Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200092

On the Performance and Energy Efficiency of Sparse Matrix-Vector Multiplication on FPGAs

Panagiotis MPAKOS [a], Nikela PAPADOPOULOU [a], Chloe ALVERTI [a], Georgios GOUMAS [a], and Nectarios KOZIRIS [a]
[a] *National Technical University of Athens*
{pmpakos, nikela, xalverti, goumas, nkoziris}@cslab.ece.ntua.gr

Abstract. The Sparse Matrix-Vector Multiplication kernel (SpMV) has been one of the most popular kernels in high-performance computing, as the building block of many iterative solvers. At the same time, it has been one of the most notorious kernels, due to its low flop per byte ratio, which leads to under-utilization of modern processing system resources and a huge gap between the peak system performance and the observed performance of the kernel. However, moving forward to exascale, performance by itself is no longer the holy grail; the requirement for energy efficient high-performance computing systems is driving a trend towards processing units with better performance per watt ratios. Following this trend, FPGAs have emerged as an alternative, low-power accelerator for high-end systems. In this paper, we implement the SpMV kernel on FPGAs, towards an accelerated library for sparse matrix computations, for single-precision floating point values. Our implementation focuses on optimizing access to the data for the SpMV kernel and applies common optimizations to improve the parallelism and the performance of the SpMV kernel on FPGAs. We evaluate the performance and energy efficiency of our implementation, in comparison to modern CPUs and GPUs, for a diverse set of sparse matrices and demonstrate that FPGAs can be an energy-efficient solution for the SpMV kernel.

Keywords. sparse matrix vector multiplication, FPGA, performance, energy efficiency

1. Introduction

Sparse matrices appear in multiple scientific problems, putting sparse linear algebra at the core of high-performance scientific computing. The Sparse Matrix-Vector Multiplication kernel (*SpMV*) has been one of the most popular kernels of this category, as the building block of many iterative solvers. At the same time, it has been one of the most notorious kernels, due to its low flop per byte ratio, which leads to under-utilization of modern processing system resources and a huge gap between the peak system performance and the observed performance of the kernel. A plethora of sparse matrix formats [1] and a variety of optimizations for multi-core processors [2], many-core processors [3] and GPUs [4] have been proposed and applied, to improve the performance of the SpMV kernel.

However, moving forward to exascale, performance by itself is no longer the holy grail; the requirement for energy efficient high-performance computing systems is driving a trend towards processing units with better performance per watt ratios. Following this trend, FPGAs have emerged as an alternative, low-power accelerator for high-end systems. This trend has been further supported by the development of high-level synthesis tools, which significantly reduce the programming effort required to port applications to FPGAs. FPGAs are already used as accelerators in production in datacenters, and several efforts focus on bringing FPGAs to the HPC world. Such an effort is EuroEXA [5], the EU-funded project that aims to implement and prototype a petascale-level system, embracing FPGA acceleration.

In this paper, in the context of the EuroEXA project, we implement the SpMV kernel on FPGAs, towards an accelerated library for sparse matrix computations, for single-precision floating point values. Our implementation focuses on optimizing access to the data for the SpMV kernel and applies common optimizations to improve the parallelism and the performance of the SpMV kernel on FPGAs. We evaluate the performance and energy efficiency of our implementation, in comparison to modern CPUs and GPUs, for a diverse set of sparse matrices, and demonstrate that FPGAs can be an energy-efficient solution for the SpMV kernel.

2. An efficient implementation of SpMV on FPGAs

2.1. Experimental platform

Our experimental platform is a Xilinx Zynq UltraScale+ MPSoC ZCU102 board. The MPSoC of the board consists of a quad-core ARM Cortex A53 processor, operating at (up to) 1.5 GHz and a Zynq UltraScale ZU9EG FPGA. The FPGA contains around 600K logic cells, 32 Mbs of BlockRAM and about 2500 DSP slices. The MPSoC contains 4GB of DDR4 DRAM (referred to as main memory from now on), which is accessible from both the ARM processor and the FPGA. We use the Xilinx SDSoC environment (version 2018.1) which utilizes the Xilinx Vivado-HLS compiler and Vivado Design Suite tools to compile synthesizable C/C++ functions into programmable logic.

```
void SpMV(int nnz, int nrows, float *values, float *col_ind,
                              float *row_ptr, float *x, float *y)
{
  for (int i = 0 ; i < nrows ; i++)
    for (int j = row_ptr[i] ; j < row_ptr[i+1] ; j++)
        y[i] += values[j] * x[col_ind[i]];
  return;
}
```

Algorithm 1. The CSR-SpMV kernel

2.2. CSR-SpMV: Properties and challenges

The Compressed Sparse Row (CSR) representation is the most commonly used sparse matrix representation, since it is generic, agnostic to the sparsity pattern and leads to fair performance on CPUs with no preprocessing cost. In the CSR format, a sparse matrix is represented with three vectors: the *values* vector contains the values of all non-zero elements of the matrix, the *col_ind* vector contains the column index for each non-zero element and the *row_ptr* vector stores the row pointers. The $y = A \times x$ CSR-SpMV kernel, for the sparse matrix A with *nrows* rows and *nnz* non-zero elements is presented in Algorithm 1.

There are two key observations regarding CSR-SpMV; first, the x vector is accessed randomly, and second, there is no reuse for the elements of the A matrix, i.e., the kernel is memory-bound. This is particularly important for our FPGA implementation: accessing data on the main memory of the MPSoC is costly, due to the limited memory bandwidth, and it is preferable to move data to the FPGA BRAM. However, BRAM capacity is limited. Thus, an efficient implementation requires careful data movement and placement.

Another challenge for the FPGA implementation of CSR-SpMV using HLS tools is that the boundaries of the inner loop are unknown at compile time. This impedes efficient loop unrolling by the compiler and limits the parallelism of the implementation.

2.3. pCSR: A packed, CSR-based representation format for sparse matrices

Using our key observations about the CSR-SpMV code, we opt for an implementation where the sparse matrix A is efficiently streamed from the main memory to the FPGA. The x vector is stored locally on the FPGA BRAM, to ensure fast access. The y vector is accessed sequentially, therefore we stream it from the FPGA back to the main memory, and do not store it locally on the BRAM. To efficiently stream the sparse matrix to the FPGA, we need to exploit the four available High Performance (HP) memory ports of our MPSoC. These ports allow for the highest throughput of data transfer from the main memory to the FPGA. Extending the MCSR representation proposed in [6], we first transform the *row_ptr* vector to a *row_length* vector, where each element refers to the number of non-zero values per row. We then pack the *row_length*, *col_ind* and *values* vector into a single stream of data, as following: for a single row of the sparse matrix, we use a zero element to denote a new line, followed by the number of non-zero elements in this row. For every non-zero element in the row, the *col_ind* and the value of the element follow in pairs. In order to fully exploit the available bandwidth of HP ports, the stream is then split into 128-bit wide parts. We also use the `hls::stream` objects, which are FIFO queues, to stream the data to the FPGA through each available port. We note that, in our SpMV implementation, the x vector is copied and stored locally on the FPGA BRAM, to ensure fast access. Therefore, the largest problem size that we can solve on our FPGA depends on the number of columns of the sparse matrix, i.e. the length of the x vector. On the other hand, the length of the y vector does not constrain our design: since the y vector is accessed sequentially, it is streamed from the FPGA back to main memory.

2.4. Optimizations

2.4.1. Increasing parallelism with vectorization

To increase the parallelism of our design, we implement SIMD parallelism by partially unrolling the inner loop of the SpMV code, by a factor of II. We note that, in order to implement vectorization, zero-padding is required, so that the elements of each row are a multiple of the vectorization factor. We test our design with vectorization factors of 2, 4 and 8.

2.4.2. Increasing parallelism with 1D-blocking

To further increase parallelism in our design, as well as the resource utilization of the FPGA, we employ multiple compute units (CU). The multiple compute units implement multiple instances of the SpMV kernel. This is equivalent to row-wise partitioning or 1D-blocking: each compute unit works on a contiguous subset of rows of the sparse matrix. We test our design with 2, 4 and 8 compute units. We note that using more than four compute units leads to full utilization of the available HP ports. However, compute units cannot share data and therefore, each compute unit requires a separate copy of the vector x. Therefore, increasing the number of compute units limits the maximum size of the x vector that can fit in the FPGA BRAM.

2.4.3. Increasing problem size with 2D-blocking

To overcome the limitation of the limited BRAM capacity that arises when using multiple compute units, we employ 2D-blocking, i.e. row-wise and column-wise partitioning, of the sparse matrix. In this way, each compute unit only needs to store part of the x vector, i.e. the part that is used by the columns of the 2D-block. Intermediate results are stored in $y\_partial$ vectors, which are then accumulated on the host side. Blocking on this second dimension allows us to split the matrix to as many blocks needed to fit the multiple copies of the partial x vectors on the FPGA BRAM. However, to implement 2D-blocking, alongside vectorization, the number of elements of each row of each block needs to be a multiple of the vectorization factor, which results to additional zero padding. Compression of the zero-padded elements can be employed to alleviate the memory overhead that occurs in this case.

2.4.4. Load balancing

Depending on the sparsity pattern of the matrix, 1D-blocking, i.e. row-wise partitioning, commonly produces load imbalance among the multiple compute units. We easily mitigate this problem by equally distributing non-zero elements across compute units. In this case, each compute unit solves the SpMV kernel on variable numbers of contiguous rows, but with better load distribution.

2.5. Increasing performance with clock frequency configuration

The FPGA of our experimental platform can be configured to operate under clock frequencies ranging from 100 to 600 MHz. Our implementation meets the timing requirements for frequencies up to 300 MHz. Figure 1 shows how execution time and power

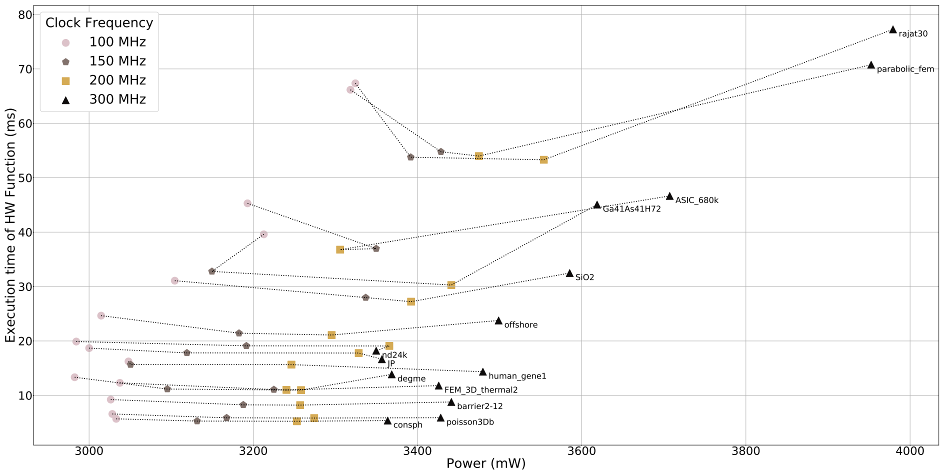

Figure 1. Comparison of execution time and power consumption of FPGA-SpMV for different clock frequencies. Each line represents a different sparse matrix.

consumption vary while we increase the clock frequency. Connected points represent measurements for the same matrix under different clock frequencies. As expected, higher clock frequency leads to higher power consumption. For all the matrices, execution time decreases as we increase the clock frequency up to 200MHz. The decrease is more significant for the larger matrices. However, when the frequency is set at 300MHz, the synthesized bitstream consumes more of the BRAM resources, limiting the available BRAM to store the x vector. This causes large size matrices to be split in more blocks, hence we observe an increase in execution time. Due to the algorithmic nature of SpMV, increasing the clock frequency of the FPGA does not proportionally improve the performance of our implementation. Therefore, we conclude that the optimal frequency, considering both performance and power consumption, is 150 MHz.

3. Evaluation

To evaluate our FPGA implementation, we use a diverse set of 19 sparse matrices from the University of Florida Sparse Matrix Collection [7], with a variety of sparsity patterns and sizes. The number of floating point operations per non-zero element is 2 (multiplication and addition). For the pCSR representation,we use 64 bits to store each row of the matrix, and 64 bits for each non-zero element of the matrix (value and col_ind index). Therefore, the flops:byte ratio is calculated by the formula $nnz/(4*nrows+4*nnz)$. We compare the performance and energy efficiency of SpMV on our FPGA against CSR-SpMV using the Intel MKL library on an Intel Xeon E5-2630V4 (Broadwell) CPU with 10 cores, 25MB LLC and 256GB of memory, and against CSR-SpMV using the cuSPARSE library on an NVIDIA Tesla K40 GPU, with 2880 cores and 12GB GDDR5 memory. We use RAPL performance counters to measure energy on the CPU. For the FPGA, we modified the power monitoring application proposed in [8], in order to execute it on our board. For the GPU, we use the GPU power sensors and compute energy according to the runtime.

Table 1. Matrix suite used for experimental evaluation

Matrix	Dimension	Non-zeros	Size(MB)	f:b ratio
human_gene1	22283	12345963	94.2772	0.2495
nd24k	72000	14393817	110.091	0.2488
JP	87616	13734559	105.121	0.2484
consph	83334	3046907	23.564	0.2433
poisson3Db	85623	2374949	18.4461	0.2413
barrier2-12	115625	3897557	30.1771	0.2428
FEM_3D_thermal2	147900	3489300	27.1854	0.2398
SiO2	155331	5719417	44.2282	0.2434
degme	185501	8127528	62.7158	0.2444
offshore	259789	2251231	18.1666	0.2241
Ga41As41H72	268096	9378286	72.5734	0.2431
parabolic_fem	525825	2100225	18.0293	0.1999
rajat30	643994	6175377	49.5711	0.2264
ASIC_680k	682862	3871773	32.1442	0.2125
Hardesty2	929901	4020731	34.2231	0.203
boneS10	914898	28191660	218.575	0.2421
audikw_1	943695	39297771	303.418	0.2441
webbase-1M	1000005	3105536	27.5081	0.1891
thermal2	1228045	4904179	42.1006	0.1999

Table 2. Hardware platforms

Device	Operating Frequency	Memory	Memory Bandwidth
Intel Xeon E5-2630V4	2.2 GHz	256 GB	40 GB/s
NVIDIA Tesla K40	745 MHz	12 GB	6 GB/s
Xilinx MPSoC ZCU102	150 MHz	4 GB	9.6 GB/s

Figure 2 shows the execution time for SpMV on the 19 matrices, for the CPU, the GPU and the FPGA. For our FPGA implementation, we showcase the results for a vectorization factor of 4, and 4 compute units. In addition, the frequency of the FPGA is set to 150MHz. We note that the execution time for the FPGA implementation includes transfers from the main memory to the FPGA and vice versa. For a fair comparison against the GPU, we compare against the performance with and without transfers over the PCIe (6GB/s). In comparison to the CPU, our FPGA implementation is slower from 7 to 62 times, with an average slowdown of 26x. We consider this performance gap to be reasonable, given the 15x difference in frequency and the 2.5x difference in cores (compute units), between the CPU and the FPGA. GPU performance is close to that of the CPU, apart from three matrices in our suite (*ASIC_680K*, *rajat30*, *degme*), which suffer from imbalance [3]. However, if we include the transfers from the host to the GPU, the average slowdown for SpMV on the FPGA is 3x.

Figure 3 shows the energy consumption for SpMV, for the CPU, the GPU and the FPGA. Despite the large difference in execution times, the CPU consumes up to 5 times more energy than the FPGA for the SpMV kernel, with the exception of the largest matrices in our dataset (*webbase_1M*, *thermal2*). The GPU consumes about the same energy with the FPGA for the computational part of the SpMV kernel, with the exception of the three imbalanced matrices. However, if we take into account the data transfers

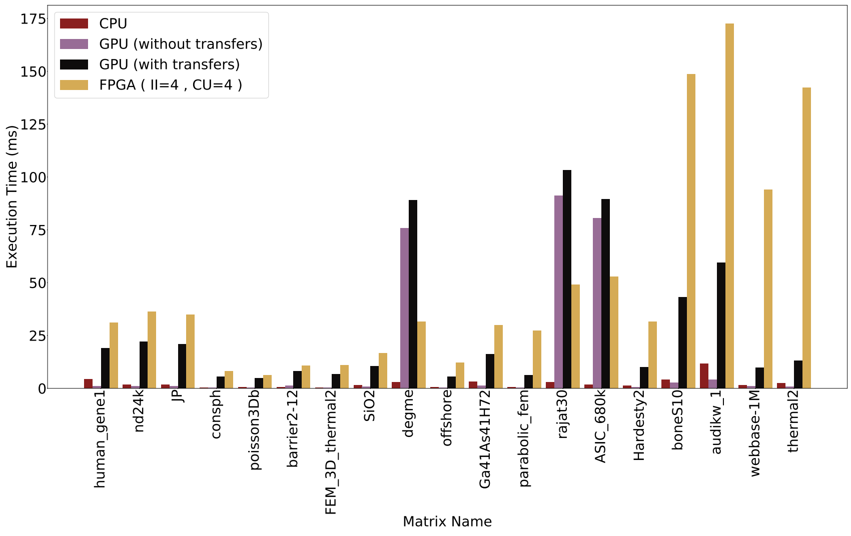

Figure 2. Comparison of performance of the CSR-SpMV kernel among different architectures.

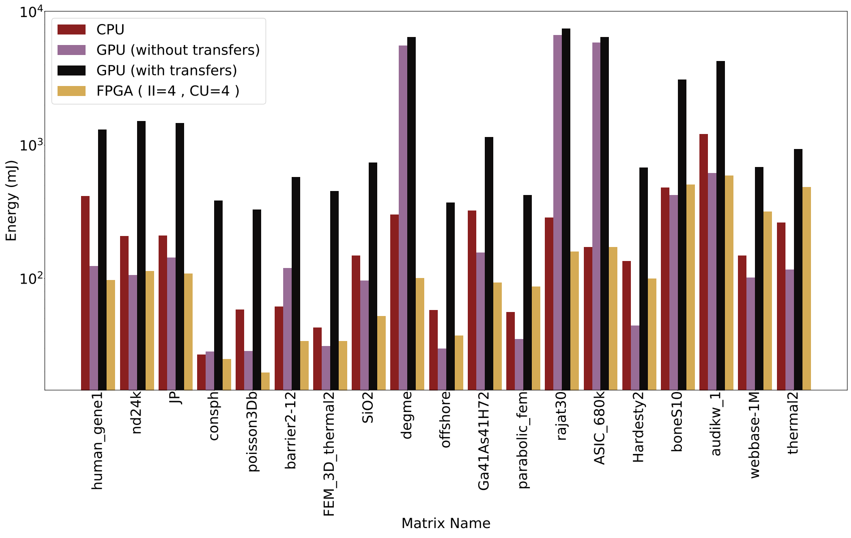

Figure 3. Comparison of the energy consumption (in Joules) of the CSR-SpMV kernel among different architectures. The y axis is in logarithmic scale.

from the host to the GPU, the GPU becomes the least energy-efficient option among the three architectures.

Another metric that can be used to measure the energy efficiency of each architecture is the performance per Watt, i.e. FLOPs per Watt [9]. In SpMV, two floating-point operations occur for each non-zero element; multiplication with the respective element of the x vector and accumulation of the result in the y vector. Thus, SpMV FLOPs are calculated by dividing the doubled number of non-zeros of the matrix with the execution time. Figure 4 shows how each architecture performs in GFLOPs/W. For smaller

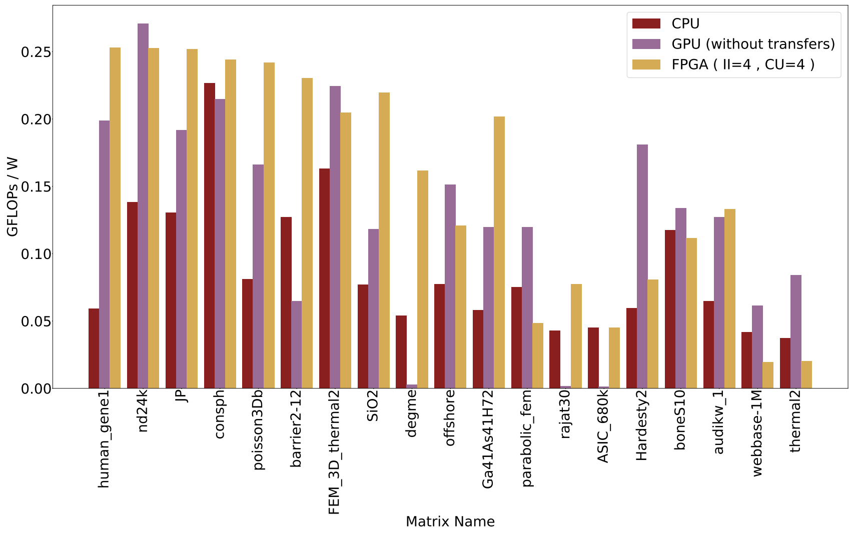

Figure 4. Comparison of the energy efficiency (in GFLOPs/W) of the CSR-SpMV kernel among the different architectures.

matrices (leftmost part of the figure), the FPGA significantly outperforms the other architectures in terms of energy efficiency. For larger matrices, although the performance of our FPGA implementation is degraded due to extensive zero-padding, the FPGA still performs well in the GFLOPs/W metric, being a viable, energy-efficient option for the SpMV kernel.

4. Related work

Multiple works present implementations of the SpMV kernel on FPGAs, using hardware design tools. Zhuo et al. [10] provide an efficient design for CSR SpMV on FPGAs, using a number of subtrees of multipliers and a specialized reduction unit. Their implementation also stores the *x* vector on the FPGA. Sun et al. [11] describe a design using multiple processing elements which include a deep pipeline with a multiplier, an accumulation circuit and FIFO queues. Their implementation uses both CSR and Row Blocked CSR. Kestur et al. [12] design a library for SpMV and propose the CVBV format, to reduce memory capacity requirements and memory bandwidth requirements. Dorrance et al. [13] implement the SpMV kernel using CSC, to make memory accesses to the *x* vector, stored in the main memory, sequential, reducing memory bandwidth requirements. Grigoras et al. [14] propose a dictionary-based compression format to improve the effective memory bandwidth of SpMV designs. Their implementation uses Maxeler tools. Several implementations of SpMV with OpenCL appear in recent work [15,9,16]. Our work extends the implementation of CSR SpMV proposed by Hosseinabady et al. in [6], which uses HLS tools to synthesize SpMV as a streaming dataflow engine.

In addition, a number of works examine the performance and energy efficiency of various algorithms on CPUs, GPUs and FPGAs. Vestias et al. [17] explore general trends in peak performance and power for CPUs, GPUs and FPGAs. Betkaoui et al. [18] compare the performance and energy efficiency of GPUs and FPGAs for four commonly used benchmarks. Fowers et al. [19] focus their analysis of performance and energy efficiency on sliding-window applications. Rucci et al. [20] focus on state-of-the-art implementations of the Smith-Waterman protein database search, on CPUs, co-processors, GPUs and FPGAs. Finally, Giefers et al. [9] perform a performance and energy-efficiency analysis for sparse matrix-vector and sparse matrix-matrix multiplication on co-processors, GPUs and FPGAs.

5. Conclusions

In this work, we examine the performance and energy efficiency of the sparse matrix-vector multiplication on FPGAs. We design and optimize the CSR-SpMV kernel for a Xilinx Zynq UltraScale+ board with a ZU9EG FPGA, using HLS tools. We evaluate the performance and energy efficiency of this kernel with the equivalent CSR-SpMV implementations on an Intel Broadwell CPU and an NVIDIA Tesla K40 GPU. Our experimental results show that the CPU and GPU outperform the FPGA in terms of performance for the SpMV kernel, however, the energy consumption of the FPGA is lower for most of the matrices in our dataset. In addition, comparing the achieved FLOPs/W for the three platforms, the FPGA is a particularly energy-efficient option for the SpMV kernel, and a further optimized design can bring in additional gains for energy consumption and energy efficiency. Future directions of this work focus on solving the SpMV on FPGAs and include further optimizations of the CSR-SpMV kernel for the FPGA, the exploration of alternative storage formats for the sparse matrices and their impact on performance and energy efficiency, as well as the evaluation of our CSR-SpMV kernel on FPGAs with higher BRAM capacity, higher bandwidth and more DSP units.

Acknowledgment

This research has received funding from the European Unions Horizon 2020 research and innovation programme under Grant Agreement no 754337 (EuroEXA project).

References

[1] D. Langr and P. Tvrdik, “Evaluation criteria for sparse matrix storage formats,” *IEEE Transactions on parallel and distributed systems*, vol. 27, no. 2, pp. 428–440, 2015.

[2] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, “Optimization of sparse matrix-vector multiplication on emerging multicore platforms,” in *SC’07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pp. 1–12, IEEE, 2007.

[3] A. Elafrou, G. Goumas, and N. Koziris, “Performance analysis and optimization of sparse matrix-vector multiplication on intel xeon phi,” in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 1389–1398, IEEE, 2017.

[4] M. M. Baskaran and R. Bordawekar, “Optimizing sparse matrix-vector multiplication on gpus,” *IBM Research Report RC24704*, no. W0812-047, 2009.

[5] G. Goumas, N. Koziris, N. Papadopoulou, K. Nikas, V. Karakostas, C. Alverti, J. Goodacre, M. Lujan, O. Palomar, M. Ashworth, *et al.*, “Co-designed innovation and system for resilient exascale computing in europe: From applications to silicon (euroexa),” 2017.

[6] M. Hosseinabady and J. L. Nunez-Yanez, “A streaming dataflow engine for sparse matrix-vector multiplication using high-level synthesis,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[7] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[8] "Zynq-7000 ap soc low power techniques part 4 - measuring zc702 power with a linux application tech tip." https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841995. Accessed: 2019-09-30.

[9] H. Giefers, P. Staar, C. Bekas, and C. Hagleitner, "Analyzing the energy-efficiency of sparse matrix multiplication on heterogeneous systems: A comparative study of gpu, xeon phi and fpga," pp. 46–56, IEEE, 2016.

[10] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on fpgas," in *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pp. 63–74, ACM, 2005.

[11] J. Sun, G. Peterson, and O. Storaasli, "Sparse matrix-vector multiplication design on fpgas," in *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*, pp. 349–352, IEEE, 2007.

[12] S. Kestur, J. D. Davis, and E. S. Chung, "Towards a universal fpga matrix-vector multiplication architecture," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pp. 9–16, IEEE, 2012.

[13] R. Dorrance, F. Ren, and D. Marković, "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*, pp. 161–170, ACM, 2014.

[14] P. Grigoras, P. Burovskiy, E. Hung, and W. Luk, "Accelerating spmv on fpgas by compressing nonzero values," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 64–67, IEEE, 2015.

[15] Q. Gautier, A. Althoff, P. Meng, and R. Kastner, "Spector: An opencl fpga benchmark suite," in *2016 International Conference on Field-Programmable Technology (FPT)*, pp. 141–148, IEEE, 2016.

[16] Y. Shan, T. Wu, Y. Wang, B. Wang, Z. Wang, N. Xu, and H. Yang, "Fpga and gpu implementation of large scale spmv," in *2010 IEEE 8th Symposium on Application Specific Processors (SASP)*, pp. 64–70, IEEE, 2010.

[17] M. Vestias and H. Neto, "Trends of cpu, gpu and fpga for high-performance computing," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–6, IEEE, 2014.

[18] B. Betkaoui, D. B. Thomas, and W. Luk, "Comparing performance and energy efficiency of fpgas and gpus for high productivity computing," in *2010 International Conference on Field-Programmable Technology*, pp. 94–101, IEEE, 2010.

[19] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "A performance and energy comparison of fpgas, gpus, and multicores for sliding-window applications," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pp. 47–56, ACM, 2012.

[20] E. Rucci, C. García, G. Botella, A. De Giusti, M. Naiouf, and M. Prieto-Matías, "State-of-the-art in smith–waterman protein database search on hpc platforms," in *Big Data Analytics in Genomics*, pp. 197–223, Springer, 2016.