Hardware Acceleration of 4×4 Matrix Operations Using a Custom Nios II Peripheral

Nabin Kumar Singh
CPE 523 – Hardware/Software Co-Design

Abstract

This report presents the design, implementation, and evaluation of a hardware accelerator for 4×4 matrix addition, subtraction, and multiplication on the DE1-SoC FPGA platform. The accelerator is implemented as a memory-mapped Avalon-MM peripheral connected to a Nios II soft processor. Using the FPGA interval timer for measurement, the hardware achieves multiple-times speedup over a software-only version while producing matching results.

1 Introduction

In this work, a custom hardware accelerator is designed to perform 4×4 matrix addition, subtraction, and multiplication using 16-bit signed inputs and 32-bit outputs. The accelerator is implemented as an Avalon-MM slave peripheral and integrated with a Nios II soft processor on the DE1-SoC FPGA. The software running on the Nios II processor handles data input, control, and result verification, while the compute-intensive matrix operations are offloaded to dedicated hardware. Unlike the software implementation, which executes nested loops sequentially, the hardware performs multiple arithmetic operations in parallel, significantly reducing execution time. Execution time is measured using the FPGA interval timer, and the performance of the hardware accelerator is compared against a software-only implementation.

1.1 Motivation and Problem Definition

Matrix operations are fundamental building blocks in signal processing, graphics, and scientific computing. Even when matrix dimensions are small and fixed, such as the 4×4 case considered in this project, these operations are often executed repeatedly inside performance-critical loops.

In a software-only implementation on an embedded processor, matrix multiplication, addition, and subtraction are performed sequentially using nested loops. This results in loop-control overhead, instruction fetch and decode latency, and limited instruction-level parallelism. As a consequence, even small fixed-size kernels can incur significant execution time when executed frequently.

This project focuses on accelerating such fixed-size matrix operations by exploiting the inherent parallelism available in FPGA hardware, rather than relying on sequential software execution.

1.1.1 Matrix Multiplication

For a given two 4×4 matrices A and B, matrix multiplication produces an output matrix C defined as:

$$C_{i,j} = \sum_{k=0}^{3} A_{i,k} \times B_{k,j}$$

Let's take an example, considering the matrices A and B as:

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}, \quad B = \begin{bmatrix} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \\ b_{30} & b_{31} & b_{32} & b_{33} \end{bmatrix}$$

Here, each element of the result matrix is computed as a dot product of one row of A and one column of B. For instance, the $C_{0,0}$ is given by:

$$C_{0,0} = a_{00}b_{00} + a_{01}b_{10} + a_{02}b_{20} + a_{03}b_{30}$$

In mathematical form, this is correct. However, on a computer, the computation is carried out as a series of scalar operations. Each of these C output elements requires four distinct multiplications and three additions. Also, these operations are performed iteratively using nested loops. In row-major memory form, the computation for each output element follows a pattern like:

$$C[i][j] = A[i][0] \times B[0][j] + A[i][1] \times B[1][j] + A[i][2] \times B[2][j] + A[i][3] \times B[3][j]$$

This results in a total of 64 multiplications and 48 additions for a single 4×4 matrix multiplication. The time complexity of matrix multiplication is

$$O(n^3)$$

where n is the matrix dimension, even for small fixed sizes, the sequential operations and nested loop method adds overhead, making this operation a good candidate for hardware acceleration. This is what I primarily aim to accelerate in this project.

1.1.2 Matrix Addition and Subtraction

Matrix addition and subtraction can be considered as element-wise operations. For the same given matrices A and B, the result matrices can be defined as:

$$C_{i,j} = A_{i,j} + B_{i,j} \quad \text{and} \quad D_{i,j} = A_{i,j} - B_{i,j}$$

For a 4×4 matrix, each operation consists of 16 independent arithmetic computations. In software, also these are implemented using nested loops that iterate over rows and columns. In a computer memory, the operations are performed as:

$$C[k] = A[k] + B[k], \quad k = 0 \dots 15$$

In general, matrix multiplication has a computational complexity of $O(n^3)$ due to its three nested loops. While the matrix size in this design is fixed at 4×4, the same nested-loop structure still exists in software and introduces control and instruction overhead.
The key performance limitation in software is therefore not the asymptotic complexity, but the strictly sequential execution of multiply-and-accumulate operations. This makes matrix multiplication a strong candidate for hardware acceleration using parallel arithmetic units.

2 System Overview

The system is implemented entirely on the FPGA fabric of the DE1-SoC platform and consists of the following components:

- Nios II soft processor running on the FPGA fabric,
- Custom 4×4 matrix accelerator implemented as an Avalon-MM slave,
- FPGA interval timer at base address `0xFF202000`, used for cycle-accurate timing measurements,
- Avalon-MM interconnect providing memory-mapped communication between the processor, accelerator, and timer.

Both the Nios II processor and the custom accelerator operate in the same 50 MHz FPGA clock domain. The interval timer runs independently and is used solely to measure execution time in clock cycles for software and hardware operations.
Nios II was selected because the entire design (CPU, interconnect, timer, and accelerator) runs on the FPGA fabric, making Avalon-MM integration and memory-mapped testing straightforward within a single Platform Designer system.

3 Hardware Architecture

3.1 Register Map

The accelerator exposes the following word addresses:

- 0–15: Matrix A input (16 elements, lower 16 bits used)
- 16–31: Matrix B input (16 elements, lower 16 bits used)
- 32–47: SUM output (32-bit each)

- 48–63: DIFF output (32-bit each)
- 64–79: PROD output (32-bit each)
- 80: CONTROL (bit 0 = START)
- 81: STATUS (bit 0 = DONE, bit 1 = BUSY)

3.2 Parallel Computation

The hardware computes SUM and DIFF for all 16 elements in one step, and performs matrix multiplication using 16 parallel multiply-accumulate paths. The inner index k runs from 0 to 3; each clock cycle accumulates one term for all 16 outputs.

3.3 FSM Control

A small FSM controls the operation:

1. **LOAD**: wait for START, compute SUM/DIFF, clear accumulators,
2. **RUN**: accumulate products for $k = 0, 1, 2, 3$,
3. **DONE**: assert DONE and hold results until next start.

3.4 Flat Matrix Representation and Index Mapping

Although matrix operations are naturally expressed in two dimensions, both the software and the hardware accelerator store matrices as flat one-dimensional arrays of length 16. The elements are stored in row-major order, meaning that each row of the matrix is laid out contiguously in memory.
For a 4×4 matrix $\mathbf{A}$, the mapping from two-dimensional indices (i, k) to a flat array index is:

$$A_{ik} \to A[i \cdot 4 + k],$$

where i denotes the row index and k denotes the column index. Similarly, for matrix $\mathbf{B}$:

$$B_{kj} \to B[k \cdot 4 + j],$$

and for the result matrix $\mathbf{C}$:

$$C_{ij} \to C[i \cdot 4 + j].$$

3.5 Partial Product Formation in Hardware

Matrix multiplication is performed using the standard formulation:

$$C_{ij} = \sum_{k=0}^{3} A_{ik} \cdot B_{kj}.$$

In the hardware accelerator, the inner index k is iterated over time, while all output elements are computed in parallel. During each clock cycle, a single value of k is selected, and all partial products corresponding to that k are computed simultaneously.
Specifically, for a fixed value of k, the hardware forms partial products using:

$$A[i \cdot 4 + k] \cdot B[k \cdot 4 + j],$$

for all combinations of i and j in the range 0 to 3. These products are accumulated into 64-bit internal registers corresponding to each output element C_{ij}.
After four clock cycles (for $k = 0, 1, 2, 3$), all partial products have been accumulated, and the final 32-bit results are written to the output product registers.

3.5.1 Illustrative Hardware Implementation

The following SystemVerilog code fragment illustrates how the matrix multiplication is realized in hardware using a flattened matrix representation and parallel multiply–accumulate logic. For a fixed value of the inner index k, all sixteen output elements are updated simultaneously.

```
if (k != 3'd4) begin
    // Row 0
    prod_accum[0] <= prod_accum[0] + (A[k] * B[(k*4) + 0]);
    prod_accum[1] <= prod_accum[1] + (A[k] * B[(k*4) + 1]);
    prod_accum[2] <= prod_accum[2] + (A[k] * B[(k*4) + 2]);
    prod_accum[3] <= prod_accum[3] + (A[k] * B[(k*4) + 3]);

    // Row 1
    prod_accum[4] <= prod_accum[4] + (A[4 + k] * B[(k*4) + 0]);
    prod_accum[5] <= prod_accum[5] + (A[4 + k] * B[(k*4) + 1]);
    prod_accum[6] <= prod_accum[6] + (A[4 + k] * B[(k*4) + 2]);
    prod_accum[7] <= prod_accum[7] + (A[4 + k] * B[(k*4) + 3]);

    // Row 2
    prod_accum[8] <= prod_accum[8] + (A[8 + k] * B[(k*4) + 0]);
    prod_accum[9] <= prod_accum[9] + (A[8 + k] * B[(k*4) + 1]);
    prod_accum[10] <= prod_accum[10] + (A[8 + k] * B[(k*4) + 2]);
    prod_accum[11] <= prod_accum[11] + (A[8 + k] * B[(k*4) + 3]);

    // Row 3
    prod_accum[12] <= prod_accum[12] + (A[12 + k] * B[(k*4) + 0]);
    prod_accum[13] <= prod_accum[13] + (A[12 + k] * B[(k*4) + 1]);
    prod_accum[14] <= prod_accum[14] + (A[12 + k] * B[(k*4) + 2]);
    prod_accum[15] <= prod_accum[15] + (A[12 + k] * B[(k*4) + 3]);

    k <= k + 3'd1; // Advance inner index each clock cycle
end
```

Listing 1: Parallel partial product accumulation for matrix multiplication

This implementation directly maps the mathematical formulation $C_{ij} = \sum_{k=0}^{3} A_{ik} \cdot B_{kj}$ into hardware. For each clock cycle, one value of k is selected, and all sixteen partial products corresponding to that k are computed in parallel. After four cycles, the complete matrix product is available. This contrasts with the software implementation, which evaluates each output element sequentially using nested loops.

3.6 Input Range and Overflow Safety Analysis

The matrix multiplication accelerator computes each output element as the sum of four products:

$$C_{ij} = \sum_{k=0}^{3} A_{ik} \cdot B_{kj}.$$

Each input element is a signed 16-bit value, and each product $A_{ik} \cdot B_{kj}$ therefore fits within 32 bits. However, because four such products are accumulated to form a single output element, care must be taken to avoid overflow in the 32-bit result register.

3.7 Worst-Case Analysis

Let M denote the maximum magnitude of any input element:

$$|A_{ik}| \leq M, \quad |B_{kj}| \leq M.$$

In the worst case, all four products have the same sign and maximum magnitude:

$$|C_{ij}| \leq 4M^2.$$

To guarantee that the result fits within a signed 32-bit integer, the following condition must be satisfied:

$$4M^2 \leq 2^{31} - 1.$$

Solving for M:

$$M \leq \sqrt{\frac{2^{31} - 1}{4}} \approx 23170.$$

Therefore, the maximum safe input range that guarantees no overflow in the matrix multiplication result is:

$$\boxed{-23170 \leq A_{ik}, B_{kj} \leq 23170.}$$

3.8 Discussion

Inputs within this range ensure mathematically correct results for all elements of the product matrix. Although the hardware accepts the full signed 16-bit range, larger input values may cause arithmetic overflow in the 32-bit output registers during accumulation. The element-wise addition and subtraction operations do not pose the same risk, since they require significantly fewer bits than matrix multiplication.

4 Software Design

The software executes on the Nios II processor and serves three roles: (1) it implements a software-only reference version of the matrix operations, (2) it controls the hardware accelerator through memory-mapped registers, and (3) it measures execution time using the FPGA interval timer. Both the software and hardware runs are timed using the same measurement method to enable a fair speedup comparison.

4.1 Data Representation (Flat 1D Layout)

Although matrices are entered as 4×4, both software and hardware treat them as a flat array of 16 elements in row-major order. The element at row i and column j is mapped to index $(i \cdot 4 + j)$. This indexing is used in the software reference implementation and also matches how the accelerator stores values in its Avalon-MM register space.

4.2 Software-Only Reference Implementation (Sequential)

The software reference version computes SUM and DIFF using a single loop over 16 elements, and computes the product using three nested loops (i, j, k). This is inherently sequential: one multiply and add is performed at a time, and the final product matrix is obtained after all loop iterations complete.

```
for (i = 0; i < 4; i++) {
  for (j = 0; j < 4; j++) {
    int32_t sum = 0;
    for (k = 0; k < 4; k++) {
      sum += (int32_t)A[i*4 + k] * (int32_t)B[k*4 + j];
    }
    SW_Prod[i*4 + j] = sum;
  }
}
```

Listing 2: Software-only matrix multiply (sequential triple loop).

In contrast to hardware, the CPU must also spend cycles on loop control, address calculations, and repeated load/store operations, which contributes to the higher cycle count.

4.3 Hardware Accelerator Invocation (Avalon-MM Access)

The accelerator is accessed as a memory-mapped Avalon-MM peripheral. Software writes the 16 elements of matrix A and B into the accelerator's input register bank, asserts START in the CONTROL register, polls STATUS until DONE is set, and finally reads back SUM, DIFF, and PROD arrays.

```
accel_base[CONTROL_OFFSET] = 1;
while ((accel_base[STATUS_OFFSET] & 0x1) == 0) { /* wait */ }

for (i = 0; i < 16; i++) HW_Sum[i] = (int32_t)accel_base[SUM_OFFSET + i];
```

```
for (i = 0; i < 16; i++) HW_Diff[i] = (int32_t)accel_base[DIFF_OFFSET + i];
for (i = 0; i < 16; i++) HW_Prod[i] = (int32_t)accel_base[PROD_OFFSET + i];
```

Listing 3: Hardware invocation: START + poll DONE + read results.

The polling approach was selected for simplicity and deterministic behavior. Results are stored into software arrays through pointer-based memory-mapped access (i.e., reads from `accel_base[...]`), which matches the peripheral register map defined in hardware.

4.4 Timing Measurement Methodology (FPGA Interval Timer)

Execution time is measured using the FPGA interval timer at base address `0xFF202000`. The timer is configured as a free-running down counter, loaded with its maximum value, and started immediately before the computation. After the computation completes, the timer is stopped and a snapshot of the counter value is taken to determine the elapsed time. The elapsed time is computed as the difference between the initial count and the snapshotted value. The same measurement procedure is used for both the software-only implementation and the hardware-accelerated implementation to ensure a fair comparison.

```
*(timer + 2) = 0xFFFF; // load period low
*(timer + 3) = 0xFFFF; // load period high
*(timer + 1) = 0x4; // start timer

// Execute software or hardware function

*(timer + 1) = 0x8; // stop timer
*(timer + 4) = 1; // snapshot counter
last_count = (*(timer + 5) << 16) | *(timer + 4);
cycles = 0xFFFFFFFF - last_count;
```

Listing 4: Timer-based measurement used for both software and hardware executions

On the DE1-SoC platform used in this project, the interval timer is clocked at **100 MHz**, corresponding to a resolution of 10 ns per timer tick. This timer clock is independent of the Nios II processor clock (50 MHz). Since both software and hardware executions are measured using the same timer source, the reported speedup values remain accurate and meaningful.

4.5 Summary of Software vs. Hardware Behavior

The software implementation evaluates matrix operations in a strictly sequential manner, executing one multiply–add operation at a time using nested loops. As a result, a significant portion of execution time is spent on loop control, instruction fetch/decode, and memory accesses.

In contrast, the hardware accelerator exploits parallelism within the FPGA by computing multiple arithmetic operations simultaneously. Once the START signal is issued, the accelerator performs matrix addition, subtraction, and multiplication independently of the Nios II processor. The CPU is only involved in initiating the computation, polling the STATUS register, and reading back the final results, leading to substantially reduced execution time.

5 Performance and Speedup

Speedup is achieved mainly due to hardware parallelism:

- Software performs multiply-add operations sequentially with loop overhead.
- Hardware computes 16 products in parallel per cycle and accumulates results.

Execution time is measured using the FPGA interval timer. Since both software and hardware are measured using the same timer source, the speedup ratio is valid.

5.1 Functional Verification

To verify correctness, the hardware accelerator outputs were compared against software reference results for multiple test cases. All tests showed exact agreement between software and hardware results for SUM, DIFF, and PROD.

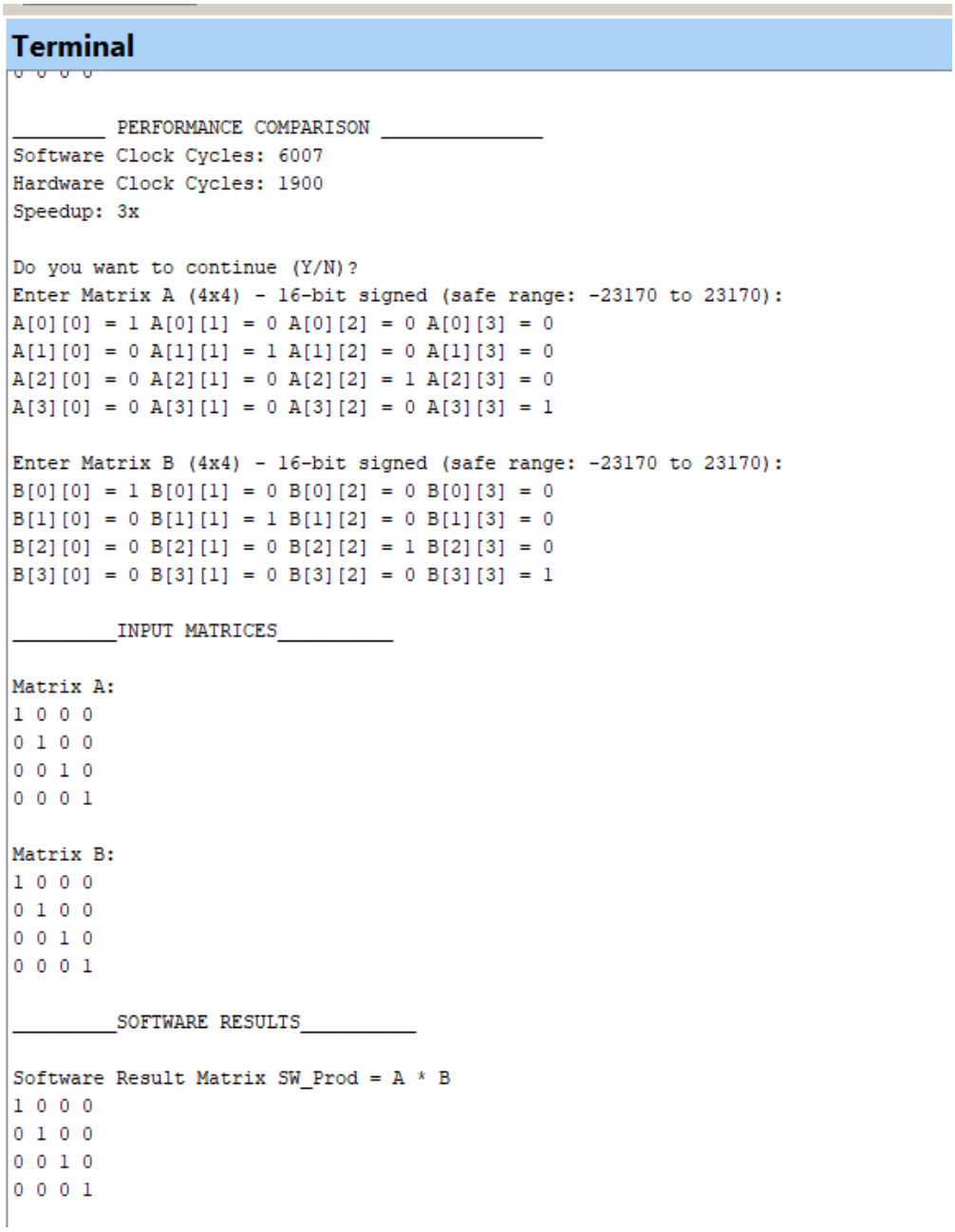

Figure 1: General Terminal View of the INTEL FPGA Monitor, for the identity matrix.

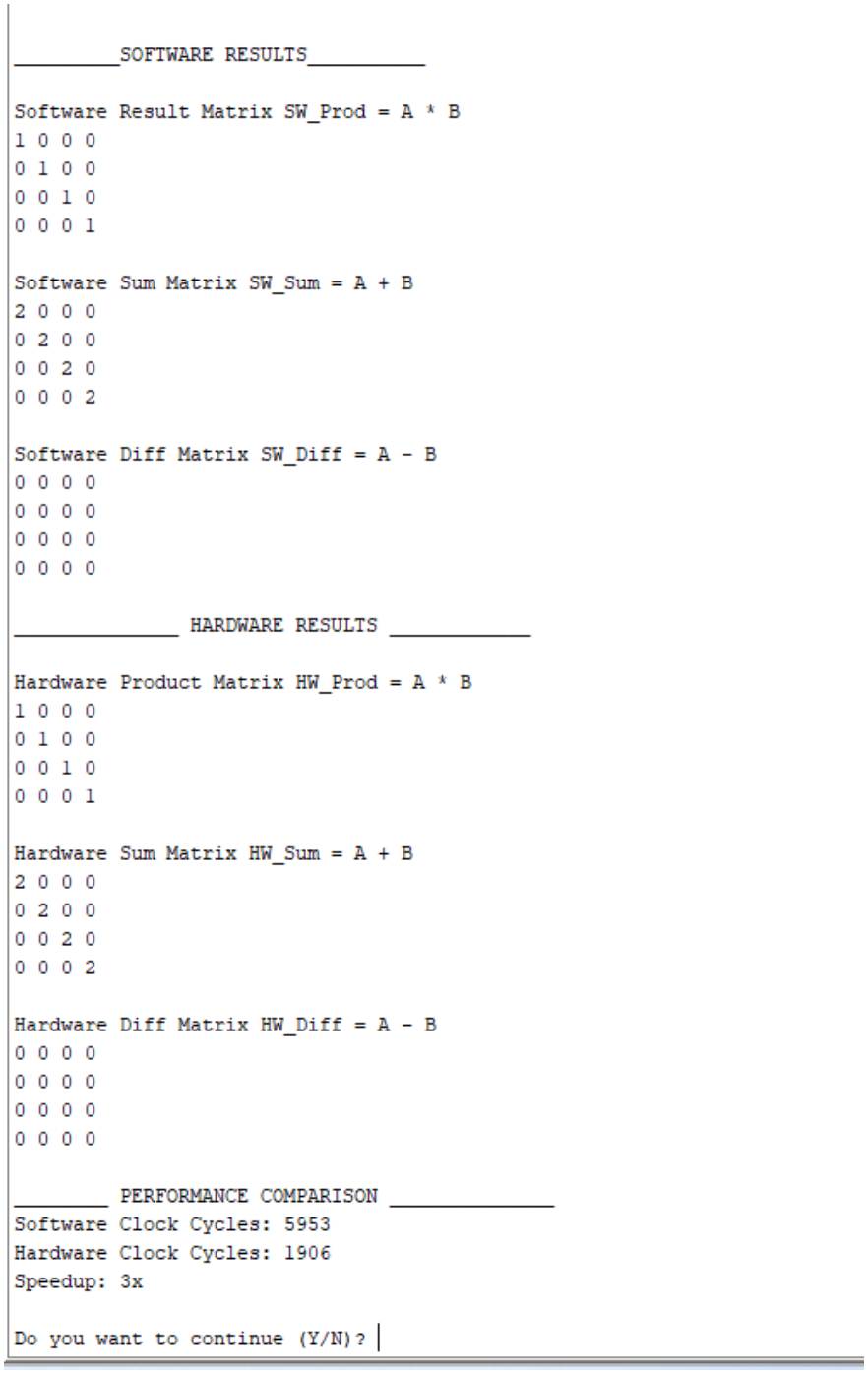

Figure 2: Identity matrix test: A $\times$ I = A. This verifies correct flattened index mapping and row/column access inside the accelerator.

```
_________INPUT MATRICES__________

Matrix A:
2300 2300 2300 2300
2300 2300 2300 2300
2300 2300 2300 2300
2300 2300 2300 2300

Matrix B:
2300 2300 2300 2300
2300 2300 2300 2300
2300 2300 2300 2300
2300 2300 2300 2300

_________SOFTWARE RESULTS__________

Software Result Matrix SW_Prod = A * B
21160000 21160000 21160000 21160000
21160000 21160000 21160000 21160000
21160000 21160000 21160000 21160000
21160000 21160000 21160000 21160000

_____________ HARDWARE RESULTS ____________

Hardware Product Matrix HW_Prod = A * B
21160000 21160000 21160000 21160000
21160000 21160000 21160000 21160000
21160000 21160000 21160000 21160000
21160000 21160000 21160000 21160000

________ PERFORMANCE COMPARISON _____________
Software Clock Cycles: 11436
Hardware Clock Cycles: 1889
Speedup: 6x
```

Figure 3: Reduced Terminal View: Sum and Diff removed to capture the major section; Uniform large-value test near the safe input range. Software and hardware results match exactly for all output elements.

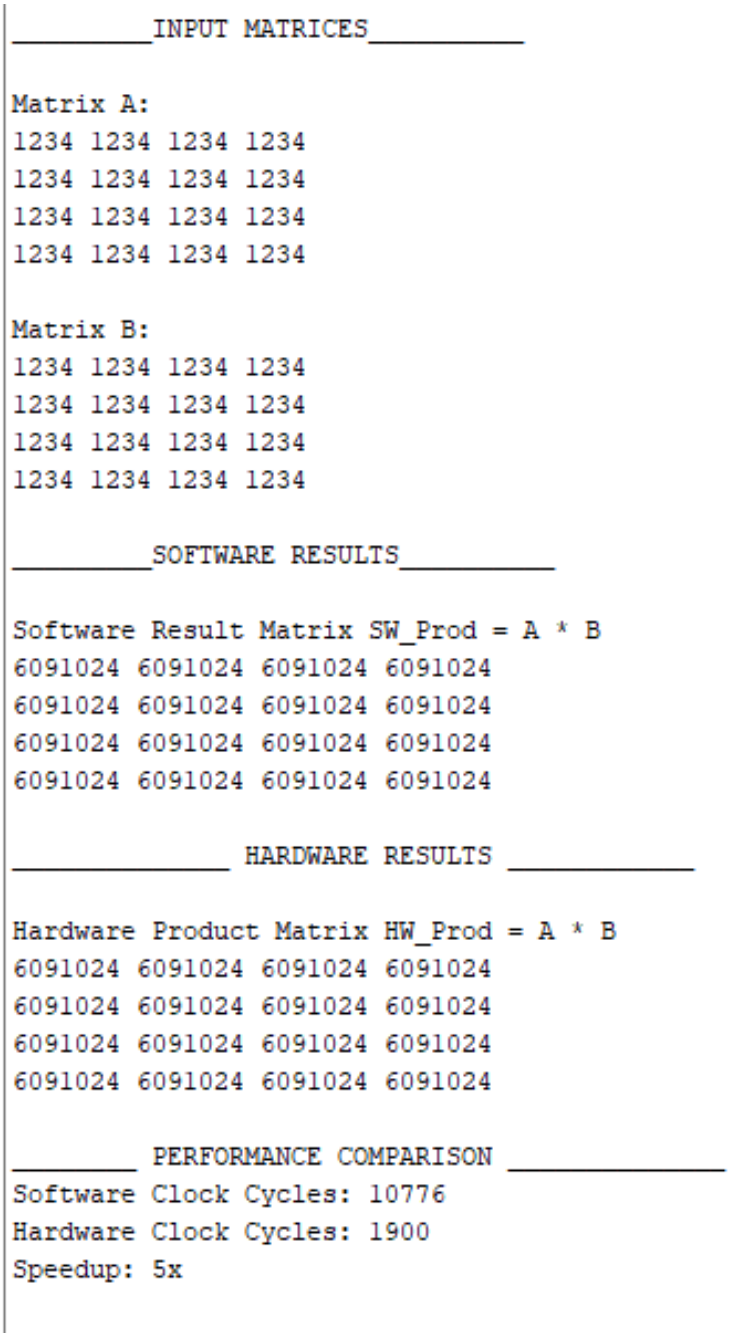

Figure 4: Uniform random-value test. Software and hardware results match exactly for all output elements.

```
_________INPUT MATRICES__________

Matrix A:
1234 2345 5234 542
5432 -5423 -5432 -5422
-2345 -5422 -2345 -542
-5422 -2345 -542 -2345

Matrix B:
-542 -542 -454 -2345
-542 -542 -542 -542
3245 2345 -2345 -542
-2345 -542 542 -222

_________SOFTWARE RESULTS__________

Software Result Matrix SW_Prod = A * B
13773522 10040148 -13811192 -7121872
-4917128 -9804194 10272454 -5650946
-2128821 -995547 9208615 9829063
7949949 4209714 3732578 14799934

_____________ HARDWARE RESULTS _____________

Hardware Product Matrix HW_Prod = A * B
13773522 10040148 -13811192 -7121872
-4917128 -9804194 10272454 -5650946
-2128821 -995547 9208615 9829063
7949949 4209714 3732578 14799934

________ PERFORMANCE COMPARISON _____________
Software Clock Cycles: 18274
Hardware Clock Cycles: 1891
Speedup: 9x
```

Figure 5: Fully random signed input matrices. This test demonstrates correct operation under realistic and irregular workloads.

5.2 Performance Results (Cycle Counts)

Table 1 summarizes the measured cycle counts for the software-only implementation versus the hardware-accelerated implementation across multiple test cases.

Table 1: Measured software vs. hardware cycle counts (FPGA interval timer).

Test case	SW cycles	HW cycles	Speedup
All zeros	5817	1905	3×
Identity matrices	6004	1884	3×
All 2300s	11436	1889	6×
All 1234s	10776	1900	5×
Random values	18274	1891	9×

5.3 Performance Summary

Across all test cases, the hardware accelerator consistently outperformed the software-only implementation. Observed speedups ranged from approximately 3× for trivial cases (zero and identity matrices) to up to 9× for random input matrices. This variation is expected, as the software execution time depends on loop behavior and data values, while the hardware execution time remains nearly constant due to the parallel computation of multiple multiply-accumulate operations.

6 Resource Utilization

6.1 Overall System Utilization (DE1-SoC Project)

Quartus reports the following overall device usage for the complete system (Nios II subsystem + interconnect + interval timer + matrix accelerator):

- Logic utilization: 24,561 / 32,070 ALMs (77%)
- Total registers: 32,521
- DSP blocks: 31 / 87 (36%)
- Block memory bits: 2,391,706 / 4,065,280 (59%)

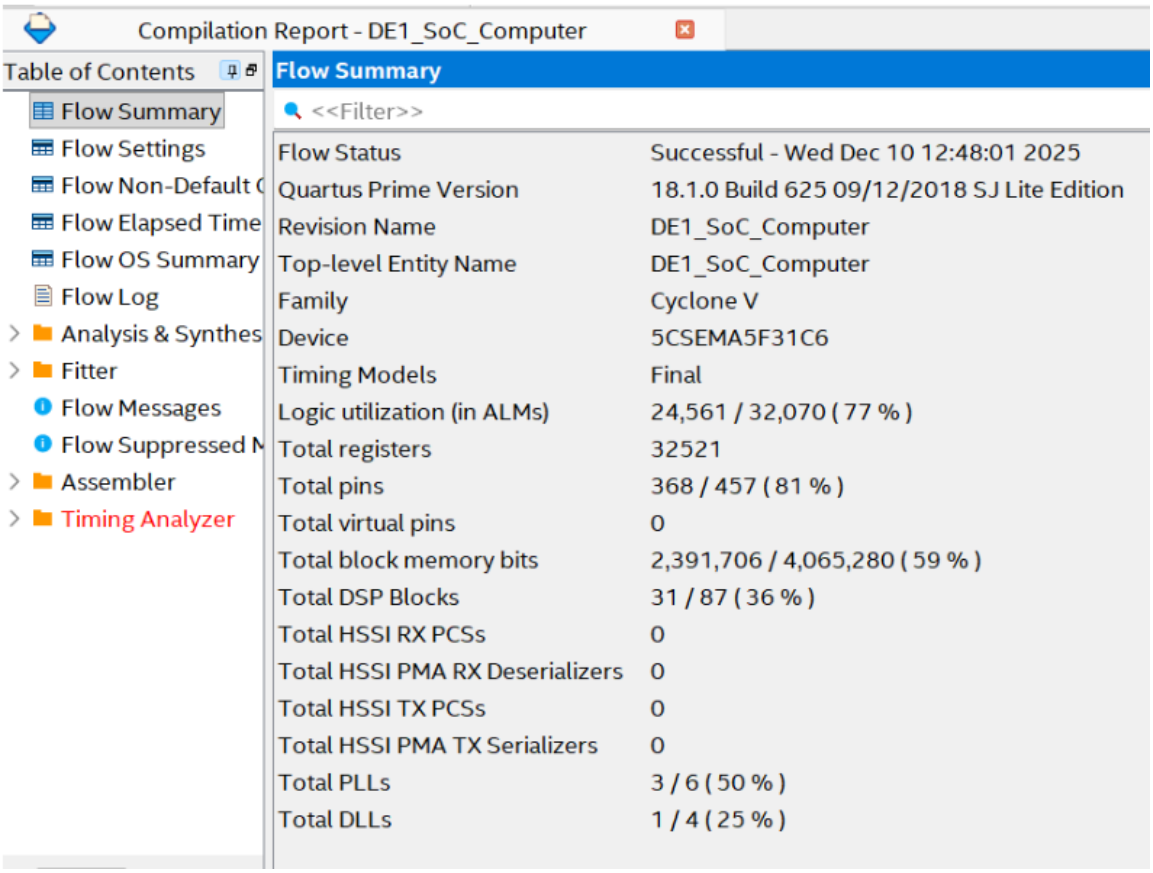

Figure 6: Quartus flow summary showing overall resource usage for the full DE1-SoC system.

6.2 Accelerator-Only Utilization (Entity Breakdown)

Figure 7 shows the fitter breakdown by hierarchy. The custom accelerator entity (`mat_mul_sub_add_all_par`) uses approximately:

- ALMs (final placement): $\approx$ 1,360
- Combinational ALUTs: $\approx$ 1,652
- Dedicated logic registers: $\approx$ 2,429
- DSP blocks: 16
- Block memory bits: 0 (no RAM used inside the accelerator)

This shows that most of the overall ALM/register/memory usage comes from the full Platform Designer system (Nios II CPU subsystem, interconnect, and memory), while the matrix accelerator itself mainly consumes DSP blocks and a modest amount of logic.

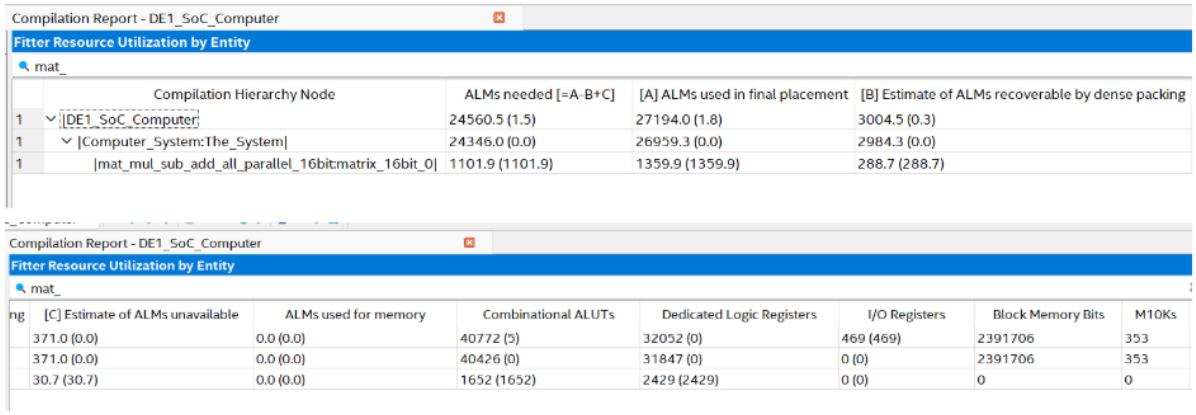

Figure 7: Fitter resource utilization by entity, highlighting the accelerator versus the full system.

7 Conclusion

A custom FPGA accelerator for 4×4 matrix addition, subtraction, and multiplication was designed and integrated as an Avalon-MM peripheral on the DE1-SoC platform. The accelerator uses parallel hardware inside the FPGA to speed up matrix operations compared to a software-only implementation running on the Nios II processor.

Using the FPGA interval timer, the hardware implementation showed speedups ranging from about 3× for simple test cases (such as zero and identity matrices) to up to 9× for random input matrices. This improvement occurs because the hardware performs many arithmetic operations in parallel, while the software executes them sequentially using nested loops.

Although the design targets a fixed 4×4 matrix size, the same approach can be extended to larger matrices or further improved using pipelining. Overall, this project demonstrates how hardware/software co-design can effectively accelerate computation-intensive tasks on FPGA-based systems.