Exploring FPGA optimizations to compute sparse Numerical Linear Algebra kernels

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

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

Abstract The solution of sparse triangular linear systems (the SPTRSV kernel) presents itself as the computational bottleneck of many numerical methods, and therefore is one of the most important building blocks in sparse Numerical Linear Algebra. For this reason, it is crucial to count with efficient implementations of such kernel, at least for the hardware platforms that are more commonly used for numerical computations. In this sense, Field–Programmable Gate Arrays (FPGAs) have evolved greatly in the last years, entering the HPC hardware ecosystem mostly due to their superior energy–efficiency relative to more established accelerators. Up until recently, the design of a solution for FPGA implied the use of low–level Hardware Description Languages (HDL) such as VHDL or Verilog. Nowadays, manufacturers are making a large effort to adopt High Level Synthesis languages like C/C++, System C or OpenCL, but the gap between their performance and that of HDLs is not yet fully studied. This work focuses on the performance offered by FPGAs to compute the SPTRSV using OpenCL. For this purpose, we implement different parallel variants of this kernel and experimentally evaluate several optimization setups, varying parameters such as the work–group size, the number of compute units, the unroll–factor and the vectorizacion–factor.

Keywords: FPGAs, sparse linear algebra, sparse triangular linear systems, power consumption

1 Introduction

Many numerical methods in engineering and scientific applications entail the solution of sparse triangular linear systems (SPTRSV kernel). A typical example is the solution of general sparse linear systems of equations by means of direct methods, or using iterative methods combined with incomplete LU preconditioners, where the SPTRSV kernel is the most computationally costly stage of the whole process [7,15]. This situation motivates the development of efficient implementations of such kernel, at least for the hardware platforms that are more commonly used for numerical computations.

The efficient parallelization of this kernel is specially difficult. Similar to other sparse linear algebra kernels, the SPTRSV is a highly memory–bound operation and presents an irregular data access pattern. In addition, the triangular structure of the nonzero entries is tied to load imbalance between threads when organizing the computations by rows or columns, and the data dependencies between equations severely constrain the available parallelism.

Propelled by the popularization of using massive-parallel devices such as GPUs and Intel Xeon Phi processors for scientific computations, High Performance Computing (HPC) hardware platforms experienced a revolution in the last decade. As a product of this revolution and the massive scale of nowadays clusters and supercomputers, there is a growing concern in the HPC community about the energy consumption of hardware and the efficiency of computing devices [1,3,8,5,12]. It is in this context that Field–Programmable Gate Arrays (FPGAs) have renewed their importance, emerging as a low-energy-consuming alternative to other hardware accelerators. As a result, the former use of FPGAs in highly specialized niches of application is now expanded to address general purpose problems.

One of the major drawbacks that prevented the massive use of FPGAs was the mandatory use of low–level Hardware Description Languages (HDL) such as VHDL or Verilog. These impose a radically different programming model than standard programming languages, with longer development periods and complex debugging. Furthermore, their use requires specialized knowledge of the underlying hardware, which explains why FPGAs are not massively adopted by the HPC community. To overcome this disadvantage, manufacturers are making efforts to adopt High Level Synthesis languages like C/C++, System C or OpenCL. The most relevant evidence of this is the introduction of SDKs for OpenCL by prominent FPGAs manufacturers such as Intel [6] and Xilinx [23].

OpenCL is an open–source, royalty– free parallel programming standard, which allows describing task parallelism using an abstract model independent of the underlying hardware. It considerably reduces development times and allows portability between platforms. Although this has enabled a greater adoption of FPGAs as hardware acceleration by the software community, there is still much to investigate regarding the performance attainable by OpenCL kernels in FPGAs, the role played by specific platform optimizations and how much knowledge of the underlying hardware is required.

In this effort we implement different parallel variants of the SPTRSV kernel for the FPGA using OpenCL. Additionally, we perform a deep evaluation of several FPGA optimization techniques. Concretely, the major contributions of this article are:

- The implementation of a NDRange and a Single Work–Item variant of the SPTRSV kernel. To the best of our knowledge, this is the first implementation of the SPTRSV for FPGAs, excluding our preliminar effort in the Power–Aware Computing – PACO2019 [13].

- The experimental evaluation of several FPGA optimizations, such as the use of threads, vectorization and unrolling, for each parallelism paradigm, employing a set of real problems extracted from the SuiteSparse Matrix Collection[3] (formerly the University of Florida Sparse Matrix Collection).
- The advance towards a characterization of the performance of sparse problems in FPGAs, that allows to choose the computational method and the optimization set efficiently.

The rest of the article is structured as follows. Section 2 summarizes the main aspects related with the solution of sparse triangular linear systems as well as the use of OpenCL to compute with the FPGAs. Later, in Section 3 we describe our proposal to estimate the degree of parallelism offered by the sparse triangular linear systems. The experimental evaluation performed follows in Section 4. Finally, in Section 5, we present the main conclusions arrived at in this work and discuss future lines of work.

2 Work context

This section presents the theorical background of our work. It starts with a mathematical description of the SPTRSV kernel and its parallelization, followed by a brief introduction to FPGAs, with special attention to the OpenCL framework. The section closes with a summary of related work about using FPGAs to tackle NLA operations.

2.1 The SPTRSV kernel

Given a (lower) triangular sparse matrix $L \in \mathbb{R}^{n \times n}$ and a vector $b \in \mathbb{R}^n$, the usual approach to obtain $x \in \mathbb{R}^n$ such that

$$Lx = b, \tag{1}$$

is the procedure known as *forward-substitution*. This procedure is presented in Algorithm 1 for the case where the sparse matrix L is stored in the CSR sparse storage format [11].

The algorithm starts by trivially solving the first equation (with only one unknown) and then, in each step, it replaces the solved unkowns by their values in the following equations, solving at least one equation per step. To obtain the unknown x_i it is necessary to multiply the sub-diagonal entries l_{ij} of row i by the value of x_j, subtracting the result from b_i and dividing by the diagonal entry l_{ii}. It is evident that if l_{ij} is nonzero, the unknown x_j needs to be solved before x_i, which constrains the parallelism of the operation.

[3] http://faculty.cse.tamu.edu/davis/suitesparse.html

Algorithm 1: Serial solution of sparse lower triangular systems for matrices stored in the CSR format. The vector *val* stores the nonzero values of L by row, while row_ptr stores the indices that correspond to the beginning of each row in vector *val*, and col_idx stores the column index of each element in the original matrix. The nonzero elements of each row are ordered by column index.

```
1 Input: row_ptr, col_idx, val, b
2 Output: x
    x = b
    for i = 0 to n − 1 do
      for j = row_ptr[i] to row_ptr[i + 1] − 2 do
        x[i] = x[i] − val[j] × x[col_idx[j]]
      end for
      x[i] = x[i]/val[row_ptr[i + 1] − 1]
    end for
```

2.2 Developing in FPGAs

Unlike other heterogeneous HPC hardware platforms (for example GPUs), FPGAs have no pre–designed high level architecture. They are composed by a matrix of configurable logic blocks (logic elements) and hard–coded blocks (such as memories, hardware multipliers and clock managers) connected through a reconfigurable routing structure. As there is no object code running over a general processor inside the FPGA, they are not "programmable" in a software sense. In fact, a real electrical circuit is synthesized inside the device, interconnecting the configurable and hard coded blocks, which allows to exploit fine–grained parallelism with a very low latency.

FPGAs are usually seen as a middle–ground between Application Specific Integrated Circuits (ASICs) and general purpose processors. The main difference with ASICs is that FPGAs can be reprogrammed after the manufacturing process. FPGAs clock operating frequency depends on the synthesized circuit but, to support reconfigurability, it is usually lower if compared to other heterogeneous devices. FPGAs also offer less peak floating point performance than GPUs and less memory bandwidth, but this may change shortly, as FPGAs manufacturers are making efforts to compete with GPUs performance.

Traditionally, FPGAs have been a good option for fixed–point, dataflow streaming applications, were they achieve high speeds with a power consumption considerably lower to that of other co-processors. However, they have been generally disregarded as HPC computing platforms due to factors such as their poor performance in floating point arithmetic, the complex development process, and difficult integration with other processors. This tendency is currently changing, as modern high–end FPGAs integrate up to millions of logic elements and thousands of DSP blocks (that provide TFLOP peformance), with a high memory bandwidth. These improvements, combined with the emergence of HLS

tools that facilitate the development, contribute to their adoption in the HPC domain.

OpenCL framework

In the past years, FPGA manufacturers have made big efforts to improve their HLS tools in order to gain more acceptance in the software community. Among the available frameworks, OpenCL is an interesting alternative because as it is a widely used, open–source, cross–platform standard. As such, it has a vast user community and allows the reuse of the code from other heterogeneous platforms.

The OpenCL framework is based on a platform–independent API that provides an abstraction model of the underlying hardware. In the typical scenario, there is a host (generally a traditional CPU) that offloads computing–intensive tasks to one or more parallel devices, hiding the complexity of controlling and communicating with the accelerator to the programmer. The device code is a C like language and its called kernel code.

In the OpenCL model, a device consist on multiple Compute Units (CU), each one having several Procesing Elements (PEs). In turn, the programming model defines a single thread as a work–item, and groups them into work–groups. Work–items in a work–group, as well as multiple work–groups can be executed in parallel. The number of threads and work–groups that can run per CU and PEs is platform dependent.

OpenCL also defines a memory model, in which every work–item has its own private memory and shares local memory with the rest of the threads in the work–group. The only memory shared between work–groups is global memory, which is usually off–chip, slow, and abundant (in the order of Gbytes). In contrast, local memory is scarce but much faster.

The number of threads per work–group is referred to as *local_work_size* and the total number of work–items required to solve a given task (execute a kernel to completion) is called *global_work_size*. The standard organizes the work–items in a work–group in up to three dimensions, in which each work–item is assigned an index 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. The OpenCL SDK allows to greatly reduce the development time, at the cost of some performance degradation.

There are some minor but relevant differences in the OpenCL approach between the usual processor based heterogeneous device (i.e. GPUs) and the FPGAs. For example, Kernels cannot be compiled at runtime because of the long compilation times (up to several hours) of the FPGA. Other difference is that as opposed as the NDRange programming model suggest, the FPGAs does not

provide real thread–level parallelism by default, instead it creates a deep pipeline which processes all work–items one after the other. Thread–level parallelism can be achieved through kernel vectorization, which increases the width of the data pipeline and process 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 is usually more resource-demanding.

In addition to NDRange, Intel introduces a different programming model called *Single Work–Item*. Instead of using of work–groups and work–items, the model structures the kernel to be executed as a single thread, were every loop is pipelined. It is the equivalent of launching a kernel with NDRange size of (1,1,1). This model adapts better to the FPGA architecture.

In general, an OpenCL NDRange kernel would launch multiple work–items to work in parallel. This model does not allow to easily share fine–grained data between threads, so Intel recommends structuring the kernel as a *Single Work–Item* for better performance.

To obtain high–throughput execution in *Single Work–Item* kernels, the compiler processes multiple loop iterations in parallel. This is achieved by pipelining the iterations of loops. In order to maximize performance, loop iterations must be processed with no delay between each other. The number of clock cycles between two consecutive loop iterations is called Initiation Interval (II). For better performance, an II of 1 is desired.

In the *Single Work–Item* model, higher parallelism can be obtained at iteration–level by using *loop unrolling*. This reduces the number of iterations in a loop at the expense of more resources.

2.3 NLA in FPGAs

Several efforts have addressed the use of FPGAs to process NLA operations, tipically considering both, performance and energy consumption. The most spread works are focused on dense NLA kernels. Some of the most prominent works are the Kestur et al. [17,16] that advance with some BLAS [9] operations, the [20] for the general matrix–matrix multplication using OpenCL, and more recently, the both efforts for achieve a BLAS implementation in FPGAs, i.e. fBLAS [18] and Vitis BLAS [2].

In the sparse counterpart, the SpMV operation is the tipically studied kernel (or other solvers building over this kernel). The importnace of this kernel on the one hand, and, in the other hand, the low–level data dependecies in comparison with other sparse NLA kernels (e.g. sptrsv), make this operation attractive to implement in HPC hardware. Some highlight efforts in this directions are the J. Fowers et al. article [14], the Umuroglu and Jahre effort [22] and the thesis of K. Townsend [21].

3 Proposal

In this section we detail our process for the development of the sptrsv kernels. Our effort is focused in the use of the OpenCL framework, and in particular

we employ two different OpenCL paradigms, NDRange and Single Work Item Kernels. For each paradigm we included a set of optimization techniques. We describe the most important ones.

3.1 NDRange Kernels

Intel recommends structuring the kernels as *Single Work–Item* whenever possible, in order to benefit from the coarse–grained parallelism available in this model. However, there may be cases where the explicit definition of concurrent threads of the NDRange model may be beneficial. In particular, when data or memory dependencies prevent achieving low II values. Given the nature of the parallel SPTRSV algorithm, where there is plenty of indirection in memory accesses, we first opted for the NDRange approach. Moreover, we choose as a starting point a GPU implementation following a similar model where good results were obtained.

We developed two NDRange kernels for the parallel SPTRSV, both based on the level–set approach. One work–group is issued for every row, following the order of the *iorder* vector (array containing the rows ordered by levels). Each work–item enters a for loop where it fetches one element from the matrix and its corresponding x vector element, performs the product and accumulation, and then moves forward *local_group_size* elements to process another pair. The loop iterates until all the non–zero elements of the row are processed.

Before processing a row, the algorithm requires that all its dependencies are resolved. The two kernels resolve this matter differently. In the first variant, referred to as NDR_{wait}, each work–item reads form memory its corresponding x element and verifies if it is solved. If it is, it performs the multiplication with its corresponding matrix element and sets the corresponding flag. When the flags from all work–items within a work–group are set, all work–items move forward to process the next *local_work_size* elements. When all elements in a row are ready, the accumulated products of each work–item are reduced and then the x value is obtained by subtracting the accumulated products to the b value and then dividing the result by the diagonal element.

To verify that x is solved, the array is preloaded with the float representation of $infinit$ before executing the kernel. This allows the kernel to determine if the x value is ready by checking if its value differs from $infinit$. The accumulated products and flags are stored in local memory in order to be visible by all threads. Threads within each work–group are synchronized using barriers.

The Intel SDK for OpenCL does not provide reduction functions across work–items, so this operation was implemented manually using operations to local memory and barriers.

As the order in which work–groups are issued is not defined, it is necessary to maintain a counter, in global memory, to go through the *iorder* vector in the correct sequence. To avoid race conditions between work–groups in the writing of this counter an atomic add is required.

In an effort so simplify the kernel and obtain a better performance a second version was explored, in which there is no need to perform the verification on the

x values. Instead, the kernel is launched as many times as levels are, meaning that each execution process only the rows on a given level. As long as the kernels are executed following the order of the *iorder* vector, it is guaranteed that all dependencies are met for each row. All kernel executions work on the same x vector, and after each execution the x values are updated in global memory. We refer to the second NDRange kernel as NDR_{multi}.

In order to control which rows are processed on each run, the *ilevels* vector must be used. This vector contains indexes pointing to the first row of each level in the *iorder* vector. By eliminating the verification on the x values, this version allows to structure the kernel in a way that the loops for memory access and the computations loop can be partially unrolled.

For both kernels, the update of the x values is performed by only one thread per work–group.

3.2 Single Work Item Kernels

For the *Single Work–Item* approach we tested three different versions: SWI_{simple}, $\text{SWI}_{channel}$ and SWI_{hash}.

The SWI_{simple} is a rather naive implementation that consist on three nested loops, where the outermost iterates over the number of levels, the middle one over the number of rows per level and the innermost over the non–zero elements of each row. The innermost loops is where computations are performed. In order to add parallelism this loop is partially unrolled.

Memory dependencies over x values prevent this kernel from being fully pipelined with II equal to 1.

For the $\text{SWI}_{channel}$ version two kernels are used. One is responsible for all memory read and write operations, and the other implements the calculations. The kernels communicate with each other using channels. These are mechanism for passing data between kernels and synchronizing kernels with high efficiency and low latency. Three channels are used, one to send the number of non–zero elements per row, other for exchanging the x values, and the last one for the matrix coefficients. Moving away the memory accesses from the computations allows the kernel to be fully pipelined.

The kernel that performs the read and write to global memory is structured as tree nested loops similarly to the SWI_{simple} version. In order to allow pipelining, the kernel is told to ignore memory dependencies using the *ivdep* pragma in the middle loop. As the outermost loop, which is responsible for issuing the levels one by one, is serialized, this guarantees that dependencies are met for every row.

We improved over this last version by adding a hash to store the solved x. This hash consists on a local memory array that stores a portion of the x values. This allows to access the x values much faster, as opposed to reading them from global memory. We refer to this last version as SWI_{hash}. We actually tested two different versions of the SWI_{hash}, one that updates the x in global memory at the same time it is stored in the hash, and another one that impacts all x values

from the hash to global memory at the end. This last version proved to be faster for most of the matrix tested.

4 Experimental Evaluation

In this section we present the experimental evaluation performed to validate our proposal. It begins with a description of the hardware platform and test cases employed, which is followed by a discussion of the experimental results.

4.1 Experimental platform

The hardware platform employed is a DE10–nano board from Terasic. This FPGA board is based on a Cyclone V SoC form Intel and includes a dual–core Cortex–A9 processor and around 110K Logic Elements of programmable logic. The board is equipped with 1GB of high–speed DDR3 memory shared between the processor and the FPGA. The FPGA has 6 MB of on–chip memory that can be used as scratchpad memory and 112 variable precision DSP blocks (capable of a peak performance of 22.4 GFLOPS).

We used the Intel FPGA SDK for OpenCL v18.1 to compile our kernels.

To measure power consumption we used a FLUKE 45 multimeter (4.5 digits, accuracy: 0.2%+6), automated using PMLIB software [4]. The runtime was obtained by the profiling functions of OpenCL.

4.2 Test cases

To perform the experimental evaluation we used a set of matrices from the SuiteSparse Matrix Collection (formerly the UF Sparse Matrix Collection). We selected 10 matrices with similar dimensions, between 17,000 and 40,000 rows, and large differences in the number of non–zero coefficients (nnz), i.e. nnz between 14,765 and 16,171,169. Table 1 summarizes the characteristics of the matrices used.

Table 1. Number of rows (n), non–zero elements (nnz) and levels of the employed sparse matrices.

Matrix	Called	n	nnz	$levels$
Bcsstm35	BcsS	30237	32645	6
Chipcool0	ChipC	20082	281150	534
Gyro	Gyro	17361	519260	2796
Godwin_40	GodW	17922	561677	739
TSOPF162	T_{162}	20374	812749	114
Thread	Thread	29736	2249892	1446
TSOPF_RS_b300	T_{300}	28338	2943887	112
Ndk	Ndk	18000	3457658	5621
TSOPF_RS_b2052	T_{2052}	25626	6761100	61

4.3 Experimental results

All the runtime results presented in this section are the average of 10 independent executions.

Considering that the static power consumption of our experimental platform is elevated in comparison with the dynamic one, in the first stage of our analysis we focused only in the runtime of the different variants.

Table 2. Runtime (in ms) for the NDR_{multi} variant of SPTRSV kernel with different optimizations.

BS	CU	UF	VW	ChipC	T_{162}	T_{300}	Gyro	GodW	BcsS	T_{2052}	Thread	Ndk
1	1	2	1	30.6	19.0	23.4	216.9	50.6	1.2	39.1	209.3	946.4
1	1	4	1	30.7	15.5	20.9	211.3	49.1	1.3	36.2	192.7	906.3
1	2	2	1	34.3	14.3	26.5	238.4	54.9	**1.1**	37.2	238.0	1121.8
1	2	4	1	31.4	13.0	22.8	221.7	52.0	1.1	35.0	205.5	995.3
2	2	2	1	31.0	12.5	21.8	205.1	48.4	1.3	39.7	172.7	774.4
2	2	2	2	33.5	22.3	60.5	225.7	54.3	1.1	132.2	230.3	1068.2
2	2	4	1	30.7	**12.1**	**20.8**	194.3	47.6	1.3	35.7	162.6	732.8
4	1	2	1	29.3	13.3	22.5	154.2	41.6	2.5	36.4	128.4	528.4
4	1	2	2	30.3	16.4	30.3	164.0	45.0	1.6	59.1	148.9	603.2
4	1	4	1	**29.5**	12.4	22.2	**150.2**	**41.5**	2.5	**33.3**	124.3	472.5
4	1	4	2	30.7	14.5	29.6	163.1	46.6	1.7	54.8	146.5	537.2
4	2	2	1	30.2	13.0	21.6	169.5	44.1	1.7	34.8	141.6	612.6
4	2	4	1	30.9	12.5	21.2	163.2	46.5	1.8	34.3	135.0	542.1
16	1	2	1	41.9	19.5	32.7	150.8	50.2	15.8	45.8	**112.2**	368.4
16	1	2	2	36.5	15.9	34.4	153.0	46.4	8.6	62.3	126.6	396.2
16	1	4	1	42.9	18.9	30.5	155.9	51.9	16.7	43.8	118.8	**361.9**
16	1	4	2	40.0	16.1	32.3	159.7	50.9	9.5	60.4	135.4	411.6

Our first study is for the NDR_{multi} variant. Table 2 presents the runtime attained to solve the different sparse matrices by the NDR_{multi} solver. We explore the use of several OpenCL optimization, particularly:

- BS: the work group size, with values of 1, 2, 4 and 16.
- CU: the number of compute unit, with values of 1 and 2.
- UF: the unroll factor, with values of 1, 2 and 4.
- VW: the vectorizacion (SIMD) factor, with values of 1, 2 and 4.

It should be noted that the vectorization factor must be less or equal than the work group size, i.e. VW $\leq$ BS. Additionally, our board does not allow many combinations of the optimization parameters due to hardware resource restrictions.

In the first place, the experimental results reached by NDR_{multi} variant show that there is not a single configuration of the optimization parameters that obtains the best results for all the matrices. From the work-group size perspective it seems that the higher the cost of solving the system is, the better it is to have

a large BS. The number of compute units (CU), a priori, does not show any recongizable pattern. On the other hand, for the unroll–factor (UF) it appears that higher numbers are better. Most of the best runtimes occur for an unroll–factor of 4 and a few for a value of 2. Finally, the vectorization does not offer any gains, in all cases the variant with a vectorization-factor of 1 outperforms the other options. This is because the compiler is failing to vectorize the memory accesses, since these are not contiguous.

Taking the general behavior of this variant into account, it can be observed that the number of levels strongly affects the performance. Thus, this feature is more important than the number of nonzeros of each matrix. This situation is aligned with other works over the SPTRSV kernel with different hardware platforms, see [19,10,19].

Table 3. Runtime (in ms) for the NDR_{wait} variant of SPTRSV kernel with different optimizations.

BS	CU	UF	VW	ChipC	T_{162}	T_{300}	Gyro	GodW	BcsS	T_{2052}	Thread	Ndk
4	1	1	1	**8.0**	**15.8**	52.5	**52.7**	**12.5**	4.5	132.5	**51.9**	**143.6**
4	1	1	2	8.6	18.9	58.0	54.4	13.7	**4.0**	143.2	67.5	144.3
8	1	1	1	9.7	25.9	86.9	78.2	20.9	7.0	200.3	75.5	162.0
8	1	1	2	8.6	18.6	**51.3**	89.5	17.5	4.4	**113.6**	57.6	200.3
16	1	1	1	18.0	36.4	116.2	102.8	29.1	17.9	264.5	99.6	224.6
16	1	1	2	16.2	18.8	59.7	115.8	30.3	9.3	133.6	65.6	208.6

Table 3 summarizes the experimental results for the NDR_{wait} variant of the SPTRSV. In this version the number of compute units and the unroll–factor are kept in 1, since incrementing the number of computational units did not produce any runtime improvements and the structure of the kernel does not allow to implement unrolling.

When comparing the attained performances with the NDR_{multi} counterpart, it is clear that NDR_{wait} version strongly improves the runtime for linear systems with a large number of levels, and it is not a good option for the smallest case and the TSOPF problem family. Additionally, the best optimization configurations in this variant are less scattered. And, more important, when this variant outperforms the previous one, the better configuration is in all cases the same (work group size equal to 4 and a vectorization value of 1).

For the Single Work Item Kernels, the OpenCL optimization space is more reduced. Only the unroll–factor is explored with values of 1, 2, 4 and 8. The other differences involve changing the algorithm strategy in each variant, i.e. wether or not using a hash as a cache memory.

Table 4 summarizes the runtime results reached by the $\text{SWI}_{channel}$ variant. The experimental results reveal that the use of 1 and 2 for the unroll–factor are the best options for all test cases. Also, the differences between both configurations are negligible. Studying the general behavior of this variant, we need to highlight that the runtime differences between the test cases is closer than

Table 4. Runtime (in ms) for the $\text{SWI}_{channel}$ variant of SPTRSV kernel with different optimizations.

BS	CU	UF	VW	ChipC	T_{162}	T_{300}	Gyro	GodW	BcsS	T_{2052}	Thread	Ndk
1	1	1	1	**11.3**	12.0	32.3	**20.7**	**11.5**	11.4	66.5	**29.4**	59.6
1	1	2	1	12.3	**11.7**	**31.4**	21.8	12.4	**10.9**	**57.3**	31.7	**58.9**
1	1	4	1	12.4	15.4	47.8	24.5	13.9	12.3	89.2	40.8	72.0
1	1	8	1	14.1	22.1	64.2	30.6	16.6	15.8	122.6	51.3	96.6

in previous kernels. In NDR_{multi} version runtime ranges from 1.1 to 361.9 and in NDR_{wait} from 4.0 to 143.6, while in the current variant the interval is between 10.9 and 58.9. Additionally, it seems that the runtime performance is more related to the nnz value of each matrix.

Table 5. Runtime (in ms) for the SWI_{hash} variant of SPTRSV kernel with different optimizations.

BS	CU	UF	VW	ChipC	T_{162}	T_{300}	Gyro	GodW	BcsS	T_{2052}	Thread	Ndk
1	1	1	1	**11.8**	22.0	69.1	24.0	12.5	**12.1**	150.5	55.3	99.9
1	1	2	1	12.7	**11.3**	22.3	**19.9**	**12.4**	13.7	42.2	25.3	46.8
1	1	4	1	14.0	12.1	**18.1**	20.9	13.4	15.3	**30.6**	**25.1**	**42.2**

The runtime results for SWI_{hash} are presented in Table 5. First, we can see in the table that the results obtained for the different matrices are closer to each other, even more than in the $\text{SWI}_{channel}$ version. This is because the SWI_{hash} variant offers more benefits for matrices with large nnz, i.e. reduced runtime for the most costly test cases. Additionally, the use of the hash in the smallest test cases does not offer any benefits, increasing the runtime only marginally. Finally, in this variant the optimization configuration (the value for unroll factor) is guided by the nnz of each matrix.

Table 6. Version, optimization configuration and runtime (in ms) for the best variant of SPTRSV kernel for the different test cases.

Matrix	Version	BS	CU	UF	VW	Runtime
ChipC	NDR_{wait}	4	1	1	1	8.0
T_{162}	SWI_{hash}	1	1	2	1	11.3
T_{300}	SWI_{hash}	1	1	4	1	18.1
Gyro	SWI_{hash}	1	1	2	1	19.9
GodW	$\text{SWI}_{channel}$	1	1	1	1	11.5
BcsS	NDR_{multi}	1	2	2	1	1.1
T_{2052}	SWI_{hash}	1	1	4	1	30.6
Thread	SWI_{hash}	1	1	4	1	25.1
Ndk	SWI_{hash}	1	1	4	1	42.2

Table 6 consolidates the runtime results. The first conclusion from the numerical values is that all version are the best for at least one case. The NDRange Kernels seem to be the best option for smallest test cases, while the Single Work Item Kernels are the best choice for matrices with large *nnz*. From the OpenCL optimization configuration perspective, our proposals are neither able to leverage the vectorization nor the use of more than one compute unit (only one best case used 2 as a CU, but the difference is negligible when comparing against the non replicating version). On the other hand, the use of different values for the unroll–factor offers some benefits. In concrete, large test cases take advantages of larger unroll factors.

Table 7. Power (in W) and Energy consumption (in mJ) for the best performing kernels.

Matrix	Version	Runtime(ms)	Power(W)	Energy(mJ)
ChipC	NDR_{wait}	8.0	5.65	45.0
T_{162}	SWI_{hash}	11.3	6.60	74.3
T_{300}	SWI_{hash}	18.1	6.55	118.3
Gyro	SWI_{hash}	19.9	5.45	108.6
GodW	$SWI_{channel}$	11.5	5.25	60.3
BcsS	NDR_{multi}	1.1	5.50	6.1
T_{2052}	SWI_{hash}	30.6	6.80	208.2
Thread	SWI_{hash}	25.1	6.15	154.4
Ndk	SWI_{hash}	42.2	6.00	253.1

The last study is from energy consumption perspective. In this line, Table 7 offers the Power and Energy consumption involving for the different test cases when the best kernel is employed. In Figure 1 we plot the number of nonzeros ($\times 10^3$) processed by mJ of energy consumed for each test case, as the *nnz* is considered the best estimation of the effort implied by the SPTRSV for a particular sparse system. It should be noted that the cases that require more Power (e.g. the T_{2052} case) are the more efficient from the perspective of this metric.

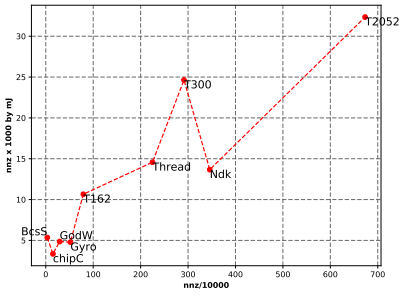

Figure 1. Thousands of *nnz* processed by energy consumption (1 mJ) for the tested sparse matrices.

5 Final remarks and future work

We have studied the performance of several kernels for the solution of sparse triangular linear systems (SPTRSV) in FPGAs. In particular, we presented OpenCL implementations for the SPTRSV kernel following two different parallel execution paradigms, the NDRange and a Single Work–Item. Additionally, our study explores the most relevant OpenCL optimization configurations, such as the use of threads, vectorization and unrolling.

The experimental evaluation performed on a low–end FPGA shows that the best method varies from one the test case to the other. This situation is aligned with other efforts for the SPTRSV kernel on masively-parallel devices. Additionally, the runtimes achieved by the best configuration of each case are competitive considering those found in the literature and our previous experience, while the energy consumption is clearly less.

In future work we plan to address the combination of OpenCL with low-level developments in order to strongly improve the kernel performance. Also, it would be interesting to evaluate the performance of our solvers in other FPGAs and for a larger number of test cases, particularly including high-end boards and large linear systems. Finally, we will try to advance in the characterization of the FPGA performance and energy consumption of each technique.

Acknowledgments

The researchers were supported by Universidad de la República and the PEDECIBA. We acknowledge the ANII – MPG Independent Research Groups: "Efficient Hetergenous Computing" with the CSC group.

References

1. The Green500 list, 2019. Available at `http://www.green500.org`.
2. The Vitis BLAS, 2019. Available at `https://www.xilinx.com/products/design-tools/vitis/vitis-libraries/vitis-blas.html`.
3. Steve Ashby *et al.* The opportunities and challenges of Exascale computing. Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee, November 2010.
4. Sergio Barrachina, Maria Barreda, Sandra Catalán, Manuel F Dolz, Germán Fabregat, Rafael Mayo, and ES Quintana-Ortí. An integrated framework for power-performance analysis of parallel scientific workloads. *Energy*, pages 114–119, 2013.
5. Peter Benner, Pablo Ezzatti, Enrique Quintana-Ortí, and Alfredo Remón. On the impact of optimization on the time-power-energy balance of dense linear algebra factorizations. In Rocco Aversa, Joanna Kołodziej, Jun Zhang, Flora Amato, and Giancarlo Fortino, editors, *Algorithms and Architectures for Parallel Processing*, pages 3–10, Cham, 2013. Springer International Publishing.
6. Tomasz S Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P Singh. From OpenCL to high-performance hardware on FPGAs. In *22nd international*

conference on field programmable logic and applications (FPL), pages 531–534. IEEE, 2012.
7. T. Davis. *Direct Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2006.
8. J. Dongarra and *et al*. The international ExaScale software project roadmap. *Int. J. of High Performance Computing & Applications*, 25(1):3–60, 2011.
9. J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, March 1990.
10. E. Dufrechou and P. Ezzatti. Solving sparse triangular linear systems in modern GPUs: A synchronization-free algorithm. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 196–203, March 2018.
11. D. Erguiz, E. Dufrechou, and P. Ezzatti. Assessing sparse triangular linear system solvers on GPUs. In *2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, pages 37–42, Oct 2017.
12. Pablo Ezzatti, Enrique S. Quintana-Ortí, Alfredo Remón, and Jens Saak. Power-aware computing. *Concurrency and Computation: Practice and Experience*, 31(6):e5034, 2019. e5034 cpe.5034.
13. F. Favaro, E. Dufrechou, P. Ezzatti, and J. P. Oliver. Unleashing the sptrsv method in fpgas. In *PACO 2019: 3rd Workshop on Power-Aware Computing*, 2019.
14. Jeremy Fowers, Kalin Ovtcharov, Karin Strauss, Eric S. Chung, and Greg Stitt. A High Memory Bandwidth FPGA Accelerator for Sparse Matrixr–Vector Multiplication. In *Proceedings of the 2014 IEEE 22Nd International Symposium on Field-Programmable Custom Computing Machines*, FCCM '14, pages 36–43, Washington, DC, USA, 2014. IEEE Computer Society.
15. Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 2013.
16. 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*, pages 9–16, April 2012.
17. S. Kestur, J. D. Davis, and O. Williams. BLAS Comparison on FPGA, CPU and GPU. In *2010 IEEE Computer Society Annual Symposium on VLSI*, pages 288–293, July 2010.
18. Tiziano De Matteis, Johannes de Fine Licht, and Torsten Hoefler. FBLAS: Streaming Linear Algebra on FPGA, 2019.
19. Maxim Naumov. Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the gpu. *NVIDIA Corp., Westford, MA, USA, Tech. Rep. NVR-2011*, 1, 2011.
20. 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)*, pages 107–113. The Steering Committee of The World Congress in Computer Science, 2018.
21. Kevin Rice Townsend. Computing SpMV on FPGAs, 2016. Graduate Theses and Dissertations. Available at `https://lib.dr.iastate.edu/etd/15227`.
22. Yaman Umuroglu and Magnus Jahre. A Vector Caching Scheme for Streaming FPGA SpMV Accelerators. In Kentaro Sano, Dimitrios Soudris, Michael Hübner, and Pedro C. Diniz, editors, *Applied Reconfigurable Computing*, pages 15–26, Cham, 2015. Springer International Publishing.
23. Loring Wirbel. Xilinx sdaccel: a unified development environment for tomorrows data center. *The Linley Group Inc*, 2014.