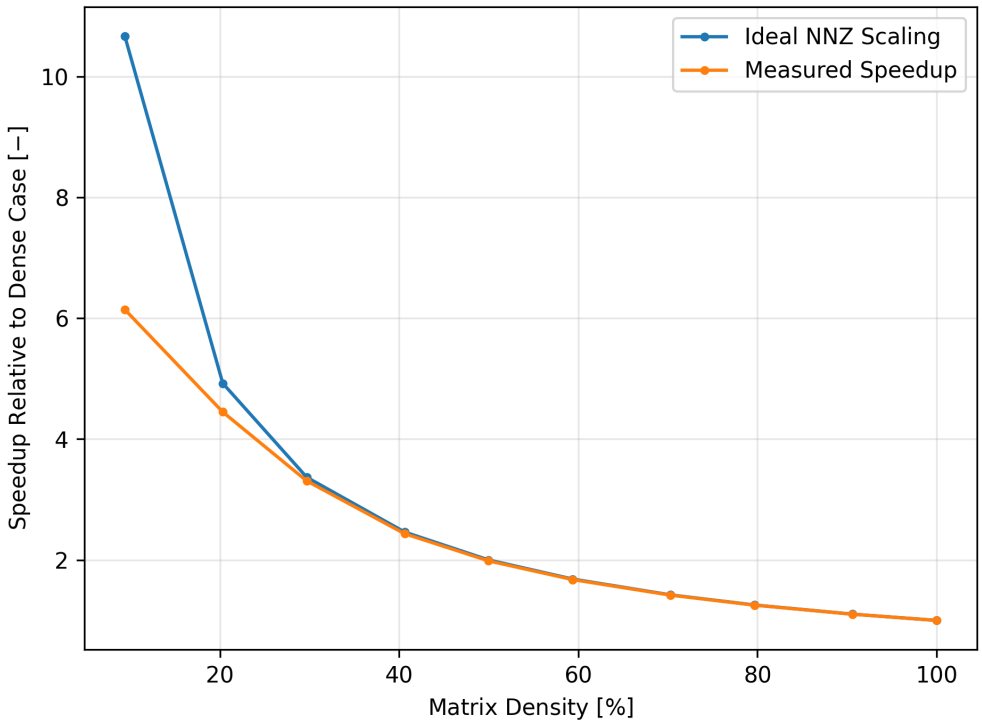


Figure 5.7: Measured speedup relative to a 100%-dense tile processed by the same CSR-based accelerator, compared with ideal scaling based on the number of non-zero coefficients.

At 50% matrix density, the accelerator requires 65 cycles compared with 129 cycles for the fully dense matrix, giving a measured speedup of approximately 1.98×. This is close to the ideal value of 2.0×. Similar near-proportional scaling is observed over the moderate-density region.

The difference between measured and ideal scaling becomes larger for highly sparse matrices. At an actual density of 9.375%, the ideal speedup based only on the ratio of non-zero coefficients is

$$\frac{64}{6} \approx 10.67, \tag{5.3}$$

whereas the measured speedup is approximately 6.14×. The baseline for this comparison is a 100%-dense tile processed by the same CSR-based accelerator, rather

than a separate dense matrix–vector implementation. The comparison isolates the effect of matrix density within the proposed architecture and excludes the cost of preparing CSR data.

The departure from ideal scaling indicates that fixed control overhead becomes increasingly important as the useful arithmetic workload decreases. In particular, the pipeline still incurs control and handshaking overhead, and the row accumulator must process the CSR row structure even when individual rows contain few or no values. These costs represent a relatively small fraction of the dense execution time but become significant at very low matrix densities.

Energy per tile-level SpMV operation

The effect of matrix density on estimated energy consumption is shown in Figure 5.8. For each density point, switching activity recorded in a VCD file during RTL simulation was supplied to Cadence Genus to obtain a workload-specific total power estimate. The estimated energy required to complete one tile-level SpMV operation was then calculated from this power estimate and the measured execution time according to

$$E_{\mathrm{SpMV}} = P_{\mathrm{total}} T_{\mathrm{exec}}. \tag{5.4}$$

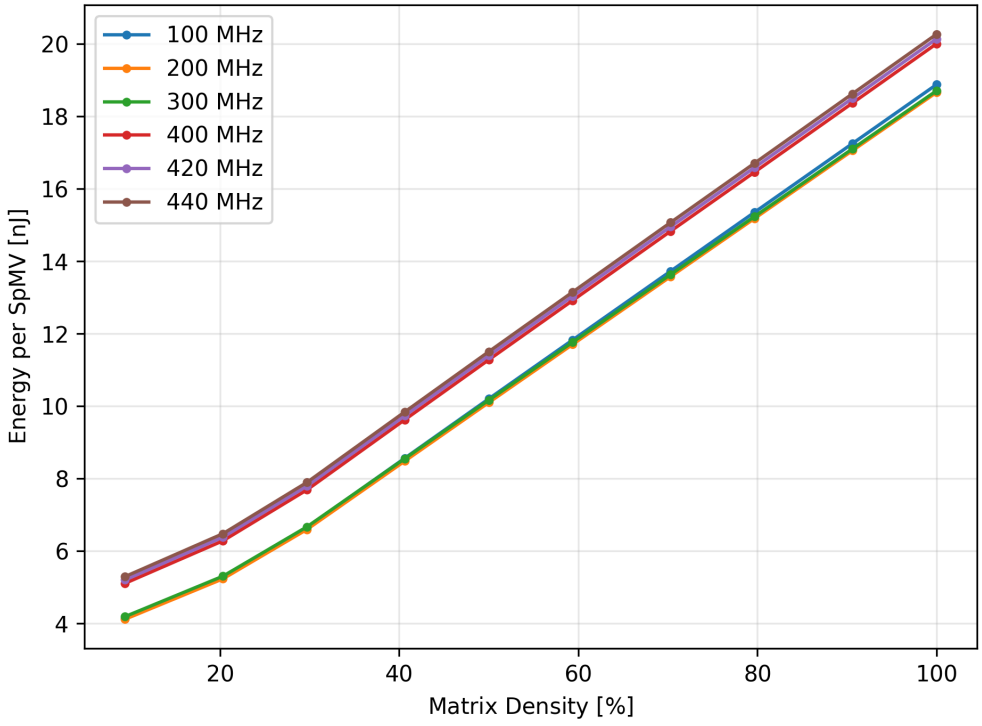


Figure 5.8: Estimated energy per tile-level SpMV operation as a function of matrix density and operating frequency.

The energy required for one tile-level SpMV operation decreases substantially as the matrix becomes more sparse. At 440 MHz, the fully dense matrix requires

approximately 20.27 nJ per operation, whereas the matrix containing six non-zero coefficients requires approximately 5.29 nJ. This corresponds to an energy reduction of approximately 73.9%.

The reduction in energy is primarily a consequence of the shorter execution time. Although the instantaneous power estimate does not decrease proportionally with sparsity, the reduction in the number of execution cycles substantially reduces the total energy required to complete the operation. Fixed pipeline and row-processing overhead prevents the energy from scaling perfectly with the number of non-zero coefficients, consistent with the speedup behaviour observed in Figure 5.7.

The frequency sweep and the sparsity experiment use different power data for different purposes. The synthesis characterization above reports the default power estimate associated with each frequency-constrained implementation, whereas the energy calculation uses the power estimate recorded for each individual density workload. The absolute power values from the two experiments are therefore not compared directly.

Across the sparsity experiment, the 100–300 MHz implementations exhibit similar energy per operation. For the fully dense matrix, the energy remains approximately 18.7–18.9 nJ across these three operating points. A higher energy requirement is observed for the 400–440 MHz implementations, where the fully dense case requires approximately 20.0–20.3 nJ. The same tendency is visible across the evaluated density range.

This result complements the synthesis characterization: increasing the target frequency reduces absolute execution latency, but the more aggressive implementation required at higher frequencies prevents a corresponding reduction in energy per operation. The accelerator therefore exhibits a trade-off between minimum execution time and energy efficiency.

Overall, the sparsity evaluation demonstrates that the proposed accelerator can exploit a reduction in the number of non-zero matrix coefficients to lower both execution latency and energy consumption. For the evaluated 8×8 matrices, reducing the number of non-zero coefficients from 64 to six reduces the execution cycle count from 129 to 21 and provides a measured speedup of approximately 6.14×. At 440 MHz, the same change reduces the estimated energy per operation by approximately 74%. The speedup remains close to the ideal non-zero-count scaling over moderate densities, while fixed processing overhead increasingly limits the benefit in the most sparse cases.

6

Discussion

The evaluation presented in Section 5.2 demonstrates two main characteristics of the proposed accelerator. First, the achievable clock frequency is limited by an increasing implementation cost in area and power, with the row accumulator becoming the timing-limiting block at the highest evaluated frequencies. Second, the execution time of the accelerator depends strongly on matrix sparsity. This section discusses the implications of these results, the limitations of the current evaluation, and possible directions for further development.

6.1 Interpretation of the design evaluation

6.1.1 Cost of Implementation

The synthesis results show that increasing the target frequency does not incur a uniform implementation cost. Between 100 and 300 MHz, the synthesized area changes comparatively little, while a substantially larger increase is observed between 300 and 400 MHz. Power follows a similar trend. This suggests that the synthesis tool can satisfy the lower-frequency constraints without significant restructuring, whereas the more aggressive timing constraints require a considerably more expensive implementation.

Timing closure was demonstrated up to 440 MHz, while the 460 MHz implementation exhibits negative slack and the 500 MHz implementation shows substantially larger timing violations. The exact maximum achievable frequency was not determined, but the synthesis results place it between 440 and 460 MHz. Among the evaluated timing-clean implementations, 440 MHz is therefore the highest demonstrated operating frequency. However, reaching this frequency comes at a significant cost: compared with the 100 MHz implementation, the 440 MHz design requires approximately 1.84 times the synthesized area and 4.39 times the estimated synthesis power.

These results indicate that the highest achievable frequency is not necessarily the most attractive operating point. If minimum latency is the primary objective, operation close to 440 MHz is preferable among the evaluated timing-clean implementations. If area or energy efficiency is more important, a lower operating frequency may provide a better trade-off. The choice of operating point should therefore depend on the requirements of the system in which the accelerator is integrated rather than on clock frequency alone.

An important architectural observation is the change in critical-path location reported by the synthesis tool. At lower target frequencies, the worst path is located in the parallel complex multiplication stage, while from 400 MHz onwards it passes through the floating-point addition logic of the row accumulator. This indicates that the row accumulator, rather than the parallel multiplier array, ultimately limits further frequency scaling in the current architecture.

6.1.2 Effect of matrix sparsity

The sparsity evaluation confirms the main architectural motivation of the design. Reducing the number of non-zero matrix coefficients reduces the number of execution cycles required for an SpMV operation. For the evaluated 8×8 matrices, the cycle count decreases from 129 cycles for the fully dense case to 21 cycles for the matrix containing six non-zero coefficients. Since the cycle count for a given sparsity pattern is unchanged across the evaluated clock frequencies, the resulting relative speedup is an architectural property of the datapath rather than an effect of a particular clock setting.

At moderate matrix densities, the measured speedup is close to the ideal scaling obtained from the reduction in the number of non-zero coefficients. For example, at 50% density the measured speedup is approximately 1.98×, compared with an ideal value of 2.0×. This indicates that, over this region, most of the execution time is associated with useful processing of the stored CSR entries.

The deviation from ideal scaling increases as the matrix becomes very sparse. At 9.375% density, the measured speedup is approximately 6.14×, whereas scaling based only on the ratio of non-zero coefficients would suggest approximately 10.67×. This behaviour is expected from the current control structure. Even when very few arithmetic operations are required, the pipeline still incurs transaction, handshaking, and row-processing overhead. In addition, the row accumulator must interpret the CSR row structure even for rows containing one or no non-zero coefficients. These costs are small in the dense case but become a significant fraction of the total execution time as the useful workload decreases.

The results therefore support the use of the accelerator for workloads in which the processed matrices exhibit varying sparsity. In the context of quantum circuit simulation, this is relevant because different quantum operations can produce matrix structures with different numbers and distributions of non-zero coefficients. The accelerator can reduce the work performed for sparse cases instead of treating every operation as a dense matrix–vector multiplication. However, the present results characterize the SpMV accelerator itself and do not yet establish the end-to-end speedup of a complete quantum-circuit simulation.

6.1.3 Trade-off between Energy and Performance

The workload-specific energy estimates show that sparsity reduces not only execution latency but also the estimated energy required to complete a tile-level SpMV operation. At 440 MHz, reducing the number of non-zero coefficients in the 8×8 tile

from 64 to six decreases the estimated energy per operation by approximately 74%. This reduction is mainly a consequence of the shorter execution time produced by the sparsity-dependent datapath.

Across the 100–300 MHz implementations, the estimated energy per tile-level operation remains relatively similar despite the substantial difference in absolute latency. The higher-frequency implementations complete the operation in fewer nanoseconds, but this benefit is largely offset by their increased estimated power. A more noticeable increase in energy per operation is observed for the 400–440 MHz implementations, consistent with the substantial area and power increase found in the synthesis sweep.

This behaviour illustrates an important distinction between performance and efficiency. Increasing the operating frequency directly reduces the absolute latency of an operation, but does not necessarily reduce the energy required to perform it. In the current design, the 440 MHz implementation provides the lowest latency among the evaluated timing-clean synthesis points, whereas the lower-frequency implementations can offer a more favourable implementation and energy cost. A final operating point should therefore be selected at the system level, where the required throughput, available silicon area, energy budget, and processor interaction can be considered together.

6.2 Limitations of the evaluation

6.2.1 Limited sampling

The sparsity experiment used one pseudorandom matrix pattern for each density. This was sufficient to demonstrate the expected sparsity-dependent behaviour, but it does not characterize the variation between different matrices having the same number of non-zero coefficients. In particular, the current row-by-row accumulation architecture means that two matrices with equal density can distribute their non-zero values differently between rows, which may affect the exact control overhead and execution cycle count.

A more complete statistical evaluation would use multiple random seeds at each density and report the mean execution time together with its variation. This was not performed because the total time required to simulate every density and frequency configuration was substantial. Consequently, the reported results should be treated as representative observations for the evaluated patterns rather than as average performance values for all matrices of the same density.

6.2.2 Synthesis-level characterization

The reported timing, area, and power values are obtained from synthesis and should be interpreted as implementation estimates rather than measurements from fabricated hardware. In particular, the final timing and power of a physical implementation can be affected by placement, routing, clock-tree implementation, and

interconnect parasitics that are not fully represented by the standalone synthesis results.

The power values used in the frequency characterization and the workload-specific sparsity experiment serve different purposes, as described in Section 5.2.2. The former are the default Cadence Genus power estimates for each frequency-constrained implementation. The latter were obtained by supplying density-specific VCD switching activity to Genus and were combined with the measured tile-level execution time to estimate the energy per tile-level SpMV operation. The absolute power values from these two analyses should therefore not be interpreted as directly interchangeable estimates.

A post-layout evaluation would be required to establish a more accurate maximum clock frequency, area, and power for a final ASIC implementation.

6.2.3 Matrix size and application-level performance

The experimental evaluation was performed using one 8×8 matrix tile and the corresponding 8×1 input-vector segment. This configuration is useful for characterizing the accelerator datapath and demonstrating its sparsity-dependent behavior, but it does not by itself demonstrate the performance of a complete quantum state-vector simulation.

In the intended CPU-assisted execution model, a larger unitary matrix would be divided into accelerator-sized tiles. The CPU would supply each tile together with the corresponding state-vector segment, store the returned partial output, and accumulate partial results when multiple tiles contribute to the same output segment. This orchestration and cross-tile accumulation were outside the scope of the present evaluation.

Consequently, the measured tile-level SpMV latency and speedup should not be interpreted as the overall speedup of a quantum simulator. The system-level benefit will also depend on data-transfer costs, memory bandwidth, CPU accumulation overhead, the frequency with which the accelerator can be invoked, and the fraction of the simulator workload that can be mapped to the proposed operation.

6.2.4 X-HEEP coprocessor integration

The accelerator was designed for eventual use as a coprocessor within the X-HEEP RISC-V platform, and an RTL interface module was developed between the processor and accelerator. However, the complete software-controlled workflow—including repeated tile scheduling, data transfers, and CPU accumulation of partial outputs was not implemented or quantitatively evaluated within the scope of this work.

The results presented in this thesis therefore characterize the standalone accelerator datapath. They do not include processor-to-coprocessor communication overhead, custom-instruction overhead, memory-access latency, or possible contention in the memory subsystem. These effects are particularly important for a sparse accelerator because the arithmetic workload can become small at low matrix densities, causing

fixed communication and data-movement costs to represent a larger fraction of the total execution time.

Consequently, the current results demonstrate the computational behaviour and implementation characteristics of the accelerator, but they do not establish the end-to-end acceleration achievable when it is used from software running on X-HEEP.

6.2.5 Absence of a dedicated dense baseline

The sparsity-dependent speedup is measured relative to a 100%-dense matrix processed by the same CSR-based accelerator. A separate dense matrix–vector implementation was not evaluated. Consequently, the results demonstrate how the proposed architecture benefits from reduced matrix density, but do not establish whether it is faster or more energy-efficient than dedicated dense hardware. The density at which sparse processing becomes preferable remains undetermined. The current evaluation also excludes CSR-data preparation costs.

6.3 Future work

6.3.1 Row accumulator optimization

The row accumulator is the clearest target for further architectural optimization. The synthesis results identify it as the timing-limiting block at higher operating frequencies, while the sparsity results show that row-processing overhead also limits the speedup achieved for very sparse matrices.

One possible improvement is to introduce additional pipelining into the accumulation datapath in order to shorten the critical combinational path. Another possibility is to replace part of the sequential accumulation process with a more parallel reduction structure. Such an approach could reduce the number of cycles required for rows containing several non-zero coefficients and could potentially increase the maximum clock frequency.

These changes would, however, introduce additional hardware resources and control complexity. A more parallel reduction structure would also increase the number of floating-point adders, potentially increasing both area and power. The synthesis results already show that aggressive frequency targets have a substantial implementation cost, so any accumulator optimization should be evaluated in terms of latency, area, and energy rather than frequency alone.

The low-density results also suggest that optimization of the control path may be worthwhile. Since fixed row-processing and transaction overhead limits speedup when only a small number of non-zero values remain, mechanisms that reduce the cost of empty or single-entry rows could improve the efficiency of highly sparse operations without necessarily increasing arithmetic parallelism.

6.3.2 System-level X-HEEP evaluation

A natural next step is to complete the integration of the accelerator with X-HEEP and evaluate representative workloads through the processor interface. Such an evaluation should include the cost of issuing the custom operation, transferring or accessing CSR and vector data, waiting for accelerator completion, and returning the result to software.

This would make it possible to distinguish accelerator execution time from complete application-visible latency and to determine the matrix sizes and sparsity levels for which offloading an operation is beneficial. In particular, the fixed system-level overhead may reduce the benefit of offloading very small or extremely sparse operations even though the standalone datapath completes them quickly.

6.3.3 Quantum-simulation evaluation

Finally, the accelerator should be evaluated as part of a representative quantum-circuit simulation rather than only through isolated SpMV operations. Such an experiment would determine how frequently the proposed sparse operation occurs in realistic circuits, how the sparsity changes between operations, and how much of the total simulation time can be accelerated.

An end-to-end evaluation would also allow comparison with a software baseline running on the host processor and would provide a more meaningful measure of application-level speedup and energy efficiency. Together with the X-HEEP integration, this would establish whether the sparsity-dependent improvements observed at the accelerator level translate into a significant reduction in total quantum-simulation execution time.

6.3.4 Comparison with a dense implementation

A future evaluation should compare the proposed accelerator with a dedicated dense complex matrix-vector implementation using the same input matrices and numerical precision. Both designs should be evaluated under comparable synthesis constraints, with their arithmetic resources, area, latency, and energy reported. This comparison would identify the matrix densities for which sparse processing provides a benefit. Where CSR conversion is required, its cost should also be included, accounting for whether the encoded matrix is reused across operations.

7

Conclusion

This thesis project investigated a CSR-based complex sparse matrix-vector multiplication accelerator intended for integration with the X-HEEP RISC-V platform. Its contribution is the implemented architecture and the experimental characterization of its sparsity-dependent execution and ASIC implementation trade-offs. The findings address the three research questions stated in the 1.

Regarding the first research question, the circuit analysis showed that sparsity varies substantially between the selected circuit unitary matrices. For the evaluated six-qubit configurations, QFT and QAOA produced fully dense matrices, while the GHZ and W-state matrices had densities of approximately 3.1% and 11.9%, respectively. These findings show that opportunities to skip zero-valued coefficients depend on the circuit and cannot be assumed for all quantum-simulation workloads. The circuit analysis provided motivation for the accelerator; its performance was evaluated separately using synthetic matrix tiles.

Regarding the second research question, reducing matrix density decreased execution latency and estimated energy within the proposed architecture. For the evaluated 8×8 tiles, reducing the number of non-zero coefficients from 64 to 6 decreased execution time from 129 to 21 clock cycles. This corresponds to a speedup of approximately $6.14\times$ relative to a 100%-dense tile processed by the same CSR-based accelerator. At 440 MHz, estimated energy per operation decreased by approximately 74%. At low densities, control and row-processing overhead became a larger fraction of execution time, limiting the speedup relative to ideal scaling based only on the number of non-zero coefficients.

Regarding the third research question, synthesis results showed that timing closure was achieved at evaluated target frequencies up to 440 MHz. Target frequencies above 300 MHz resulted in substantially higher area and estimated power. Compared with the 100 MHz implementation, the 440 MHz implementation required approximately 1.84 times the area and had 4.39 times the power estimated at synthesis. At higher frequencies, the row accumulator became the timing-limiting block, making it a priority for further architectural optimization.

These results characterize the standalone accelerator. A dedicated dense matrix–vector implementation was not evaluated, so the results do not establish the density at which the proposed sparse hardware becomes preferable to dense hardware. CSR-data preparation and processor–coprocessor communication costs were also excluded. Although an RTL interface module was implemented for X-HEEP integration, the complete processor–coprocessor workflow and end-to-end quantum

simulation performance were not evaluated.

Future work should compare sparse and dense implementations under comparable conditions, including relevant data-preparation costs, to determine when sparse processing is beneficial. It should also evaluate the complete X-HEEP workflow, additional sparsity patterns, and representative quantum-simulation workloads. Row-accumulator optimization and post-layout ASIC evaluation would further characterize the achievable performance and implementation cost.

Bibliography

[1] Qiskit contributors, "Qiskit aer," 2025. [Online]. Available: https://github.com/Qiskit/qiskit-aer

[2] B. Zhang, B. Fang, F. Ye, L. Guo, F. Song, N. Tallent, and D. Tao, "BMQSim: Overcoming memory constraints in quantum circuit simulation with a high-fidelity compression framework," in *Proceedings of the 39th ACM International Conference on Supercomputing (ICS '25)*. New York, NY, USA: Association for Computing Machinery, 2025, pp. 689–704.

[3] Y. Zhao, Y. Guo, Y. Yao, A. Dumi, D. M. Mulvey, S. Upadhyay, Y. Zhang, K. D. Jordan, J. Yang, and X. Tang, "Q-GPU: A recipe of optimizations for quantum circuit simulation using gpus," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022.

[4] S. Jiang, Y.-H. Chung, C.-C. Chang, T.-Y. Ho, and T.-W. Huang, "BQSim: Gpu-accelerated batch quantum circuit simulation using decision diagram," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25)*. Rotterdam, Netherlands: Association for Computing Machinery, 2025.

[5] NVIDIA Corporation, "GPUDirect storage overview guide," https://docs.nvidia.com/gpudirect-storage/overview-guide/index.html, 2024, accessed: 2026-04-10.

[6] N. Quetschlich, L. Burgholzer, and R. Wille, "MQT Bench: Benchmarking software and design automation tools for quantum computing," *Quantum*, vol. 7, p. 1062, 2023.

[7] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, "ASAP7: A 7-nm FinFET predictive process design kit," *Microelectronics Journal*, vol. 53, pp. 105–115, 2016.

[8] P. D. Schiavone, S. Machetti, M. Peón-Quirós, J. Miranda, B. Denkinger, T. C. Müller, R. Rodríguez, S. Nastruzio, and D. Atienza, "X-HEEP: An open-source, configurable and extendible RISC-V microcontroller," in *Proceedings of the 20th ACM International Conference on Computing Frontiers*. ACM, 2023, pp. 379–380.

[9] S. Mach, F. Schuiki, F. Zaruba, and L. Benini, "FPnew: An open-source multiformat floating-point unit architecture for energy-proportional transprecision computing," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 29, no. 4, pp. 774–787, 2021.

[10] OpenHW Group, “Core-V eXtension Interface (CV-X-IF) Specification,” https://docs.openhwgroup.org/projects/openhw-group-core-v-xif/en/latest/, accessed: 2026-09-28.

[11] X-HEEP Project, “Core-V eXtension Interface (CV-X-IF) Configuration,” https://x-heep.readthedocs.io/en/latest/Configuration/XIFConfiguration.html, accessed: 2026-09-28.