Journal of Electrical and Electronics Engineering (IOSR-JEEE)
e-ISSN: 2278-1676, p-ISSN: 2320-3331, Volume 21, Issue 3 Ser. 1 (May. – June. 2026), PP 14-22
www.iosrjournals.org

FPGA Based Matrix Multiplication Accelerator

Sayan Kumar Singha

Abstract:
In Big data and Deep learning structure involve complex computation including sequential matrix multiplication which may reduce overall system clock period. Whereas FPGA offers the parallel processing and hardware acceleration. In this project, implement the matrix multiplication process of the CPU part on the PYNQ-Z2 FPGA development board to design a matrix multiplication accelerator. A custom matrix multiplication IP core was developed using Vitis HLS and integrated with the PYNQ framework for the design. By the offloading computation and parallel process nature of the FPGA, the hardware-software system achieves improved performance in terms of timing analysis compared with sequential processing on CPU framework.

Keywords: *Accelerator, GEMM, Inner product, Outer product, Row-wise product, FPGA, HLS, IP*

Date of Submission: 06-05-2026 Date of Acceptance: 16-05-2026

I. Introduction:

Matrix multiplication is a fundamental operation in deep learning and scientific computing. General Matrix-Matrix Multiplication (GEMM) forms the backbone of fully connected and convolutional layers. The GEMM operation is mathematically defined as:

$C=\alpha A\times B+\beta C$

where A, B, and C are matrices, and α, β are scalar coefficients. This operation performs a scaled multiplication of two matrices A and B, followed by a scaled addition of a third matrix C [1]. FPGAs provide low-latency, parallel execution, making them ideal for matrix operations in embedded systems. This project uses the PYNQ-Z2 board which has a dual-core ARM Cortex-A9 processor coupled with programmable logic fabric to provide a heterogeneous computing platform with a Zynq-7020 SoC [2] to accelerate 128x128 matrix multiplication using a custom IP core developed in Vitis HLS.

Challenges Faced:

1. **AXI Stream Debugging:** Ensuring correct TVALID, TREADY, and TLAST signaling was vital to prevent data loss or stalls during matrix streaming.
2. **Control-Data Synchronization:** The IP had to wait until both input matrices were fully received before starting computation. Handshaking and status flags resolved timing mismatches**.**
3. **Memory Constraints on PYNQ-Z2:** Limited BRAM required offloading large matrices to DDR via DMA, introducing latency. Efficient tiling, buffer management, and memory access optimization were key to handling large datasets**.**

Background:

Matrix multiplication is one of the most critical computational kernels in scientific computing, signal processing, and deep learning. At the core of many machine learning algorithms, especially deep neural networks, is the need to process and transform large data sets represented as matrices. A standard and highly optimized formulation for such operations is the **General Matrix-Matrix Multiplication (GEMM)**.

Matrix Multiplication in Deep Learning:

In CNNs, convolution can be transformed into matrix multiplication using **im2col** [3], enabling acceleration via GEMM. Efficient GEMM is crucial due to growing demands in filter size, input volume, and precision.

Three Key Matrix Multiplication Views for Hardware Optimization:

- **Inner Product**: Computes each output element as a dot product of row (A) and column (B) as shown in figure-1A [4]. This is efficient for dense ops, MAC-friendly, highly parallelizable [5]. **Mathematical expression**: For matrices $A \in m\times k$ and $B \in k\times n$, the element Cij of the output matrix $C \in m\times n$ is calculated as:

$$Cij = \sum_{r=1}^{k} Air \cdot Brj$$

- **Outer Product**: Forms output by summing rank-1 matrices (column of A × row of B) as shown is figure 1B [4]. This

maximizes data reuse, ideal for systolic arrays and tiling [6].

Mathematical expression:

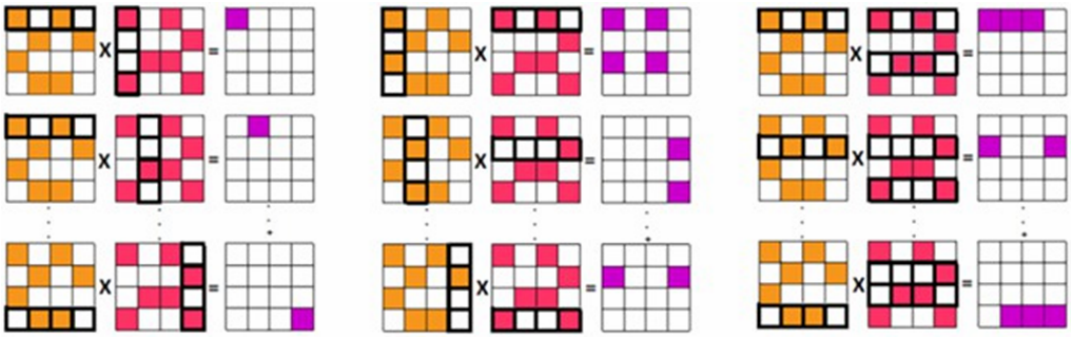
$$C = \sum_{r=1}^{k} A_{:,r} \cdot Br_{,:}$$

Here, $A_{:,r}$ is the rth column of matrix A, and $B_{r,:}$ is the rth row of matrix B. Each outer product produces a full matrix, and their sum forms the final result.

- **Row-wise Product**: Generates one output row at a time (row of A × matrix B) as shown in figure 1C [4]. This is good for streaming/pipelined designs with row buffering [7].

Mathematical expression:

$$Ci_{,:} = Ai_{,:} \cdot B$$



A. Inner product B. Outer product C. Row-wise product

Fig-1: Matrix multiplication visualization

FPGA Acceleration:

An **FPGA (Field Programmable Gate Array)** is a reconfigurable chip enabling **true parallel execution** by creating custom hardware circuits. This flexibility makes FPGAs ideal for compute-heavy tasks like **matrix multiplication**, where many multiply-accumulate operations can run simultaneously [8][9].

Key FPGA Components:

- **Logic Blocks (CLBs):** Contain LUTs, flip-flops, and multiplexers for creating digital logic [10].
- **I/O Blocks (IOBs):** Interface with memory, sensors, and communication protocols [10].
- **Programmable Interconnect:** Routes signals between blocks, crucial for performance [10].

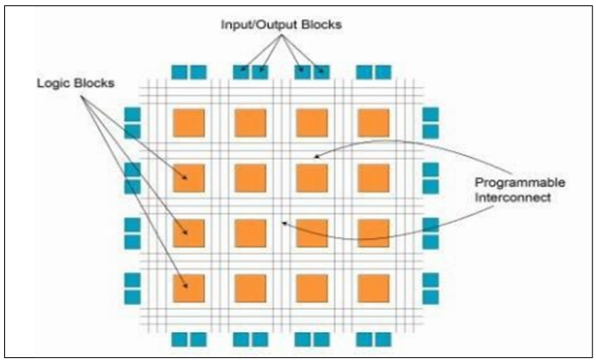


Fig-2: FPGA Architecture

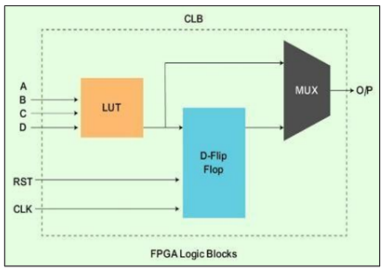


Fig-3: CLB

Using tools like **Vitis HLS**, developers can convert C/C++ code into hardware. On platforms like **PYNQ-Z2 (Zynq-7020)**, the ARM processor handles control while FPGA logic accelerates computations. Optimizations like **loop pipelining, tiling**, and **double-buffering** enhance performance by efficiently using BRAM and DSPs.

Proposed Design:

The matrix multiplication system was implemented on the PYNQ-Z2 FPGA development board using the Xilinx Vivado Design Suite (2024.2) and the PYNQ Python framework. The proposed design flow diagram is shown in the figure

4:

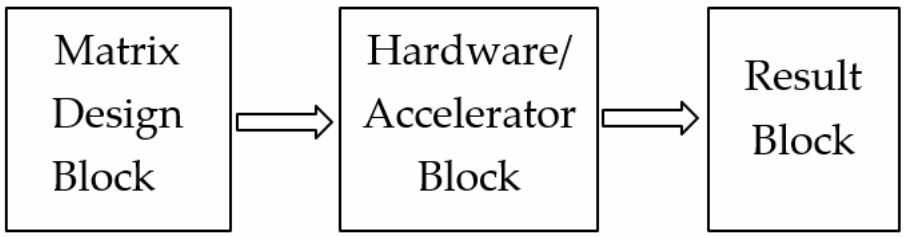


Fig-4: Flow diagram of the proposed design

Matrix Design Block: Designs two 128 x 128 matrices using numpy in software.
Hardware/Accelerator Block: Multiply the two matrices by channelling the matrices to the multiplication IP. Direct Memory Access (AXI- DMA) has to be used to send and receive data from software to hardware and then hardware to software.
Result Block: Result channels from hardware to software and the times of multiplication of the matrices in software and hardware is been compared.

Now the complete detailed steps of the above block diagram are as followed:
High-Level Synthesis: Matrix Multiplication IP in C++ [11]

IP or **intellectual property** cores or IP blocks, which are reusable functional blocks of code are designed in a high-level language. The matrix multiplication logic was implemented in C++ using Vitis HLS. The function reads two matrices A and B, performs multiplication, and outputs the result C. The code below demonstrates the top-level function used for synthesis:

Header file(matmult.h):

```
#ifndef _MMULT_ #define _MMULT_

#include"ap_axi_sdata.h" #include "ap_int.h" #include <inttypes.h>

#define N 128
#define N2 16384 // N*N
#define DWIDTH 512
typedef ap_axiu<DWIDTH, 0, 0, 0> axis_t;

typedef ap_uint<512> uint512_t; typedef float DataType;
const int DataTypeSize = sizeof(DataType) * 8; typedef ap_uint<DataTypeSize> DataTypeInt; typedef union converter {
DataTyped; uint32_t i;
} converter_t;
template <typename T> void kernel_mmult(T a[N2], T b[N2], T c[N2]); #endif
```

Source code file(matmult_accel.cpp):

```
#include "matmult.h" #include"hls_stream.h"

template <typename T> void kernel_mmult(T a[N2], T b[N2], T out[N2]) { L1:
for (int m = 0; m < N; ++m) { L2:
for (int n = 0; n < N; ++n) { #pragma HLS PIPELINE II = 1
T sum = 0; L3:
for (int k = 0; k < N; ++k)
sum += a[m * N + k] * b[k * N + n]; out[m * N + n] = sum;
}
}
return;
}
extern "C" {
void matmult_accel(hls::stream<axis_t> &in, hls::stream<axis_t> &out) { #pragma HLS INTERFACE s_axilite port =
return bundle = control
#pragma HLS INTERFACE axis port = in #pragma HLS INTERFACE axis port = out
DataTypel_A[N2]; DataTypel_B[N2]; DataTypel_C[N2];
#pragma HLS ARRAY_PARTITION variable = l_A factor = 16 dim = 1 cyclic #pragma HLS ARRAY_PARTITION
```

```
variable = l_B factor = 16 dim = 1 block #pragma HLS ARRAY_PARTITION variable = l_C factor = 16 dim = 1 cyclic

int j_limit = 512 / DataTypeSize; int i_limit = N2 / j_limit; converter_t converter;
load_A:
for (int i = 0; i < i_limit; i++) { axis_t temp = in.read();
for (int j = 0; j < j_limit; j++) {
int high = j * DataTypeSize + DataTypeSize - 1; int low = j * DataTypeSize;
int index = i * 16 + j;

converter.i = temp.data.range(high, low); l_A[index] = converter.d;
}
}

load_B:
for (int i = 0; i < i_limit; i++) { axis_t temp = in.read();
for (int j = 0; j < j_limit; j++) {
int high = j * DataTypeSize + DataTypeSize - 1; int low = j * DataTypeSize;
int index = i * 16 + j;
converter.i = temp.data.range(high, low); l_B[index] = converter.d;
}
}
kernel_mmult<DataType>(l_A, l_B, l_C);

writeC:
for (int i = 0; i < i_limit; i++) { axis_t temp;
for (int j = 0; j < j_limit; j++) {
int high = j * DataTypeSize + DataTypeSize - 1; int low = j * DataTypeSize;
converter.d = l_C[i * 16 + j];
temp.data.range(high, low) = converter.i;
}
ap_uint<1> last = 0;
if (i == i_limit - 1) { last = 1;
}
temp.last = last;
temp.keep = -1; // enabling all bytes out.write(temp);
}
}
}
```

Testbench file(matmul_accel_tb.cpp):

```
#include "matmult.h"

void mmult_sw(DataType a[N2], DataType b[N2], DataType out[N2])
{
for (int ia = 0; ia < N; ++ia) for (int ib = 0; ib < N; ++ib)
{
float sum = 0;
for (int id = 0; id < N; ++id)
sum += a[ia * N + id] * b[id * N + ib];
out[ia * N + ib] = sum;
}
}
int main(void)
{

int ret_val = 0; int i, j, err;
DataType matOp1[N2]; DataType matOp2[N2]; DataTypematMult_sw[N2]; DataType matMult_hw[N2];
/** Matrix Initiation */ for (i = 0; i < N; i++)
for (j = 0; j < N; j++)
```

```
matOp1[i * N + j] = (DataType)(i + j);

for (i = 0; i < N; i++) for (j = 0; j < N; j++)
matOp2[i * N + j] = (DataType)(i * j);
/** End of Initiation */ kernel_mmult<DataType>(matOp1, matOp2, matMult_hw);
/* reference Matrix Multiplication */ mmult_sw(matOp1, matOp2, matMult_sw);
/** Matrix comparison */ err = 0;
for (i = 0; (i < N && !err); i++) for (j = 0; (j < N && !err); j++)
if (matMult_sw[i * N + j] != matMult_hw[i * N + j]) err++;

if (err == 0)
printf("Matrixes identical ... Test successful!\r\n"); else
printf("Testfailed!\r\n"); return err;
}
```

After verifying functionality with C simulation and C/RTL co-simulation, the function was synthesized into an AXI-Stream-compatible IP using Vitis HLS and imported to IP repository in Vivado.

Vivado Block Design:

To enable communication between the ARM processor and the custom matrix multiplication IP, a Vivado block design was created with AXI-based connectivity. Key components:

A. ZYNQ7 Processing System: Controls the system, manages DDR, initiates computation, and handles data transfers.
B. Matrix Multiplication IP (matmul_accel_0): **s_axi_control**: AXI4-Lite for config/status.
in_r: AXI-Stream input for matrix A & B (via DMA MM2S).
out_r: AXI-Stream output for matrix C (to DMA S2MM).
ap_clk / ap_rst_n: Clock/reset from PS.
interrupt: Optional interrupt to signal completion.
C. AXI DMA (axi_dma_0): Transfers data between DDR and IP using AXI-Stream (MM2S for input, S2MM for output).
D. AXI Interconnects: Route control signals between PS, DMA, and the custom IP.
E. Processor System Reset: Generates synchronized reset signals for the design.
This setup ensures efficient, high-speed matrix data handling and control between PS and PL.

Block Design Diagram: The hardware design on the Vivado platform is shown in Fig-5:

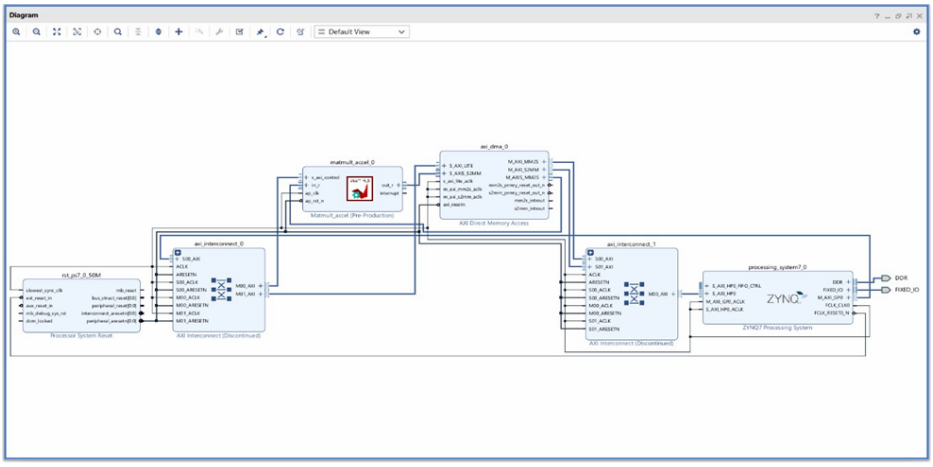


Fig-5: Hardware Accelerator on Vivado Platform

Interconnection Detail: The interconnection between different paths are discuss bellow:

I. Control Path (AXI-Lite): ZYNQ PS configures both the DMA and custom IP via axi_interconnect_1 using AXI4-Lite.

II. Data Path (AXI-Stream):

MM2S: Streams matrix A & B from DDR to the IP. S2MM: Writes the output matrix C back to DDR.

III. Clock & Reset:

Clock: Driven by FCLK_CLK0 from ZYNQ PS.
Reset: Managed by Processor System Reset using FCLK_RESET0_N.

Address Mapping

The AXI-Lite interface of both the **DMA controller** and the **custom IP** are assigned unique address ranges using the **Address Editor** in Vivado as shown in the figure-6:

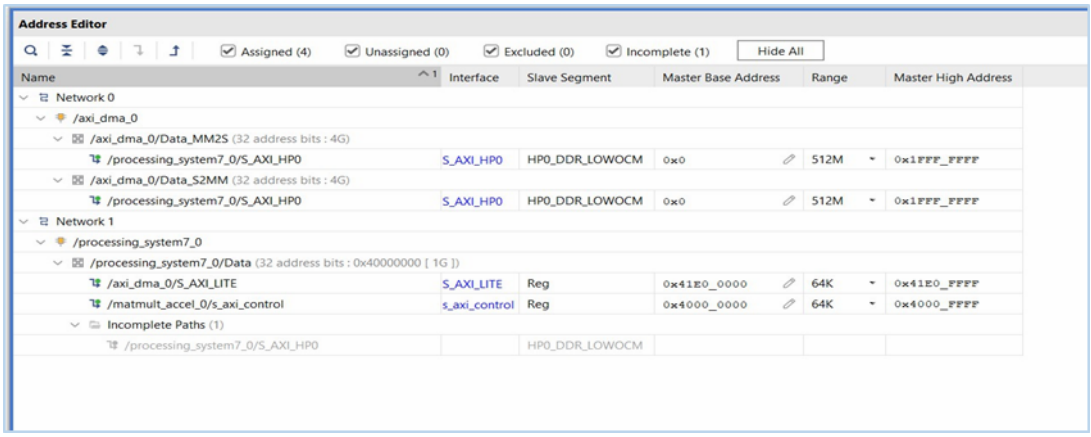


Fig-6: Adress mapping using Vivado Adress Editor

Vivado's **auto address assignment** feature was used to avoid address conflicts and simplify mapping.

Bitstream Generation:

After successfully validating the design:

- The block design was wrapped in a top-level HDL wrapper.
- The design was synthesized, implemented, and the **.bit** and **.hwh** files were generated.

Integration with PYNQ:

- Exported bitstream and .hwh file from Vivado.
- Loaded overlay in a Jupyter Notebook on PYNQ-Z2.
- Used NumPy to prepare matrices and transfer data via DMA.
- Collected and verified results using **Python.**

The python codes are as followed [11]:

```
In [1]: from pynq import Overlay

        overlay = Overlay("/home/xilinx/jupyter_notebooks/design_matmult_accel_wrapper.bit", dtbo=None)

        print("Bitstream loaded successfully!")

        Bitstream loaded successfully!
```

```
In [2]: from pynq import (allocate, Overlay)
        import numpy as np
```

```
In [3]: from pynq import Overlay

        # Load the bitstream
        overlay = Overlay("/home/xilinx/jupyter_notebooks/design_matmult_accel_wrapper.bit")

        # Print the IP dictionary
        print(overlay.ip_dict.keys())

        dict_keys(['matmult_accel_0', 'axi_dma_0', 'processing_system7_0'])
```

```
In [4]: ol = Overlay('/home/xilinx/jupyter_notebooks/design_matmult_accel_wrapper.bit')

        dma = ol.axi_dma_0
        mmult_ip = ol.matmult_accel_0
```

```
In [5]: DIM = 128
        in_buffer = allocate(shape=(2, DIM, DIM), dtype=np.float32, cacheable=False)
        out_buffer = allocate(shape=(DIM, DIM), dtype=np.float32, cacheable=False)
```

```
In [6]: CTRL_REG = 0x0
        AP_START = (1<<0) # bit 0
        AUTO_RESTART = (1<<7) # bit 7
        mmult_ip.register_map.k = DIM
        mmult_ip.register_map.m = DIM
        mmult_ip.register_map.n = DIM

        def run_kernel():
            dma.sendchannel.transfer(in_buffer)
            dma.recvchannel.transfer(out_buffer)
            mmult_ip.write(CTRL_REG, (AP_START | AUTO_RESTART))  # initialize the module
            dma.sendchannel.wait()
            dma.recvchannel.wait()

In [14]: A = np.random.rand(128, 128).astype(dtype=np.float32)
         B = np.random.rand(128, 128).astype(dtype=np.float32)

         in_buffer[:] = np.stack((A, B))

In [15]: %%timeit
         run_kernel()

         3.38 ms ± 11.1 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [16]: %timeit A @ B

         5.93 ms ± 80 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [17]: np.array_equal(A @ B, out_buffer)
Out[17]: True

In [20]: print("Shape=",np.shape(out_buffer))

         Shape= (128, 128)
```

II. Result Verification:

Co-simulation of software and hardware is performed to ensure seamless interaction between matrix generation in software and matrix multiplication in FPGA hardware, enabling accurate verification and efficient execution. The following table (Table-1) summarizes the hardware resource utilization of different IP blocks in the design, as reported by Vivado post-implementation. The metrics include Slice LUTs, Slice Registers, Slice usage, LUTs used as logic and memory, and the number of DSP slices shown in Table-1:

Name	Slice LUTs	Slice Registers	Slice	LUT as Logic	LUT as Memory	DSPs
> design_matmult_accel_wrapper	37219	44350	12866	25159	12060	160
>design_matmult_accel_i (design_matmult_accel)	37219	44350	12866	25159	12060	160
>axi_dma_0 (design_matmult_accel_axi_dma_0_0)	5139	7801	2159	4972	167	0
>axi_interconnect_0 (design_matmult_accel_axi_interconnect_0_0)	514	655	279	453	61	0
>axi_interconnect_1 (design_matmult_accel_axi_interconnect_1_0)	2555	2753	1153	2486	69	0
>matmult_accel_0 (design_matmult_accel_matmult_accel_0_0)	28998	33108	10698	17236	11762	160
>processing_system7_0 (design_matmult_accel_processing_system7_0_0)	0	0	0	0	0	0
>rst_ps7_0_50M (design_matmult_accel_rst_ps7_0_50M_0)	17	33	13	16	1	0

Table-1: Table of FPGA resource utilization

Here this can be observed that the matrix multiplication IP (matmult_accel_0) uses 160 DSPs and a moderate amount of LUTs.

The output matrix was compared with NumPy's np.array_equal(A @ B, out_buffer) to verify correctness. The output is true, which indicates the results matched, confirming functional correctness of the design. The time of execution of the multiplication in Software and Hardware are as shown in Table-2 and graphical representation is shown in figure-7:

Matrix Size	Software Time (Python/NumPy)	Hardware Time (FPGA)
128x128	5.93 ms ± 80μs per loop	3.38 ms ± 11.1μs per loop

Table-2: Execution time comparison

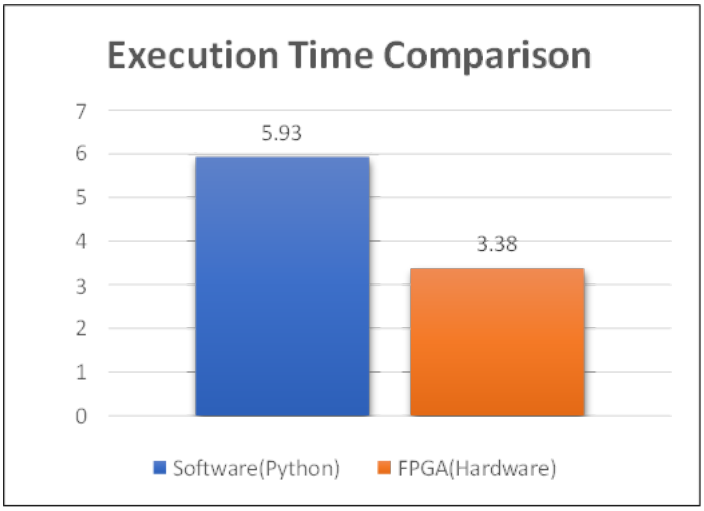


Fig-7: Bar diagram of comparing execution times

The FPGA implementation significantly outperformed compare to the software implementation or CPU execution.

The execution time of matrix multiplication using the FPGA accelerator and the standard CPU (NumPy) implementation was measured over 50 runs with randomly generated 128×128 matrices using the code given below and a plot has been generated.

```
import numpy as np import time
import matplotlib.pyplot as plt

# Run parameters
N = 128
runs = 50

# Lists to store execution times
cpu_times = [] fpga_times = []
# Repeat measurements
for i in range(runs):
A = np.random.rand(N, N).astype(np.float32)
B = np.random.rand(N, N).astype(np.float32)

# Prepare input buffer for FPGA
in_buffer[:] = np.stack((A, B))

# Time FPGA kernel
start = time.time()
run_kernel()  # Your custom function to trigger FPGA execution fpga_time = (time.time() - start) * 1000  # ms
fpga_times.append(fpga_time)
# Time CPU (NumPy)
start = time.time()
_ = np.matmul(A, B)
cpu_time = (time.time() - start) * 1000  # ms
cpu_times.append(cpu_time)
# Plotting
plt.figure(figsize=(10, 6))
plt.plot(fpga_times, label='FPGA (Accelerated)', marker='o') plt.plot(cpu_times, label='CPU (NumPy)', marker='x')
plt.xlabel('Run Index')
plt.ylabel('Execution Time (ms)')
plt.title('Matrix Multiplication Execution Time: FPGA vs CPU') plt.grid(True)
plt.legend() plt.tight_layout() plt.show()
# Summary statistics
print(f"FPGA:  Mean = {np.mean(fpga_times):.2f} ms | Std Dev = {np.std(fpga_tim es):.2f} ms")
print(f"CPU:                Mean = {np.mean(cpu_times):.2f} ms | Std Dev = {np.std(cpu_times
):.2f} ms")
```

The output provided in figure-8 shows the time taken by matrices to multiply across iterations.

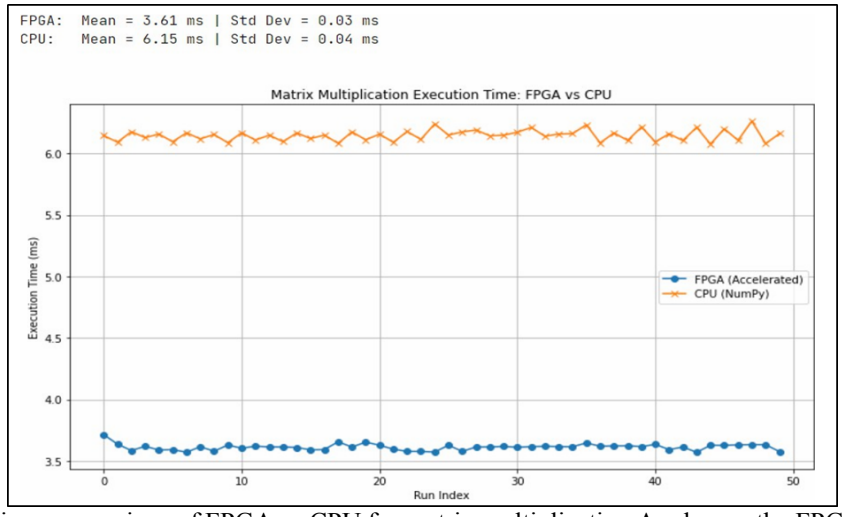


Fig-8: Execution time comparison of FPGA vs CPU for matrix multiplication As shown, the FPGA implementation consistently performs faster, with significantly lower variability.

The average execution times were:

- FPGA: 3.61 ms ± 0.03 ms
- CPU: 6.15 ms ± 0.04 ms

This confirms the performance benefit of offloading matrix multiplication to the FPGA, validating the efficiency of the hardware.

III. Conclusion:

This project demonstrates efficient, hardware-accelerated matrix multiplication on the PYNQ-Z2 FPGA using a custom IP core, AXI Stream, and DMA for high-throughput data transfer. Integrated with the Zynq Processing System, the design achieves reliable and parallel computation. Despite challenges like AXI synchronization and memory limits, the system performs effectively. The work highlights the benefits of FPGA-based acceleration and sets the stage for future improvements like tiling, floating-point support, and deep learning integration.

Future Work:

1. **Tiling support for Large Matrices:** Implement matrix tiling to handle large datasets by dividing them into smaller blocks, reducing BRAM usage and improving scalability [12].
2. **Convolution Acceleration:** Map 2D convolution operations to hardware using systolic arrays or line buffers to speed up deep learning workloads [13].
3. **PYNQ Overlay & GUI:** Develop a user-friendly interface for uploading matrices, running computations, and visualizing results via a web-based GUI.
4. **Benchmarking:** Evaluate performance and energy efficiency against CPU/GPU baselines using standard datasets to validate the FPGA system's effectiveness.

References:

[1]. Http://Petewarden.Com/2015/04/20/Why-Gemm-Is-At-The-Heart-Of-Deep-Learning/
[2]. Https://Www.Tulembedded.Com/Fpga/Productspynq-Z2.Html
[3]. A. V. Trusov, E. E. Limonova, D. P. Nikolaev, V. V. Arlazarov, P-Im2col: Simple Yet Efficient Convolution Algorithm With Flexibly Controlled Memory Overhead, Ieee Access.
[4]. G. Noblea, S. Naleshb, S. Kalaa, A. Kumar, Configurable Sparse Matrix -Matrix Multiplication Accelerator On Fpga: A Systematic Design Space Exploration Approach With Quantization Effects, Alexandria Engineering Journal.
[5]. Https://En.Wikipedia.Org/Wiki/Dot_Product
[6]. Https://Www.Sciencedirect.Com/Topics/Engineering/Outer-Product
[7]. Https://Www.Youtube.Com/Watch?V=1sxy73f3eha
[8]. Https://Www.Reddit.Com/R/Fpga/Comments/1ch565g/What_Is_Fpga_Acceleration/
[9]. Https://Indico.Cern.Ch/Event/1170079/Attachments/2484554/4269719/Cern%20openlab%20lect Ure%2021-07-2022.Pdf
[10]. Https://Www.Logic-Fruit.Com/Blog/Fpga/Fpga-Design-An-Ultimate-Guide-For-Fpga-Enthusiasts/
[11]. Https://Github.Com/Twaclaw
[12]. K. Xie, Y. Lu, D, Yi, H. Dong, Y. Chen, Winols: A Large-Tiling Sparse Winograd Cnn Accelerator On Fpgas, Acm Digital Library.
[13]. Https://Dl.Acm.Org/Doi/10.1145/3688636.3688655