Sparse Matrix-Dense Matrix Multiplication on Heterogeneous CPU+FPGA Embedded System

Mohammad Hosseinabady
Jose Nunez-Yanez
mohammad@hosseinabady.com
J.L.Nunez-Yanez@bristol.ac.uk
University of Bristol
Bristol

ABSTRACT

Embedded intelligence is becoming the primary driver for new applications in industry, healthcare, and automotive, to name a few. The main characteristics of these applications are high computational demand, real-time interaction with the environment, security, low power consumption, and local autonomy, among others. Addressing these diverse characteristics, researchers have proposed heterogeneous multicore embedded systems comprising CPUs, GPUs, FPGAs, and ASICs. Whereas each computing element provides a unique capability to enable one of the application characteristics, collaborating these processing cores in running an application to get the maximum performance is a crucial challenge. This paper considers the collaborative usage of a multicore CPU and an FPGA in a heterogeneous embedded system to improve the performance of sparse matrix operations, which have been essential techniques in reducing the inference complexity in machine learning techniques, especially deep convolutional neural networks. Experimental results show that the collaborative execution of sparse-matrix-dense-matrix multiplication on the Xilinx Zynq MPSoC, a heterogeneous CPU+FPGA embedded system, can improve the performance by a factor of up to 42% compared with just using the FPGA as an accelerator.

CCS CONCEPTS

• **Hardware → High-level and register-transfer level synthesis**; **Hardware accelerators**; • **Computer systems organization → Multicore architectures**; **Heterogeneous (hybrid) systems**; **Embedded systems**.

KEYWORDS

Sparse Matrix, Heterogeneous System, Embedded FPGA, High-level Synthesis

ACM Reference Format:
Mohammad Hosseinabady and Jose Nunez-Yanez. 2020. Sparse Matrix-Dense Matrix Multiplication on Heterogeneous CPU+FPGA Embedded System. In *11th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures / 9th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM'20), January 21, 2020, Bologna, Italy*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3381427.3381428

PARMA-DITAM'20, January 21, 2020, Bologna, Italy

ACM ISBN 978-1-4503-7545-0/20/01…$15.00
https://doi.org/10.1145/3381427.3381428

1 INTRODUCTION

Smart systems are on the rise, primarily as the underlying technology in many areas such as automotive, manufacturing industry, healthcare, smart cities, voice-activated personal assistants, robotics, online shopping, and retail marketing, to name a few. All these areas and their related applications rely on edge computing technology, which commonly requires an embedded system that delivers computational power, security, real-time processing, local automation, and low power consumption, among others. To address these diverse characteristics, researchers have proposed heterogeneous multicore embedded systems comprising computing elements with different architectures and micro-architectures such as multicore CPUs, manycore GPUs, FPGAs, and ASICs. Each of these processing elements can support one of the smart application characteristics. For example, compute-intensive applications can benefit from GPU/FPGA-based accelerators, and embedded FPGAs can reduce energy consumption compared to the embedded CPUs and GPUs. There has been a significant amount of research activities studying each computing element; however, their collaboration to execute a single task more efficiently is less investigated.

This paper explores the usage of a multicore embedded CPU and an embedded FPGA running a shared task concurrently. For this purpose, the given task is divided into several sub-tasks that are distributed between processing elements according to their computational power. Load balancing is the objective of this subtask distribution that leads to higher performance. As the task workload and the irregularity in the task algorithm has a significant impact on the task division technique, this paper considers sparse data operations which recently have been the main focus of several research activities [10, 12] in the area of machine learning thanks to the sparsity in the training data and the sparsification techniques in deep neural network (DNN) graphs [15] to reduce their memory and hardware footprint. One of the sparsification techniques in DNN is pruning that increases the weights with zero value during the training phase; this requires that sparse-based operators can cope with this dynamic behavior and do not assume a specific pattern for the non-zero weights. Therefore, researchers have studied and proposed techniques to accelerate sparse-based operators such as

sparse matrix-vector multiplication (SpMV) or sparse matrix-dense matrix multiplication (SpMDM) [14].

Sparse matrix-dense matrix multiplication is an operator that can model the DNN training and inference phase [9]. Therefore, accelerating this operator has a direct impact on the efficiency of a sparse DNN. Traditionally, most of the efficient proposed SpMV algorithms rely on considering a specific sparsity pattern in the input matrix, such as diagonal, triangular, block-based sparse data, or they consider a matrix preprocessing mechanism which results in a high preprocessing overhead. Although these algorithms have benefits in scenarios in which the sparse matrix is fixed and used many timed in an iterative algorithm, they cannot be used in the DNN training phase in which the sparsity has a dynamic behavior due to the pruning algorithms. Besides, most of the presented accelerators in the literature focus on one single architecture (such as FPGA, CPU, GPU, or ASIC) to accelerate the sparse operators. However, the ubiquitous heterogeneous computing architectures requires new techniques to accelerate such operators utilizing all the potential computational power in the given platforms. These challenges have motivated us to propose an SpMDM algorithm considering the dynamic sparsity pattern and a heterogeneous platform consisting of a multi-core CPU and an embedded FPGA.

The rest of this paper is organized as follows. The next section defines some definitions and points out the preliminary concepts that are used in the rest of this paper. Section 3 reviews the previous work and stands out the benefit of the proposed techniques compared to the others. The proposed techniques are explained in Section 4. Experimental results are discussed in Section 5. Finally, Section 6 concludes the paper.

2 DEFINITIONS AND PRELIMINARIES

This section explains the preliminary concepts and definitions that will be used throughout this paper. Sparse matrices are matrices that most of their elements are zeros. These matrices appear in many applications such as image processing, simulations, machine learning, and deep convolutional neural network, among others. Fig. 1-a shows an example of a 4×6 sparse matrix with seven non-zero elements (i.e., $nnz = 7$). Traditionally, a compressed format of this matrix is saved to reduce the corresponding memory footprint. The Compressed Sparse Row (CSR) format is the most common general representation which uses three vectors to keep all the information in the matrix [1]. Fig. 1-b shows the corresponding CSR format for the sparse matrix of Fig. 1-a. The three vectors, which define the matrix, are *rowPtr*, *values* and *cols*. The *values* vector (of size *nnz*) contains all the non-zero elements in the matrix in the row-major order. The corresponding column index for each element of the *values* vector is saved in the *cols* vector of size *nnz*. The elements of *rowPtr* vector of size $n + 1$ denotes the index of elements in the *values* vector that contain the first non-zero element in each row of the original matrix, and the last element points to the next index of the last item in the *values* vector.

Sparse-matrix dense-vector multiplication (SpMV) is one of the primary operators that is used in many applications. Listing 1 shows its corresponding code that reads the sparse matrix in its CSR format.

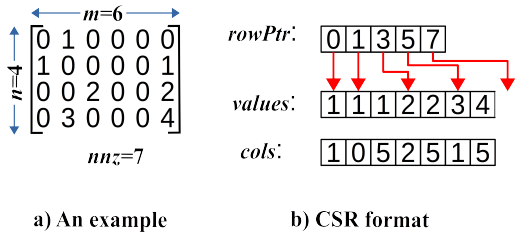

Figure 1: An example of a sparse matrix

```
void SpMV (float *values, int *cols, int *rowPtr, float *x, float *y, int n) {
  for (int i = 0; i < n; i+) {
    float y0 = 0.0;
    for (int j=rowPtr[i]; j<rowPtr[i+1]; j++){
      y0 += value[j] * x[cols[j]]; {
    y[i] = y0;
  }
}
```

Listing 1: SpMV pseudo code

The efficient implementation of this operation is studied by several researchers considering different types of computing platforms, including multicore CPUs, many-core GPUs, FPGA, and ASICs. This operation can be the basis of other tasks such as sparse-matrix dense-matrix multiplication (SpMDM), which is shown in Fig. 2. Note that, for the sake of simplicity, throughout this paper, we assume that the X matrix is saved in its transpose format. This assumption is true in the machine learning area, as feature vectors (i.e., X) are saved sequentially in the memory. Listing 2 shows the corresponding algorithm using the SpMV operator.

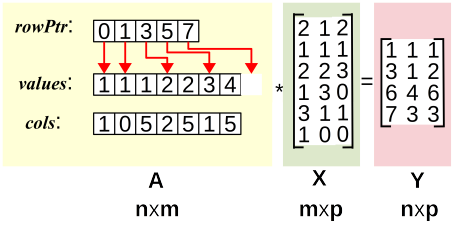

Figure 2: Sparse matrix dense matrix multiplication example

```
void SpMDM (float *values, int *cols, int *rowPtr,
            float **X, float **Y, int n, int m, int p) {
  for (int i = 0; i < p; i+)
    SpMV (values, cols, rowPtr, X[i], Y[i], n);
}
```

Listing 2: SpMDM pseudo code

3 PREVIOUS WORK

Sparse matrix-related operations are not a new subject. However, because of their importance in deep machine learning, recently, a new wave of research tries to find better implementations considering the constraints in new advanced computing platforms. In 2019, the MIT/IEEE/Amazon GraphChallenge.org [10] encourage the community to develop new solutions for sparse data and graph problems for DNN. There have been several implementations,

libraries, and techniques to implement sparse data operations in different computing architectures. A wide range of research activities considers multi-core CPUs [5, 11] or many-core GPUs [2, 6] as the underlying hardware platforms. Another group of research studies using the heterogeneous CPU+GPU architectures [13]. Recently, FPGAs have received particular attention for sparse operations [4, 8]. In contrast to these research activities, we focus on heterogeneous embedded CPU+FPGA computing platforms and study the potential of achieving a better performance.

4 PROPOSED TECHNIQUE

This section explains the optimization techniques for running the SpMDM algorithm on a heterogeneous FPGA+CPU platform. For this purpose, firstly, we propose an optimum FPGA implementation that utilizes the maximum achievable memory bandwidth. Secondly, we briefly explain the multi-core CPU implementation, using OpenMP platform. Finally, the optimized heterogeneous execution on FPGA+CPU is studied.

4.1 FPGA

The code in Listing 2 is the underlying idea of our proposed FPGA implementation. According to this code, the optimum implementation of SpMV has a direct impact on SpMDM performance. High-level synthesis techniques can be used directly to synthesis the SpMV code in Listing 1 for a target FPGA, such as Xilinx Zynq UltraScale+ MPSoC. However, the resulted performance is limited mainly due to the dynamic behavior of the inner *for*-loop index at Line 4 of Listing 1, which results in a high initiation interval (*II*) for the pipelined implementation. To address this issue, we have used a streaming computation similar to the technique in [8]. The SpMV code in Listing 1 sequentially reads the data elements in two *values* and *cols* vectors. Therefore, the stream computing scheme can be used to implement this algorithm on an FPGA. To make the inner *for*-loop at Line 4 of Listing 1 more predictable to reduce the pipeline initiation interval in the HLS implementation, we use the number of non-zero elements in each row instead of pointers to the first element of each row. Similar to the technique in [8], we consider a few zero elements in each row to make its size production of the *II* of the pipelined loop. This *zero-padding* technique can help us to unroll the pipelined loop with a factor of *II* to improve the performance and reduce the overall initiation interval to one.

Fig. 3 shows the architecture of this streaming implementation. It consists of three pipelined modules denoted by *M1*, *M2*, and *M3*. The initiation intervals of these modules are one. The *M1* module reads the *rowPtr* vector and calculates the number of non-zero elements in each row by differentiating two adjacent elements. The results are saved into the *rowSize* vector. This module also modifies these values by considering the zero-padding technique and save them in *rowSizeModified* vector. Module *M2* reads the *x* vector into a local array to be used by *M3*. Module *M3* implements the SpMV vector using a pipelined streaming fashion. Listing 3 shows the corresponding code for this module. *M1* and *M2* modules can be run in parallel as there are no dependencies between them. As these two modules are pipelined with *II=1*, they take *n* and *m* clock cycles to finish, respectively, where *n* is the number of elements in *rowPtr* and *m* in the number of elements in *x*. The *M3* module starts after finishing the first two modules, as it requires their results. This module takes *new_nnz* clock cycles, which is the length of values vector after zero-padding.

4.2 Multiple hardware threads

New advanced FPGAs, such as Xilinx Zynq MPSoC, support multiple memory interfaces that enables parallel memory access mechanisms for multiple computational hardware threads. This multi-thread hardware has motivated us to divide rows in a sparse matrix into different regions. Then, a few parallel hardware threads can run the algorithm in Listing 3 on each sparse region. Fig. 4 shows the architecture of this scenario, in which four hardware threads read sparse matrix regions in parallel and perform the SpMV algorithm. Apart from the three modules *M1*, *M2*, and *M3* shown in Fig. 3, the multi-thread implementation requires a load balancer to distribute data evenly among hardware threads. We have combined the load balancer into the *M1* module, while its initiation interval is still one. Fig. 5 shows the hardware structure of an iteration of the pipelined loop in *M'1*.

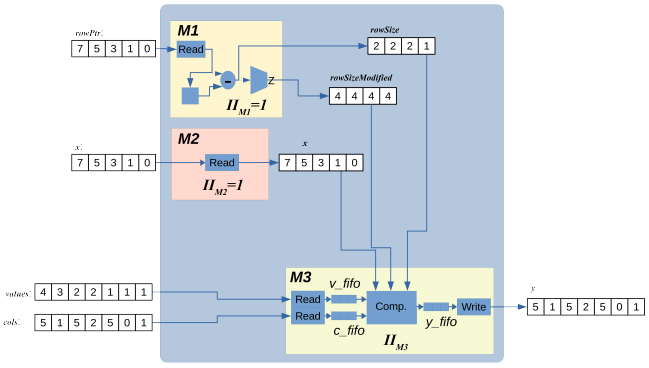

Figure 3: Streaming SpMV architecture

```
1  u32 row_size_remains = 0;
2  for (u32 i = 0; i < new_nnz; i+=IIC {
3  #pragma HLS pipeline
4    if (row_size_tmp == 0) {
5      row_size_tmp = rowSizeModified[j];
6      row_size_remains = 0;
7      y_tmp = 0;
8      row_counter = rowSize [j++];
9    }
10   DATA_TYPE y_local = 0;
11   for (u32 p = 0; p < II; p++) {
12     row_size_remains++;
13     if (row_size_remains > row_counter) {
14       y_local += 0;
15     } else {
16       DATA_TYPE v = values_fifo.read();
17       u32 ci = col_indices_fifo.read ();
18       y_local += v*x_local[ci];
19     }
20   }
21   y_tmp += y_local;
22   row_size_tmp-=II;
23   if (row_size_tmp == 0) {
24     y_fifo << y_tmp;
25   }
26 }
```

Listing 3: Stream computing code [8]

The main difference between our proposed technique for SpMV and the one in [8] is that in this approach, we do not need any data preparation before calling the accelerator and do not need to save extra zeros from zero-padding in the memory. In addition, in contrast to [8], the load balancer module is also in the hardware that is implemented by a pipelined loop with *II=1*. The interested reader can refer to the open-source code of this research [7].

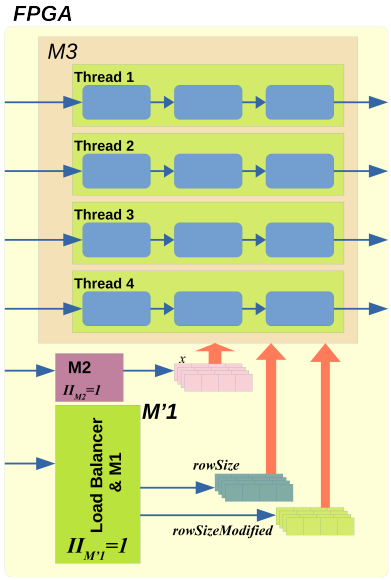

Figure 4: Multithread streaming SpMV

We use the multithread streaming SpMV in Fig. 4 to implement the SpMDM code in Listing 1. Algorithm 1 shows this implementation, where *rowPtr*,*values*, and *cols* represent the sparse matrix; the constant *THR* defines the number of hardware threads; n denotes the number of rows in the sparse matrix; m and p represent the vector size and the number of vectors in dense matrix, respectively. Line 3 in Algorithm 1 implements the *M'1* module of Fig. 4 using a pipelined loop with *II=1*. Hence it takes n clock cycles to finish. The rest of the algorithm consists of two nested loops. The outer loop iterates over the number of vectors (i.e., p) and execute sequentially. Line 4 performs the *M2* module of Fig. 4, reading a vector from matrix X using a pipelined loop with *II=1*, which takes m clock cycles to complete. Then, the inner loop, at Line 5, performs the *M2* module of Fig. 4, which is the multi-threaded SpMV. As each inner loop iteration implemented by three pipelined loops with *II=1*, it takes $max(nnz[0], nnz[1], ..., nnz[THR-1])$ clock cycles, which represents the slowest thread.

According to the above discussion, Equ. 1 represents the performance model of SpMDM in the FPGA, where α is a constant. Running the SpMDM on the Xilinx ZCU102 board, Fig. 6 depicts this equation empirically.

$$t_{fpga} = \alpha(n + p(m + max(nnz_j))) \quad (1)$$

Algorithm 1: SpMDM Multi-thread FPGA algorithm

```
   Data: rowPtr,values,cols,X,n,m,p
   Result: y
1  int nnz[THR];
2  nnz[] = loadBalancing(rowPtr) // n cycles;
3  SERIAL:for i ← 1 to p do
4  |   X= readX(i,x) // m cycles ;
5  |   PARALLEL:for i ← 1 to p do
6  |   |   SpMV(nnz[j]) // nnz[j] cycles
7  |   end
8  end
```

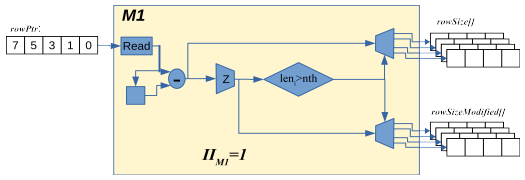

Figure 5: Load balancer and M1 module in Fig. 4

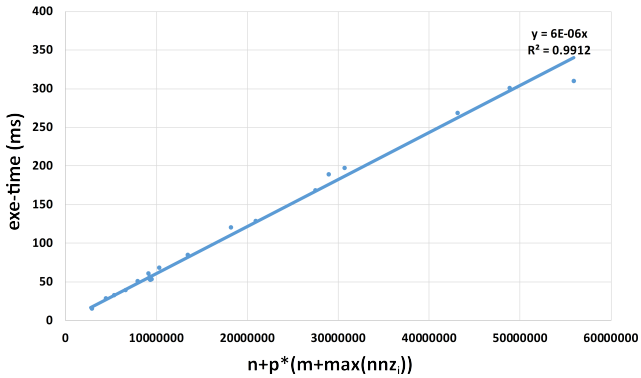

Figure 6: SpMDM execution time on ZynqMPSoC FPGA (p=100)

4.3 Multi-Core CPU

Listing 4 shows the OpenMP implementation of the SpMDM on a multicore CPU.

```
1  #pragma omp parallel for private(k, j, v)
2  for (v = 0; v < p; v++) {
3    for (i = 0; i < n; i++) {
4      y[v*m+i] = 0;
5      for (k = rowPtr[i]; k < rowPtr[i + 1]; k++) {
6        j = colIndices[k];
7        tmp = values[k] * x[v*m+j];
8        y[v*m+i] += tmp;
9      }
10   }
11 }
```

Listing 4: Multicore CPU implementation for SpMDM (p=100)

Considering 100 vectors in the dense matrix, Fig. 7 depicts the execution time of the OpenMP implantation on Xilinx Zynq MPSoC

quad-core ARM Cortex-A53 for different sparse matrix sizes. The horizontal axis shows the sum of row size (i.e., n) and the number of non-zero elements (i.e., nnz) in the sparse matrix. The vertical axis represents the execution time in ms. This diagram shows a linear relationship between the performance and $(n + nnz)$. Equ. 2 formulate this behavior, where β is a constant. Note that as we consider moderate matrix size in this paper, mainly because of the assumption of considering an embedded system as the underlying hardware, the feature size (i.e., m) does not have an impact on this formulation. The main reason for this phenomenon is that the CPU microarchitecture can hide its overhead by fetching the vector data into the cache memory.

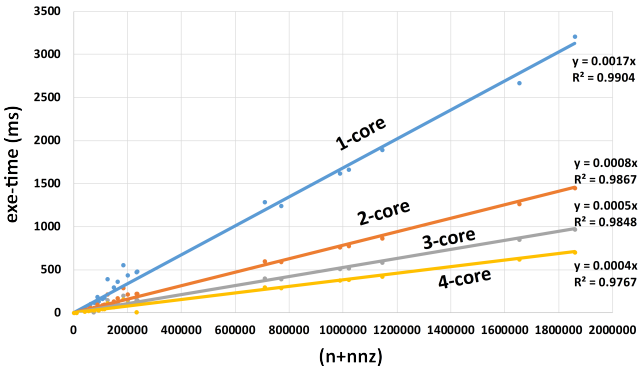

Figure 7: SpMDM execution time on quad-core A53 CPU

$$t_{cpu} = \beta p(n + nnz) \tag{2}$$

4.4 FPGA+CPU Heterogeneous

To enable the CPU and FPGA collaboration in executing the SpMDM operation, we should partition the input matrices and distribute parts between the two processing elements. Hence, a proper memory space should save the data, considering each architecture constraints. Whereas an efficient FPGA memory access requires a contiguous area in the memory, the CPU expects a cacheable memory. Therefore, a contiguous cacheable memory should save the matrices. This constraint is accessible in the new advanced FPGA-based embedded system such as Xilinx ZynqMPSoC through *sds_alloc* function available in the Xilinx SDSoC design environment. Two data partitioning mechanisms are possible:

- **Sparse matrix partitioning** (SMP): In this case, rows in the sparse matrix are divided between the CPU and FPGA. Note that according to the FPGA performance presented by Equ. 1, partitioning an input matrix into two parts and calling the FPGA implementation twice takes more time than calling the FPGA once over the whole data. Let us assume that we divide the sparse matrix into two parts: $p0 = (a_0, ..., a_{(q-1)})$ and $p1 = (a_q, ..., a_{(n-1)})$ and represents their execution times by t_{p0} and t_{p1}, respectively. Then there is no guarantee than $t_{p0} + t_{p1} \leq t$, where t is the execution time of the FPGA over the whole sparse matrix. However, this phenomenon is negligible in the CPU implementation. Therefore, we partition the sparse matrix such that the FPGA only receives one region. If the first and the last index define each group of consecutive rows, then expressions 3 to 5 show an example of the sparse matrix partitioning. In a possible scenario, FPGA can accept the $p1$ partition, and $p0$ and $p2$ partitions are assigned to the CPU. Note that each $p0$ and $p2$ parttion can be empty.

$$p0 = (a_0, ..., a_(l-1)) \tag{3}$$
$$p1 = (a_l, ..., a_(u-1)) \tag{4}$$
$$p2 = (a_u, ..., a_(n-1)) \tag{5}$$

- **Dense matrix partitioning** (DMP): The FPGA performance limitation that can accept only one sparse matrix partition can restrict the number of tasks assigned to the CPU. Therefore, we also consider the dense matrix partitioning. In this approach, the dense matrix is divided into two parts such that one part is only assigned to the CPU, and the other is processed, collaboratively, by the CPU and FPGA, considering the sparse matrix partitioning technique.

Fig. 8 depicts our proposed data partitioning scheme. Three parameters denoted by l, u, and z define the proposed partitioning. Whereas the l and u determines the partition areas in the sparse matrix, the z parameter determines the dense matrix partitioning.

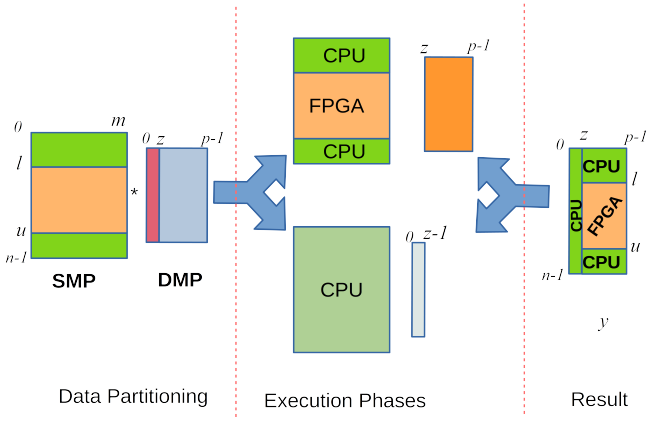

Figure 8: Sparse and dense matrix partitioning

5 EXPERIMENTAL RESULTS

To evaluate the proposed methodologies, we selected the Xilinx ZCU102 evaluation board that features Zynq Ultrascale+ MPSoC as the FPGA-based embedded system. In addition, we have used the Xilinx SDSoC toolset to synthesize the FPGA accelerator described in the C language [7].

Table 1 shows the statistics of seven sparse matrices [3] that are used in this section. The first column is the name of matrices. The second, third, and fourth columns are the number of rows, columns, and non-zero elements, respectively. The last column shows the length of the modified nnz after zero-padding. In the following experiments, we have considered 100 vectors in the dense matrix.

Three cores of the quad-core Cortex A53 CPU in the Zynq are used for running the OpenMP software version of the SpMDM, and

Table 1: Sparse matrix statistics

Matrix name	row	col	nnz	nnz-new
abtaha2	37932	332	137228	303456
bayer10	13436	13436	71509	138368
brack2	62631	62631	366559	867600
mixtank_new	29960	29960	1990385	2114272
c-45	13206	13206	93829	165088
opt1	15449	15449	973052	1028224
exdata_1	6001	6001	1137751	1168512

Table 2: CPU and FPGA execution time (ms)

Matrix name	3-core CPU	FPGA
abtaha2	112.7	50.93
bayer10	47.43	33.36
brack2	257.64	313.46
mixtank_new	1192.65	345.56
c-45	58.38	46.61
opt1	514.138	169.29
exdata_1	582.51	198.15

Table 3: Heterogenous CPU+FPGA execution

Matrix name	z (%)	l (%)	u (%)	exe-time(ms)	speed-up (%)
abtaha2	10	20	90	34.84	31.59
bayer10	35	20	100	22.55	32.40
brack2	55	10	100	170.5	33.82
mixtank_new	24	0	100	273	20.99
c-45	25	64	100	26.71	42.69
opt1	15	20	100	155.21	8.317
exdata_1	27	22	50	163.7	17.38

the fourth core is allocated to schedule subtasks onto the CPU and FPGA. Table 2 shows the execution time in *ms* of the SpMDM on the 3-core CPU and FPGA, separately. The FPGA implementation utilizes four hardware threads running at the frequency of 200MHz. It also uses two 128-bit HPC cacheable memory ports to transfer 32-bit floating-point data type between the main memory and FPGA hardware threads. Table 3 shows the data partitioning parameters and the execution time of the heterogeneous CPU+FPGA implementation of the SpMDM algorithm. The partitioning parameters *z*, *l*, and *u*, which are found manually in this paper, are defined base on the percentage of the columns on the left-hand side in the corresponding matrix. For example, in the first row z=10%, l=20%, and u=90%, then 10% of the dense matrix is assigned to the CPU only, and the rest is shared between CPU and FPGA; the first 20% and the last (100-90)=10% of the rows in the sparse matrix are assigned to the CPU, and the rest of the sparse matrix, i.e., 70% of rows, is assigned to the FPGA as shown in Fig. 8. The last column of this table shows the speed-up of the proposed techniques compared to the fastest implementation.

6 CONCLUSIONS

In this paper, the collaborative execution of the irregular sparse matrix-dense matrix operation on an embedded FPGA and multi-core CPU has been studied. We showed that with proper data partitioning between CPU and FPGA, the performance could be improved by a factor of up to 42%. The future work of this research can focus on finding the optimum data partitioning automatically with minimum overhead and apply the proposed techniques to real DNN problems.

ACKNOWLEDGMENTS

The authors would like to thank the support received from EPSRC for this work as part of the ENEAC project (EP/N002539/1). The open-source code of this research is available in the author Github site [7].

REFERENCES

[1] Nathan Bell and Michael Garland. 2009. Implementing Sparse Matrix-vector Multiplication on Throughput-oriented Processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*. ACM, New York, NY, USA, 18:1—-18:11. https://doi.org/10.1145/1654059.1654078

[2] Jee W Choi, Amik Singh, and Richard W Vuduc. 2010. Model-driven Autotuning of Sparse Matrix-vector Multiply on GPUs. *SIGPLAN Not.* 45, 5 (jan 2010), 115–126. https://doi.org/10.1145/1837853.1693471

[3] Tim Davis. 2019. SuiteSparse Matrix Collection. https://sparse.tamu.edu/

[4] Richard Dorrance, Fengbo Ren, and Dejan Marković. 2014. A Scalable Sparse Matrix-vector Multiplication Kernel for Energy-efficient Sparse-blas on FPGAs. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays (FPGA '14)*. ACM, New York, NY, USA, 161–170. https://doi.org/10.1145/2554688.2554785

[5] A Elafrou, G Goumas, and N Koziris. 2017. Performance Analysis and Optimization of Sparse Matrix-Vector Multiplication on Intel Xeon Phi. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1389–1398. https://doi.org/10.1109/IPDPSW.2017.134

[6] Michael Garland. 2008. Sparse Matrix Computations on Manycore GPU's. In *Proceedings of the 45th Annual Design Automation Conference (DAC '08)*. ACM, New York, NY, USA, 2–6. https://doi.org/10.1145/1391469.1391473

[7] Mohammad Hosseinabady. 2020. Sparse Matrix-Dense Matrix Multiplication (SpMDM) implementation. https://github.com/Hosseinabady/SDSoC-Benchmarks/tree/master/SpMDM

[8] Mohammad Hosseinabady and Jose Nunez-Yanez. 2019. A Streaming Dataflow Engine for Sparse Matrix-Vector Multiplication using High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2019). https://doi.org/10.1109/TCAD.2019.2912923

[9] Forrest N Iandola, Matthew W Moskewicz, Khalid Ashraf, Song Han, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR* abs/1602.0 (2016). arXiv:1602.07360 http://arxiv.org/abs/1602.07360

[10] Sid Samsi Jeremy Kepner, Simon Alford, Vijay Gadepally, Michael Jones, Lauren Milechin, Ryan Robinett. 2019. Sparse deep neural network graph challenge. , 7 pages. https://graphchallenge.mit.edu/

[11] M Krotkiewski and M Dabrowski. 2010. Parallel Symmetric Sparse Matrix-vector Product on Scalar Multi-core CPUs. *Parallel Comput.* 36, 4 (apr 2010), 181–198. https://doi.org/10.1016/j.parco.2010.02.003

[12] L Lu, J Xie, R Huang, J Zhang, W Lin, and Y Liang. 2019. An Efficient Hardware Accelerator for Sparse Convolutional Neural Networks on FPGAs. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 17–25. https://doi.org/10.1109/FCCM.2019.00013

[13] Jing Nie, Chunlei Zhang, Dan Zou, Fei Xia, Lina Lu, Xiang Wang, and Fei Zhao. 2019. Adaptive Sparse Matrix-Vector Multiplication on CPU-GPU Heterogeneous Architecture. In *Proceedings of the 2019 3rd High Performance Computing and Cluster Technologies Conference (HPCCT 2019)*. ACM, New York, NY, USA, 6–10. https://doi.org/10.1145/3341069.3341072

[14] Leonid Yavits and Ran Ginosar. 2018. Accelerator for Sparse Machine Learning. *IEEE Comput. Archit. Lett.* 17, 1 (jan 2018), 21–24. https://doi.org/10.1109/LCA.2017.2714667

[15] J Yu, A Lukefahr, D Palframan, G Dasika, R Das, and S Mahlke. 2017. Scalpel: Customizing DNN pruning to the underlying hardware parallelism. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 548–560. https://doi.org/10.1145/3079856.3080215