- ORIGINAL ARTICLE -

Energy-Efficient Algebra Kernels in FPGA for High Performance Computing

Núcleos de Álgebra Energéticamente Eficientes en FPGA para Computación de Altas Prestaciones

Federico Favaro[1], Ernesto Dufrechou[2], Pablo Ezzatti[2], and Juan P Oliver[1]

[1] *Instituto de Ingeniería Eléctrica, Universidad de la República, Uruguay*
{ffavaro, jpo}@fing.edu.uy
[2] *Instituto de Computación, Universidad de la República, Uruguay*
{edufrechou, pezzatti}@fing.edu.uy

Abstract

The dissemination of multi-core architectures and the later irruption of massively parallel devices, led to a revolution in High-Performance Computing (HPC) platforms in the last decades. As a result, Field-Programmable Gate Arrays (FPGAs) are re-emerging as a versatile and more energy-efficient alternative to other platforms. Traditional FPGA design implies using low-level Hardware Description Languages (HDL) such as VHDL or Verilog, which follow an entirely different programming model than standard software languages, and their use requires specialized knowledge of the underlying hardware. In the last years, manufacturers started to make big efforts to provide High-Level Synthesis (HLS) tools, in order to allow a grater adoption of FPGAs in the HPC community.

Our work studies the use of multi-core hardware and different FPGAs to address Numerical Linear Algebra (NLA) kernels such as the general matrix multiplication (GEMM) and the sparse matrix-vector multiplication (SPMV). Specifically, we compare the behavior of fine-tuned kernels in a multi-core CPU processor and HLS implementations on FPGAs. We perform the experimental evaluation of our implementations on a low-end and a cutting-edge FPGA platform, in terms of runtime and energy consumption, and compare the results against the Intel MKL library in CPU.

Keywords: dense and sparse NLA, FPGA, HLS, energy consumption

Resumen

La masificación de arquitecturas de multinúcleo y la posterior irrupción de dispositivos masivamente paralelos produjeron una revolución en las plataformas de computación de altas prestaciones. Como resultado, las FPGAs (del inglés, Field-Programmable Gate Arrays) están resurgiendo como una alternativa versátil y más eficiente desde el punto de vista energético. El flujo de diseño tradicional en FPGAs implica el uso de lenguajes de descripción de hardware de bajo nivel, como VHDL o Verilog, que siguen un modelo de programación completamente diferente al de los lenguajes de software estándar, y su uso requiere un conocimiento especializado del hardware subyacente. En los últimos años, los fabricantes comenzaron a hacer grandes esfuerzos para proporcionar herramientas de síntesis de alto nivel, con el fin de permitir una mayor adopción de las FPGAs en la comunidad de computación de altas prestaciones.

Nuestro trabajo estudia el uso de plataformas multinúcleo y diferentes FPGAs para abordar problemas de álgebra lineal numérica (NLA) como la multiplicación de matrices (GEMM) y la multiplicación de matriz dispersa por vector (SPMV). Específicamente, comparamos el comportamiento de implementacióne optimizadas para un procesador multinúcleo y las implementaciones con síntesis de alto nivel en FPGAs. Realizamos la evaluación experimental de nuestras implementaciones en una plataforma FPGA de gama baja y otra de gama alta, analizando tiempo de ejecución y consumo de energía, y comparamos los resultados con la biblioteca Intel MKL para CPU.

Palabras claves: algebra densa y dispersa, FPGA, HLS, consumo de energía

1 Introduction

There is an increasing concern for energy consumption in HPC [1], as it has become one of the main constraints of hardware platform design. This concern is due to both the economic cost of electricity and the environmental impact. Also, the massive dissemination of the multi-core processor, less than two decades ago, and the later adoption of GPUs as general-purpose computing devices, started an important revolution for HPC hardware. More recently, this revolution has reached Field-Programmable Gate Arrays (FPGAs). This scenario makes the FPGAs an attractive, more energy-efficient alternative to other many-core HPC devices.

The classic approach for FPGA design implies using low-level Hardware Description Languages (HDL) such as VHDL or Verilog. These impose a different programming models than standard software languages, with more extended development periods and complex debugging. Furthermore, their use requires specialized knowledge of the underlying hardware, explaining why the HPC community does not massively adopt FPGAs. To overcome this disadvantage, manufacturers are making efforts to adopt HLS languages like C/C++ and OpenCL. The most relevant evidence of this is the introduction of OpenCL frameworks by the largest FPGAs manufacturers, Intel [2] and Xilinx [3]. This enables more significant adoption of FPGAs as hardware accelerators by the software community.

The rapid growth of information technology in the recent decades gave society the ability to generate, collect and store huge volumes of data. This data is growing exponentially, and it is expected to continue to do so in the following decades. A large amount of data comes from internet applications, including social networks or web searches. Due to its nature, graphs often represent this data, which has led to significant efforts in efficiently handling these structures. An example in this direction is GraphBLAS[1], a sincere effort to define standard building blocks for graph algorithms in the language of linear algebra. In general, graphs can be represented as sparse matrices to process them more efficiently. In the big data field, this means matrices of enormous proportions.

Numerical Linear Algebra (NLA) is a research field characterized by the use of kernel-libraries that are *de facto* standards. Some key examples are the BLAS specification for elementary dense matrix operations and the sparse matrix-vector multiplication (SPMV) for the sparse case. These kernels are the central part of several scientific programs and, in general, the most costly stage from the execution time and energy consumption perspectives [4]. This has motivated several efforts to improve the performance and energy consumption of these building blocks [5].

This work's main objective is to understand the contexts in which each hardware platform is competitive, in particular, contrasting traditional multi-core CPUs and FPGAs with different characteristics. For this purpose, we select the two most representative kernels from the dense and sparse NLA fields, respectively, the general matrix multiplication (GEMM) and the sparse matrix-vector product (SPMV). We develop and evaluate both NLA kernels for FPGAs using HLS languages. The experimental evaluation carried out in a low-end FPGA platform from Intel (Cyclone V SoC) and a cutting-edge FPGA from Xilinx (Alveo U50), compares both implementations with those included in the Intel MKL library for traditional CPUs, in terms of runtime and energy consumption. This paper is an extension of our previous work [6] in several lines. We highlight the inclusion of a cutting-edge FPGA, the fine-tuned development for the Xilinx FPGA and the addition of several new test cases and experiments.

The paper is structured as follows. In Section 2 we describe the two addressed NLA kernels (GEMM and SPMV), and we include a brief introduction to FPGAs and a review of literature related to the use of FPGAs to compute NLA kernels. In Section 4, we present our designs for both operations using HLS Languages. This is followed by the experimental evaluation and the comparison with the Intel MKL variants in Section 5. Finally, Section 6 contains the concluding remarks and an outline of future research.

[1] https://graphblas.github.io/

2 NLA kernels and FPGAs

In this section we first describe the selected NLA kernels. Later, we present the FPGA technology and the use of HLS (such as OpenCL) to perform computations on these devices. Finally, the section closes with a summary of related work on leveraging FPGAs to accelerate NLA operations.

2.1 NLA kernels

Firstly, we describe the two NLA kernels employed in our work as a proof of concept.

2.1.1 General matrix-matrix multiplication

This operation (GEMM) is defined as follows:

$$C = \alpha A * B + \beta C \quad (1)$$

where A, B and C are matrices and α and β are scalars. This kernel is considered the main building block in dense linear algebra because many other operations can be expressed in terms of several GEMM invocations [7]. GEMM belongs to Level 3 of the BLAS specification [8].

2.1.2 Sparse matrix-vector multiplication

This operation is the base of iterative linear-system and eigenvalue solvers. The Algorithm 1 summarizes the serial version of the sparse matrix-vector multiplication (SPMV), where the sparse matrix A is stored in Compressed Sparse Row (CSR) format [9].

2.2 FPGA technology

Unlike other heterogeneous platforms (for example, GPUs), FPGAs have no pre-defined high-level architecture. They are composed of a matrix of configurable logic blocks (known as logic elements) and hardcoded blocks such as memories, hardware multipliers and clock managers. The interconnection of the different blocks is achieved through a programmable routing structure. To interface with the outside world, they

Algorithm 1 Serially computed sparse matrix-vector multiplication (SPMV). The matrix A is stored using the CSR format; *val* stores the nonzero elements; *row_ptr* stores the index of the first element for each row in vector *val*, and *col_idx* stores the column index of each element in the matrix A. The nonzero elements within each row are ordered by column index.

Input: row_ptr, col_idx, val, x
Output: y

1: $y = 0$
2: **for** $i = 0$ **to** $n-1$ **do**
3: **for** $j = row_ptr[i]$ **to** $row_ptr[i+1]-1$ **do**
4: $y[i] = y[i] + val[j] \cdot x[col_idx[j]]$
5: **end for**
6: **end for**

have several programmable input/output pins that allow the implementation of multiple communication protocols.

FPGAs are not *programmed* in a software sense. This means there is no code running (in general) inside the FPGA. Instead, an actual electrical circuit is synthesized inside the device through the programmable logic's interconnection-elements and hard-coded blocks. This allows processing with low latency (as there is little control overhead), excellent flexibility, and fine-grained parallelism. FPGAs stand in the middle between Application-Specific Integrated Circuits (ASICs) and general-purpose processors from a technological perspective. One of the main differences with ASICs is that FPGAs can be reprogrammed after the manufacturing process.

The clock's operating frequency of a given design depends on the synthesized circuit, but it is usually lower than other heterogeneous devices. This is a consequence of supporting reconfigurability. FPGAs also offer lower peak floating-point performance than GPUs and less memory bandwidth, but this may change shortly, as FPGA manufacturers are making efforts to compete with GPU performance in these contexts.

The logic blocks (or logic elements) are the smallest programmable blocks of the FPGA. In their simplest form, these blocks are composed of a programmable Look-Up Table (LUT), a Flip-Flop and a programmable selector that takes either the LUT output or the register output. LUTs can implement small logic functions, and their value is set during the FPGA programming stage. Combining multiple (up to thousands or millions) LUTs and registers allow the generation of complex logic functions or state machines. When a design is synthesized in the FPGA, the different functions or tasks are mapped to its different resources, and then the circuit is completed by interconnecting these resources.

Traditionally, FPGAs have been a good alternative in fixed-point, dataflow streaming applications, where they reach high speeds at excellent energy efficiency. However, their poor performance in floating-point arithmetic, in addition to the complex design flow and difficult integration with other processors kept them apart from the mainstream HPC world. This started to change recently, as modern high-end FPGA devices offer up to millions of logic elements, thousands of DSP blocks (that allow TFLOP performance) and high memory bandwidth. These characteristics, in combination with the HLS tools available, are making these devices increasingly attractive in the HPC domain.

2.3 HLS development for FPGAs

High-Level Synthesis tools create HDL code from a highly abstract source code like C, C++, or SystemC. The generated hardware description can then be synthesized to a real digital circuit in an FPGA, allowing for a much faster design than with traditional HDL languages.

FPGA design using HDL languages allows creating a highly-optimized solution for a given problem. However, the design takes a long time and requires significant expertise regarding the underlying hardware. The use of HLS languages brings FPGA design to software development times, and it is approachable by non-hardware experts. However, to create a truly optimized HLS code for a specific platform, certain knowledge of the underlying hardware is still required. The verification of the design is also faster as it is performed in HLS.

In the following subsections, we briefly describe the HLS languages and tools used in this work.

2.3.1 OpenCL

This open-source, cross-platform, parallel programming standard is based on a platform-independent API that abstracts the underlying hardware. It consists of a host code running on a CPU that offloads computing-intensive tasks to an heterogeneous device, hiding from the programmer the complexity of controlling and communicating with the accelerator. The device code is written in a C-like language, and it is called kernel code.

The OpenCL model defines a single thread as a work-item, and gathers them into workgroups. Parallelism can be achieved at the thread level and also between workgroups. Every work-item has its private memory and shares local memory with all threads within a workgroup. However, the only memory shared between workgroups is the global memory, which is usually off-chip, slow, and abundant (in the order of GBytes) compared to the on-chip, fast, and scarce local memory (of up to a few MBytes). OpenCL refers to the number of threads per workgroup as *local work size*, and calls *global work size* the total number of work-items required to solve a task (execute a kernel to completion). The standard organizes the

work-items in a workgroup in up to three dimensions. A multidimensional index then identifies each work-item according to the number of dimensions used. This is called NDRange model.

The Intel FPGA SDK for OpenCL allows the user to interface with the FPGA accelerator using the device-agnostic OpenCL programming model. This hides the complexity of interfacing and exchanging data between FPGA and host CPU, which is not an easy task to do in traditional HDL-based design. It considerably accelerates the development time at the cost of some performance degradation.

The NDRange programming model for FPGAs does not provide thread-level parallelism by default. Instead, it creates a deep pipeline that processes all work-items one after the other. Thread-level parallelism can be achieved through kernel vectorization, which increases the data pipeline's width processed by work-items in a SIMD fashion. Also, the pipeline can be replicated using Compute Unit Replication, which has the same effect of adding thread-level parallelism, but it is usually more resource-demanding.

2.3.2 Xilinx HLS C/C++

Despite all the benefits mentioned about OpenCL, for Xilinx development we used HLS C/C++, which is recommended to achieve better performance in Xilinx devices. If OpenCLwere to be used regardless, it is not advisable to use multiple workgroup dimensions (NDRange paradigm).

High-Level Synthesis C/C++ is fundamentally C/C++ language, where the compiler —aided by the designer by the use of pragmas and a specific coding style— takes advantage of the available FGPA resources to parallelize the code.

Parallelism can be explicit by mapping the same function to different FPGA resources that work simultaneously. For example, this can be achieved by loop unrolling. Another way of accelerating an application is by pipelining, which can be achieved by implementing the application as a deep pipeline and launch several independent tasks with short intervals. Even if the latency of the whole pipeline is high, when fully utilized, it will produce high throughput. When applying pipelining to a loop, the best-case scenario is when the pipeline can process new data at every clock cycle. This means that the loop is pipelined with an Initiation Interval (II) equal to 1.

For the Xilinx FPGA design, we used the software Vitis, a relatively new tool from Xilinx to develop embedded software and accelerated applications on heterogeneous platforms. It integrates all previous software development platforms into one unified environment.

3 Related work

There are numerous efforts to use FPGA to accelerate NLA operations in both sparse and dense contexts. In this section, we describe those that are most relevant to this work.

Early works from [10] propose an efficient implementation of BLAS level 2 routines on FPGA and compare the execution time and energy consumption with CPU and GPU platforms. For the GAXPY implementation on the BEE3 FPGA, the authors claim a 293× improvement in energy efficiency against a Tesla C1060 GPU. However, their design requires storing the matrix in internal memory, limiting the size of the problem. In a later work [11] the authors present a universal matrix-vector multiplication (MVM) library to accelerate matrix computations using FPGAs. They propose a flexible and scalable design and support a variety of matrix formats for the sparse case.

In [12] the authors analyze the energy efficiency of a dense matrix multiplication kernel in a hybrid CPU/FPGA system. Based on their measurements, they conclude that for double-precision arithmetic the FPGA does not speed up the computations. Regarding energy consumption, the FPGA is more efficient if only the dynamic power is considered. However, due to the high static power consumption, it is less energy efficient if the total system power is considered.

The authors in [13] evaluate the performance of a matrix multiplication OpenCL kernel in a Stratix V FPGA and compare the results against the CPU implementations using Intel MKL and OpenBLAS. The OpenCL kernel is based on blocked matrix multiplication according to the NDRange OpenCL paradigm. They study the impact on throughput and power consumption of modifying the block size and applying vectorization. The resulting kernel is slower than the CPU implementation but more energy efficient.

In [14] the authors propose a model to optimize matrix multiplication for FPGA platforms by maximizing performance (computations) and minimizing off-chip I/O accesses. They apply their model to a particular implementation in FPGA using HLS obtaining competitive performance while maintaining high levels of abstraction in the code that allows portability between platforms.

In [15] the authors present a double-precision parallel implementation of the sparse direct KLU matrix solver on a Virtex-5 FPGA. They compare the results against the implementation on an Intel Core i7 for various matrices generated from spice3f5 circuit simulations. They obtain speedups in the computations that range from 1.2 to 64×. The authors in [16] present the implementation of a sparse matrix solver in an FPGA applied to matrices from SPICE circuit simulations. They compare their hardware prototype's performance against state-of-the-art software packages running on a CPU, obtaining average speedups of 9.65×, 11.83×,

and 17.21× against UMFPACK, KLU, and Kundert. Another effort in this field is the study about the implementation of sparse triangular solver (SPTRSV) in FPGA from energy and runtime perspective [17].

In [18] the authors propose a streaming dataflow architecture to perform SPMV operation in an embedded platform containing a Xilinx ZynqMP FPGA. The proposed solution consists of a deep pipeline that is constantly consuming input data with no stalls. To speed up the operation, they process several rows in parallel and propose a solution to the limited number of ports in the device by interleaving input data in a single port. Their solution achieves speedups 3.25 for small and medium-size matrices against an embedded GPU but is slower by a factor of 1.58. for large matrices. Their FPGA implementation is more energy efficient in all cases.

Several reviewed works about FPGAs for NLA applications develop kernels especially designed to tackle the target test cases. In other words, the synthesized hardware is optimized to achieve the best results possible for the matrices underlying each problem. Our approach is slightly different. We are working on general implementations that, after being tested on several different matrices, allow us to determine which hardware platform (CPU or FPGA) is the best alternative according to the problem's characteristics.

4 NLA kernels design

In this section, we offer some details on the design of the routines that tackle the selected NLA kernels. Firstly, we describe the low-end (embedded) FPGA designs, continuing with the high-end (data center acceleration) variant later.

4.1 Embedded FPGA

GEMM and SPMV kernels are designed following the NDRange OpenCL paradigm.

4.1.1 GEMM implementation

The kernel is based on the tiled matrix multiplication algorithm, where the input matrices (A and B) are subdivided into blocks, and these blocks are multiplied by generating partial results, which are accumulated until completing the corresponding block of the solution matrix (C). This computational method enhances data locality and thus optimizes memory accesses.

The implementation follows the OpenCL 2–dimensional NDRange model. Each work–item within a workgroup loads an element of matrices A and B into local memory. When the blocks of size $N_b \times N_b$ are complete (the load is synchronized using barriers), the block-wise product is carried out, and the partial results are accumulated. After the blocks are completely consumed, the subsequent blocks are loaded and consumed until the calculation of a block of the matrix C is completed. Each work-item within the workgroup is responsible for computing an element of the matrix C. This means the global work size is the same as the elements in C. See pseudocode in Listing 1.

For a given block, each work-item performs N_b products, which are done in parallel using loop unrolling. Another degree of parallelism is added by using kernel vectorization, which adds work-item parallelism. Increasing the size of the blocks and adding vectorization improves the computing performance (as long as the memory bandwidth limit is not reached) but has a substantial impact on the number of resources used.

```
1  sum = 0.0f;
2  for...//Iterate over all blocks of size
       NbxNb
3  {
4      //Load block elements to local memory
5      A_mem[] = A[];
6      B_mem[] = B[];
7      //Wait for the entire block to be
       loaded
8      barrier();
9      //Dot product accumulation within
       this block
10     #pragma unroll
11     for (int k = 0; k < Nb; ++k)
12         sum += A_mem[][k] * B_mem[][k];
13     //Wait for the block to be fully
       consumed
14     barrier();
15 }
16 //Store result in matrix C
17 C[] = sum;
```

Listing 1: Pseudocode of GEMM OpenCL implementation

We perform a design space exploration by varying both parameters (block size and vectorization) to determine the best combination. Unfortunately, our FPGA has relatively low resources and only allows for a moderate increase of both parameters. The combination of parameters that we could synthesize in the FPGA and achieved the best execution time were $N_b = 8$ and $SIMD = 4$. More parallelism could be added by replicating the pipeline (Compute Unit replication), but it has a similar effect to that of kernel vectorization and consumes even more resources.

4.1.2 SPMV implementation

The SPMV kernel is based on the Spector OpenCL benchmark suite [19], which provides a mechanism to perform a design space exploration of the different optimization parameters. The kernel uses the 1-dimensional NDRange model. It processes one row per work-item, following the row-based strategy used in the massively parallel context (as the *scalar* version in [20]), allowing the computation of multiple rows in parallel (controlled by the workgroup size parameter). More parallelism is added by unrolling the loops that read the data from global into local memory and adding vectorization. Finally, it is possible to replicate the pipeline using Compute Unit replication.

We compiled kernels with different parameters and tested their performance on all the matrices, selecting the kernel that yielded the best execution time.

4.2 High-End FPGA

For the Xilinx's Data Center platform we tested the GEMM implementation from [14] and developed our version of SPMV based on the work in [18].

4.2.1 GEMM implementation

The GEMM is an efficient version of the tiled matrix multiplication, with highly-optimized memory access. It follows a systolic array architecture, where N_p processing elements (PE) consume prefetched elements of the matrices A and B in a stream-like fashion. Each PE holds N_c compute units (CU), and each one of them is capable of producing one output product (a partial result of matrix C) every clock cycle. The design is parametrizable in the number of PEs, the size of PEs (number of CUs), and the tiles' size.

4.2.2 SPMV implementation

The SPMV kernel follows a streaming dataflow architecture, where input data is streamed through a deep pipeline at every clock cycle. The kernel was implemented in such a way that different tasks are compartmentalized into different functions or blocks. These blocks then exchange data through FIFO memories (referred to as Streams in Xilinx HLS). All these functions are then instantiated with the dataflow pragma to allow concurrent execution. One block for each input port gets the data from global memory and places it in its corresponding Stream. The computation block reads data from the Streams, performs the product and accumulation, and writes the results into another Stream. Finally, the output function is in charge of writing the results in global memory. Figure 1 shows a block diagram of the implemented kernel.

The computation block iterates over the *nnz* performing the floating-point product and accumulation of the matrix elements and their corresponding vector elements. When all the elements in a row are processed, the result is sent to the output Stream. This computation enforces a loop-carried dependency between the partial results in each iteration. As the floating-point operation has a latency of L clock cycles in our FPGA and there is a loop-carried dependency in the variable that holds the accumulation, it is not possible to pipeline the loop with an II of 1. In order to overcome this and improve the performance, in each iteration L elements are processed in parallel using loop unrolling, and the II of the pipeline is set to L.

Parallelism is achieved by processing several rows at the same time, instantiating replications of the functions in the dataflow region and adding the corresponding memory buffers and I/O ports. The design takes advantage of the multiple ports to access the High Bandwidth Memory (HBM) available. The vector x is copied into the FPGA's internal RAM (very fast and has very low latency) to avoid stalls due to non-contiguous global memory accesses. In order to run the parallel version of the kernel, the input data must be divided among the host before starting the kernel.

5 Experimental Evaluation

This section presents the experiments conducted to assess the performance of our kernels. First, we describe our computing platforms and the equipment used to evaluate runtime and power consumption. Afterward, we introduce the test cases and the numerical results, accompanied by the corresponding analysis.

5.1 Experimental Setup

The experiments were carried out in two distinct FPGAs:

1. The DE10-nano board from Terasic. This platform is based on a Cyclone V SoC FPGA and includes a dual-core Cortex-A9 processor and around 110K Logic Elements of programmable logic. The board is equipped with 1GB of DDR3 memory, shared between the FPGA and the processor. The FPGA has 6 MB of on-chip memory used as scratch-pad memory and 112 variable precision DSP blocks (22.4 GFLOPS). The kernels were compiled using Intel FPGA SDK for OpenCL v18.1. In the following sections, we refer to this platform as EMBFPGA.

2. The Alveo U50 is an FPGA-based data center acceleration card based on Xilinx UltraScale+ architecture. It includes 8GB of on-board HBM RAM and QSFP28 connections for 100 GbE applications. Its typical power consumption is 75 W, which adds to the high computing capacities and input/output bandwidth and achieves excellent energy efficiencies. It has 872K logical elements, 1,743K registers, 28 MB of internal RAM, and 5,952 DSPs blocks. The kernels for the Alveo platform were compiled using Xilinx Vitis 2019.2. In the following sections, we refer to this platform as HIGHFPGA.

The traditional processor consist of a quad-core Intel i7-4770 CPU @3.40GHz installed with 16 GB of RAM.

To measure runtime and power consumption we use:

- **DE10-nano**: Measured DC input current with FLUKE 45 (4.5 digits, accuracy: 0.2%+6).
- **Alveo U50**: Measured input current and voltage of the board using Xilinx Runtime (XRT) profiling capabilities.

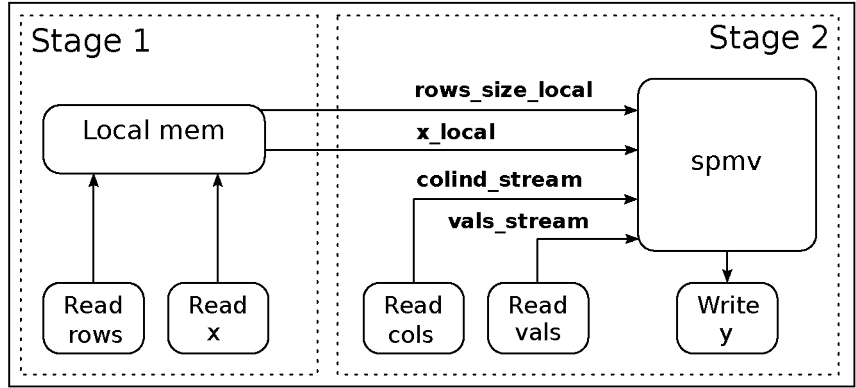


Figure 1: Block diagram of SPMV implementation for HIGHFPGA.

- **Intel CPU**. Two independent measurements, using PMLIB software [21].
  - Processor and memory power consumption using RAPL.
  - Full system power using Zes Zimmer LMG-640 meter (AC Power. High accuracy: 0.015%+1% of range).
- Execution times: obtained using OS libraries in CPU and OpenCL profiling functions in FPGA.

Runtimes were computed as the average value of several measurements in consecutive kernel executions. The number of iterations was determined individually for each test case in order to produce a total runtime above 5 minutes. During the first two minutes no measurements were taken, as its considered the warm-up time. Power consumption was measured once per second during the remaining time.

5.2 Evaluation of dense kernel

In this section, we first show the test cases employed for the experimental evaluation, and later we analyze the obtained results.

5.2.1 Test cases

For the dense kernel ($C := A * B$) evaluation, we employ eight different data sets, composed, in all cases, by random matrices of increasing dimensions, ranging from 64×32 to $8k \times 4k$. The test cases are:

- GEMM1$\rightarrow A_{64\times 32} \times B_{32\times 32}$
- GEMM2$\rightarrow A_{128\times 64} \times B_{64\times 64}$
- GEMM3$\rightarrow A_{256\times 128} \times B_{128\times 128}$
- GEMM4$\rightarrow A_{512\times 256} \times B_{256\times 256}$
- GEMM5$\rightarrow A_{1k\times 512} \times B_{512\times 512}$
- GEMM6$\rightarrow A_{2k\times 1k} \times B_{1k\times 1k}$
- GEMM7$\rightarrow A_{4k\times 2k} \times B_{2k\times 2k}$
- GEMM8$\rightarrow A_{8k\times 4k} \times B_{4k\times 4k}$

5.2.2 Experimental results

Table 1 summarizes the results obtained for runtime and energy consumption. Specifically, we include the runtime and the full–system power and energy consumption yielded by each hardware platform. For the CPU platform, we present both its internal and external power and energy consumption. In contrast, for EMBFPGA we include only the external values (as it is a standalone board), and for the HIGHFPGA we show only the internal ones. There are no results of the smaller matrices for HIGHFPGA because of implementation restrictions (the matrix size cannot be smaller than the tiles' size). For the comparison within CPU and the different FPGA platforms we used the following criteria:

- The EMBFPGA power and energy were compared against CPU external measurements, because the FPGA is an embedded board ad measurements consider not only the FPGA chip but also all the necessary peripherals required by the board.
- For the HIGHFPGA we compared against only internal CPU and memory consumption. This is because both devices require to be installed in a computing node in order to work, so we did not consider the power consumption of all remaining parts, such as motherboard, network cards, hard drive, etc. It may be arguable that the FPGA platform still requires a CPU in order to work, but in the scenario in which some task is offloaded to the FPGA, the CPU would be idle (consuming little power) or performing some other tasks.

On the one hand, the results show that in execution time, the CPU strongly outperforms the EMBFPGA with differences between 20× and 150×. The differences with HIGHFPGA are also in favor of the CPU but only in ratios of approximately 2×. On the other hand, it can be stated that the power consumed by the FPGAs is notoriously less (in the order of 20× for EMBFPGA –comparing external values– and 2× for HIGHFPGA –comparing internal values–)

Table 1: Runtime, power and energy consumption for the tested matrices sizes with the GEMM kernel.

	Variant	Exec T (ms)	P(W)	EnergyExt(mJ)	P(W)	EnergyInt(mJ)
GEMM1	CPU	3.40×10^{-3}	115	3.91×10^{-1}	—	—
	HIGHFPGA	—	—	—	—	—
	EMBFPGA	7.44×10^{-2}	5.35	3.98×10^{-1}	—	—
GEMM2	CPU	8.69×10^{-3}	123	1.07×10^{0}	—	—
	HIGHFPGA	—	—	—	—	—
	EMBFPGA	2.06×10^{-1}	5.80	1.19×10^{0}	—	—
GEMM3	CPU	2.81×10^{-2}	113	3.18×10^{0}	—	—
	HIGHFPGA	—	—	—	—	—
	EMBFPGA	1.24×10^{0}	6.90	8.56×10^{0}	—	—
GEMM4	CPU	1.88×10^{-1}	113	2.12×10^{1}	—	—
	HIGHFPGA	—	—	—	—	—
	EMBFPGA	9.18×10^{0}	7.05	6.47×10^{1}	—	—
GEMM5	CPU	1.48×10^{0}	114	1.69×10^{2}	74.3	1.10×10^{5}
	HIGHFPGA	3.21×10^{0}	—	—	39.5	1.27×10^{5}
	EMBFPGA	7.24×10^{1}	7.20	5.21×10^{2}	—	—
GEMM6	CPU	1.32×10^{1}	121	1.60×10^{3}	78.7	1.04×10^{6}
	HIGHFPGA	2.09×10^{1}	—	—	39.5	8.26×10^{5}
	EMBFPGA	5.79×10^{2}	7.45	4.31×10^{3}	—	—
GEMM7	CPU	9.83×10^{1}	123	1.21×10^{4}	79.7	7.83×10^{6}
	HIGHFPGA	1.52×10^{2}	—	—	39.5	6.00×10^{6}
	EMBFPGA	9.09×10^{3}	6.75	6.14×10^{4}	—	—
GEMM8	CPU	7.56×10^{2}	124	9.37×10^{4}	82.9	6.27×10^{7}
	HIGHFPGA	1.17×10^{3}	—	—	39.5	4.62×10^{7}
	EMBFPGA	1.14×10^{5}	6.05	6.90×10^{5}	—	—

than that consumed by the CPU. These results reveal two completely different scenarios. First, in the case of EMBFPGA, the lower runtimes presented by the CPU variant are enough to also position this variant as the most energy-efficient. Opposite to this, the HIGHFPGA is the most energy-efficient variant for the three larger test cases.

The results also show that the runtime differences between the CPU and the EMBFPGA decrease with the matrices' dimension. Thus, this kind of FPGA is more competitive for small matrices. More in detail, from the standpoint of energy consumption, in the smallest test case (GEMM1) both platforms are comparable. This is a common scenario in the AI field, which usually implies several small matrix multiplications. However, when the target is a medium or large dense problem, the computational power offered by the traditional multi-core CPUs in conjunction with the efficient data access (smart use of their cache levels), make the multi-core platform a better option than the small FPGA. On the other side, the experiments with the HIGHFPGA show that this device allows similar performances to traditional multi-core CPUs (e.g., better energy consumption and comparable runtimes). Additionally, the performance of HIGHFPGA improves for larger problems, i.e. better scalability.

To verify the importance of the efficient use of cache levels in multi-core we include the following experiment. We generate a parallel version of the GEMM kernel in CPU based on the DOT implementation of BLAS library, i.e., a GEMM based on BLAS-1 operations. Following the same principle, we implement a fundamental modification of our EMBFPGA kernel by removing the use of tiles in the matrix product. Figure 2 offers the comparison of the execution times involved in the computation of each test case for both platforms.

At first sight, the most notorious result is the performance reduction of the GEMM kernel in the multi-core CPU, which decreases between 20× and 80×. These results are aligned with other efforts [22], which highlights the importance of properly exploiting the cache memory in the GEMM kernel. Regarding the EMBFPGA results, it can be observed that its performance is reduced only between 3× and 10×. This allows us to conclude that our FPGA tiled variant of GEMM is not capable of exploiting the memory hierarchy as efficiently as the multi-core CPU, which is one reason for the much higher performance presented by the latter.

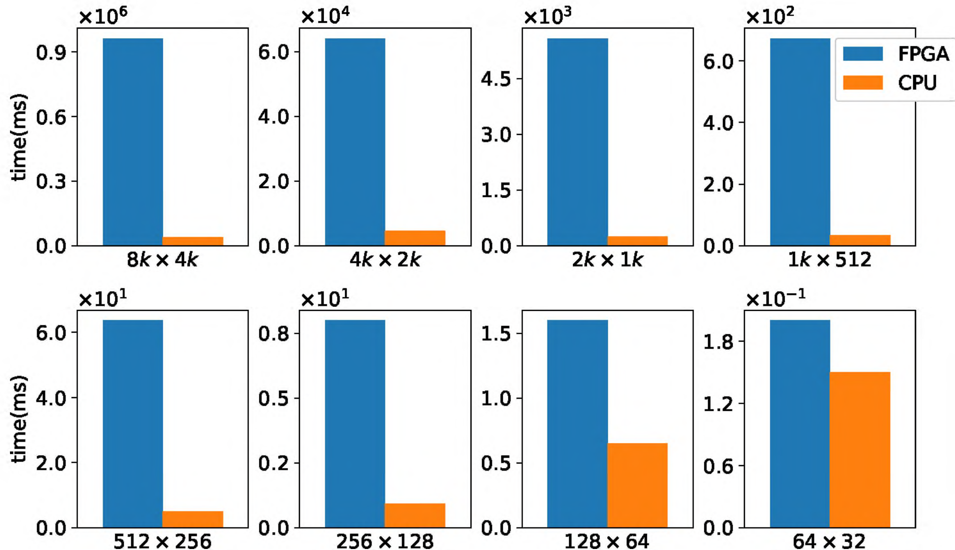


Figure 2: EMBFPGA vs. CPU runtime (in ms) for basic versions of GEMM.

5.3 Evaluation of sparse kernel

In this section, we present the test cases employed in the experimental evaluation of the sparse scenarios, and later we analyze and discuss the experimental results.

5.3.1 Test cases

We employ several matrices from the SuiteSparse Matrix Collection (formerly the UF Sparse Matrix Collection) for the sparse context. We select 12 matrices with similar dimensions that range from 17,000 to 40,000, with a highly different number of non-zero coefficients (*nnz*), i.e., *nnz* between 14,765 and 16,171,169, as well as different distribution of these coefficients. Table 2 summarizes the information related to the employed matrices.

Table 2: Number of rows (n) and non-zero elements (*nnz*) of the sparse matrices.

Matrix	n	*nnz*
bcsstm37	25503	14765
bcsstm35	30237	18211
qpband	20000	30000
chipcool0	20082	281150
gyro_k	17361	519260
Godwin_40	17922	561677
TSOPF162	20374	812749
thread	29736	2249892
TSOPF_RS_b300	28338	2943887
ndk	18000	3457658
TSOPF_RS_b2052	25626	6761100
TSOPF_RS_b2383	38120	16171169

5.3.2 Parametrization of FPGA versions

In this section, we study the optimization of parameters required by our FPGA implementation of the sparse kernel.

EMBFPGA *configuration:* The parameters included in the design space exploration were: the *work-group size* (BS), the number of Compute Units (CU), the Unroll Factor (UF) and the vectorization (VC). Different combinations of the mentioned parameters were explored. These were: **BS:** 2, 16, 32 and 64; **CU:** 1 and 2; **UF:** 2, 4 and 8; **VC:** 2, 4 and 8.

Some combinations could not be synthesized because of resource restrictions of the FPGA platform. After measuring the runtime of all the synthesized kernels for all matrices, the ones with the best performance were selected for each matrix. The parameters used in the selected kernels and the resource utilization (as a percentage) and the clock operating frequency are shown in Table 3.

HIGHFPGA *configuration:* The only available parameters for this kernel are the Initiation Interval of the compute kernel and the number of rows to be processed in parallel. The first parameter was set to 4, which allows running the kernel at around 200 Mhz of clock frequency. The last parameter is limited by the amount of available internal memory of the device, as a copy of the vector x is required to process each row. For this experiment, the number of rows to be processed in parallel is 4.

5.3.3 Experimental results

The obtained results for the sparse context on the best performing kernels are presented in Table 4. For the EMBFPGA, the first two results are aligned with the dense evaluation: i) the CPU version outperforms the FPGA implementation in runtime and ii) the power

Table 3: Parameters, resource utilization and clock operating frequency of best performing kernels for each matrix in EMBFPGA.

Problem	Parameters				Resource Utilization (%)				Fclk (MHz)
	BS	CU	UF	VC	ALMs	FFs	RAMs	DSPs	
bcsstm37 bcsstm35 qpband	16	2	2	1	45.6	26.5	53.6	5.3	111
chipcool0	2	2	2	1	45.5	26.6	53.6	5.3	111
gyro_k Goodwin_40	64	1	2	2	33.2	19.0	34.0	4.4	120
TSOPF_RS_b162	16	2	2	1	45.6	26.5	53.6	5.3	111
thread	16	1	2	4	45.9	27.1	42.1	8.0	115
TSOPF_RS_b300	32	1	4	1	33.8	19.4	34.3	4.4	120
nd6k	64	1	8	2	78.8	44.5	72.5	15.1	107
TSOPF_RS_b2052	2	1	4	4	78.6	44.4	71.5	15.1	95
TSOPF_RS_b2383	2	1	2	4	45.8	27.0	42.1	8.0	112

consumed by the FPGA is notoriously less than that of the CPU.

Another result that can be highlighted is the direct relation between the *nnz* of the matrix and the associated runtimes. Specifically, there is a linear relation (after a concrete threshold) between the *nnz* of each matrix and the runtime. This behavior is completely aligned with the theory and is expected for the SPMV because it is a memory-bound operation [23].

From the energy consumption perspective, the SPMV kernel presents varied results. When $nnz(A)$ is small, the CPU implementation notoriously consumes less energy than the FPGA counterparts. However, when $nnz(A)$ increases, the FPGA variants become competitive, even offering important reductions in the energy consumption for the largest cases in the EMBFPGA. See Figure 3a. These values are based on two contrasting situations. On the one hand, the power required by the FPGA increases with the *nnz* of each matrix. However, in the CPU case, the behavior is the opposite (i.e., large matrices require less power). The under-utilization of the CPU resources could explain this situation. On the other hand, the runtime differences between platforms for small matrices are considerable, with values higher than 100×, while for the larger matrices, these gaps decrease to approximately 10×.

Finally, in the HIGHFPGA case, the power consumption reduction reached by the FPGA does not still compensate the runtime differences between both hardware platforms, see Figure 3b. However, we need to remark that we are comparing only internal energy in this case, which is not entirely fair with the FPGA device.

6 Concluding remarks and Future work

We have explored using traditional multi-core hardware and FPGAs to address Numerical Linear Algebra (NLA) kernels. Specifically, we studied the behavior of highly tuned kernels in a multi-core CPU processor and HLS implementations over FPGAs, analyzing the execution time and the energy consumption. For this purpose, we employed the most critical kernels from dense and sparse NLA fields, i.e., the GEMM and SPMV operations. Our work includes a brief introduction to the use of FPGAs and HLS, such as OpenCL and a revision of related work in this field. Additionally, we design and fit the implementation of the two NLA kernels in two contrasting platforms from the major FPGA vendors, Xilinx and Intel. The selected platforms are the embedded DE10–nano board from Terasic and the modern Alveo U50 from Xilinx.

The experimental evaluation shows that, for dense problems, the CPU strongly outperforms the FPGA variants in runtime, showing a higher energy efficiency than the small FPGA as well. However, in the sparse case, the differences in runtime are more modest, and the performance in terms of energy consumption depends on the matrix characteristics. Specifically, for matrices with many nonzeros, the EMBFPGA offers substantial reductions in energy consumption.

As part of future work, we plan to advance in three distinct directions:

- Exploring other paradigms of parallelism to address the SPMV operation.
- Employing HDL tools to improve the performance of FPGA kernels.
- Advancing in the automatic prediction of which hardware platform will offer the best performance regarding both runtime and energy consumption.

Competing interests

The authors have declared that no competing interests exist.

Table 4: Runtime, power and energy consumption for the tested matrices with the SPMV kernels.

	Variant	Exec T (ms)	P(W)	EnergyExt(mJ)	P(W)	EnergyInt(mJ)
bcsstm37	CPU	1.10×10^{-2}	125.9	1.38×10^{0}	73.1	8.04×10^{-1}
	HIGHFPGA	1.59×10^{-1}	—	—	19.9	3.16×10^{0}
	EMBFPGA	1.36×10^{0}	5.7	7.75×10^{0}	—	—
bcsstm35	CPU	1.20×10^{-2}	128.7	1.54×10^{0}	73.7	8.84×10^{-1}
	HIGHFPGA	2.26×10^{-1}	—	—	20.0	4.52×10^{0}
	EMBFPGA	1.43×10^{0}	5.7	8.15×10^{0}	—	—
qpband	CPU	1.50×10^{-2}	115.3	1.73×10^{0}	62.9	9.44×10^{-1}
	HIGHFPGA	1.57×10^{-1}	—	—	19.8	3.11×10^{0}
	EMBFPGA	1.33×10^{0}	5.8	7.65×10^{0}	—	—
chipcool0	CPU	6.00×10^{-2}	127.0	7.62×10^{0}	72.9	4.37×10^{0}
	HIGHFPGA	5.00×10^{-1}	—	—	20.5	1.03×10^{1}
	EMBFPGA	3.00×10^{0}	6.2	1.85×10^{1}	—	—
gyro_k	CPU	7.10×10^{-2}	132.6	9.41×10^{0}	76.8	5.45×10^{0}
	HIGHFPGA	7.56×10^{-1}	—	—	20.5	1.55×10^{1}
	EMBFPGA	4.15×10^{0}	6.0	2.49×10^{1}	—	—
Goodwin_40	CPU	6.90×10^{-2}	132.1	9.11×10^{0}	80.3	5.54×10^{0}
	HIGHFPGA	8.02×10^{-1}	—	—	20.4	1.64×10^{1}
	EMBFPGA	4.39×10^{0}	6.0	2.63×10^{1}	—	—
TSOPF_RS_b162	CPU	1.65×10^{-1}	119.3	1.97×10^{1}	30.3	1.04×10^{1}
	HIGHFPGA	1.09×10^{0}	—	—	20.5	2.23×10^{1}
	EMBFPGA	5.62×10^{0}	6.3	3.51×10^{1}	—	—
thread	CPU	7.86×10^{-1}	112.4	8.83×10^{1}	31.7	4.10×10^{1}
	HIGHFPGA	2.73×10^{0}	—	—	20.6	5.62×10^{1}
	EMBFPGA	1.60×10^{1}	6.4	1.02×10^{2}	—	—
TSOPF_RS_b300	CPU	1.62×10^{0}	80.7	1.31×10^{2}	32.1	5.54×10^{1}
	HIGHFPGA	3.46×10^{0}	—	—	20.5	7.09×10^{1}
	EMBFPGA	1.56×10^{1}	6.1	9.52×10^{1}	—	—
nd6k	CPU	1.58×10^{0}	96.1	1.52×10^{2}	31.2	6.35×10^{1}
	HIGHFPGA	3.96×10^{0}	—	—	20.6	8.16×10^{1}
	EMBFPGA	2.43×10^{1}	6.8	1.65×10^{2}	—	—
TSOPF_RS_b2052	CPU	3.78×10^{0}	80.2	3.03×10^{2}	30.9	1.19×10^{2}
	HIGHFPGA	7.58×10^{0}	—	—	20.7	1.57×10^{2}
	EMBFPGA	3.48×10^{1}	6.7	2.31×10^{2}	—	—
TSOPF_RS_b2383	CPU	9.13×10^{0}	79.0	7.21×10^{2}	30.7	2.83×10^{2}
	HIGHFPGA	1.79×10^{1}	—	—	21.3	3.81×10^{2}
	EMBFPGA	9.30×10^{1}	6.4	5.95×10^{2}	—	—

Authors' contribution

FF wrote the program, conducted the experiments, analyzed the results and wrote the manuscript; PE conceived the idea and analyzed the results; ED, PE and JPO analyzed the results and revised the manuscript. All authors read and approved the final manuscript.

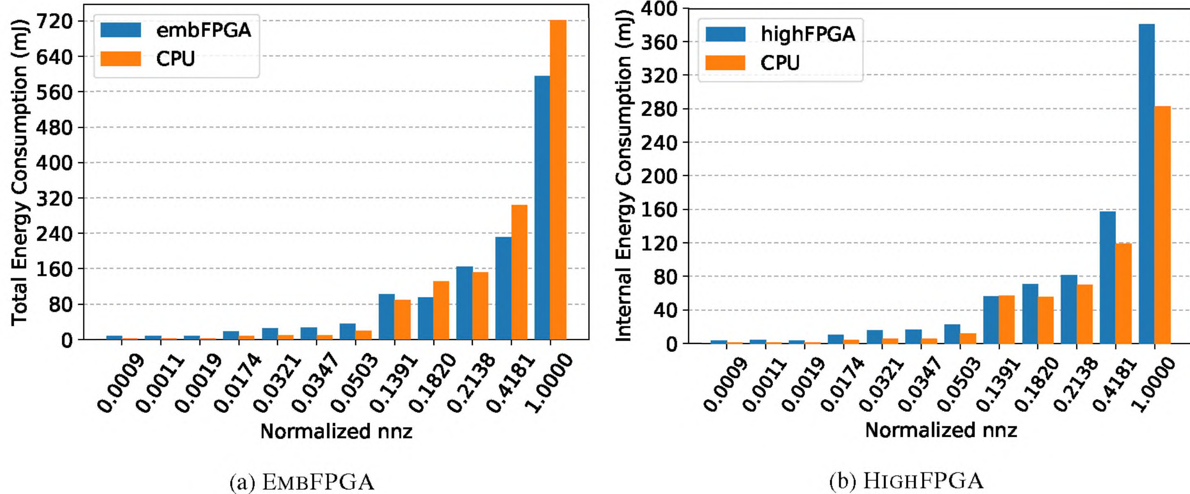
Acknowledgements

We acknowledge the ANII – MPG Independent Research Groups: "Efficient Hetergenous Computing" with the CSC group.

References

[1] P. Ezzatti, E. S. Quintana-Ortí, A. Remón, and J. Saak, "Power-aware computing," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 6, p. e5034, 2019. e5034 cpe.5034.

[2] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From opencl to high-performance hardware on fpgas," in *22nd international conference on field programmable logic and applications (FPL)*, pp. 531–534, IEEE, 2012.



(a) EMBFPGA

(b) HIGHFPGA

Figure 3: Full system (a) and internal (b) energy consumption (in mJ) of EMBFPGA, HIGHFPGA and CPU versus normalized *nnz* for the tested sparse matrices. The normalization is performed over the largest *nnz* value.

[3] L. Wirbel, "Xilinx sdaccel: a unified development environment for tomorrows data center," *The Linley Group Inc*, 2014.

[4] G. H. Golub and C. F. V. Loan, *Matrix Computations*. Baltimore: The Johns Hopkins University Press, 2013.

[5] P. Benner, P. Ezzatti, E. Quintana-Ortí, and A. Remón, "On the impact of optimization on the time-power-energy balance of dense linear algebra factorizations," in *Algorithms and Architectures for Parallel Processing* (R. Aversa, J. Kołodziej, J. Zhang, F. Amato, and G. Fortino, eds.), (Cham), pp. 3–10, Springer International Publishing, 2013.

[6] F. Favaro, J. P. Oliver, E. Dufrechou, and P. Ezzatti, "Understanding the performance of elementary NLA kernels in fpgas," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2020, New Orleans, LA, USA, May 18-22, 2020*, pp. 479–482, IEEE, 2020.

[7] P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintana-Ortí, T. Rhodes, R. A. van de Geijn, and F. G. Van Zee, "Deriving dense linear algebra libraries," *Formal Aspects of Computing*, vol. 25, pp. 933–945, Nov 2013.

[8] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 16, pp. 1–17, Mar. 1990.

[9] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2nd ed., 2003.

[10] S. Kestur, J. D. Davis, and O. Williams, "BLAS Comparison on FPGA, CPU and GPU," in *2010 IEEE Computer Society Annual Symposium on VLSI*, pp. 288–293, July 2010.

[11] 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, April 2012.

[12] H. Giefers, R. Polig, and C. Hagleitner, "Analyzing the energy-efficiency of dense linear algebra kernels by power-profiling a hybrid CPU/FPGA system," in *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, pp. 92–99, June 2014.

[13] Y. Tan and T. Imamura, "Performance evaluation and tuning of an opencl based matrix multiplier," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pp. 107–113, 2018.

[14] J. de Fine Licht, G. Kwasniewski, and T. Hoefler, "Flexible communication avoiding matrix multiplication on fpga with high-level synthesis," in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 244–254, 2020.

[15] N. Kapre and A. DeHon, "Parallelizing sparse matrix solve for spice circuit simulation using fpgas," in *2009 International Conference on Field-Programmable Technology*, pp. 190–198, Dec 2009.

[16] T. Nechma and M. Zwolinski, "Parallel sparse matrix solution for circuit simulation on fpgas," *IEEE Transactions on Computers*, vol. 64, pp. 1090–1103, April 2015.

[17] F. Favaro, E. Dufrechou, P. Ezzatti, and J. P. Oliver, "Exploring FPGA optimizations to compute sparse numerical linear algebra kernels," in *Applied Reconfigurable Computing. Architectures, Tools, and Applications - 16th International Symposium, ARC 2020, Toledo, Spain, April 1-3, 2020, Proceedings [postponed]* (F. Rincón, J. Barba, H. K. So, P. C. Diniz, and J. Caba, eds.), vol. 12083 of *Lecture Notes in Computer Science*, pp. 258–268, Springer, 2020.

[18] M. Hosseinabady and J. L. Nunez-Yanez, "A streaming dataflow engine for sparse matrix-vector multiplication using high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 6, pp. 1272–1285, 2019.

[19] Q. Gautier, A. Althoff, P. Meng, and R. Kastner, "Spector: An opencl fpga benchmark suite," in *2016 Interna-*

tional Conference on Field-Programmable Technology (FPT), pp. 141–148, Dec 2016.

[20] N. Bell and M. Garland, “Implementing sparse matrix-vector multiplication on throughput-oriented processors,” in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC ’09, (New York, NY, USA), Association for Computing Machinery, 2009.

[21] S. Barrachina, M. Barreda, S. Catalán, M. F. Dolz, G. Fabregat, R. Mayo, and E. Quintana-Ortí, “An integrated framework for power-performance analysis of parallel scientific workloads,” *Energy*, pp. 114–119, 2013.

[22] P. Benner, P. Ezzatti, E. S. Quintana-Ortí, A. Remón, and J. P. Silva, “Tuning the blocksize for dense linear algebra factorization routines with the roofline model,” in *Algorithms and Architectures for Parallel Processing* (J. Carretero, J. Garcia-Blas, V. Gergel, V. Voevodin, I. Meyerov, J. A. Rico-Gallego, J. C. Díaz-Martín, P. Alonso, J. Durillo, J. D. Garcia Sánchez, A. L. Lastovetsky, F. Marozzo, Q. Liu, Z. A. Bhuiyan, K. Fürlinger, J. Weidendorfer, and J. Gracia, eds.), (Cham), pp. 18–29, Springer International Publishing, 2016.

[23] S. Williams, A. Waterman, and D. Patterson, “Roofline: An insightful visual performance model for multicore architectures,” *Commun. ACM*, vol. 52, pp. 65–76, Apr. 2009.

Citation: F. Favaro, E. Dufrechou, P. Ezzatti and J.P. Oliver. *Energy-Efficient Algebra Kernels in FPGA for High Performance Computing*. Journal of Computer Science & Technology, vol. 21, no. 2, pp. 80–92, 2021.
DOI:10.24215/16666038.21.e09
Received: April 9, 2021 **Accepted:** September 7, 2021.
Copyright: This article is distributed under the terms of the Creative Commons License CC-BY-NC.