A Streaming Dataflow Engine for Sparse Matrix-Vector Multiplication Using High-Level Synthesis

Mohammad Hosseinabady and Jose Luis Nunez-Yanez

***Abstract*—Using high-level synthesis techniques, this paper proposes an adaptable high-performance streaming dataflow engine for sparse matrix dense vector multiplication (SpMV) suitable for embedded FPGAs. As the SpMV is a memory-bound algorithm, this engine combines the three concepts of *loop pipelining*, *dataflow graph*, and *data streaming* to utilize most of the memory bandwidth available to the FPGA. The main goal of this paper is to show that FPGAs can provide comparable performance for memory-bound applications to that of the corresponding CPUs and GPUs but with significantly less energy consumption. The experimental results indicate that the FPGA provides higher performance compared to that of embedded GPUs for small and medium-size matrices by an average factor of 3.25 whereas the embedded GPU is faster for larger size matrices by an average factor of 1.58. In addition, the FPGA implementation is more energy efficient for the range of considered matrices by an average factor of 8.9 compared to the embedded CPU and GPU. A case study based on adapting the proposed SpMV optimization to accelerate the support vector machine (SVM) algorithm, one of the successful classification techniques in the machine learning literature, justifies the benefits of utilizing the proposed FPGA-based SpMV compared to that of the embedded CPU and GPU. The experimental results show that the FPGA is faster by an average factor of 1.7 and consumes less energy by an average factor of 6.8 compared to the GPU.**

***Index Terms*—Edge computing, energy, FPGA, high-level synthesis (HLS), sparse-matrix-vector, support vector machine (SVM).**

I. Introduction

SPARSE matrix-vector multiplication (SpMV) is one of the common operations used in several areas, such as scientific optimization, circuit simulation, and machine learning [1]. Although SpMV has been known for a long time, recent progress in utilizing new architectures that consist of multicore CPUs, many-core GPUs, and FPGAs has led to a renewed interest in research activities toward optimizing its performance for the corresponding applications [2].

Manuscript received June 3, 2018; revised October 19, 2018 and January 17, 2019; accepted March 21, 2019. Date of publication April 23, 2019; date of current version May 22, 2020. This work was supported by the Engineering and Physical Sciences Research Council through ENEAC Project under Grant EP/N002539/1. This paper was recommended by Associate Editor W. Hung. *(Corresponding author: Mohammad Hosseinabady.)*

The authors are with the Department of Electrical and Electronic Engineering, University of Bristol, Bristol BS8 1UB, U.K. (e-mail: m.hosseinabady@bristol.ac.uk; j.l.nunez-yanez@bristol.ac.uk).

Digital Object Identifier 10.1109/TCAD.2019.2912923

Generally, cloud-based big-data computing and analysis are the main application framework for SpMV, especially in machine learning areas. However, with the challenges arising from centralized cloud-based computing, such as scalability and security, modern machine learning techniques are utilizing distributed architectures, relying on the edge computing framework. In this approach, the edge processors consume the locally generated data to train or refine a model. These data usually collected by a group of local sensors, hence, their size is limited. Recently, it has been shown that this scenario can provide a highly accurate model [3] by proposing CoCoA framework. An extension of the CoCaA called Mocha [3] focuses on the nascent federated machined learning scheme that has been empirically evaluated by academia and industry [4], corroborate the theoretical studies. This new approach has motivated us to focus on efficiently developing the SpMV on edge candidate devices considering moderate datasets (i.e., training data) and limited dimension sizes (i.e., features and training points).

Embedded FPGAs are potential candidates for accelerating computations on the edge thanks to their low energy consumption, fine-grained parallelism, and multiprecision capabilities that help efficient implementation of compute-intensive applications such as deep learning algorithms [5] on small devices. This has inspired us to optimize the SpMV targeting on embedded FPGAs.

Traditionally, FPGA accelerators are designed by hardware description languages (HDLs) that can potentially provide a high-performance implementation. However, the HDL-based design flow is tedious and time-consuming. In addition, the design is not easily adaptable (modifiable) to the versatile edge computing environment that includes a variety of algorithms with different configurations and complexity. To cope with these issues, we study the use of the high-level synthesis (HLS) that is increasingly popular for accelerating algorithms in embedded heterogeneous platforms. Studies have shown that HLS can provide high-performance and energy-efficient implementations with shortening time-to-market and addressing today's system complexity [6].

The SpMV is known as a memory-bound algorithm with irregular memory access operations and its implementation on FPGA should be optimized for maximum memory bandwidth utilization. This requires optimizing the number of computational hardware threads and load balancing to keep them busy. To achieve these optimization objectives, this paper proposes a streaming dataflow engine (SDE) architecture for SpMV running on an FPGA using HLS. To utilize the streaming data

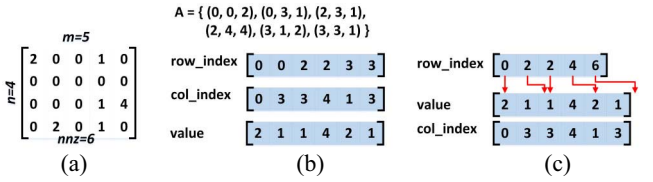


Fig. 1. Sparse matrix representation example ($n = 4$, $m = 5$, and $nnz = 6$). (a) Original matrix. (b) Coordinate format. (c) Compressed row format.

transfer capabilities provided by HLS tools via the burst data transfer protocol, this engine integrates the loop level and process level pipelining in the code enabling high memory access throughput by saturating the memory bandwidth.

Novelties and contributions of this paper are as follows.

1) Proposing a streaming dataflow (SDF) engine for SpMV, composed of multiple hardware threads that can be used as a template in an HLS environment.
2) Explaining the adaptivity of the proposed SDF for the versatile algorithm and configurations in the machine learning techniques on the edge computing paradigm.
3) Proposing a simple analytical model to understand the algorithm and platform bottlenecks and overheads.
4) Comparing the embedded FPGA implementation of the SpMV with multicore embedded CPU and many-core embedded GPU versions and studying in which cases the FPGA implementation is more efficient.
5) Optimizing the support vector machine (SVM) algorithm as a real application that uses SpMV as an operator by merging it with other operators needed in SVM.

The rest of this paper is organized as follows. Preliminary concepts, definitions, and requirements are explained in the next section. Section III reviews previous work and clarifies the motivations and contributions of this paper. The dataflow engine as the underlying structure of the proposed techniques is discussed in Section IV. Section V goes through the details of the proposed methodology. Section VI investigates the experimental results. Finally, Section VII concludes this paper.

II. Preliminaries

This section briefly explains concepts, techniques, and definitions that are considered throughout this paper.

A. Sparse Matrix

Most of the elements in a sparse matrix are zeros. Fig. 1(a) shows such a matrix with four rows (denoted by n) and five columns (represented by m) which has 14 zero elements and 6 nonzero elements (denoted by nnz). Operators involving these matrices (such as multiplications) usually suffer from low compute-per-byte ratio which makes their traditional implementations inefficient. Using new computation techniques with associated matrix representations to achieve high performance (HP) and reduce the memory utilization have been proposed for sparse matrix manipulations. Using coordinate list (COO) in the form of (*row index, column index,* and *value*) tuples for nonzero elements, as shown in Fig. 1(b), is one way to reduce the matrix memory footprint. However, one of the row and column vectors has redundancy that can be removed. This leads to the compressed sparse row (CSR) representation, shown in Fig. 1(c) which is the common representation for sparse matrices. Three vectors, named *value*, *col_index*, and *row_index*, represent the matrix. The *value* vector contains the nonzero elements in row-order and their corresponding column indices are saved in *col_index* vector, therefore, the number of nonzero elements, denoted by *nnz*, determines their sizes. The *row_index* elements are the indices of the *value* vector that contains the first element of each row in the original matrix. In other words, *row_index* elements point to the first element of each row in the *values* vector.

There are several different sparse matrix representations [7], especially used among HPC community and some of them rely heavily on the matrix sparsity pattern or the underlying computer architecture. These representations can be categorized into three main groups: 1) general format (GF); 2) architecture specific format (ASF); and 3) sparsity pattern aware format (SPF).

1) GF: Examples of the first group are CSR, COO, and CSC [7] that are more suitable for stream computing platforms, as the data is saved in a sequential orders. They are also suitable for computing architectures with high cache memory [7]. However, they may not show high-performance in GPUs which utilize the coalesce memory access scheme.

2) ASF: ELLPACK formats are among the second groups suitable for vector architectures and GPUs with coalesce memory access pattern [7].

3) SPF: Block-based CSR formats used for matrices in which zeros show a regular patterns such as block of zeros are repeating in the matrix. Diagonal formats (DIAs) is another example of this group that show the high-performance computation for diagonal matrices where nonzero elements are around the diagonal of the matrix. However, they are not suitable for representing general sparse matrices.

In summary, we have selected a sparse matrix format that:

1) represents a wide range of matrices;
2) is suitable for streaming data computing;
3) requires a light-weight preprocessing step.

Therefore, we have used the general CSR format that makes no assumption on the sparsity of the matrix and data are saved in a sequential order.

The SpMV algorithm based on the CSR representation is shown in the code snippet of Listing 1 which multiplies the sparse matrix A, represented by *value*, *col_index*, and *row_index* vectors, by a dense vector x and generates the output vector y, i.e., $y = Ax$. It consists of two nested *for* loops. The outer loop iterates through rows and the inner loop accesses each element in a row. The inner loop performs the dot-product of a row and the x vector by finding the proper element in the x vector with the index denoted by k at line 7 of Listing 1.

B. High-Level Synthesis

HLS tools, which transform a high-level description of a task usually written in C/C++ into the equivalent HDL code, have been used recently to efficiently implement many computational or memory-intensive algorithms, especially on FPGA platforms [8]. The main goal of current HLS tools is to provide parallel implementations of the concurrencies that are modeled by designers in the input code using compiler directives (such as pragmas) or following a specific coding style suggested by

```
void SpMV_Ref(int n, float *value, int *col_index, int *
      row_index, float *x, float *y) {
  int rowStart = 0, rowEnd = n;

  for (int i = rowStart; i < rowEnd; ++i) {
    float y0 = 0.0;
    for (int j=row_index[i]; j<row_index[i+1]; j++) {
      int k = col_index[j];
      y0 += value[j] * x[k];
    }
    y[i] = y0;
  }
}
```

Listing 1. SpMV operator.

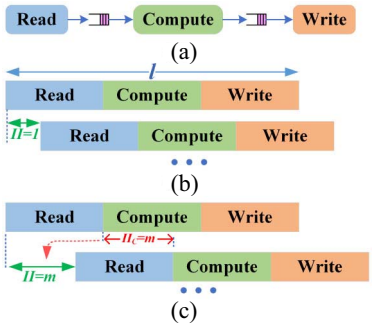


Fig. 2. Pipelined stream computing. (a) Stream computing dataflow. (b) Ideal pipelined streaming computation th $II = 1$. (C) Pipelined streaming computation with $II > 1$.

the tools [9]. These concurrency models can be categorized in two main groups: 1) statement level and 2) process level.

1) Statement Level: Independent expressions and assignments are automatically implemented in hardware running in parallel if there are enough resources in the underlying hardware. Extending this feature to iterative statements by completely or partially unrolling the iterations can be useful.

Current HLS tools leverage compile-time code analysis and optimization techniques to provide a static scheduling for the single statements and loops in the code. Hence, this requires resolving ambiguity and dependencies among variables, especially in iterative statements, at compile-time to achieve maximum hardware performance. The efficiency of loop pipelining depends heavily on the result of the static dependency and hazard analysis during which the compiler determines the fixed minimum loop iteration initiation intervals (*IIs*). The loop *II* is the minimum number of clock cycles before a loop iteration can start processing data by finding free resources. In other words, the *II* indicates the minimum interval between two consecutive loop iterations without encountering any hazards in the pipeline [9].

2) Process Level: A *process* is a stand-alone block of statements without any side-effect, including loops, with specific inputs and outputs. Data dependency between processes can be represented by a dataflow graph in general. FPGAs can provide HP running streaming dataflow processes. Streaming dataflow requires pipelining among processes and streaming data communication. Fig. 2(a) shows a dataflow of simple stream computing scheme which consists of three processes, *read*, *compute*, and *write*, communicating through buffers. Each of these processes can be implemented with a *for* loop in HLS. In the ideal case, which *IIs* of all loops are 1, this dataflow can be run at its highest performance as shown in Fig. 2(b) and it takes $N * II + l = N + l$ clock cycles to finish, where N is the loop iterations and l is a latency of one dataflow iteration. However, if the *II* of one process is higher than 1, it determines the *II* of the whole dataflow, consequently reducing the performance. For instance, if the *II* of *compute* process is d as shown in Fig. 2(c), then the *II* of the design would be d. In this case, the design takes $N * II + l = Nd + l$ clock cycles, which is d times slower than the ideal case if l is negligible compared to N. Therefore, the main goal of stream computing in HLS is to minimize the processes' *II* or compensate for its negative impacts. In the sequel, this paper will explain some of the techniques to design an optimum stream computation engine for the sparse matrix multiplication.

III. Previous Work

Sparse matrix operations are well-known problems in scientific computations and optimizations, especially in high-performance computing. Recently, a new wave of implementations is proposed [10] to support the application of these operations in the machine learning field. These algorithms mainly utilize multicore CPUs or many-core GPUs [11], [12].

Several studies have investigated the optimization of SpMV on hardware and FPGAs [13]–[15].

Most of these research activities are focusing on high-end FPGAs and big-data such as approaches proposed in [13], [14], and [16]. To get high-performance, they usually benefit from a complex data preprocessing, thanks to their powerful underlying computational hardware. In contrast to these approaches, our methodology targets embedded systems used in edge computing frameworks which process only parts of the big-data in a distributed computing scheme such as federated learning. In terms of the target sparse matrices, some work consider the sparsity pattern in a matrix and propose optimization techniques toward specific patterns such as the methods introduced in [14], whereas others make no assumptions about the sparsity structure of the matrix, such as [17]. Our method in this paper fits the second group.

Sadi *et al.* [16] proposed a streaming SpMV accelerator utilizing 3-D stacked high bandwidth memory (HBM) to overcome the memory wall issue. To consider large matrices whose x and y vectors do not fit into the on-chip memory, they propose matrix partitioning to fit the vector x into the on-chip memory. They also propose a two-step stream processing approach that is suitable for their architecture but would have high overhead in embedded FPGAs. In contrast to their approach, we use one step stream computing suitable for optimization on an embedded FPGA which does not benefit from the HBM technology.

Fowers *et al.* [18] introduced an FPGA-based SpMV architecture and a sparse matrix decoding to exploit the parallelism across matrix rows. They have assumed the availability of two separate DRAMs on the system which may not be available in most current embedded systems.

Designing an efficient floating point accumulator (i.e., multiplier and adder) to improve the performance of the SpMV is the main theme in [2] and [19]. Opposed to these approaches, our technique can be used with any accumulator design and only its latency is required (as explained in

Section V) to determine the number of hardware threads to achieve a high-performance.

In terms of the parallelism, some previous work exploit the row-based parallelism (such as [2], [17], and [19]) and pad each row with zeros to make their sizes a product of the parallelization factor k. Similar to these approaches, we utilize the row-based parallelism however, we clearly explain the minimum value of the zero-padding for a given accumulator. In addition, our proposed techniques exploit the parallelism in a row and between rows using two main techniques in HLS that are loop pipelining and unrolling.

Umuroglu and Jahre [15] utilized multiport memory interfaces to increase the memory bandwidth. Similarly, we utilize multiple ports as well as the wide-buses on each port and compare the results with embedded GPU and CPU.

Finally, in contrast to other work, we show that the proposed approach is easily adaptable to the environment of a real application since the SpMV hardware description can be extended with other compute intensive operators maintaining the performance level. These concepts are explained through case studies in Section VI.

IV. Proposed Streaming Dataflow Engine

This section explains the structure of the proposed SDE with its performance model.

Fig. 3 shows the structure of the proposed SDE for implementing the SpMV. The related subtasks are distributed into three main stages connecting through *stream mapping layer*s. Whereas each stage consists of a few processes performing computation or data transfer between the FPGA and the main memory, a stream mapping layer *reformats* and *distributes* the data received from its input buffers among its output buffers. In addition, it resolves the data-type mismatch problem between two consecutive stages. For example, if the input stage uses a 128-bit bus to transfer data to the FPGA while the compute stage uses 32-bit *float* data-type, then the stream data mapper should provide this transformation by mapping an input stream data into four output stream data utilizing proper buffers and pipelined concatenation or splitting assignments.

The *input stage*, as shown in Fig. 3, consists of a few processes (denoted by s) each of which is responsible for reading data from the main memory, through a dedicated port, using a burst data transfer scheme. Each process is implemented by a pipelined loop with a specific II which has a direct impact on the bandwidth utilization. The maximum input bandwidth utilization associated with a process is determined based on the number of bytes read per second which can be represented by (1) in which II_{in} is the initiation interval of the process reported by the HLS tool, b_{in} is the bus-width of the corresponding memory port, and f_{in} is the clock frequency of the memory interface.

The *compute stage* in Fig. 3 receives sequences of data from its predecessor stream data mapping layer and performs its task. This stage comprises of p processes each of which consists of t pipelined threads that can be run in parallel. The SpMV computation tasks are divided among these parallel processes. The maximum performance of a process in terms of the number of operations per second is denoted by (2) in which II_{comp} is the initiation interval of processes' loop, c_{comp} denotes the number of operations in each loop iteration and f_{comp} determines the frequency of the operations.

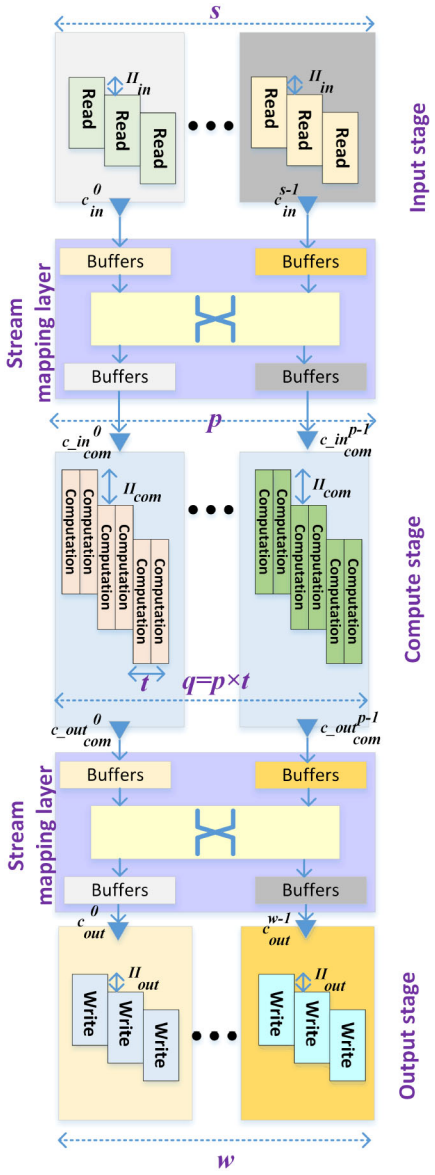


Fig. 3. SDE: structure.

The *output stage* consisting of w processes is responsible for writing the results to the main memory. It has a similar structure to the input stage.

One of the features of this structure is its adaptability to a specific application, such that it can be adapted to a given target application by adding operations to the code of each process as long as the added operator does not incur any loop dependency which results in no changes in II during the synthesis, in case of having enough resources on the FPGA

$$BWin_i^{\max} = (b_{in} \times f_{in})/(II_{in}) \quad (1)$$

$$Perfcomp_i^{\max} = t * \left(c_{\text{comp}} \times f_{\text{comp}}\right)/(II_{\text{comp}}) \quad (2)$$

$$BWout_i^{\max} = (b_{\text{out}} \times f_{\text{out}})/(II_{\text{out}}). \quad (3)$$

A. Performance Model

We propose a performance model to determine the contribution of algorithm and platform on the design efficiency.

This model simply clarifies the bottleneck of the whole design and can be used as a guideline to propose algorithmic or architectural optimization techniques. This model calculates the execution time of the design as shown in (4), where t^{alg} represents the time required by the algorithm which includes the ideal execution time denoted by $t^{\text{alg}}_{\text{ideal}}$ and the algorithm overhead represented by $t^{\text{alg}}_{\text{over}}$. Moreover, t^{plat} denotes the platform overhead which consists of hardware ($t^{\text{plat}}_{\text{hard}}$) and library ($t^{\text{plat}}_{\text{lib}}$) overheads. An example of $t^{\text{plat}}_{\text{lib}}$ is the high-latency of the floating-point operators that can have a negative impact in pipelined design. The hardware module initialization is an example of $t^{\text{plat}}_{\text{hard}}$

$$T = t^{\text{alg}} + t^{\text{plat}} = \left(t^{\text{alg}}_{\text{ideal}} + t^{\text{alg}}_{\text{over}}\right) + \left(t^{\text{plat}}_{\text{lib}} + t^{\text{plat}}_{\text{hard}}\right). \quad (4)$$

In addition, in the rest of this paper, we define M_{BRAM} and M_{BW} as the amount of FPGA internal memory (i.e., BRAM) and the main memory bandwidth used by the design.

V. SpMV: Proposed Methodology

Considering the dataflow engine of Fig. 3, this section explains the proposed streaming computation architecture in C language that can be synthesized by an HLS tool supporting the dataflow pipelining such as Xilinx Vivado-HLS. We also explain a sparse matrix representation suitable for the data stream communication.

The proposed SpMV implementation consists of three main tasks.

Task 1: Transferring the entire dense vector x into the FPGA memory (i.e., BRAM).

Task 2: Invoking the stream computation engine.

Task 3: Transferring the results from the FPGA to the main memory.

In the sequel, we explain how to utilize different optimization techniques to implement these three tasks.

A. Naïve Stream Computing

The code presented in Listing 1 receives the data in *values* and *col_index* vectors in a streaming fashion as their indices in the algorithm (i.e., j at lines 7 and 8) is ascending during the execution. The first step of stream computing in [16] implements this algorithm in ASIC with their own designed processing elements (PEs) which their details have not been explained. Although, this algorithm can be synthesized by available HLS tools, exploiting the parallelism in the code is not straightforward (in the context of FPGA and HLS) as the number of iterations of the inner loop at line 6 is known at runtime. Therefore, the code static analysis performed by an HLS tool cannot resolve the dependency among the statements; consequently, the outer loop cannot be pipelined or unrolled and should be executed sequentially which makes its stream computing inefficient due to the high iteration latency. To solve this problem, we modify the sparse matrix CSR representation as explained in the sequel.

The key point of the solution is making the length of the inner loop in Listing 1 predictable for each iteration of the outer loop. For this purpose, we modify the *row-index vector* in Fig. 1(b) such that each value represents the number of data element involved in the inner loop performing the dot-product

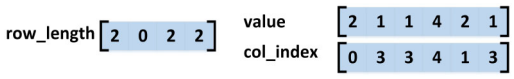


Fig. 4. MCSR format of the sparse matrix in Fig. 1(a).

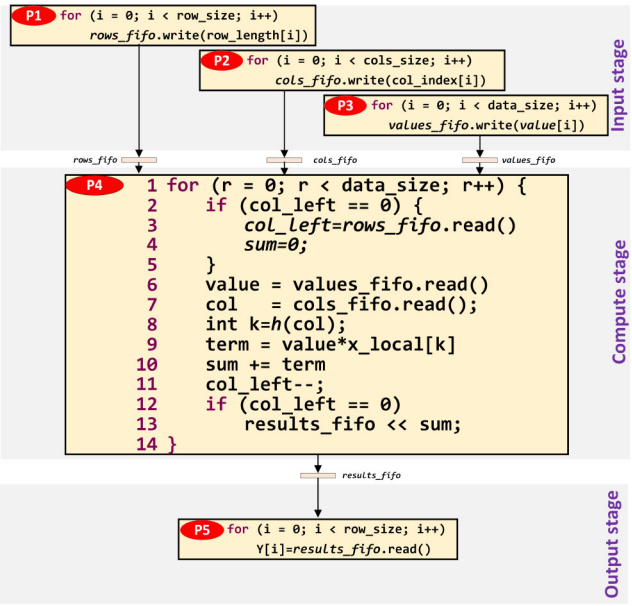


Fig. 5. Naïve stream computation pseudocode.

at line 8 in Listing 1. The new vector is called *row_length* as shown in Fig. 4 for the same matrix of Fig. 1(a). This representation is referred to as modified CSR (MCSR) throughout this paper. This technique is similar to the one presented in [20]. Note that, the *row_length* elements can be computed by differentiating two consecutive elements in the *row_index* vector. Therefore, its computation can be done in the hardware along with Task 1. The interested reader can refer to the open source code of this paper for more information [21]. The overheads associated with this technique are explained in Section VI-C.

The aforementioned three tasks of this implementation are as follows. Task 1 transfers the entire x into the FPGA memory using the burst data transfer which takes about m (i.e., its length) clock cycles. To implement Tasks 2 and 3 the SDE structure of Fig. 3 can be used. Considering this structure and the MCSR representation, Fig. 5 depicts the pseudocode of the naïve stream computation for the SpMV. The dense vector x transferred to the FPGA is denoted by *x_local* in this pseudocode.

The input stage consists of three processes, **P1**, **P2**, and **P3**, to read *row_length* and *col_index* indices as well as *value* vectors (as shown in Fig. 4) from the main memory in a streaming manner using the burst data transfer protocol. In this case $s = 3$, and as each process uses a dedicated memory port and the burst data transfer is used to read vectors, the minimum II_{in}, reported by the synthesis tool, for each process is 1. As the data-types in the three stages are the same, the streaming data mapping layer is very simple and only consists of buffers as shown between stages.

The code in process **P4** of Fig. 5 converts the nested loops in Listing 1 into a single loop that can easily be pipelined. The intra- and inter-loop iteration dependencies due

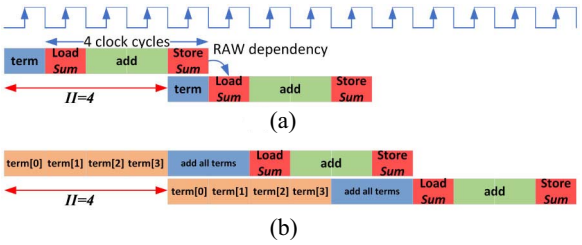


Fig. 6. (a) Naïve: RAW dependency hazard in the code of Fig. 5. (b) Fast stream: unrolling to compensate the $II = 4$.

to read-after-write potential hazards on *col_left* variable and the accumulation on the *sum* variable (at line 10 of the P4 process in Fig. 5) restricts the timing relation between the two consecutive loop iterations. This can cause an II higher than one. For example, in our experiment, we obtained the initiation interval of 4 (i.e., $II = 4$) by synthesizing the code for Xilinx Zynq-MPSoC. The main reason for this high II is the high latency of the accumulate operator with the *float* data-type. Fig. 6(a) shows the simplified pipeline timing diagram. This restricts the whole task throughput and performance. In addition, as just one process is considered without any loop unrolling, therefore $p = 1$ and $t = 1$, according to Fig. 3. The last stage (i.e., output stage) consists of one process which writes back the results into the main memory. In this case, $w = 1$ and $II_{\text{out}} = 1$. If there are enough memory ports to transfer data into the FPGA, then this process requires about $(nnz * II_{P4})$ clock cycles to complete. Taking the number of clock cycles for Task 1 into account, the entire SpMV takes about $(m + nnz * II_{P4})$ cycles to execute. Therefore, (5)–(7) show the performance model, BRAM usage, and memory bandwidth utilization, respectively. Note that in this case $t^{\text{alg}}_{\text{ideal}} = (m+nnz)/f$ and $t^{\text{plat}}_{\text{lib}} = (II_{P4}-1)nnz/f$ because it is caused by the floating point operation latency

$$T = (m + nnz)/f + (II_{P4} - 1)nnz/f + t^{\text{plat}}_{\text{hard}} \quad (5)$$

$$M_{\text{BRAM}} = m * \text{sizeof(DATATYPE)} \quad (6)$$

$$M_{MW} = (m + nnz)/T. \quad (7)$$

Our experimental results show that $t^{\text{plat}}_{\text{hard}}$ is negligible, hence, according to (5), the main bottleneck of this design is the high initiation interval of process **P4** in the compute stage. The next section explains how to cope with this issue.

B. Fast Stream Computing

One way to overcome the high initiation interval bottleneck of the **P4** process is processing multiple data in one iteration of the process's loop. Hence, the loop can be unrolled with a factor of II_{com}, i.e., $t = II_{\text{com}}$ in the SDE shown in Fig. 3. For example, according to our experiment, since $II_{\text{com}} = 4$ here, then it is enough to unroll the loop just four times. As such, the **P4** process can consume the data generated by the input stage processes without causing any wait state in processes of the input stage. Listing 2 shows the corresponding snippet code. The corresponding simplified timing diagram is shown in Fig. 6(b). Although the II is not changed, using four elements in each iteration increases the throughput by a factor of 4 which cancels the negative impact of $II = 4$. Note that this technique increases the number of utilized adder/multipliers by

```
1  for (r=0; r<data_size; r+= II_com) { //pipelined
2    if (col_left == 0) {
3      col_left=rows_fifo.read()
4      sum=0;
5    }
6    for (int i = 0; i < II_com; i++) {//unrolled
7      value = values_fifo.read();
8      col = col_fifo.read();
9      int k = h(col);
10     y[i] = y0;
11     term[i] = value * x[k];
12   }
13   DATA_TYPE sum_tmp=0;
14   for (int i = 0; i < II_com; i++) {//unrolled
15     sum_tmp += term[i];
16   }
17   sum += sum_tmp;
18   col_left-=II_com;
19   if (col_left == 0) {
20     results_fifo << sum;
21   }
22 }
```

Listing 2. Fast stream computing code.

a factor of II compared to the naive implementation. Utilizing multiple multipliers/adders has proposed by researchers who use the HDL design flow to cancel the high-latency of floating-point multipliers in a pipeline, such as scheme [18]. However, they have proposed their own fused accumulator which is not directly applicable to the context of HLS.

As each iteration of the *for* loop at line 1 of the snippet code in Listing 2 processes II_{com} data items of a row, the number of processed data element in each row should be a product of II_{com}. To satisfy this constraint some zero elements should be added to each row in the matrix representation in Fig. 4. This is referred to as zero-padding in the sequel of this paper. This zero-padding adds an overhead to the performance that will be examined later in Section VI for a set of matrices. The number of elements processed in each row is denoted by elements under process (eup) which is greater than *nnz*. Therefore, considering the number of Task 1 clock cycles, this implementation requires $(m + \text{eup})$ clock cycles to complete. Equation (8) shows the corresponding performance model, where $t^{\text{alg}}_{\text{ideal}} = (m + nnz)/f$ and $t^{\text{alg}}_{\text{over}} = (\text{eup} - nnz)/f$. Note, this algorithm address the platform library overhead and introduce the algorithm overhead. The experimental results show that this ends up to improve the performance

$$T = (m + nnz)/f + (\text{eup} - nnz)/f + t^{\text{plat}}_{\text{hard}} \quad (8)$$

$$M_{\text{BRAM}} = m * \text{sizeof(DATATYPE)} \quad (9)$$

$$M_{MW} = (m + equ)/T. \quad (10)$$

Note that, for implementing the zero-padding process, only the *row_length* vector should be modified and there is no need to modify the *values* and *col_indices* to contain zeros. The extra zeros can be inserted into the stream computing during the computation [21]. The complexity of the *row_length* modification algorithm is $O(n)$ and can be done in hardware (by a loop $II = 1$ [21]) along with Task 1 which does not have any impact on the total performance.

This algorithm can be modified to cover other sparse matrix formats. This illustrates that the proposed HLS technique is easily modifiable and adaptable to new situations in contrast

```
1  for (r=0; r<data_size; r+ = IIcom) { //pipelined
2    if (col_left == 0) {
3      col_left=rows_fifo.read()
4      sum=0;
5    }
6    for (int i = 0; i < IIcom; i++) {//unrolled
7      value = values_fifo.read();
8      col = col_fifo.read();
9      int k = h(col);
10     y[i] = y0;
11     term[i] = value * x[k];
12     if (col != r)
13       y_local[i][col] += value * x_local[r];
14   }
15   DATA_TYPE sum_tmp=0;
16   for (int i = 0; i < IIcom; i++) {//unrolled
17     sum_tmp += term[i];
18   }
19   sum += sum_tmp;
20   col_left-=IIcom;
21   if (col_left == 0) {
22     DATA_TYPE tmp=0;
23     for (int i = 0; i < II; i++)
24       tmp += y_local[i][r];
25     results_fifo << sum+tmp;
26   }
27 }
```

Listing 3. Fast stream computing code for sparse symmetric matrix.

to the traditional HDL approach. For example, it can be modified as Listing 3 to support symmetric sparse matrices in which only lower-left or upper-right triangular shape of data should be saved in the CSR format. In this algorithm, each matrix element (i.e., *value*) should modify two elements in the y output vector that has been shown in lines 12 and 13. This requires keeping the y vector in the FPGA. However, multiple access to the y elements in one iteration of the outer pipelined loop increases the II, mainly because of the shortage in the number of ports on y for parallel data access. To solve this problem, line 13 utilizes II_{com} copy of the y vector to save partial results. Lines 22–25 show how to merge these partial results to get the final y vector elements. The resource overheads of this modification are 61.1%, 15.2%, 11.2%, and 19.3% on DSP, FF, LUTRAM, and LUT of the FPGA. It also reduces the maximum size of the sparse matrix by a factor of 2. The performance improvement of this modification is 63% for a symmetric sparse matrix of size $40\,960 \times 40\,960$ with $nnz = 139\,264$. Note that further optimization of this modified algorithm is beyond the scope of this paper and requires a separate article.

Although Listing 2 provides a fast streaming computation for SpMV, it utilizes three memory ports (i.e., memory interconnects on the FPGA) that restricts its scalability to utilize more ports for performing parallel threads, mainly due to the limited number of memory ports available in embedded systems. The next section explains how to reduce the number of utilized ports and increase the number of computation processes.

C. Reduced-Port Stream Computing

To reduce the number of ports used by the design in Listing 2, the row and column indices can be combined and read through a single port. As shown in Fig. 7, the new

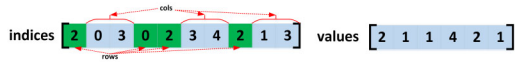


Fig. 7. Two-port streaming CSR.

format is defined by concatenating the number of elements in a row and the column indices of those elements. The new combined vector which is called *indices* has a length of $n + \text{eup}$, where n is the number of rows. The corresponding implementation requires $(m + (n + \text{eup}))$ clock cycles to complete. Therefore, (11) shows the performance model, where $t^{\text{alg}}_{\text{ideal}} = (m + nnz)/f$ and $t^{\text{alg}}_{\text{over}} = (n + \text{eup} - nnz)/f$

$$T = (m + nnz)/f + (n + \text{eup} - nnz)/f + t^{\text{plat}}_{\text{hard}} \quad (11)$$

$$M_{\text{BRAM}} = n * \text{sizeof(DATATYPE)} \quad (12)$$

$$M_{MW} = (m + n + equ)/T. \quad (13)$$

It should be noted that interleaving the row and column indices is a very simple process and does not include any computation and can be done during the process of receiving the locally generated data by the embedded system on the edge computing platform which results in no overhead on preprocessing data. This can be done by buffering each row's data at the edge platform before merging column and row indices. The goal of this optimization is to reduce the number of used ports and not to improve the performance, instead, it prepares the algorithm to utilize multiple ports to improve the performance. The next section clarifies the benefits of this approach.

D. Multiport Stream Computing

One way to increase the design throughput is utilizing multiple ports to transfer data from memory to the FPGA in parallel. If the embedded FPGA contains P memory ports each having B bits, and the number of bits of each element in *value* and *indices* vectors are g and h, respectively, then the number of computing processes, denoted by p in Fig. 3 satisfies (14). In this case, the rows in the input sparse matrix can be divided into p parts, each processed by a computing process, resulting in a maximum of p times speed-up. However, the maximum speed-up is limited by the part that contains more data elements. As each hardware thread calculates a part of the output vector y, Tasks 2 and 3, mentioned earlier in this section, execute sequentially. Therefore, the entire y is saved into the FPGA BRAM and transferred to the memory after Task 2 finishes

$$p \leq \frac{P * B}{g + h}. \quad (14)$$

Tasks 1 and 3 can also benefit from multiple port utilization. If we use k ports to transfer these vectors, then this implementation requires about $m/k + (n + \text{eup})/p + n/k$ clock cycles to complete. Therefore, (15) shows the performance model, where $t^{\text{alg}}_{\text{ideal}} = (n/k + (n)/p)/f$ and $t^{\text{alg}}_{\text{over}} = ((n + \text{eup} - nnz)/p + m/k)/f$

$$T = (m/k + (nnz)/p)/f + ((n + \text{eup} - nnz)/p + m/k)/f + t^{\text{plat}}_{\text{hard}} \quad (15)$$

$$M_{\text{BRAM}} = (n * p + m) * \text{sizeof(DATATYPE)} \quad (16)$$

$$M_{MW} = (n + m + \text{eup})/T. \quad (17)$$

Algorithm 1: Load Balancing Algorithm

Data: *no_part*: number of partition
Data: *eup*: number of total eup
Data: $R =< r_0, r_1, \ldots, r_{N-1} >$:
Result: $< P_0, P_1, P_{p-1} >$:
ideal_part_size $= eup/no\_part$;
$P_0 = r_0$
$j = 0$;
for $i \leftarrow 1$ **to** $N-1$ **do**
  if $|P_j| + |r_i| <$ *ideal_part_size* **then**
    $P_j = P_j + r_i$
  else
    if $j + 1 <$ *no_part* **then**
      $j++$;
    end
    $P_j = P_j + r_i$
  end
end

One concern with this multiprocess design is the unbalanced number of data elements divided among the parallel computing processes. The next section explains how to deal with this problem.

E. Load Balancing

To get the maximum performance by utilizing multiple computing processes, their workloads should be balanced through a proper matrix partitioning. This preprocessed matrix partitioning problem can be modeled using the 1-D chains-on-chains partitioning (CCP) problem [22], [23]. If all rows are denoted by the $R = <r_0, r_1, \ldots, r_{N-1}>$ chain (i.e., an ordered set) and the eup in each row denoted by $\text{EUP} = <\text{eup}_0, \text{eup}_1, \ldots, \text{eup}_{N-1}>$, then the partitioning problem is dividing R into p disjoint and nonempty sub-chains denoted by $R = <P_0, P_1, \ldots, P_{p-1}>$ in which $P_i = <r_j, r_{j+1}, \ldots, r_{j+k-1}>$. If the number of elements being processed in each partition is denoted by $\text{eup}_{P_i} = \sum_{t=j}^{t=k-1} \text{eup}_t$, then the objective of the partitioning is to minimize the largest value of eup_{P_i}, where $0 \leq i \leq p$, as all partitions are running in parallel and the optimum case is when the execution of the largest partition is minimized. In the ideal case, this minimum happens when all partitions have the same number of eup.

For the sake of simplicity, we use a greedy algorithm as shown in Algorithm 1 to solve this load balancing. The ideal case of eup_{P_i} is $\text{eup}_{equ} = (\sum_{i=0}^{N-1} \text{eup}_i)/p$ such that all partitions have the same number of eup. Starting at the first partition and first line, the algorithm adds lines to the partition until the difference between the partitions eup_{P_i} and eup_{equ} is decreasing. Note that the complexity of the load balancing process is $O(n)$, where n is the number of rows, and can be done on the hardware with a pipelined loop with $II = 1$ [21] or on the processor available in the embedded system and does not have a low overhead.

VI. Experimental Results

This section evaluates the proposed SpMV optimization techniques. For this purpose, first, several sparse matrices, selected as benchmarks, are used to study the impact of each optimization technique explained in Section V. Then, the performance results are compared with the performance of a multicore embedded CPU and two many-core embedded GPUs running the corresponding SpMV. Finally, two case studies are examined to explain the efficiency of the proposed methods in practice. Before delving into the detailed analysis and comparison, the next section explains the experimental set-up used for generating results.

A. Experimental Setup

To evaluate the proposed methods, we use three state-of-the-art embedded platforms available on the market. Xilinx ZCU102 evaluation board featuring the Zynq UltraScale XCZU9EG-2FFVB1156 FPGA [24], referred to as Zynq-MPSoC in the sequel, is used to run the proposed SpMV on its embedded FPGA. In addition, this platform is used to execute the multithreaded version of SpMV on its quad-core embedded processor. Nvidia Jetson TX1 and TX2 as two available commercial embedded GPU are used for running the corresponding SpMV on their embedded GPUs.

1) Zynq-MPSoC: The Xilinx Zynq Ultrascale+ MPSoC consists of two main parts: the multicore ARM processing system (PS) and the programmable logic (PL). This embedded system is supported by an external 64-bit DDR4 memory as the main memory for program code and data that is shared between the PL and PS through dedicated ports. Our design utilizes the four 128-bit HP ports on the Zynq-MPSoC to transfer data between the main memory and the PL. In this system, the FPGA and the CPU power domain supply voltages are provided by 23 voltage rails [24] among them VCCINT and VCC_PSINTFP supply the main powers for the FPGA and CPU in this paper, respectively. The corresponding voltage regulator, provided these voltage rails, supports the power management bus (PMBUS) and I2C protocol, so the power consumption can be monitored through software using the proper I2C APIs in Linux. We use the Xilinx SDSoC environment [25] which utilizes Xilinx Vivado-HLS and Vivado as the synthesis tool-chain to generate the bitstream file for the FPGA configuration and related drivers and software in Linux to invoke the accelerator.

2) TX1 and TX2: These embedded systems are based on NVIDIA Tegra X1 and X2 SoC [26], respectively. Whereas the TX1 consists of an NVIDIA Maxwell GPU with 256 CUDA cores, a Quad ARM A57, and 4 GB 64-bit memory, the TX2 encompasses the NVIDIA Pascal architecture with 256 CUDA cores, a Quad ARM A57, and 8 GB 128-bit memory. The cuSPARSE library [27], one of the most efficient industrial libraries provided by Nvidia for sparse matrix operation, is used in this paper for SpMV implementation on the embedded GPUs. In this system, the GPU and the CPU power domain supply voltages are provided by VDD_GPU and VDD_CPU voltage rails. The powers drawn from these rails are measured as the power consumption of each part. Note that these two platforms utilize programmable voltage regulators that can be monitored at runtime through the I2C protocol. Consequently, a software thread can read the power consumption of these modules at runtime using the I2C software library available in the Linux OS [21].

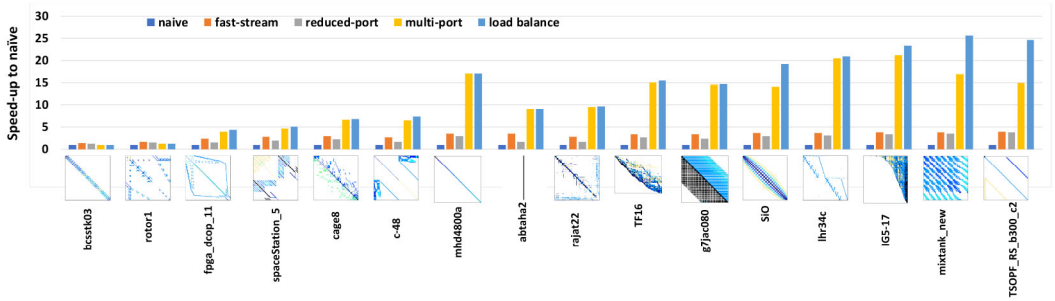


Fig. 8. SpMV: speed-up of each optimization compared to the naïve.

TABLE I
SPARSE MATRIX STATISTICS

Matrix name	n	m	nnz
bcsstk03	112	112	376
rotor1	100	100	708
fpga_dcop_11	1220	1220	5892
spaceStation_5	1020	1020	7895
cage8	1016	1016	11003
c-48	18354	18354	92217
mhd4800a	4800	4800	102252
abtaha2	37932	332	137228
rajat22	39900	39900	197264
TF16	15437	19321	216173
g7jac080	23672	23672	293976
SiO	33404	33404	675528
lhr34c	35152	35152	764014
IG5-17	30162	27944	1035008
mixtank_new	29960	29960	1995041
TSOPF_RS_b300_c2	28338	28338	2943887

TABLE II
eup FOR DIFFERENT II

Matrix name	II = 4 (overhead%)	II = 8 (overhead%)
bcsstk03	448 (16.1%)	896 (58.0%)
rotor1	832 (14.9%)	1064 (33.5%)
fpga_dcop_11	8144 (27.6%)	10400 (43.4%)
spaceStation_5	9472 (16.6%)	12320 (35.9%)
cage8	12440 (11.6%)	14144 (22.2%)
c-48	130304 (29.2%)	185584 (50.3%)
mhd4800a	110096 (7.1%)	117312 (12.8%)
abtaha2	151728 (9.6%)	303456 (54.8%)
rajat22	253108 (22.1%)	359944 (45.2%)
TF16	236728 (8.7%)	272056 (20.5%)
g7jac080	295052 (0.4%)	351296 (16.3%)
SiO	718880 (6.0%)	819584 (17.6%)
lhr34c	814728 (6.2%)	872416 (12.4%)
IG5-17	1080672 (4.2%)	1152040(10.2%)
mixtank_new	2061692 (3.2%)	2114464(5.6%)
TSOPF_RS_b300_c2	2985696 (1.4%)	3097096(4.9%)

Note that the power measurements in the experimental results do not include the *cool-down* power in which capacitors in the accelerator are discharged after the kernel execution has completed. However, for the sake of completeness and giving a value for the energy consumed during this period, we measured the power consumption after finishing a task. Our measurements show that for the GPU this period takes about 4 ms and consumes 11388 uJ energy, and for the FPGA it takes around 57 ms and consumes 6071 uJ. Note that studying and Optimization of the cool-down energy consumption requires separate research that is beyond the scope of this paper.

B. Benchmarking

A group of sparse matrices from the University of Florida sparse matrix collection [28] has been considered as our benchmarks in this section. According to the histogram of these matrices [28] the dimension and the number of nonzero elements for most of matrices are less than 10^5 and 10^6, respectively. Therefore, the benchmarks chosen here are among the most frequently occurring mid-range matrices to be processed by our underlying FPGA. This is in line with the motivations of this research, explained in Section I, in which an embedded system only processes a part of a big data. Table I shows the statistics of these sparse matrices. The first column is the name of the matrix as it appears in [28], the number of rows and columns are shown in Columns 2 and 3, respectively. The last column represents the number of nonzero elements in each matrix.

C. FPGA Accelerator Results

This section examines the performance of the proposed techniques on each sparse matrix benchmark and points out to the corresponding resource utilization and limitations as well as its scalability.

1) Performance: To evaluate the performance of the proposed methodology, we consider two different FPGA clock frequencies (100 and 200 MHz) and two single and double precision floating data type, denoted by SP and DP, respectively. The FPGA clock frequency has a direct impact on the II of the stream computing engine of Fig. 3. Increasing the clock frequency increases the latency of the floating-point operation used for the accumulation at line 17 of Listing 2. The synthesis results show $II = 4$ and $II = 8$ for the SP data type at design frequencies of 100 and 200 MHz, respectively. For the DP data type, the II change to 5 and 10 at design frequencies of 100 and 200 MHz, respectively. The different IIs result in a different number of eup after applying the zero-padding technique, which has been shown in Table II for the single precision.

As mentioned in Section V-A, transforming the CSR to MCSR can be done in the hardware along with transferring the vector x into FPGA. This design comes with hardware and energy overheads. The synthesized hardware shows 0.8%, 13.3%, and 2.1% overhead on *FF*, *LUTRAM*, and *LUT* in FPGA resource utilization, respectively. In addition, its energy overhead for *bcsstk03* and *TSOPF_RS_b300_c2* sparse matrices are 18% and 0.9%, respectively, corresponding to the smallest and largest matrix in our benchmark.

As mentioned in Section V-E the load balancing algorithm, that can be done on the CPU, has a very low overhead as its complexity is $O(n)$, where n is the number of rows. This overhead is 0.3% for *bcsstk03* matrix and 7.6% for *TSOPF_RS_b300_c2*, corresponding to the smallest and largest matrix in our benchmark.

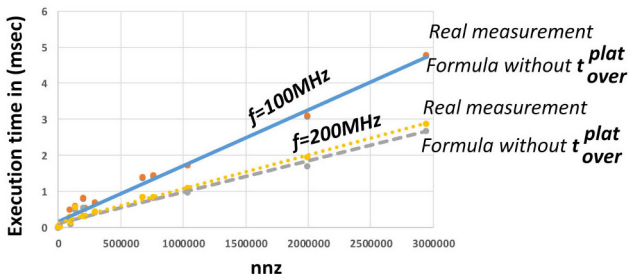


Fig. 9. SpMV: FPGA performance at different frequency for SP data type.

TABLE III
UTILIZED MEMORY BANDWIDTH (GB/S)

	$100MHz$		$200MHz$	
Matrix name	Formula	Empirical	Formula	Empirical
bcsstk03	5.1	0.5	7.0	0.7
rotor1	5.5	0.8	7.2	1.0
fpga_dcop_11	5.1	2.8	7.5	3.6
spaceStation_5	5.2	3.2	8.9	4.9
cage8	5.5	3.4	9.0	5.0
c-48	5.1	4.9	8.9	8.3
mhd4800a	5.8	5.2	10.4	8.2
abtaha2	5.6	5.4	6.5	5.4
rajat22	5.1	5.0	7.7	12.4
TF16	5.5	5.3	9.3	8.8
g7jac080	5.5	5.3	8.8	8.5
SiO	5.7	5.6	9.4	9.2
lhr34c	5.8	5.7	10.2	9.8
IG5-17	6.0	5.9	10.4	9.2
mixtank_new	6.1	6.0	11.1	9.6
TSOPF_RS_b300_c2	6.1	6.0	10.8	10.2

Fig. 8 compares the speed-up of the different levels of optimizations explained in Section V to the naïve version. According to this diagram, the fast-stream version speeds up the naïve version up to 3.91 times that is quite close to the upper bound of $II_{\text{com}} = 4$. Merging the two indices vectors in the reduced-port case abates this speed-up; however, it enables increasing the number of hardware processes in the multiport option that eventually increases the speed-up factor to 21.1. As can be seen from this diagram, the load balancing technique has a great impact on the large matrices with an unbalanced distribution of *nnz* elements, such as *mixtank_new*.

Fig. 9 shows the execution time of the proposed SpMV for the two different frequencies. For each frequency two diagrams are plotted, one based on the performance formula of (15) without the platform overhead (i.e., $t_{\text{over}}^{\text{plat}}$) and the other based on the real measurement. As can be seen, the platform overhead is almost zero for $f = 100$ MHz and it is negligible for $f = 200$ MHz. This shows the low overhead of using HLS for implementing the proposed algorithm.

Table III shows the memory bandwidth utilization for each sparse matrix for the two different design frequencies. The theoretical limit for using four 128-bit HP memory ports in Zynq MPSoC are $((128 \times 4)/8) \times 100$ MHz $= 6.4$ GB/s and $((128 \times 4)/8) \times 200$ MHz $= 12.8$ GB/s at design frequencies of 100 and 200 MHz, respectively. This table shows that the proposed methodology has managed to achieve up to 93.8% and 79.7% of these theoretical memory bandwidths, respectively.

Table IV Shows a brief comparison with three other SpMV design on FPGA. The first row shows the number of giga floating point operation per second (GFLOPS), the second row is the maximum memory bandwidth utilization. The maximum sparse matrix dimension handled in each case is shown in the third row and the last row shows the maximum design frequency. Note that [18] utilizes an Altera Startix V D5 FPGA with two DRAM supporting an aggregate memory bandwidth of 21.3 GB/s, so this is the reason for achieving a performance of 3.9 GFLOPS.

TABLE IV
COMPARISON WITH OTHER SPMV IMPLEMENTATION ON FPGA

	Our method	Ref.[17]	Ref.[18]
GFLOPS	2.5	< 2.5	3.9
Memory BW (GB/s)	10.2	14.1	–
Max. matrix dimension size	50000	30237	16000
Max. frequency	200	160	150

TABLE V
MAXIMUM MATRIX DIMENSIONS FOR THE ZYNQ-MPSOC FPGA

Resource	naïve	fast-stream	reduced-port	multi-port
MAX n	–	–	–	50000
MAX m	980000	980000	980000	50000

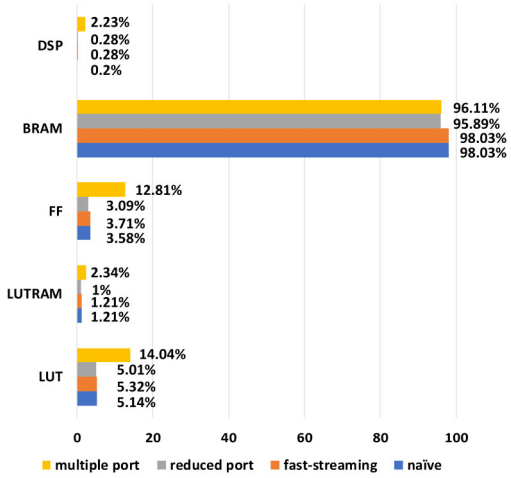


Fig. 10. SpMV: FPGA resource utilization.

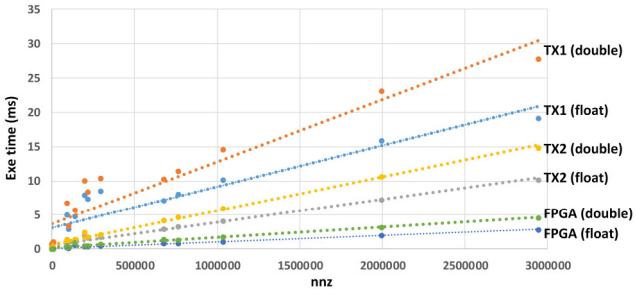


Fig. 11. Data transfer plus computation execution time: FPGA versus GPU.

2) Resource Utilization and Limitation: Fig. 10 shows the percentage of resource utilization for each optimization level explained in Section V.

Regarding the data sizes, as the proposed techniques keep the x or y dense vectors into the FPGA BRAM, the implementations should allocate almost all the BRAM to be able to process large matrices. This is the reason for high BRAM utilizations. Table V shows the maximum matrix dimension sizes that can be processed by each optimization technique using the Zynq-MPSoC. However, there is no any limitation on the number of nonzero elements in each matrix due to the streaming mechanism of reading these data. The first

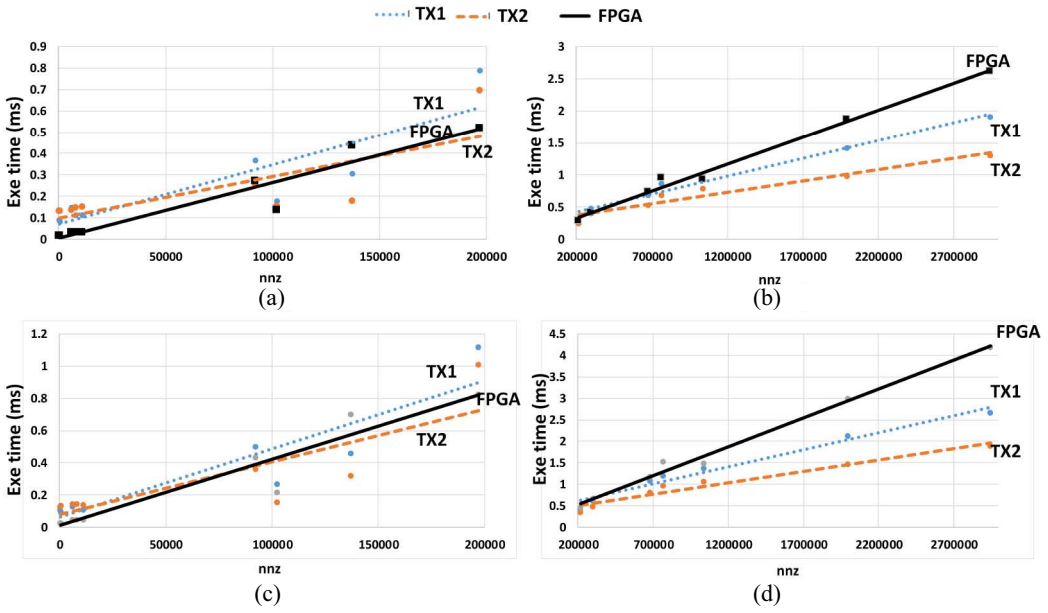


Fig. 12. Execution time for SP and DP in FPGA versus GPU (only computation) for two range of *nnz*. (a) Low *nnz*, SP data type. (b) High *nnz*, SP data type. (c) Low nnz, DP data type. (d) High nnz, DP data type.

TABLE VI
SPMV FPGA EXECUTION TIME (*msec*) COMPARISON WITH EMBEDDED CPU

benchmarks	Cortex-A53 1-core	Cortex-A53 2-core	Cortex-A53 4-core	FPGA	FPGA Speed-up
bcsstk03	0.008187	0.005817	0.005319	0.0168	0.32
rotor1	0.013165	0.010049	0.007779	0.0245	0.32
fpga_dcop_11	0.107124	0.068131	0.050811	0.0338	1.50
spaceStation_5	0.135262	0.117783	0.073115	0.0342	2.13
cage8	0.18639	0.122833	0.078148	0.035	2.23
c-48	1.7283	1.696485	1.233822	0.3021	4.08
mhd4800a	1.699987	0.994339	0.575088	0.1399	4.11
abtaha2	2.710926	1.827024	1.2892	0.586	2.2
rajat22	4.253527	2.782051	1.957403	0.3343	5.86
TF16	4.179026	2.897752	1.677043	0.3344	5.02
g7jac080	5.233087	3.298531	2.297791	0.4411	5.21
SiO	12.727249	8.903879	5.098102	0.8615	5.92
lhr34c	13.584376	8.411587	4.973272	0.8482	5.86
IG5-17	19.719413	12.602667	7.99485	1.111	7.20
mixtank_new	36.28779	24.690021	13.812478	1.9559	7.06
TSOPF_RS_b300_c2	48.882794	48.14978	30.80637	2.8723	10.7

three optimization techniques do not impose any restriction on the number of rows as they only keep the x dense vector into the BRAM. However, the last technique restricts both the number of rows and columns of the sparse matrices as it requires to save both the x and y vectors into the BRAM.

In terms of the scalability, the limiting factor of our design for higher performance is memory bandwidth and not the available hardware resources. Therefore, a larger chip will not help to improve performance but performance scalability will be obtained with several devices working in parallel to benefit from the aggregated memory bandwidth.

D. Comparison With Embedded CPU and GPU

This section compares the performance and energy consumption of our proposed FPGA design for SpMV with the corresponding ones running on embedded CPU and GPUs.

1) Embedded CPU: Table VI compares the performance of the proposed SpMV on the FPGA with the corresponding software implementation running on the quad-core Corex-A53

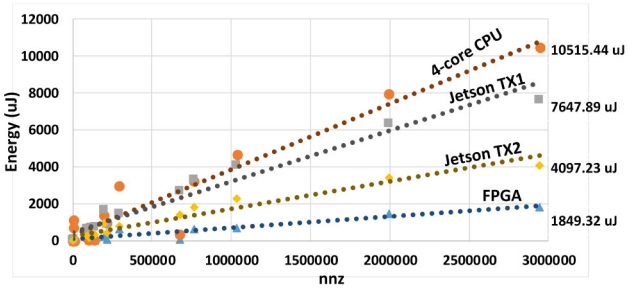


Fig. 13. SpMV: energy consumption comparison—only computation time.

available on the Zynq-MPSoC. The software implementation utilizes the OpenMP parallel programming model to employ multiple cores available on the processor. The first column of this table shows the benchmark names, Columns 2, 3, and 4 represent the execution time in *msec* after running the software implementation of the SpMV on one, two, and four cores, respectively. The execution time of the fastest FPGA implementation is presented in the fifth column. The amount of speed-up achievable by using the FPGA is shown in the last column. As a general rule, the FPGA implementation shows better performance compared to the quad-core CPU version when the *nnz* factor increases.

2) Embedded GPU: This section uses Nvidia Jetson-TX1&TX2 embedded GPUs to compare with our SpMV implementation on the FPGA. GPUs provide massive parallelisms which are suitable for implementing regular algorithms. In addition, utilizing different types of on-chip memory, such as scratch-pad, they overcome the high latency of accessing data in the off-chip global memory. Fig. 11 compares the SpMV execution time running on 4-Core CPUs, GPUs, and FPGA for the SP and DP data types. The GPU execution times include the data transfer and computation similar to that of the CPU and FPGA.

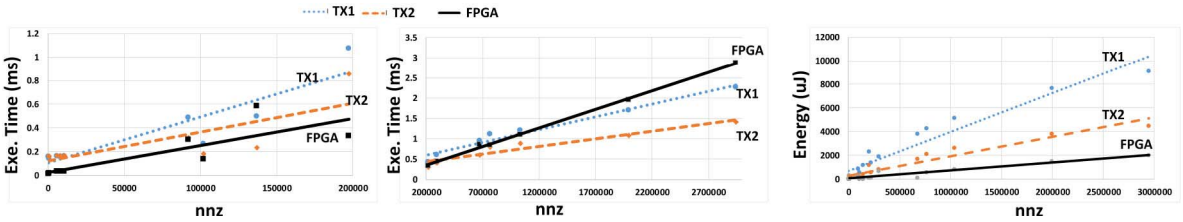


Fig. 14. SAXPY: FPGA and GPUs execution time comparison for two range of *nnz*. (a) Execution time for small *nnz*. (b) Execution time for large *nnz*. (c) Energy consumption.

As can be seen, FPGA shows better performance compared to GPUs and CPU if both data transfer and computation are considered in an application. However, it is also worth considering only the computation for the GPU implementation as in some iterative applications the data transfer performs only once. Considering this assumption, Fig. 12 compares the GPU and FPGA performance for two different range of matrix sizes. According to these diagrams, whereas the FPGA generally shows better performance for low *nnz* factor, GPU presents a higher performance for larger *nnz* factors.

Fig. 12 depicts two performance trends of running SpMV on embedded GPUs and FPGA for low and high values of *nnz* considering SP and DP data types. As can be seen, the embedded FPGA shows better performance with small and medium-size matrices and the embedded GPUs show better performance when the value of *nnz* is large. However, both FPGA and GPU provide higher performances than that of the embedded CPU. According to the measured data, the speed-up factor of the FPGA implementation to that of the GPU for small and medium-size matrices is 3.25 on average whereas the speed-up factor of the GPU for large size matrices is 1.58 on average.

The CPU usually benefits from their extensive cache memory to cope with memory intensive applications. Therefore, in cases that the data being processed fits the cache memory, the CPU can show better performance than other architectures. This justifies the better performance of CPU for smaller sparse matrices. On the other hand, GPU benefits from utilizing a large number of hardware threads and coalesce memory access, therefore they can show better performance in tasks that provide a large amount of data to keep all threads busy. This explains why GPU can show better performance for large sparse matrices.

3) Energy Consumption: This section compares the energy consumption of running SpMV on the embedded FPGA, the multicore embedded CPU, and the many-core embedded GPU. Fig. 13 compares the energy consumption of the SpMV running on the FPGA, CPU, and GPU in μJ. As can be seen the energy consumptions of the CPU and GPU implementations are much more than that of the FPGA. For example, for the last case as shown in Fig. 13, the FPGA consumes $4097.23/1849.32 = 2.21$ times less energy. According to the measured energy consumption for all benchmark matrices, on average the FPGA implementation consumes 8.9 times less energy compared to the GPU. This confirms the benefit of using FPGA instead of CPU and GPU in situations that energy consumption is an important factor, such as mobile edge devices.

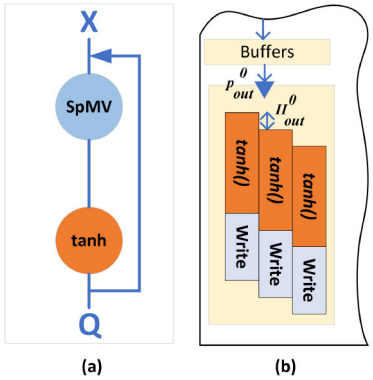


Fig. 15. Pipelined loop of Q computation. (a) Q computation graph. (b) Adapted SpMV.

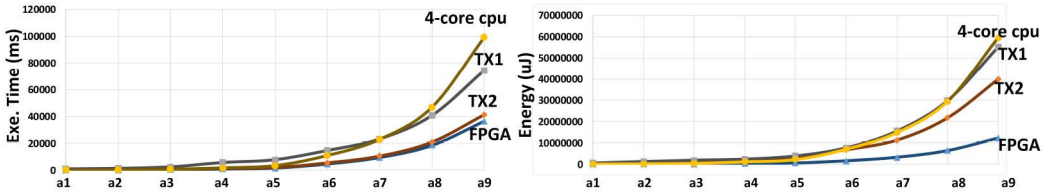
E. Case Study 1: SAXPY

In some applications and libraries, the SpMV kernel usually computes $y = y + \alpha Ax + \beta$. This kernel requires to read the previous y vector as an input, as it appears on the right-hand side. The proposed techniques can be easily modified to efficiently implement this kernel without adding overhead. For this purpose, the streaming data format presented in Fig. 7 can be modified to interleave the y vector elements with the *values* vector similar to interleaving the column and row indices. In this case, both vectors in Fig. 7 have the same length. Processes in the stream mapping layer (Fig. 3) can separate the y values into an FIFO what will be used later in the compute stage layer. For example, the y elements can be read and used an initialization value for the sum variable at line 4 of P4 process in Fig. 5. Fig. 14(a) and (b) compares the SAXPY execution time running on GPU and FPGA for two range of matrix sizes similar to Fig. 12. In addition, the amount of energy consumption is shown in Fig. 14(c).

F. Case Study 2: Support Vector Machine

This section puts the proposed SDE in Fig. 3 into practice to show its adaptability and efficiency in real applications. For this purpose, we have chosen SVM which is one of the successful classification algorithms [29]. We have modified LIBSVM [30], one of the state-of-the-art SVM implementations, to use embedded FPGA, GPU, and CPU implementations of the SpMV.

To make this paper self-contained, the C-SVM [30] as one type of the SVM is briefly explained here. C-SVM solves the optimization problem in (18) subject to (19), where $x_i \in R^n$, $i = 1, \ldots, l$ are the training vectors, $y \in R^l$, $y_i \in \{1, -1\}$



(a) (b)

Fig. 16. SVM results. (a) Execution time. (b) Energy consumption.

represents the class labels, C is the regularization parameter, w is the vector of model coefficients, b is a constant, and ξ_i denotes parameters for handling nonseparable data. The function ϕ is used to transform data from the input space to the feature space

$$\min_{w,b,\xi} \quad \frac{1}{2}w^T w + C\sum_{i=1}^{l}\xi_i \tag{18}$$

$$y_i\big(w^T\phi(x_i)+b\big) \geq 1-\xi_i,\ \xi_i \geq 0;\ i=1,\ldots,l. \tag{19}$$

The dual form of the problem which is more suitable for iterative optimization is shown in (20) and (21), where $e = [1,\ldots,1]^T$ and Q is an $l \times l$ matrix as shown in (22). The $K(x_i,x_j) = \phi(x_i)^T\phi(x_j)$ in (22) is the kernel function

$$\min_{\alpha} \quad \frac{1}{2}\alpha^T Q\alpha + e^T\alpha \tag{20}$$

$$\text{subject to} \quad y^T\alpha = 0,\ \ 0 \leq \alpha_i \leq C;\ \ i=1,\ldots,l \tag{21}$$

$$Q_{ij} = y_i y_j K\big(x_i,x_j\big). \tag{22}$$

After solving this problem, the model coefficients can be obtained using

$$w = \sum_{i=1}^{l} y_i\alpha_i\phi(x_i). \tag{23}$$

In our implementation, we have considered the sigmoid kernel function which can be represented as

$$\text{kernel:}\ \ K\big(x_i,x_j\big) = \tanh\big(\gamma x_i^T x_j + r\big). \tag{24}$$

The calculation of the Q matrix in (22), which its computation graph is depicted in Fig. 15(a), is compute-intensive part of this algorithm and it takes up to 80% of the total SVM execution time with the sigmoid kernel. Each $x_i^T x_j$ term in (24) is the result obtained by invoking the SpMV operation.

Other researchers also follow this mechanism to accelerate the SVM on multicore CPU, GPUs and ASIC design. Among them are [30] which utilizes multicore CPU and GPU. Nurvitadhi *et al.* [31] proposed an ASIC accelerator for SpMV to perform the Q computation in SVM. An ASIC accelerator for SpMV is proposed by [1] that has been used to speed-up the SVM execution. They have used a simulation approach to evaluate their designs.

Our proposed SpMV can be used to perform this operation. However, we would like to emphasize that the proposed SpMV can be easily adapted to the requirement of Q computation (which is invoking tanh function after SpMV) to improve the performance. Note that the hyperbolic tangent (i.e., tanh) function calculation is also time-consuming. As these operations apply to each element of the SpMV output individually, it can be merged into the SpMV pipeline in our proposed FPGA implementation while the initiation interval remains intact. For this purpose, only the processes in the output stage of Fig. 3 should be modified. For example, the assignment in the loop body of the **P4** process in Fig. 5 can be modified to $Y[i] = \tanh(\gamma * results\_fifo.read() + r)$ without any changed in the initiation interval of the corresponding loop. Fig. 15(b) shows the modified output stage corresponding to the SDE of Fig. 3. As the pipeline structure can hide the latency of the tanh function, the overall performance remains unchanged.

TABLE VII
SVM TRAINING DATA SET

	a1a	a2a	a3a	a4a	a5a	a6a	a7a	a8a	a9a
n	524	2265	3185	4782	6414	11220	16100	22696	32561
m	122	122	122	122	122	122	122	122	122
nnz	7248	31404	44162	66304	88939	155608	223304	314815	451592

To evaluate the impact of SpMV implementations used in the SVM, we have considered nine training data sets, taken from [30], with different sizes shown in Table VII. Fig. 16(a) and (b) compares the performance and energy consumption of the SVM training phase running on the embedded FPGA, GPU, and quad-core CPU. As can be seen, running the adapted SpMV on the FPGA slightly improves the performance and significantly reduces the energy consumption. Averaging all the measurements for the given datasets in Table VII, the FPGA-2 implementation of the SVM is 1.7 times faster consumes 6.8 times less energy compared to the embedded GPU version.

G. Challenges and Lessons

The challenges and take away lessons for using HLS as design flow are as follows.

1) Reusing the software-based algorithm in HLS is not straightforward and may need lots of modifications to allow synthesis tools exploit enough parallelism to provide the required performance. Thinking in stream computing can be helpful to cope with this issue.
2) Taking advantage of loop pipelining is the key technique to provide scalable design as it can provide parallelism with minimum resource utilization.
3) Utilizing all the memory ports available on the FPGA side can provide enough data for several pipelined stream computing threads on the FPGA to maximize performance.

VII. CONCLUSION

This paper has proposed an efficient SpMV to be used in an HLS approach and run on an embedded FPGA. The proposed

method is based on stream computing techniques in which computation and data transfer between the FPGA and the main memory are executing in a pipelined fashion. The experimental results indicate that the FPGA implementation of SpMV can be more performance-efficient for small and medium-size matrices compared to the GPU versions while the GPU can show better performance in large size matrices.

REFERENCES

[1] L. Yavits and R. Ginosar, "Accelerator for sparse machine learning," *IEEE Comput. Archit. Lett.*, vol. 17, no. 1, pp. 21–24, Jan./Jun. 2018.

[2] Y. Zhang, Y. H. Shalabi, R. Jain, K. K. Nagar, and J. D. Bakos, "FPGA vs. GPU for sparse matrix vector multiply," in *Proc. Int. Conf. Field Program. Technol.*, Dec. 2009, pp. 255–262.

[3] V. Smith, "System-aware optimization for machine learning at scale," Ph.D. dissertation, EECS Dept., Univers. California at Berkeley, Berkeley, CA, USA, Aug. 2017.

[4] J. Konečný, H. B. McMahan, D. Ramage, and P. Richtárik, "Federated optimization: Distributed machine learning for on-device intelligence," *CoRR*, vol. abs/1610.02527, Oct. 2016. [Online]. Available: http://arxiv.org/abs/1610.02527

[5] Y. Umuroglu *et al.*, "FINN: A framework for fast, scalable binarized neural network inference," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays (FPGA)*, 2017, pp. 65–74.

[6] R. Nane *et al.*, "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 10, pp. 1591–1604, Oct. 2016.

[7] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proc. Conf. High Perform. Comput. Netw. Stor. Anal.*, Portland, OR, USA, Nov. 2009, pp. 1–11.

[8] M. Hosseinabady and J. L. Nunez-Yanez, "A systematic approach to design and optimise streaming applications on FPGA using high-level synthesis," in *Proc. 27th Int. Conf. Field Program. Logic Appl. (FPL)*, Sep. 2017, pp. 1–4.

[9] *Vivado Design Suite User Guide High-Level Synthesis*, Xilinx Inc., San Jose, CA, USA, 2018.

[10] S. Sun, M. Monga, P. H. Jones, and J. Zambreno, "An I/O bandwidth-sensitive sparse matrix-vector multiplication engine on FPGAs," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 59, no. 1, pp. 113–123, Jan. 2012.

[11] Y. Liang *et al.*, "Scale-free sparse matrix-vector multiplication on many-core architectures," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 36, no. 12, pp. 2106–2119, Dec. 2017.

[12] W. T. Tang, W. J. Tan, R. S. M. Goh, S. J. Turner, and W. F. Wong, "A family of bit-representation-optimized formats for fast sparse matrix-vector multiplication on the GPU," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 9, pp. 2373–2385, Sep. 2015.

[13] E. S. Chung, J. C. Hoe, and K. Mai, "CoRAM: An in-fabric memory architecture for FPGA-based computing," in *Proc. 19th ACM/SIGDA Int. Symp. Field Program. Gate Arrays (FPGA)*, 2011, pp. 97–106.

[14] S. Li *et al.*, "A data locality-aware design framework for reconfigurable sparse matrix-vector multiplication kernel," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Austin, TX, USA, Nov. 2016, pp. 1–6.

[15] Y. Umuroglu and M. Jahre, "An energy efficient column-major backend for FPGA SPMV accelerators," in *Proc. IEEE 32nd Int. Conf. Comput. Design (ICCD)*, Oct. 2014, pp. 432–439.

[16] F. Sadi, L. Fileggi, and F. Franchetti, "Algorithm and hardware co-optimized solution for large SPMV problems," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, Sep. 2017, pp. 1–7.

[17] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on FPGAs," in *Proc. ACM/SIGDA 13th Int. Symp. Field Program. Gate Arrays (FPGA)*, 2005, pp. 63–74.

[18] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, "A high memory bandwidth FPGA accelerator for sparse matrix-vector multiplication," in *Proc. IEEE 22nd Annu. Int. Symp. Field Program. Custom Comput. Mach.*, Boston, MA, USA, May 2014, pp. 36–43.

[19] L. Zhuo, G. R. Morris, and V. K. Prasanna, "High-performance reduction circuits using deeply pipelined operators on FPGAs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 10, pp. 1377–1392, Oct. 2007.

[20] J. Sun, G. Peterson, and O. O. Storaasli, "Mapping sparse matrix-vector multiplication on FPGAs," in *Proc. Reconfig. Syst. Summer Inst. (RSSI)*, 2007.

[21] M. Hosseinabady. (2018). *Sparse Matrix Vector Multiplication on ZYNQ FPGA*. [Online]. Available: https://github.com/Hosseinabady/SDSoC-Benchmarks/tree/master/SpMV

[22] A. Pinar and C. Aykanat, "Fast optimal load balancing algorithms for 1D partitioning," *J. Parallel Distrib. Comput.*, vol. 64, no. 8, pp. 974–996, Aug. 2004.

[23] S. H. Bokhari, "Partitioning problems in parallel, pipeline, and distributed computing," *IEEE Trans. Comput.*, vol. C-37, no. 1, pp. 48–57, Jan. 1988.

[24] *Zynq UltraScale+ MPSoC Technical Reference Manual, UG1085 (V1.1)*, Xilinx Inc., San Jose, CA, USA, Mar. 2016.

[25] *SDSoC Environment User Guide, UG1027 (V2017.4)*, Xilinx, San Jose, CA, USA, 2018. [Online]. Available: https://www.xilinx.com/html_docs/xilinx2018_3/sdsoc_doc/zld1544032151946.html

[26] Nvidia. (2017). *Jetson Tx1–Tx2 Developer Kit Carrier Board Specification*. [Online]. Available: https://developer.nvidia.com/embedded/downloads

[27] *cuSPARSE Library*, Nvidia, Santa Clara, CA, USA, 2018. [Online]. Available: https://docs.nvidia.com/cuda/cusparse/index.html

[28] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, p. 1, Dec. 2011.

[29] V. Kecman, *Learning and Soft Computing: Support Vector Machines, Neural Networks, and Fuzzy Logic Models*. Cambridge, MA, USA: MIT Press, 2001.

[30] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, pp. 1–27, 2011. [Online]. Available: http://www.csie.ntu.edu.tw/ cjlin/libsvm

[31] E. Nurvitadhi, A. Mishra, and D. Marr, "A sparse matrix vector multiply accelerator for support vector machine," in *Proc. Int. Conf. Compilers Archit. Synth. Embedded Syst. (CASES)*, Amsterdam, The Netherlands, Oct. 2015, pp. 109–116.

Mohammad Hosseinabady received the B.S. degree in electrical engineering from the Sharif University of Technology, Sharif, Iran, in 1992, and the M.S. degree in electrical engineering and the Ph.D. degree in computer engineering from the University of Tehran, Tehran, Iran, in 1995 and 2006, respectively.

He is currently a Researcher with the University of Bristol, Bristol, U.K., researching on energy proportional computing based on the reconfigurable platforms. His current research interests include high-level reliability and testability, reconfigurable architectures, dynamic resource management, and runtime power management. He has published several papers in journals and conference proceedings in the above areas.

Jose Luis Nunez-Yanez received the Ph.D. degree in hardware-based parallel data compression from the University of Loughborough, Loughborough, U.K., with three patents awarded on the topic of high-speed parallel data compression.

He is a Senior Lecturer of digital systems with the University of Bristol, Bristol, U.K., and a member of the Microelectronics Group. His current research interests include design of reconfigurable architectures for signal processing with a focus on run-time adaptation, parallelism, and energy-efficiency. He is the PI in several industrial (e.g., TSB, DSTL, and ESA) and U.K. research council projects, including a CASE award by ARM in the field of run-time energy prediction and energy-aware scheduling. He is currently also involved as a Co-Investigator in the EU ENTRA and Energy-ICT FP7 projects.