

# An approximate model SpMV on FPGA assisting HLS optimizations for low power and high performance

Alden C. Shaji, Zainab Aizaz, Kavita Khare

Department of Electronics and Communication, Maulana Azad National Institute of Technology, Bhopal, India

## Article Info

### Article history:

Received May 4, 2024

Revised Jun 4, 2025

Accepted Jun 10, 2025

### Keywords:

Field-programmable gate array  
High level synthesis  
High performance computing  
Optimization methods  
Sparse matrix-vector  
multiplication

## ABSTRACT

High performance computing (HPC) in embedded systems is particularly relevant with the rise of artificial intelligence (AI) and machine learning at the edge. Deep learning models require substantial computational power, and running these models on embedded systems with limited resources poses significant challenges. The energy-efficient nature of field-programmable gate arrays (FPGAs), coupled with their adaptability, positions them as compelling choices for optimizing the performance of sparse matrix-vector multiplication (SpMV), which plays a significant role in various computational tasks within these fields. This article initially did analysis to find a power and delay efficient SpMV model kernel using high level synthesis (HLS) optimizations which incorporates loop pipelining, varied memory access patterns, and data partitioning strategies, all of this exert influence on the underlying hardware architecture. After identifying the minimum resource utilization model, we propose an approximate model algorithm on SpMV kernel to reduce the execution time in Xilinx Zynq-7000 FPGA. The experimental results shows that the FPGA power consumption was reduced by 50% when compared to a previously implemented streaming dataflow engine (SDE) flow, and the proposed approximate model improved performance by 2× times compared to that of original compressed sparse row (CSR) sparse matrix.

This is an open access article under the [CC BY-SA](#) license.



## Corresponding Author:

Alden C. Shaji

Department of Electronics and Communication, Maulana Azad National Institute of Technology  
Bhopal, India

Email: aldencshaji@gmail.com

## 1. INTRODUCTION

High performance computing (HPC) plays a vital role in the field of artificial intelligence (AI) by providing computational power required for training and running complex models. Many HPC systems incorporate specialized hardware accelerators, such as field-programmable gate arrays (FPGAs) or graphics processing units (GPUs), to offload floating-point computations from traditional CPUs. These accelerators are optimized for parallel processing and can significantly boost performance for floating-point-intensive workloads [1]. Floating-point accelerators are engineered to deliver robust computational capabilities while also taking into account energy efficiency, a critical consideration in HPC systems where power consumption and heat management are major challenges. Also, some of the key challenges associated with sparse matrix-vector multiplication (SpMV) computation on FPGAs are irregular memory access patterns, load imbalance, limited on-chip memory resources and energy efficiency. The flexibility of FPGAs, being programmable hardware, allows for the customization of floating-point accelerators to suit specific workloads. This adaptability give advantage in HPC applications, by providing solutions that can deliver optimal performance for diverse computational tasks [2], [3].

SpMV is a foundational operation within HPC and holds vital significance across scientific, engineering, and data analysis domains. This operation entails the multiplication of a sparse matrix distinguished by a substantial volume of zero elements with a dense vector. SpMV utility extends to tasks such as solving linear equation systems, simulating physical phenomena, and conducting graph computations, under-scoring its essential role in diverse applications [4]. Custom memory access patterns are critical for improving the performance of SpMV on embedded FPGAs. By tailoring memory hierarchies and data structures, FPGAs can minimize memory latency and maximize bandwidth, leading to enhanced efficiency in SpMV computations [5].

SpMV involves accessing non-contiguous memory locations owing to the sparse nature of matrices. This irregular memory access pattern can lead to cache misses, causing increased memory latency and affecting the overall performance of both CPUs and GPUs [6], [7]. In addition, the workload in SpMV is not evenly distributed among the processing elements owing to the varying sparsity of the matrices. This load imbalance can lead to inefficient utilization of resources, especially on GPUs where thread-level parallelism is crucial. Consequently, CPUs and GPUs may not be the most suitable platforms for accelerating SpMV kernels. In contrast, FPGAs emerge as a promising solution for SpMV acceleration. FPGAs boast large off-chip storage bandwidth, allowing them to efficiently handle memory bound applications. Their tailored logical components and efficient floating-point computations enhance its standing even more in FPGAs as an attractive platform for accelerating SpMV computations [8].

Research findings indicate that high level synthesis (HLS) holds promise for furnishing high-performance, energy-efficient solutions, thereby expediting time-to-market and tackling the complexities of modern systems concurrently [9], [10]. Our investigation focuses on exploring the application of HLS, a technique that is gaining popularity for accelerating algorithms on embedded heterogeneous platforms. After finding the efficient optimization technique we calculated the kernel power consumption in the embedded FPGA, then propose two novel approximate compressed spectral regression (CSR) matrix to minimize the execution time for the kernel in the hardware.

The remainder of this paper is organized as follows. Section 2 provides background information and related work on the SpMV and HLS flows. Section 3 presents the methodology used in this study paper to bring out the results, including the usage of pragmas and a novel approximation model algorithm. The result and discussion are presented in section 4. The paper concludes with the conclusion and future scope in section 5.

## 2. THE COMPREHENSIVE THEORETICAL BASIS

### 2.1. Sparse matrix-vector multiplication

Sparse matrices, in contrast to dense matrices that hold a substantial amount of redundant information, primarily consist of zero values, leading to more efficient memory usage. SpMV involves the multiplication of a sparse matrix with a dense vector, ultimately producing a new vector that represents the linear transformation of the original data expressed as (1).

$$Y_i = \sum_{j=0}^{Row} \sum_{k=0}^{Col} A_{ij} \times X_j, \text{ if } A_{ij} \neq 0 \quad (1)$$

Sparse matrices are typically encoded in condensed formats that only contain the non-zero members in order to restrict the data collection needed. The ratio of total zero elements to total elements in a sparse matrix determines the matrix's sparsity. Figure 1 provides an overview of the SpMV process along with the common compressed formats used to store sparse matrices. The example SpMV kernel in Figure 1(a) is represented using three commonly used compressed formats COOrdinate (COO), compressed sparse column (CSC), and compressed sparse row (CSR) as shown in Figure 1(b). Out of these widely used is CSR format. The val vector holds the non-zero elements mentioned by *nnz* determines their size and their corresponding column indices are saved in col vector. In ptr vector, the difference between adjacent cells gives the no. of non-zero elements (*nnz*) present in corresponding row in sparse matrix. The CSR format is appropriate for computing with streaming data and only requires a brief preparation stage [11]. Also, CSR reduce the memory needed from  $O(m \times n)$  to  $O(2nnz+m)$  due to this we have used it in our work.

SpMV is a versatile operation with applications in a wide range of fields, offering computational efficiency and memory savings when dealing with sparse data structures. Its broad applicability makes it a fundamental operation in various scientific, engineering, and data-driven disciplines. SpMV application in convolutional neural networks (CNN) training is shown in Figure 2, this training scheme was used in [12] to get fast execution of CNN on GPUs. During the forward pass of CNN training, SpMV is applied when computing the output of convolutional layers. The sparse weight matrices are multiplied by the input data vectors, and the resulting sparse vector contributes to the activation of neurons in subsequent layers. In the

backward pass (backpropagation) during training, gradients with respect to the weights are calculated efficiently, taking advantage of the sparsity in both the input data and the weight matrices. This enables faster updates to the weights during optimization.

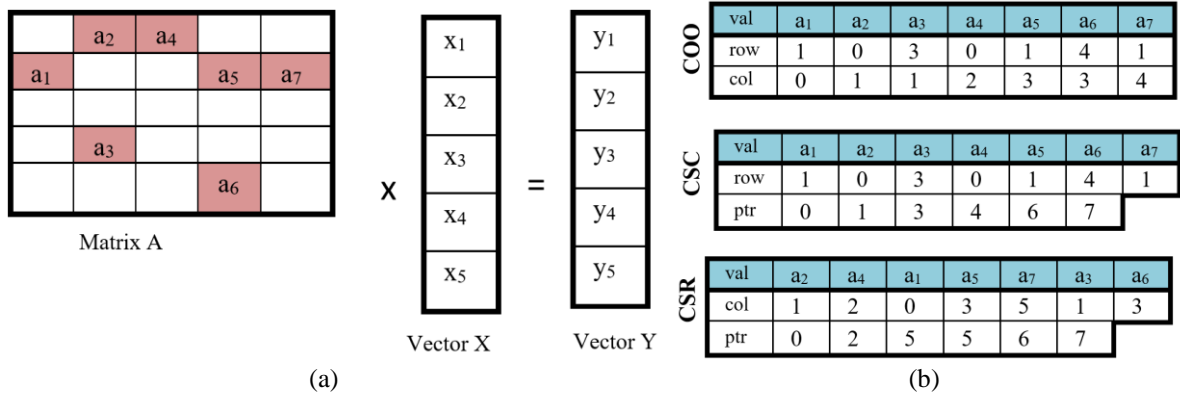


Figure 1. Sparse matrix-vector multiplication and conventional compress format (a) an example of SpMV and (b) conventional compressed formats

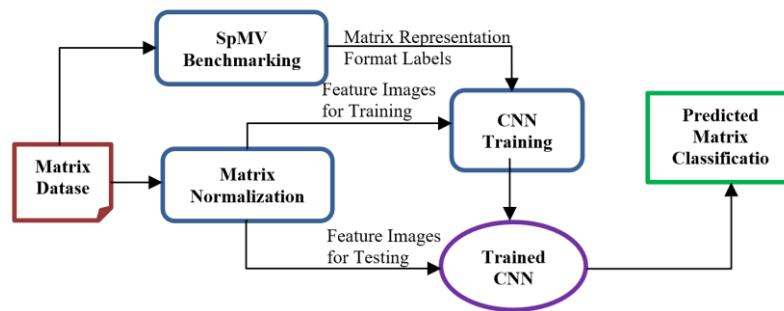


Figure 2. SpMV application in CNN training

## 2.2. High level synthesis

Vitis HLS is a tool provided by Xilinx that takes high-level C or C++ functions and translates them into RTL code, which can then be implemented in the programmable logic region of a system on chip (SoC). It generates a hardware solution by considering the defined target flow, default tool settings, design constraints, and optimization pragmas provided. Optimization directives are utilized to customize and manage the internal logic and input/output ports implementation, superseding the tool's default actions and configurations. To attain optimal performance from the hardware generated, the HLS tool needs to identify and utilize parallelism inherent in sequential code, enhancing overall performance. SpMV pseudo code implemented using HLS is shown in Figure 3 [8].

```

1 void SpMV_Ref(int n, float *value, int *col_index, int *
  row_index, float *x, float *y) {
2   int rowStart = 0, rowEnd = n;
3
4   for (int i = rowStart; i < rowEnd; ++i) {
5     float y0 = 0.0;
6     for (int j=row_index[i]; j<row_index[i+1]; j++) {
7       int k = col_index[j];
8       y0 += value[j] * x[k];
9     }
10    y[i] = y0;
11  }
12 }

```

Figure 3. SpMV kernel pseudo code in HLS environment

In high level language programs, the arrays are essential for storing and managing data. When translating this to hardware, arrays are realized as either memory or registers during synthesis. Memory can be either local or global, with global memory often corresponding to double data rate (DDR) or high-bandwidth memory (HBM) memory banks. Accessing global memory incurs higher latency and multiple cycles, whereas local memory access is faster and typically completed within a few cycles. Efficient memory access is essential to minimize the overhead associated with accessing global memory. One strategy for optimization involves consolidating access, maximising consecutive accesses to enable bursting. Burst access effectively masks memory access latency and enhances memory bandwidth. While the process of blocks of data, several loops or nested loops are needed. With the combination of micro-level HLS pragmas, it can perform unroll, pipeline operations for a loop or nested loops [13].

HLS tools apply various optimizations to improve the performance, area, and power characteristics of the generated hardware. HLS tools often provide simulation and verification capabilities, it helps to simulate the behaviour of the generated hardware before the actual synthesis. The output of HLS can be targeted for implementation on FPGAs for rapid prototyping providing flexibility in the choice of hardware platform.

### 2.3. Related work

The concept of sparse matrices and related operations like SpMV can be traced back to the early days of numerical computing and finite element analysis. Various algorithms and storage formats were developed to accelerate SpMV, finding unique characteristics of sparse matrices in [14], [15]. In various studies they have investigated the optimization of SpMV on FPGAs [16], [17]. The majority of these research endeavors concentrate on leveraging high-end FPGAs and implementing approaches geared towards processing big data efficiently in [18]. Du *et al.* [19] investigate a sparse matrix format specifically designed for HBM. Another comparable effort, ReDESK [20] has examined SpMV optimizations in the context of heterogeneous computing, it is designed to enable streaming process on the FPGA side and data prefetching on the CPU side.

Design of bandwidth efficient SpMV on FPGA is the main theme in [5], [21]. Fowers *et al.* [22] proposed an architecture for SpMV based on FPGA, along with a technique for sparse matrix decoding to leverage parallelism across matrix rows. The design assumes the presence of two distinct DRAM modules in the system, a feature that may not be commonly found in many existing embedded systems.

Hosseiniabady and Nunez-Yanez [8] investigated on how parallelization and pipelining can be effectively applied using HLS to increase the performance of SpMV on FPGA platforms. This includes strategies for optimizing data movement and memory accesses. Garibotti *et al.* [10] suggested employing commercial HLS tools along with dynamic analysis to produce higher quality designs. Creating an effective floating-point accumulator, which encompasses both multiplier and adder components, to improve the performance of SpMV is the objective in [23], [24]. Recently, a work [25] compares the SpMV calculation, showcasing the performance and energy computation on GPU and FPGA. Furthermore, current optimizations primarily target half precision floating point data types, overlooking support for reduced precision fixed point arithmetic [26]. However, recent studies have investigated strategies for blending single and double precision floating-point arithmetic [27].

Finally, in contrast to other works, this paper proposed a novel approximation model SpMV to reduce the power and execution time, using which can significantly transform the FPGA accelerator. We have compared the power consumption and execution time of same matrices with the implementation in [8] in the results. Experimental work on HLS pragmas and approximate algorithms are discussed in detail in further sections.

## 3. METHOD AND EXPERIMENTAL SETUP

In this section, the details of the design model used for SpMV implementation on FPGA are provided. Initially, we find out the trade-off between execution time and the resource utilization using the HLS optimization techniques. Then we select the efficient technique based on hardware emulation and implementation. After that we applied the approximation model algorithm to the sparse matrix and did the analysis. We have compared the results obtained in our target hardware Zybo Z7-20 board with Xilinx ZCU102 evaluation board used in [8].

### 3.1. High level synthesis optimization techniques

Utilization of HLS tools is to harness the productivity benefits of translating C/C++ code into RTL for hardware, or objective is to accelerate a subset of a C/C++ algorithm by running it on a specialized hardware built with programming logic. Functions implemented in C/C++ and transformed into custom

hardware using programmable logic can operate at notable higher speeds compared to what is attainable on typical GPU/CPU setups, resulting in higher throughput and performance. The proposed SpMV implementation have 3 primary tasks involved: *Task A*: reading of data into the FPGA memory, *Task B*: initiating the stream computation engine, and *Task C*: transferring output of FPGA to main memory. The following optimization techniques are used to implement the SpMV computation.

### 3.1.1. Loop pipelining

Loops are crucial constructs within an SpMV. Since loop body is executed repeatedly, this characteristic can be effectively leveraged to enhance parallelism and optimize performance. To enhance throughput and make more efficient use of computational resources, a valuable approach is to introduce pipelining in operators, loops and functions. Figure 4 shows the example flow of 3 tasks (each took 10 units to complete) before and after pipelining. To finish the first workload it took 30 units, is called the iteration latency. After the completion of first workload, next two workloads only take 10 units each, called the initiation interval (II). The overall completion of all the workloads is called the total latency, which is 50 here. The general formula for finding total latency for N no. of workloads is given in (2).

$$\text{Total Latency} = \text{Iteration Latency} + II \times (N - 1) \quad (2)$$

In a pipelined function or loop, new inputs can be processed every specified  $II$  clock cycles.  $II=1$  implies processing a new input every clock cycle. The maximum throughput that a pipelined loop can reach without unrolling is attained at this point. Sometimes this not possible, due to resource constraints and loop carried dependencies. The pipelined loop will automatically unwind any nesting loops. A common issue in pipelined loop is memory conflict. There are four loops in the kernel code in which we applied pipeline on every loop with  $II=1$ , considered both the cases of with and without pipelining the loops.

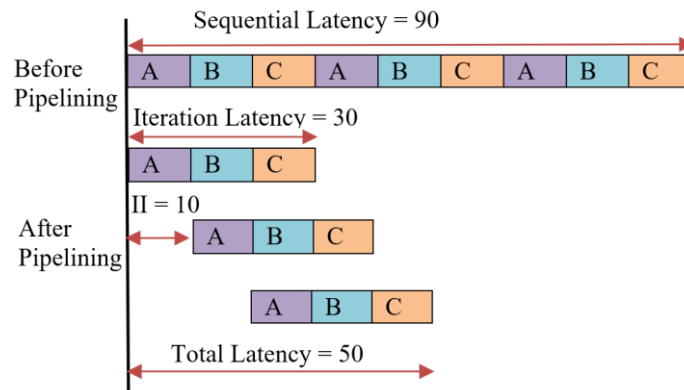


Figure 4. Pipelining flow for three tasks

Pipelining reduces the latency of the computation by allowing the creation of new loop iterations before the completion of previous ones. This is crucial in SpMV where the data dependencies are often sparse, and overlapping computations can significantly improve overall performance. It can contribute to achieving higher clock frequencies by breaking down the computation into smaller, more manageable stages.

### 3.1.2. AXI burst transfer

Bursting is an optimization strategy aimed at smartly consolidating memory accesses to DDR in order to reduce latency and increase throughput bandwidth. The AXI4 protocol's burst functionality boosts the load-store function's throughput by enabling them to read or write a large group of data to or from the global memory in a single request. The throughput increases with the size of the data being transferred. Optimising different HLS interface metrics, such as port width, burst access, latency, numerous ports, and the number of unfinished reads and writes, is necessary to develop effective load-store functions. Utilizing local buffers to store segments of the matrix or vector during computation proves beneficial by mitigating memory access latency and enhancing data reuse. This strategy involves temporarily holding subsets of the data within the fast on-chip memory, allowing the processor to access and manipulate the information more efficiently. By minimizing the need for frequent off-chip memory accesses, local buffers contribute to

optimizing overall computation performance. In the hardware we have four 64-bit AXI high-performance memory ports to transfer data from programming logic to DDR memory.

### 3.1.3. Loop unrolling

Aims to improve performance by reducing loop overhead and increasing parallelism. In loop unrolling multiple iterations of the same loop are performed within a single iteration. Unrolling factor is the no. of iterations to execute in each unrolled iteration. The loop can be partially or completely unrolled with the UNROLL pragma. Fully unrolling makes a duplicate of the loop body for each iteration, enabling concurrent operation of the whole loop. On the other hand, partial unrolling entails setting a factor N to make N copies of the loop body and decrease the loop iterations appropriately. The limits of a loop must be known at compile time in order to fully unroll it. Loop unrolling needs more computation and storage resources hence it is a trade-off between performance and resources. In our experiments we implemented the kernel code with unroll factor of 2 and 4.

The most effective unroll factor for loop unrolling in SpMV using Vitis HLS depends on the unique characteristics of the targeted FPGA architecture and the specific attributes of the SpMV problem being addressed. Conducting experiments with various unroll factors and leveraging performance profiling through Vitis HLS reports is crucial for identifying the optimal configuration that maximizes computational efficiency. This iterative process enables fine-tuning and customization, ensuring the SpMV kernel is tailored for optimal performance on the specific FPGA platform and problem domain.

### 3.1.4. Array partitioning

It involves breaking down a single array into smaller, independent parts or subsets such that each part can be implemented as a BRAM, so that can access them at the same time. Aggregate types can be divided into smaller memories or into their component parts, which increases the memory bandwidth and increases the number of memory accesses on each cycle. Block, cyclic, and complete array partitioning are the three types available. The options like type and dim for the memory partition pragma specify the partition type and dimension, respectively. Large array size will be synthesized into BRAMs in FPGA. Here we declared the variables with array partition in cyclic factor with dim=1.

Using #pragma HLS ARRAY\_PARTITION variable=x complete dim=1 partitions the input matrix x along its rows (specified by dim=1). Complete partitioning is employed, indicating that each partition comprises an entire set of rows from the matrix. This optimization aims to boost parallelism by enabling concurrent processing of multiple rows within the matrix. Within the computation loop, partial sums are calculated for each row by leveraging the partitioned matrix and the input vector. This approach enhances parallel execution, thereby optimizing memory access patterns and improving the overall performance of SpMV kernels on FPGA platforms.

### 3.1.5. Bind storage

It links a code variable to a certain memory type in the RTL. The memory type associated with the array influences the number and kind of ports required in the RTL, making this element important for the arrays on the top-level function interface. These variables must use the storage\_type and storage\_impl options of the BIND\_STORAGE pragma to specify the memory type and implementation. The latency option for BRAMs on the interface enables the memory to be implemented using more pipelined stages. Timing issues that arise during RTL synthesis can be effectively resolved by adding extra pipeline stages.

## 3.2. Sparse matrix-vector multiplication-kernel

A kernel typically refers to a computational routine or algorithm that is specialized for a particular operation. SpMV kernel is a specific implementation or routine designed to efficiently perform the multiplication of a sparse matrix with a dense vector. Minimize dynamic memory allocations and deallocations during the computation to avoid unnecessary overhead. Access patterns should be designed to minimize cache misses during the multiplication and accumulation steps.

The source code for the SpMV kernel is given in Figure 5 which is referenced from [8]. The pseudo code contains four for loops, in which data fetching from the sparse matrix is happened first, then it will uncompress the CSR format matrix by fetching each data value from each row of the matrix. After getting all the nnz's then multiplication with the corresponding element in the dense vector and accumulate the result from the multiplication step into the corresponding entry of the output vector. Repeat the multiplication and accumulation steps for all non-zero elements in the sparse matrix. The last loop is for the transfer of the output data containing the result of the SpMV operation to the output terminal.

```

for (unsigned int i = 0; i < m; i++) { //Input Data Fetching
    x_local[i] = x[i];
}
for (unsigned int i = 0; i < n+1; i++) { //uncompressing CSR
    unsigned int row_index = row_indices[i];
    if (i > 0) {
        row_indices_diff_local[i-1] = row_index-previous_row_index;
        nnz += row_index-previous_row_index;;
    }
    previous_row_index = row_index;
}
double y_previous_break = 0.0;
double y_all_row = 0.0;
unsigned int j = 0;
unsigned int remained_row_index = row_indices_diff_local[j++];
for (int i = 0; i < nnz; ++i) { //SpMV
    int k = col_indices[i];
    float y_t = values[i] * x_local[k];
    y_all_row += y_t;

    remained_row_index--;
    if (remained_row_index == 0) {
        y_local[j-1] = y_all_row - y_previous_break;
        y_previous_break = y_all_row;
        remained_row_index = row_indices_diff_local[j++];
    }
}
for (unsigned int i = 0; i < n; i++) { //Output Data Transfer
    y[i] = y_local[i];
}

```

Figure 5. SpMV kernel pseudo code with pipelining

### 3.3. Approximate sparse matrix-vector multiplication algorithm

As far as we are aware, there is currently no existing research that focuses on optimizing the computation of approximate model SpMV on FPGA. Despite previous studies addressing optimizing techniques on FPGA for dense matrix multiplications and deep learning, there appears to be a gap in the literature regarding the specific optimization of approximate model SpMV on these hardware platforms. The computational performance of CPUs in this task is inherently limited by their restricted memory bandwidth and the difficulty of efficiently executing frequent random accesses. This limitation arises from the fact that there are no assurances that the required values have not been taken from the cache, impeding the ability to access data quickly and reliably.

The motivation behind approximate SpMV algorithms is to accelerate the computation of matrix-vector multiplication in scenarios where an exact solution is not strictly necessary. This is common in machine learning, signal processing, and other applications where an approximate result is acceptable. The key trade-off in approximate SpMV algorithms is between computational speed and solution accuracy. Approximate SpMV models can be designed to scale better with increasing matrix sizes. This is especially beneficial when dealing with large datasets in scientific simulations or machine learning applications.

In this section, we suggested a unique approximate model approach for SpMV, to shorten the execution time. Efficiency in SpMV is often achieved through algorithms and data structures that lessen the number of arithmetic operations and memory access by taking advantage of the matrix's sparsity. We have taken the sparse matrix  $S_o$  as the input and obtain the approximate CSR format matrices  $S_t$  and  $S_v$  as outputs. Implementation results are shown in section 5. This algorithm contains two types of approximation:

AX-1: here the approximation of SpMV is based on thresholding the row count of the matrix. Only taking the data values which are higher than the threshold and stores it in  $S_t$ . Threshold is calculated as the mean of max and min row count values of the sparse matrix. In Algorithm 1, step 4–11 corresponds to this approximation.

AX-2: here the approximation is based on the accuracy of the data value. After sorting the input sparse matrix based on data value, it is classified into positive matrix ( $S_p$ ) and negative matrices ( $S_n$ ). Then taken only the high accuracy values of 70% of total NNZ's. Both matrixes are joined together in  $S_v$  and again

do the sorting based on the row values. It is then converted into the CSR format. In Algorithm 1, step 12–19 corresponds to this approximation.

#### Algorithm 1. Approximate SpMV algorithm

Input:

The original sparse matrix,  $S_o$ ;

Output:

The target approximate CSR format,  $S_t$  and  $S_v$ ;

```

1: Obtain the matrix parameters from  $S_o$ ;
2: Count the no. of  $NNZ$ 's in each row and store in  $row\_count$ ;
3: Initialize the matrix  $S_t$ ,  $S_v$  with rows of specified  $NNZ$ 's,  $j=0$ ;
4: Obtain the max and min  $row\_count$  values;
5: Find the  $threshold$  by taking mean of values from 4;
6: while  $j < NNZ$  do
7:   if  $row\_count < threshold$  then
8:     skip adding those row's values to  $S_t$ ;
9:   else add those rows to  $S_t$  and increment the size;
10:  add corresponding col      and data value to  $S_t$ ;
11: end while
12: sort  $S_o$  based on absolute data value;
13: Classify  $S_o$  into two matrices as  $S_p$ ,  $S_n$  based on integer;
14: Count the no. of elements in  $S_p$ ,  $S_n$  and store in  $p$ ,  $n$ ;
15: for  $i$ :  $p \times 0.3$  to  $p$ 
16:  copy the corresponding values from  $S_p$  to  $S_v$ ;
17: for  $i$ : 1 to  $n \times 0.7$ 
18:  copy the corresponding values from  $S_n$  to  $S_v$ ;
19: sort  $S_v$  based on row value;
20: Convert matrix  $S_t$  and  $S_v$  into CSR format;

```

### 3.4. Experimental setup

To evaluate the proposed methods, we setup the host environment with processor Intel Core i5-7500 @ 3.4 GHz×4, Memory of 7.6 GiB. The target device as Zybo Z7-20 contains the FPGA Zynq-7000 platform which consists of XC7Z020-1CLG400C chip and it also contains dual-core ARM Cortex-A9 processor. Our design utilizes the four 64-bit high performance memory ports in the programmable logic. It supports data transfer from processing system to DDR memory with memory bandwidth of 12.2 GB/s. It also contains an on-chip memory of 256 KB which is useful in reducing the throughput while computing. We set the frequency to 150 MHz on FPGA for the execution of SpMV accelerator. The tool used for implementation of the design is Vitis HLS 2020.2.

Installed PetaLinux and Vitis HLS, two distinct tools provided by Xilinx that serve different purposes. PetaLinux is primarily used for building and customizing Linux distributions for Xilinx devices, while Vitis HLS is focused on HLS, converting C, C++, and OpenCL code into hardware implementations. By running config, build and package for PetaLinux tool will generate the boot image that includes the first stage boot loader (FSBL), bitstream, and other necessary components.

As input data for the experiment, we used a set of sparse matrices from the University of Florida's sparse matrix collection [28]. The selected matrices dimension and  $NNZ$ 's are less than  $10^6$ , respectively. The features of the sparse matrices are given in Table 1, according to that all the matrices have sparsity above 98% and four matrixes in the selected are floating point type and remaining one is integer type value. The optimization techniques used in the SpMV kernel are modelled into 8 types as shown in Table 2. Combination of different optimization techniques are used with HLS pragmas in modelling and obtained the performance results in Vitis HLS.

Table 1. Sparse matrix statistics

Matrix name	Row size	Col size	NNZ	Sparsity (%)	Type
c-48.mtx	18354	18354	92217	99.97	FP value
cage8.mtx	1015	1015	11003	98.93	FP value
g7jac080.mtx	23670	23670	293976	99.95	FP value
mhd4800a.mtx	4800	4800	102252	99.56	FP value
TF16.mtx	19321	15437	216173	99.93	INT value



Table 2. Optimization model

Model	Optimization technique
A	Pipeline OFF
B	Pipeline OFF and unroll factor=2
C	Pipeline OFF and unroll factor=4
D	Pipeline OFF and array partition
E	Pipeline ON
F	Pipeline ON and unroll factor=2
G	Pipeline ON and array partition
H	Pipeline ON and bind storage

#### 4. RESULTS AND DISCUSSION

This segment analyses the proposed SpMV approximate algorithm with optimization techniques. Initially, a series of 5 sparse matrices chosen as benchmarks for the purpose of examining the effects of each optimization technique described. Zybo Z7-20 taken as our target board, which contains Zynq 7000 FPGA and with Vitis HLS tool we implemented software emulation, hardware emulation, and hardware build. The execution time in Figure 6 is taken after passing the test while running software emulation. In hardware emulation, it checks the functional correctness of the RTL code synthesized from the OpenCL kernel code. It gives the resource utilization, estimated frequency and the number of cycles taken for the execution of the task in the target device. In hardware build the tool will generate the FPGA bitstream for the corresponding device after running several steps including logic placement, optimization, routing, timing optimization in Vitis tool. The resource utilization for each model is given in Table 3, according to that we have calculated the power consumption for each model using Xilinx power estimator.

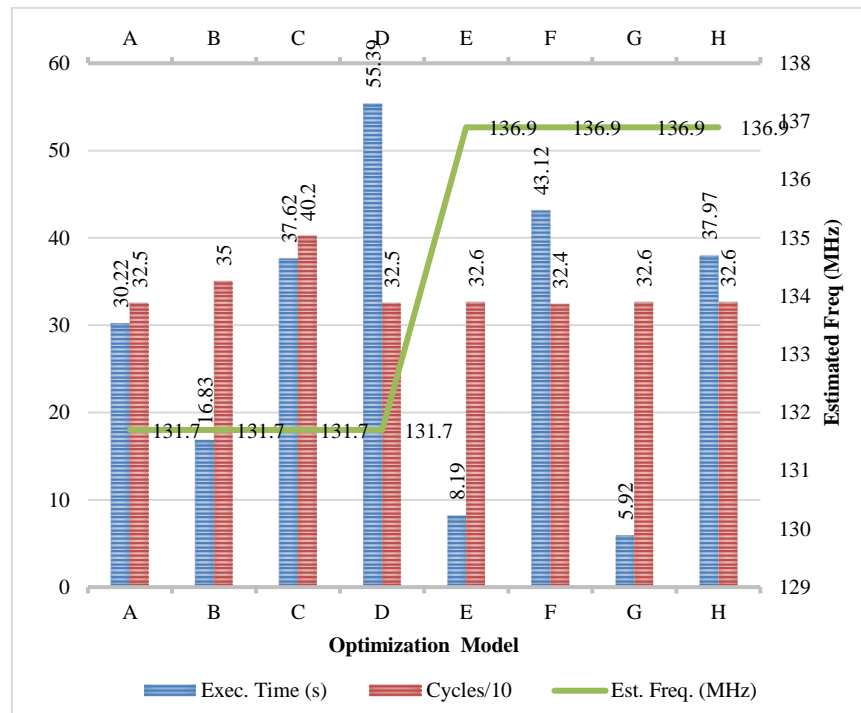


Figure 6. Evaluation of design performance in Vitis HLS

Table 3. Optimization model

Model	DSP (220)	BRAM_18K (280)	FF (106400)	LUT (53200)
A	12	100	4115	7645
B	6	100	4392	7879
C	6	100	5022	9805
D	12	100	4099	7783
E	12	100	4462	7751
F	6	100	5225	8008
G	12	100	4571	7951
H	12	100	4462	7717

From the results of software and hardware emulations, we conclude that model G which is using Pipeline with array partition is giving better results for a common sparse matrix as shown in Figure 5. The models which use pipeline is taking an estimated time of 7.3 ns for a target time of 6.7 ns with a slack of -0.6 ns, and the models without pipelining is taking an estimated time of 7.6 ns with a slack of -0.9 ns. Thus, pipelining is efficient for reducing the slack and improving the performance in HLS. Except B and C models, remaining all models taking 326-324 cycles to complete the task.

Power consumption depends on various factors, including the hardware architecture, clock frequency, resource utilization, and the nature of the operations performed by the models. Examine the resource utilization reports from Vitis HLS to understand how much of the FPGA resources each model is consuming. This includes details on look-up tables (LUTs), flip-flops (FFs), BRAMs, digital signal processors (DSPs), and other resources. Higher clock frequencies generally lead to better performance but can also increase power consumption. Dynamic power consumption is influenced by the activity and switching of logic elements during operation whereas static power consumption is associated with the leakage power of the FPGA. This component is independent of the activity and can be significant in low-power applications.

Power utilization for each model is calculated by Xilinx power estimator with the resource utilization got from the Vitis HLS tool. From the calculation model B is taking the lowest power consumption. The power utilization comparison of our model with the streaming dataflow engine (SDE) SpMV model from [8] is given in Figure 7. They have used ZCU102 evaluation board which contains Zynq UltraScale+ MPSoC as their target device, which is higher end device as compared to Zynq-7000. From the comparison chart for the SpMV model, our model is taking almost half of the power consumption taken by SDE model.

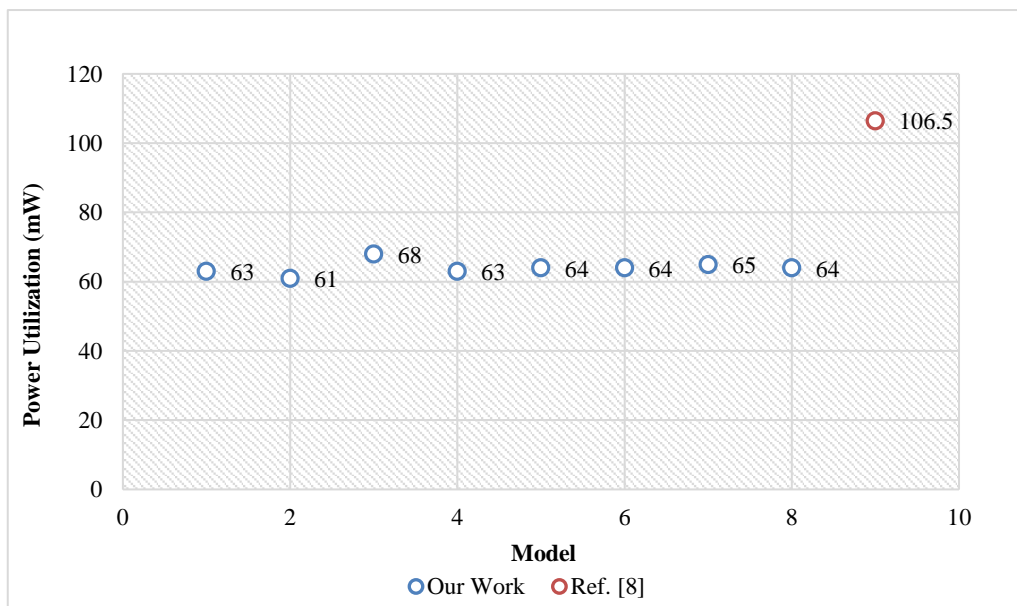


Figure 7. Power utilization comparison chart

In optimization model G we calculated the execution time (milli seconds) is shown in Figure 8 for 5 selected sparse matrices mentioned in Table 1. The approximate algorithm applied to the matrices given to the G model SpMV kernel and analyzed the execution time in the target FPGA. We find out that AX-1 approximation model is reaching 50% reduction in execution time for floating point value sparse matrix compared to its original. But for integer type value sparse matrix AX-2 approximation model getting much reduction than other. Also observed that the reduction is higher when the *NNZ*'s are larger. Execution time taken for both approximate model AX-1 and AX-2 for the sparse matrices in comparison with SDE model in [8] is shown in Figure 8. In [8] they have used target device as Zynq UltraScale+ MPSoC FPGA with a frequency of 200 MHz, which reduces the execution time significantly.

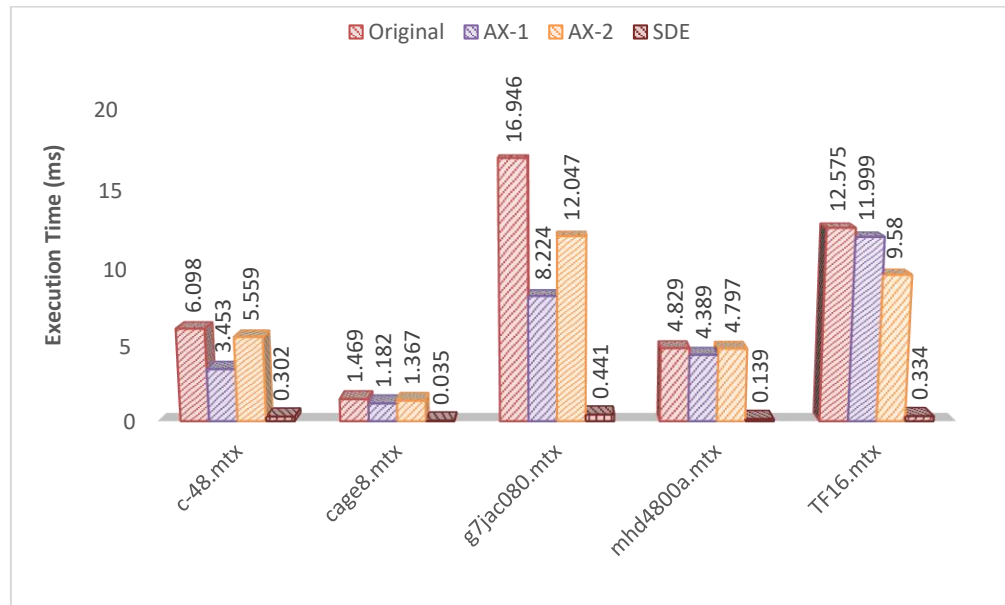


Figure 8. Execution time comparison for each model

## 5. CONCLUSION

Efficient approximate SpMV implementations developed through HLS offer versatile applications spanning scientific computing, machine learning, and signal processing domains. However, deploying these solutions effectively demands seamless integration, robustness, and vigilant performance monitoring in real-world scenarios. Approximation techniques can significantly accelerate SpMV computations by trading off accuracy for speed. By approximating complex operations or reducing precision where acceptable, HLS generated kernel can execute SpMV operations faster than conventional methods.

In this article, we introduce a novel approximate model in the HLS optimized SpMV kernel. These sparse kernels have a CSR sparse matrix format, on the heterogeneous platform it can execute non-blocking pipelined operations efficiently, optimizing bandwidth usage by assigning irregular memory access and control patterns to appropriate processes. The approach relies on stream computing methodologies, where computation and data transfer between main memory and the FPGA occur concurrently in a pipelined manner. The experimental results point towards a power and delay efficient SpMV kernel on Zynq 7000 FPGA. It also figured out that pipeline and array partition have given higher throughput, low resource utilization and execution time is reduced by more than 50% and also approximate model sparse matrices achieved 2× performance. Overall, our research focused on bridging the gap between approximate computing and hardware optimization, leveraging the unique capabilities of FPGAs to enhance the performance of SpMV operations in reconfigurable and embedded systems while addressing key challenges related to power, efficiency, and adaptability.




Edge computing is the future of HPC, so execution time and power consumption plays a significant role. Investigating hybrid precision approaches that combine low-precision arithmetic with higher precision where necessary could offer a balance between performance and accuracy. Developing dynamic adaptation mechanisms that adjust approximation levels based on runtime conditions and application requirements could enhance flexibility and adaptability. This could involve techniques such as runtime monitoring and feedback mechanisms to dynamically tune approximation parameters.

## REFERENCES




- [1] W. Mao *et al.*, "A Configurable Floating-Point Multiple-Precision Processing Element for HPC and AI Converged Computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 30, no. 2, pp. 213-226, Jan. 2022, doi: 10.1109/TVLSI.2021.3128435.
- [2] V. Sze, Yu-Hsin Chen, Tien-Ju Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295-2329, Dec. 2017, doi: 10.1109/JPROC.2017.2761740.
- [3] S. Song and J. Zambreno, "A floating-point accumulator for fpga-based high performance computing applications," in *2009 International Conference on Field-Programmable Technology*, Sydney, NSW, Australia, 2009, pp. 493-499, doi: 10.1109/FPT.2009.5377624.
- [4] L. Yavits and R. Ginosar, "Accelerator for sparse machine learning," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 21-24, Jan.-Jun. 2018, doi: 10.1109/LCA.2017.2714667.

- [5] B. Liu and D. Liu, "Towards high-bandwidth-utilization SpMV on FPGAs via partial vector duplication," in *2023 28th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Tokyo, Japan, 2023, pp. 33-38.
- [6] Min Li, Y. Ao, and C. Yang, "Adaptive SpMV/SpMSpV on GPUs for input vectors of varied sparsity," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 7, pp. 1842-1853, Jul. 2021, doi: 10.1109/TPDS.2020.3040150.
- [7] W. Yang, K. Li, Z. Mo and K. Li, "Performance optimization using partitioned SpMV on GPUs and multicore CPUs," *IEEE Transactions on Computers*, vol. 64, no. 9, pp. 2623-2636, Sep. 2015, doi: 10.1109/TC.2014.2366731.
- [8] M. Hosseinabady and J. L. Nunez-Yanez, "A streaming dataflow engine for sparse matrix-vector multiplication using high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 6, pp. 1272-1285, June 2020, doi: 10.1109/TCAD.2019.2912923.
- [9] M. Alle, A. Morvan, and S. Derrien, "Runtime dependency analysis for loop pipelining in high-level synthesis", in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, Austin, TX, USA, 2013, pp. 1-10, doi: 10.1145/2463209.2488796.
- [10] R. Garibotti, B. Reagen, Y. S. Shao, G. Y. Wei, and D. Brooks, "Assisting high-level synthesis improve SpMV benchmark through dynamic dependence analysis," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 10, pp. 1440-1444, Oct. 2018, doi: 10.1109/TCSII.2018.2860122.
- [11] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, Portland, OR, USA, 2009, pp. 1-11, doi: 10.1145/1654059.1654078.
- [12] P. Guo and C. Zhang, "Sparse matrix selection for CSR-based SpMV using deep learning," in *2019 IEEE 5th International Conference on Computer and Communications (ICCC)*, Chengdu, China, 2019, pp. 2097-2101, doi: 10.1109/ICCC47050.2019.9064309.
- [13] *Vitis High-Level Synthesis User Guide*, Xilinx Inc., San Jose, CA, USA, 2023, [Online]: Available: <https://docs.amd.com/r/2023.1-English/ug1399-vitis-hls>.
- [14] S. Kestur, J. D. Davis, and E. S. Chung, "Towards a universal FPGA matrix-vector multiplication architecture," *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, Toronto, ON, Canada, 2012, pp. 9-16, doi: 10.1109/FCCM.2012.12.
- [15] J. Naher, C. Gloster, S. S. Jadhav, and C. C. Doss, "Using machine learning to estimate utilization and throughput for OpenCL-based SpMV implementation on an FPGA," in *2020 SoutheastCon*, Raleigh, NC, USA, 2020, pp. 1-8, doi: 10.1109/SoutheastCon44009.2020.9249711.
- [16] E. S. Chung, J. C. Hoe, and K. Mai, "CoRAM: An in-fabric memory architecture for FPGA-based computing," in *FPGA '11: Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2011, pp. 97-106, doi: 10.1145/1950413.1950435.
- [17] Y. Umuroglu and M. Jahre, "An energy efficient column-major backend for FPGA SPMV accelerators," in *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, Seoul, Korea (South), 2014, pp. 432-439, doi: 10.1109/ICCD.2014.6974716.
- [18] S. Li *et al.*, "A data locality-aware design framework for reconfigurable sparse matrix-vector multiplication kernel," *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Austin, TX, USA, 2016, pp. 1-6, doi: 10.1145/2966986.2966987.
- [19] Y. Du, Y. Hu, Z. Zhou, and Z. Zhang, "High-performance sparse linear Algebra on HBM-equipped FPGAs Using HLS: a case study on SpMV," in *FPGA '22: Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2022, pp. 54-64, doi: 10.1145/3490422.3502368.
- [20] K. Lu *et al.*, "ReDESK: a reconfigurable dataflow engine for sparse kernels on heterogeneous platforms," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Westminster, CO, USA, 2019, pp. 1-8, doi: 10.1109/ICCAD45719.2019.8942089.
- [21] M. M. N. S. and K. S., "Bandwidth-efficeint sparse matrix multiplier architechure for deep neural networks on FPGA," in *2021 IEEE 34th International System-on-Chip Conference (SOCC)*, Las Vegas, NV, USA, 2021, pp. 7-12, doi: 10.1109/SOCC52499.2021.9739346.
- [22] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, "A high memory bandwidth FPGA accelerator for sparse matrix-vector multiplication," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, Boston, MA, USA, 2014, pp. 36-43, doi: 10.1109/FCCM.2014.23.
- [23] Y. Zhang, Y. H. Shalabi, R. Jain, K. K. Nagar, and J. D. Bakos, "FPGA vs. GPU for sparse matrix vector multiply," in *2009 International Conference on Field-Programmable Technology*, Sydney, NSW, Australia, 2009, pp. 255-262, doi: 10.1109/FPT.2009.5377620.
- [24] L. Zhuo, G. R. Morris, and V. K. Prasanna, "High-performance reduction circuits using deeply pipelined operators on FPGAs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 10, pp. 1377-1392, Oct. 2007, doi: 10.1109/TPDS.2007.1068.
- [25] T. Laan and A. L. Varbanescu, "Heterogeneous GPU and FPGA computing: a VexCL case study," *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Lyon, France, 2022, pp. 382-390, doi: 10.1109/IPDPSW55747.2022.00073.
- [26] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, "Harnessing gpu tensor cores for fast fp16 arithmetic to speed up mixed-precision iterative refinement solvers," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, Dallas, TX, USA, 2018, pp. 603-613, doi: 10.1109/SC.2018.00050.
- [27] K. Ahmad, H. Sundar, and M. Hall, "Data-driven mixed precision sparse matrix vector multiplication for gpus," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 4, pp. 1-24, 2019, doi: 10.1145/3371275.
- [28] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1-25, 2011, doi: 10.1145/2049662.204966.




**BIOGRAPHIES OF AUTHORS**

**Alden C. Shaji**    graduated in Electronics and Communication Engineering from Rajiv Gandhi Institute of Technology, Kottayam, Kerala and completed the Master's degree in VLSI design and embedded systems at Maulana Azad National Institute of Technology, Bhopal, Madhya Pradesh, in 2024. He is currently working in an MNC at Bangalore as Physical Design Engineer. He also published 2 conference papers related to SpMV optimization using HLS. His fields of interest are VLSI design with low power techniques, interconnects at lower tech nodes. He can be contacted at email: aldencshaji@gmail.com.



**Zainab Aizaz**    completed her Ph.D. from Maulana Azad National Institute of Technology, Bhopal, MP in 2024 in Digital VLSI design. Her subjects of interest are digital circuits, CPU design, and hardware accelerators for machine learning and artificial intelligence. She is a research fellow in AI hardware at University of Sussex. She can be contacted at email: aizazzainab@gmail.com.



**Kavita Khare**    received her B.Tech. degree in Electronics and Communication Engineering in 1989, and M.Tech. degree in Digital Communication Systems in 1993 and a Ph.D. degree in VLSI design in 2004. She has 30 years of teaching experience with more than 200 publications in reputed journals and conferences of IEEE, Springer, and Elsevier. She has guided 40 M.Tech. and 19 Ph.D. theses. She is a consistent reviewer in many tier 1 journals and is the Editor/Chief Advisory Board Member of some reputed International Journals. Currently, she is working as Professor, in the Department of Electronics and Communication Engineering at MANIT, Bhopal, India. She had Served as Head of the Department of Electronics and Communication Engineering for 3 consecutive years. She has two ongoing govt of India projects. She has three books published. She got 4 best paper awards. Her fields of interest are VLSI design of arithmetic circuits and ultra-low power VLSI. She can be contacted at email: kavita\_khare1@yahoo.co.in.