PSyGS Gen: A Generator of Domain Specific Architecture to Accelerate Sparse Linear System Resolution

Francesco Negri, Niccolò Nicolosi

June 23, 2023

POLITECNICO
MILANO 1863

POLITECNICO MILANO 1863
NECST laboratory

Abstract

Sparse linear systems frequently arise in diverse scientific and engineering domains, in problems like network analysis, computational fluid dynamics, optimization problems and more. Techniques to solve this type of systems efficiently, by taking advantage of their sparsity, already exist, but accelerating their resolution is an important topic in scientific research. In this report, we propose a generator of prototype domain specific architectures (DSAs) on FPGA that accelerate sparse linear system resolution by making use of the symmetric Gauss-Seidel algorithm and existing parallelization techniques. In particular, the generated DSAs are designed to exploit data preprocessed with coloring algorithms that make it possible to parallelize the computation.

1 Introduction

The aim of this report is to describe PSyGS Gen: a generator of prototype DSAs on FPGA that accelerates sparse linear system resolution. We will briefly present the algorithm that we aim to accelerate, analyze the state of the art regarding the acceleration of the chosen algorithm and finally describe the designed architecture, discussing the obtained results.

2 The algorithm: Symmetric Gauss-Seidel

The Gauss-Seidel algorithm (GS) is a known iterative algorithm to solve sparse linear systems in the form:

$$A\overline{x} = \overline{b}$$

At each iteration, a new approximation of $\overline{x}$ is computed using the following update formula, in a loop over i from 1 to n:

$$x_i \coloneqq \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i}^{n} a_{ij} x_j \right) \tag{1}$$

where $j \in \{1, n\}$

Note that the computation of x_i always needs the most recently calculated value of x_j $\forall j \in \{1, n\}$, so the updates are usually done sequentially.

After an arbitrary number of iterations, the convergence of the algorithm is checked by substituting the newly computed $\overline{x}$ into the system and verifying that:

$$\|A\overline{x} - \overline{b}\| < T \tag{2}$$

where T is a fixed threshold

Some sufficient conditions for the convergence of the algorithm are the following:

- A is symmetric positive-definite
- A is strictly diagonally dominant
- A is irreducibly diagonally dominant

A variant of this algorithm which is known to have a better convergence rate is the Symmetric Gauss-Seidel algorithm (SyGS), which divides every iteration into two steps:

- A forward sweep, during which the update formula is called in a loop over i from 1 to n
- A backward sweep, during which the update formula is called in a loop over i from n to 1

From now on, we will always refer to the Gauss-Seidel algorithm and its symmetric variant by their abbreviations: GS and SyGS, respectively.

3 State of the art

Accelerating sparse linear system resolution is an important topic in scientific research. In facts, different parallelization techniques have already been applied specifically to GS, in order speed up its execution [3]. Moreover, various implementations of DSAs that embed GS-dedicated hardware modules have already been proposed in scientific literature and could benefit from a faster SyGS accelerator.

3.1 GS parallelization: coloring techniques

In the context of GS, in order to allow the computation of multiple components of $\overline{x}$ to happen in parallel, it must be verified that such computations are not dependent on one another.

We shorten "the computation of x_i directly depends on the one of x_j" or in other words "x_i directly depends on x_j", by the notation: $x_j \rightarrow x_i$.

Referring to the update formula 1:

$$x_j \rightarrow x_i \iff a_{ij} \neq 0$$

Two computations i and j are directly dependent on one another *iff* either $x_j \rightarrow x_i$ or $x_i \rightarrow x_j$, that we shorten by the notation: $x_i \leftrightarrow x_j$.

A computation i depends on a computation j *iff* one of the following conditions is satisfied:

$$x_j \rightarrow x_i$$

$$\exists k \,|\, x_j \rightarrow x_0 \rightarrow \cdots \rightarrow x_k \rightarrow x_i$$

Two computations i and j are dependent on one another *iff* either x_i depends on x_j or x_j depends on x_i.

Based on this considerations, various techniques to highlight this type of dependencies have been applied to GS in different studies. The main idea behind these techniques is to partition the components of $\overline{x}$ such that each partition contains elements that are not dependent on one another (directly or indirectly) and to assign a different color to each partition. Each element of a partition is said to be of the color of its partition and components of the same color can be computed in parallel. We will introduce how three of these coloring techniques, that are central for the purpose of this study, are applied to GS (the same applies for SyGS).

3.1.1 Graph coloring

In order to apply graph coloring to GS, an undirected graph is constructed as such:

- A node for each component of $\overline{x}$
- An edge between node i and node $j \iff x_i \leftrightarrow x_j$

Then, each node is assigned a color by following a greedy policy that guarantees that no adjacent nodes share the same color. As the reader may notice, this only preserves direct dependencies and therefore, the resulting parallel computation of GS is not equivalent to the sequential one in the general case. However, it has been empirically shown that this still allows the algorithm to converge, at the price of a penalty on the number of iterations required. This method allows for the largest amount of $\overline{x}$ components to be updated in parallel.

3.1.2 Block coloring

Block coloring is a variant of graph coloring, in which the components of $\overline{x}$ are partitioned into blocks of a fixed size. Hence, the graph is constructed as such:

- A node for each block
- An edge between node i and node $j \iff$ block i and block j contain x_{i0} and x_{j0} respectively, such that $x_i \leftrightarrow x_j$

Then, the graph is the colored with the same greedy policy used by graph coloring. Blocks of the same color can be computed in parallel, but components within the same block are updated sequentially. Graph coloring is a special case of block coloring, in which the block size is equal to one. Increasing the block size, the amount of components that are updated in parallel decreases, but the penalty in the number of iterations for convergence slightly decreases.

3.1.3 Dependency graph

To apply dependency graph to GS, an undirected graph is constructed in the same way as in graph coloring. The graph is then colored by following a policy that considers both direct and indirect dependencies. This method preserves all dependencies, therefore having no penalty in the number of iterations required for convergence. However, with respect to the previous techniques, this causes the least possible number of components to be updated in parallel.

3.2 Related work: DSAs embedding GS

In the existing literature there are examples of DSAs dedicated to solve specific problems, that embed hardware modules responsible of executing GS to solve linear systems. Unfortunately, the performance of these GS modules is not directly comparable with that achieved by our architectures as it is

not explicitly reported in the papers. In spite of this, some of them, such as [1], [4] and [2], are examples of GS use cases in scientific and engineering domains that could benefit from a better SyGS accelerator.

3.3 From state of the art to our solution

Given the present state of the art, we propose a new solution to the problem. We designed PSyGS Gen: a generator of DSAs to execute the symmetric Gauss-Seidel algorithm in parrallel, exploiting dependency-free data preprocessed with coloring algorithms.

4 Architecture

4.1 Framework

PSyGS Gen (Parallel Symmetric Gauss-Seidel Generator) is implemented in Chisel, an open-source hardware description language embedded in Scala, that allows the development of parametric architectures. Specifically because of the parametric nature of the Chisel language, we have been able to design a template DSA that can be deployed in different configurations by adjusting parameters like the number of processing elements, the memory size and others. Different configurations may better suit different use cases and most importantly they leave room for future experimentation.

In particular, our generator was developed within the Chipyard framework, a well known prototyping environment which provided us with the design of the Rocket Core. Rocket Core is a 5-stage in-order scalar processor that we synthesized on FPGA along with our template to be able to run the C code that leads the execution of the algorithm by sending commands to our accelerator. We needed a soft-core processor in the absence of a hard-core on the board we used to test our DSAs.

4.2 Overview

As anticipated above, our DSAs are designed to handle the parallel execution of the algorithm. This is done by assigning independent computations of $\overline{x}$ components to different processing elements (PEs), which are coordinated by a controller unit.

The controller communicates with the host Rocket Core processor, that leads the executions of the algorithm.

The memory is divided in local memory banks and shared memory banks. The former are dedicated to a single PE each, to store constant data needed by the specific PE. The latter are shared among all PEs: they are used to store the components of $\overline{x}$, so they are constantly read and updated during the execution of the algorithm.

An overview of the whole architecture is shown in figure 1.

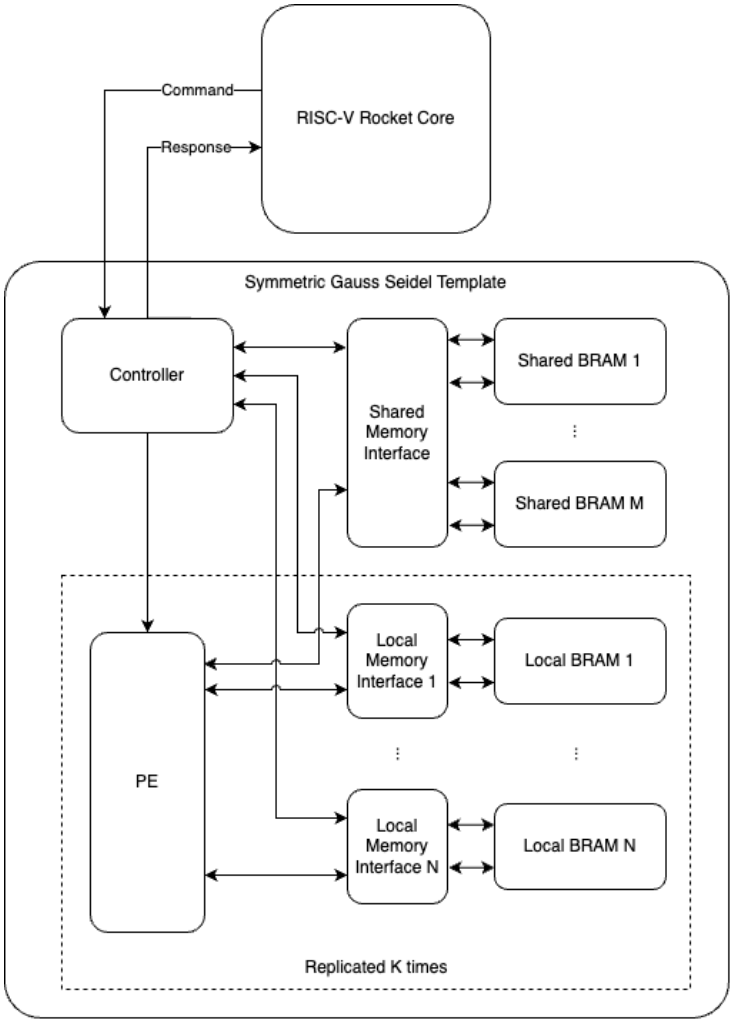

Figure 1: Overview

Before diving into the details of the PEs, which are the core of the architec-

ture, we will introduce how the data needed by the algorithm is stored into memory and how it can be retrieved by other components.

4.2.1 Local memory layout

As already anticipated, local memory banks are dedicated to store data that are only needed by a specific PE. Each PE needs to have access to metadata that describe the structure of data, data about the precomputed coloring and data about a portion of the system to solve. In facts, each PE is assigned the computation of specific components of $\overline{x}$ and consequently needs to read only the related rows of matrix A. Multiple local memory banks can be dedicated to each PE, depending on the number of accumulators they contain (see section 4.3.1 for details).

A schematic representation of the local memory layout is shown in figure 2

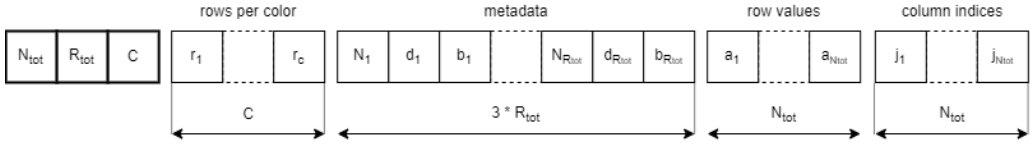

Figure 2: Local memory layout

Specifically the first three entries are:

1. N_{tot}: the total number of non-zero values ($a_{ij} \neq 0$) of the rows assigned to the PE
2. R_{tot}: the total number of rows assigned to the PE
3. C: the total number of colors

The successive C entries are the number of rows assigned to the PE for each color.

After that, the memory contains the metadata: three values for each row assigned to the PE:

1. N: the total number of non-zeros of the k-th row assigned to the PE
2. d: the index of the diagonal value (a_{ii}) within the list of non-zeros values of the k-th row assigned to the PE
3. b: the value of vector $\overline{b}$ associated to the k-th row assigned to the PE

The last entries are the list of non-zeros values and column indices (a_{ij} and j respectively) of each of the R rows assigned to the PE.

4.2.2 Shared memory layout

The shared memory banks contain vector $\overline{x}$. The components of $\overline{x}$ are split among the banks according to figure 3.

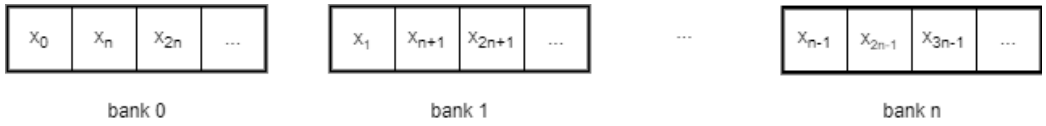

Figure 3: Shared memory layout

Each bank has two access ports and a single clock cycle latency. During the computation, each PE requests to read from one of these banks at each clock cycle, so the higher the number of banks, the easier is to serve the requests.

4.2.3 Memory interfaces

The memory interfaces are the components that handle incoming read and write requests form PEs to the memory and the outgoing responses. The requests are handled based on a fixed priority.

There is one memory interface for each local memory bank. The shared memory banks, instead, are all connected to a single memory interface that manages read and write requests for all the PEs. In our design, write requests are always prioritary with respect to read requests.

4.3 Processing element

The PE is the component which handles the main computation. It is composed by two main submodules:

- The address generator (AG), which handles the interactions with memory
- The data flow engine (DFE), which makes the actual computation, communicating with the AG

The PE is designed to be instantiated multiple times to increase the parallelism of the algorithm. Each PE is independent from the others.

An overview of the whole PE architecture is shown in figure 4. The details are explained in the next sections.

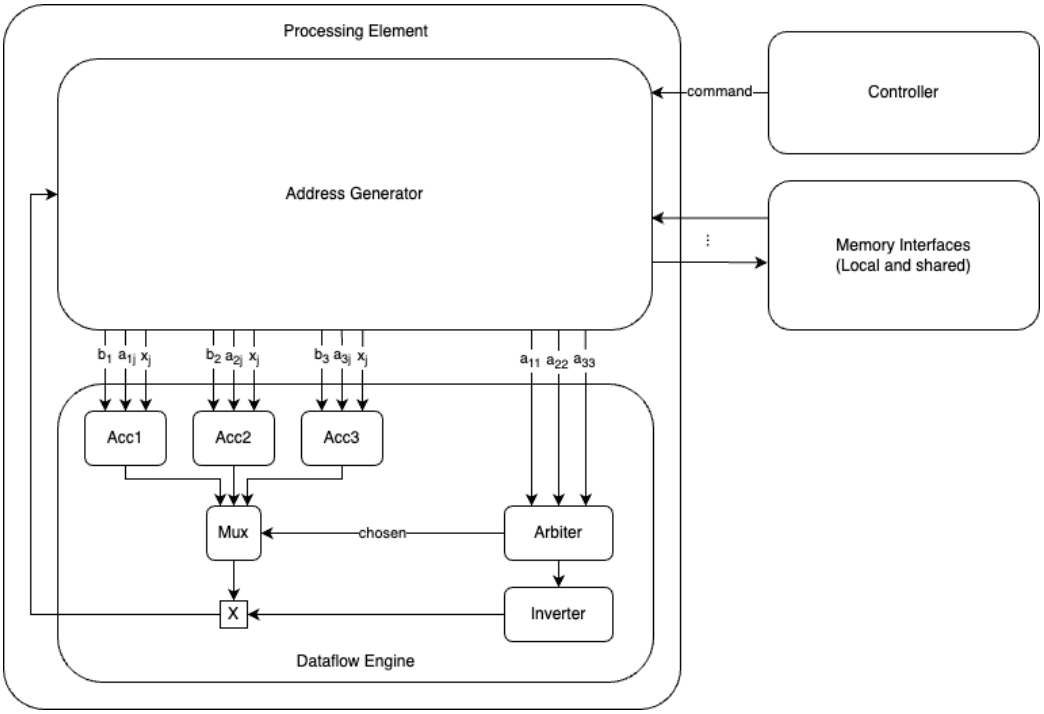

Figure 4: Processing Element

4.3.1 Data Flow Engine

The DFE is the component that performs the actual computation of GS. According to formula 1, the update of a component can be split into three main operations:

1. multiplication:
$$m_j = a_{ij} x_j$$
2. accumulation:
$$acc = b_i - \sum_{j \neq i}^{n} m_j$$
3. division:
$$x_i = \frac{acc}{a_{ii}}$$

Given a row i of matrix A, the DFE executes these three operations to compute the new x_i. First, the accumulator unit takes in a_{ij} and x_j $\forall j \in \{1, N_i - 1\}$, where N_i is defined as the number of non-zero elements of row i, and multiplies them together. Then, the results are sent to the next module

which computes the $N_i - 1$ successive subtractions from b_i. Meanwhile, a_{ii} is sent to the inverter unit, which computes $\frac{1}{a_{ii}}$ and sends the result back to the accumulator. When both the accumulation and the reciprocal have been computed, they are multiplied together and are delivered in output.

In order to absorb latencies introduced by the interactions with memory and achieve a better throughput, all the inner modules are interconnected by queues.

To better support the computation of less sparse systems, the accumulator unit can be replicated while maintaining a single inverter unit, so that more computations of reciprocals can happen during long accumulations. A detailed picture of the DFE implementation with multiple accumulators is shown in figure 5.

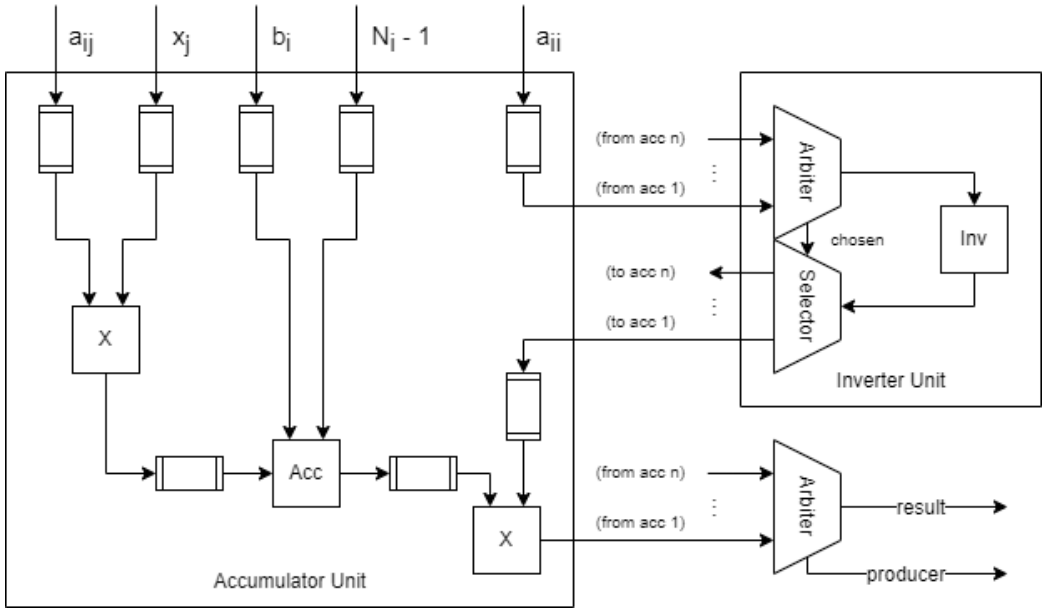

Figure 5: Data Flow Engine with multiple accumulators

To perform floating point operations in our implementation, we used the hardware modules provided by Hardfloat, a Chisel library dedicated specifically to floating point operations. None of the provided modules is pipelined, except for a 64bit divider module, and this is obviously is a severe constraint on the clock frequency of our FPGA implementation. Even the implementation of the 64bit float divider is not fully pipelined, in facts being able to

start the execution of a new division once every six cycles unless exceptions. For this reason, we decided to also allow the possibility of replicating the inverter unit while having a single accumulator, in order to overcome the delay that it introduces.

4.3.2 Address Generator

The AG is responsible of producing memory read and write requests, in particular by generating the correct addresses to access the data stored in local and shared memory banks, based on the computation to perform. In facts, the module receives a "start forward" or a "start backwards" command from the controller, which makes it start the forward or backward stage of a SyGS iteration respectively. From the moment the AG receives a start command, it starts sending read requests to the memory interfaces and passing the obtained data to the DFE. Then, it waits for DFE to finish its computation and generates the write requests to write the results back on memory.

The AG is composed by three submodules:

- The metadata memory request generator (MMRG)
- The data memory request generator (DMRG)
- The variable memory request generator (VMRG)

The MMRG is the supervisor of the other two modules. It starts by reading N_{tot}, R_{tot} and C. Then, for each color:

1. it reads r_i
2. it reads the metadata of the next r_i rows in memory, keeping count of them to know when the color is fully computed

N_{tot}, R_{tot} and C are mainly used to calculate offsets in local memory. C is also used to keep count of how many colors have been computed and set the busy bit of the MMRG to false once a complete iteration has been done.

At the same time, r_i is propagated to the other modules to keep count of how many rows have to be read for the current color.

The metadata are directly sent to the DFE (see section 4.3.1).

The DMRG reads the row values and column indices. They are essential for the computation of the DFE to proceed.

This module needs N_{tot}, R_{tot} and C to compute the addresses of its requests. It uses r_i to keep count of the rows done for color i. It's important to keep track of them because this module tries to send requests in advance to speed up the data transfer, but it must not exceed the requests of the current color, due to the coloring constraints.

The module also needs N and d. They are used to infer the amount of row values to read for each row and to know which of them is the diagonal value. The row values (a_{ij}) are collected from the memory and directly sent to the DFE (see section 4.3.1). Instead, the column indices (j) are sent directly to the VMRG.

The VMRG handles the requests to the shared memory banks. It requires r_i to keep count of how many variables have to be written before its "busy" bit can be set to false. For each column index received from the DMRG, this module sends the corresponding request to the shared memory.

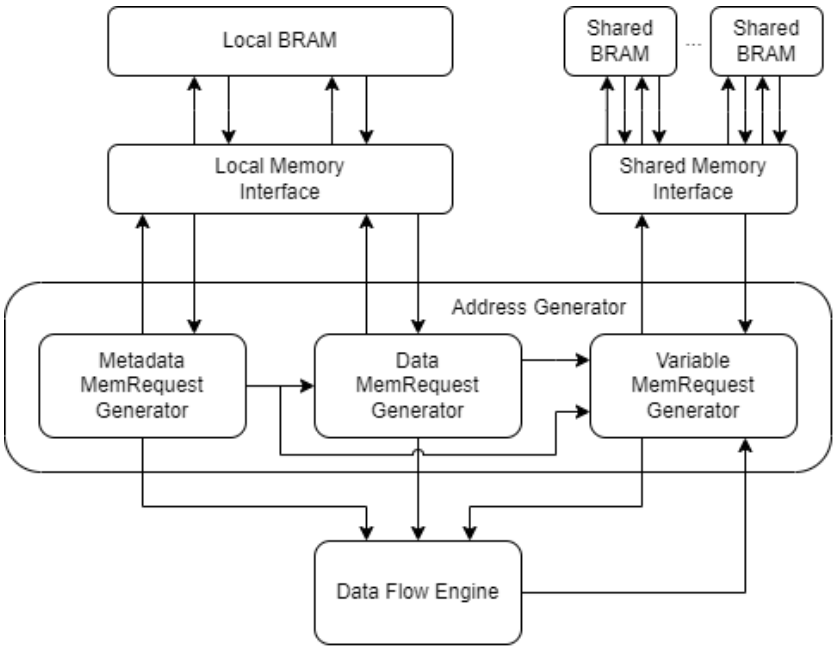

Figure 6: Address Generator

5 Preprocessing

In order to start the execution of the algorithm on the DSA, we need to preprocess the data by computing the coloring with one of the algorithms covered in section 3.1 and organize the data layout as described in section 4.2.1. From the tests we conducted, the overhead time caused by the preprocessing stage proved to be negligible (in the order of 10^{-5} of the total execution time).

6 Deployment

To deploy our prototype DSAs on the Virtex-7 FPGA VC707 evaluation kit, we developed an automated prototyping flow which is shown in figure 7

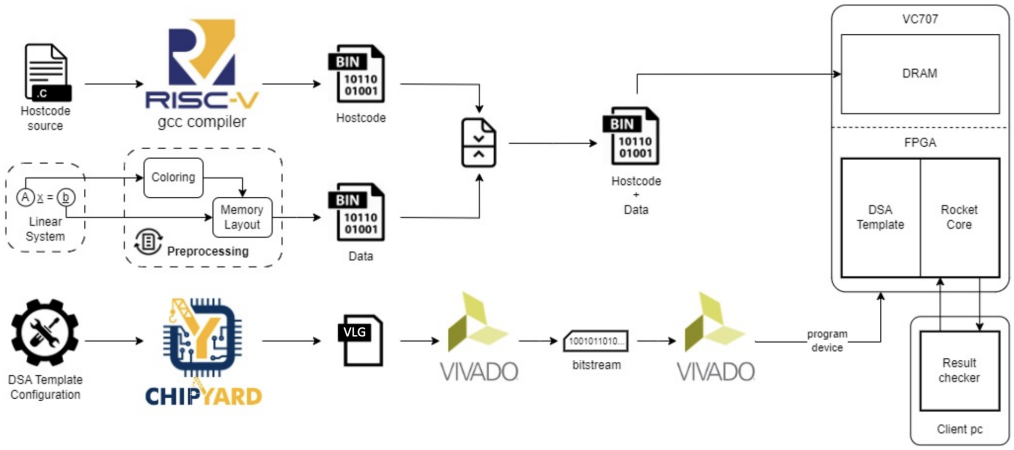

Figure 7: Prototyping flow

The flow consists two distinct streams of computation. On one hand we obtain the verilog of the DSA under test, produce the bitstream and program the FPGA. On the other hand we compile the hostcode that will run on the Rocket Core, preprocess the data of the linear system and concatenate the resulting binaries into a single binary that we load into the DRAM of the board. After that, we can start the execution of the algorithm on the data under test by sending a start command from the client PC connected to the board.

6.1 Client

The client PC starts the serial communication with the Rocket Core on the FPGA to send and receive data. Its main functionalities are:

- determine the amount of iterations to execute before checking the convergence of $\overline{x}$
- check the convergence of $\overline{x}$ according to formula 2

Every time the architecture finishes a block of iterations, it sends $\overline{x}$ back to the client. The client computes the product between A and the updated vector $\overline{x}$, then it calculates the error as the mean square error of all the differences between the components of the newly calculated vector and the ones of $\overline{b}$. When the error is less than a fixed error threshold the process stops and outputs the result.

To estimate the amount of iterations to be done until the next check, the client tries to fit an exponential function with the previously calculated error and the current one. This method results to be efficient due exponential trend of the error over time: in facts, the nearer we get to the solution the more iterations we need to reduce the error by a fixed amount.

6.2 Hostcode

The hostcode running on the Rocket Core leads the execution of the algorithm by sending commands to the controller of the DSA. At first, it writes all the data relative to the linear system, which are already on the DRAM of the board, to the memory banks of the DSA. Then, after it receives the amount of iterations to execute from the client, it starts that number of iteration sending the start commands to the DSA. After completing a block of iterations, it sends back the resulting $\overline{x}$ to the client PC, reading it from the DSA memory.

7 Results

We compared the performance of the generated DSAs with the performance of the software implementation of the algorithm, using the same coloring algorithms in the case of the parallel implementations.

The software tests were performed on a AMD Ryzen 7 3750H CPU, using OpenMP for the parallelization of the algorithm loop.

7.1 Dataset choice

The dataset consists of linear systems taken from real use cases. The matrices of the systems vary in the number of rows (494 to 4960), number of nonzero values (1080 to 23884) and distribution of nonzero values.

7.2 Overall speed-up

As shown in figures 8, 9 and 10, the generated DSAs outperform the software counterparts with the same level of parallelism and the same coloring algorithm, in terms of clock cycles.

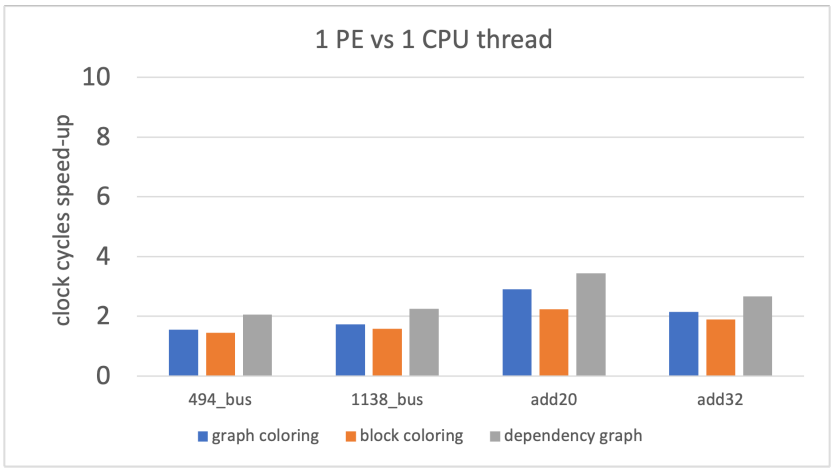

Figure 8: Speed-up comparison 1PE vs sequential GS

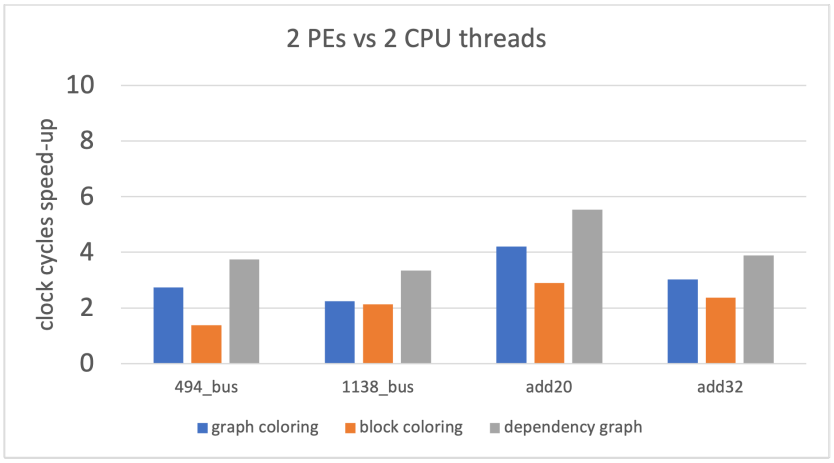

Figure 9: Speed-up comparison 2 PEs vs 2 CPU threads

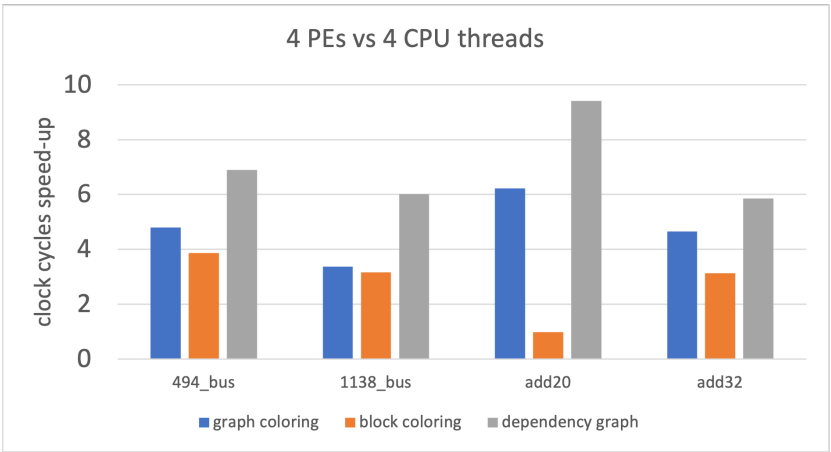

Figure 10: Speed-up comparison 4 PEs vs 4 CPU threads

The actual number of execution cycles of the different DSAs and software implementations is shown in figures 11, 12, and 13.

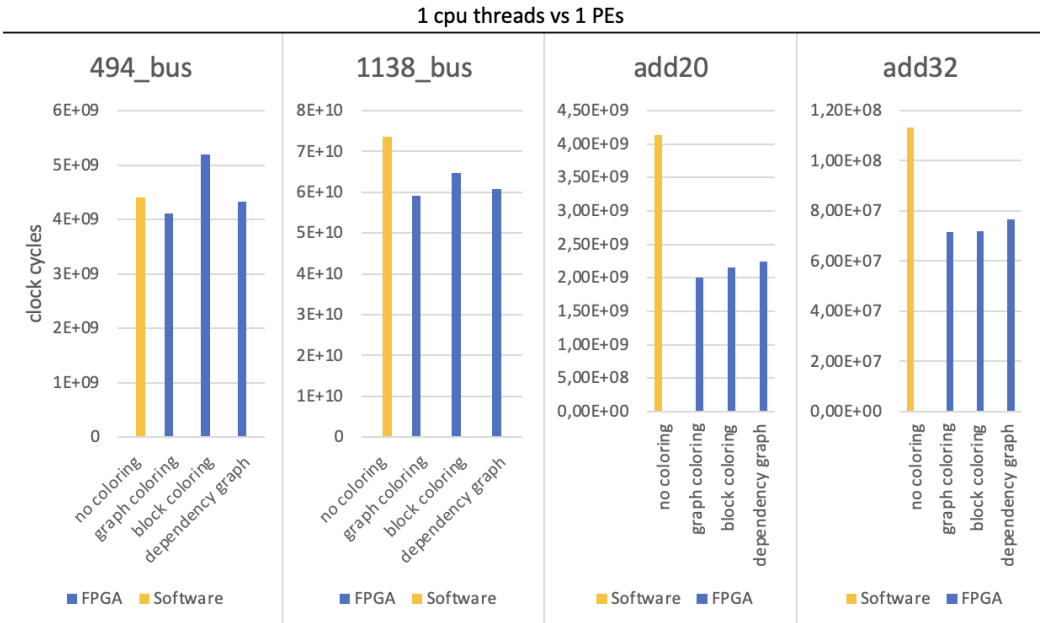

Figure 11: Clock cycles sequential GS vs 1 PE

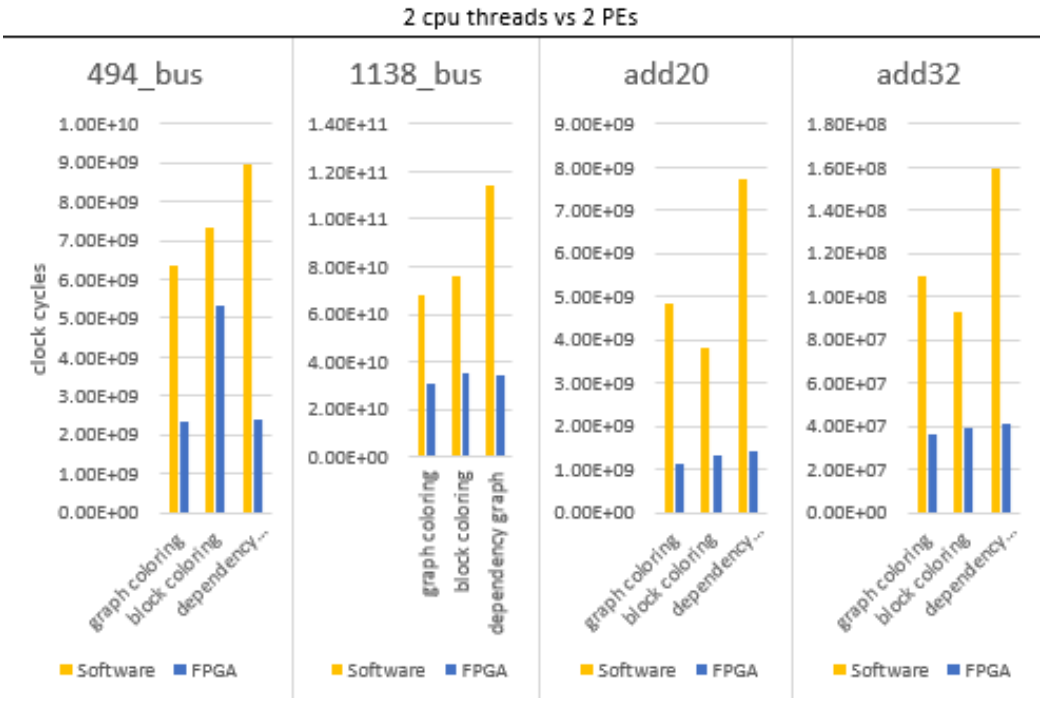

Figure 12: Clock cycles 2 threads vs 2 PEs

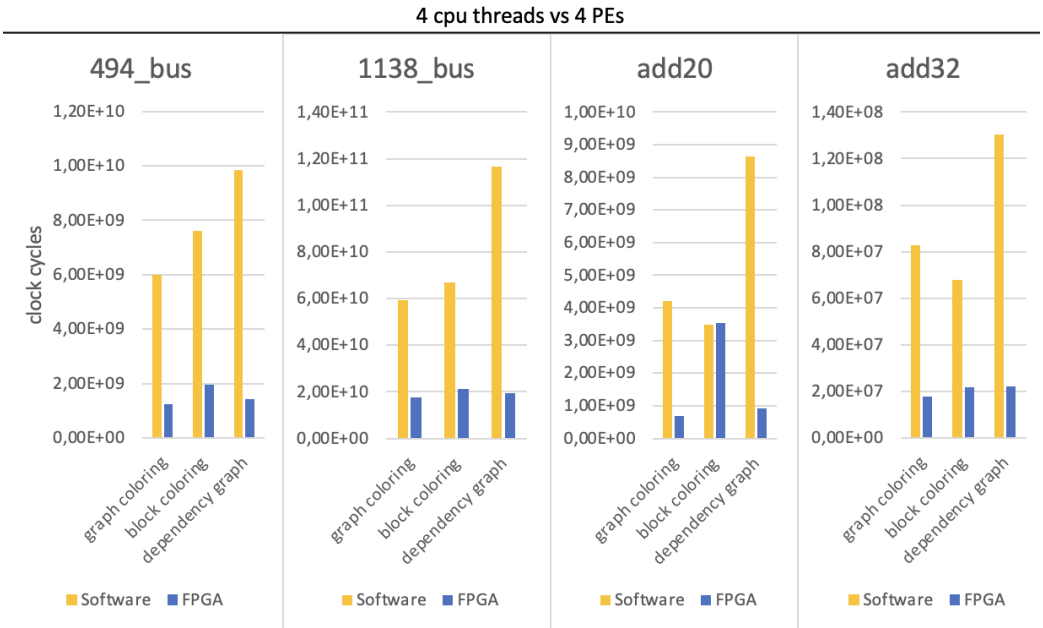

Figure 13: Clock cycles 4 threads vs 4 PEs

The average speed-up trends with respect to the number of PEs/Threads

shown in figure 14 demonstrate that while the software parallel implementation of GS improves very little or even gets worse when using dependency graph, our DSAs scale progressively better.

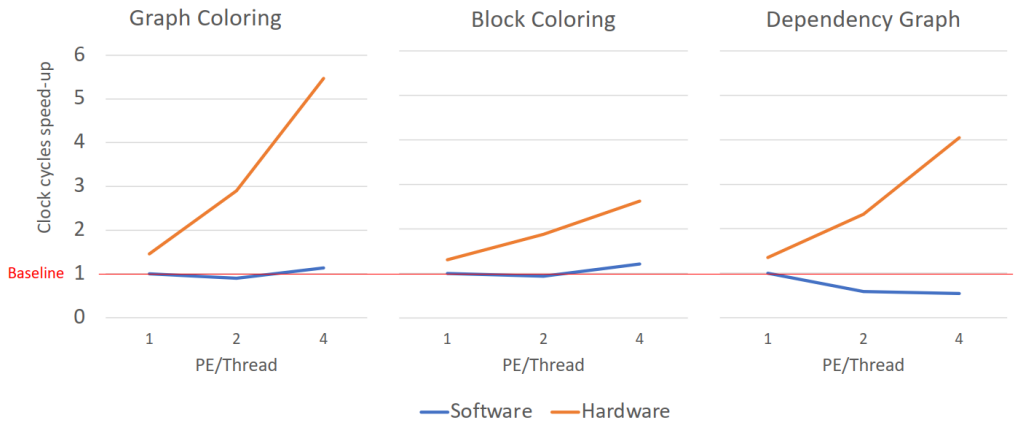

Figure 14: Average speed-up trends

Lastly, being graph coloring the algorithm that lead to the best performing parallelization for both software and hardware in terms of clock cycles, in figure 15 we show in more detail the scaling comparison between our DSAs and the software implementations. In particular, we can observe from the picture, that we always surpass the best performing software implementation, reaching up to 6 times fewer clock cycles in the best cases.

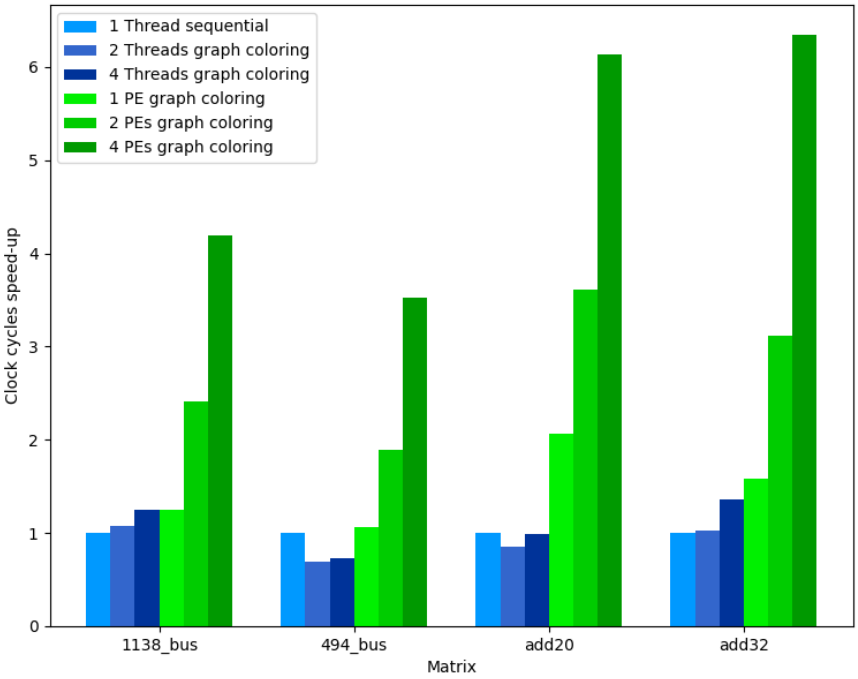

Figure 15: Graph coloring speed-up

7.3 Resource utilization

The usage of FPGA resources by the generated DSAs scales promisingly with the number of PEs, as shown in figure 16. This implies that DSAs with even more PEs can be generated to reach better performance, without requiring excessive resources.

Resource	Total available
LUTs	303600
FFs	607200
RAMB36	1030
DSP	2800

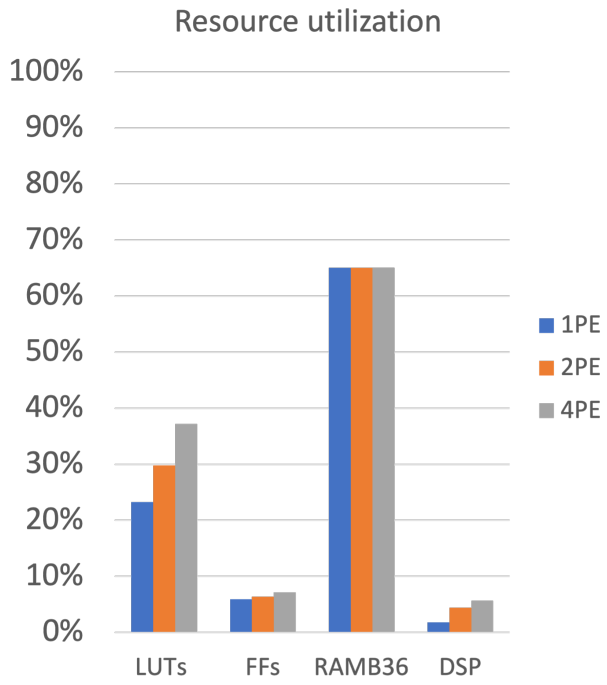

Figure 16: Resource utilization

8 Conclusion

The results obtained confirm that PSyGS Gen is capable of generating DSAs that outperform the software counterparts in terms of number of execution clock cycles.

Even if the execution times are still worse than the software implementation due the relevant difference in clock frequency, we demonstrated that by properly scaling the DSA and increasing its clock frequency enough, we can surpass the performance of the software implementation.

8.1 Future work

Some possibilities for future work are:

- optimizing the data flow engine with piplined float modules to increase throughput and allow higher clock frequency
- switching from the current soft-core host processor (Rocket Core) to a hard-core to increase the overall performance by decoupling the ac-

celerator from the host, allowing higher clock frequency and reducing FPGA resources utilization

References

[1] Jong-Ho Byun et al. "Accelerating the Gauss-Seidel Power Flow Solver on a High Performance Reconfigurable Computer". In: Jan. 2009, pp. 227–230. DOI: 10.1109/FCCM.2009.23.

[2] D.P. Chassin et al. "Gauss-Seidel accelerated: implementing flow solvers on field programmable gate arrays". In: *2006 IEEE Power Engineering Society General Meeting*. 2006, 5 pp.-. DOI: 10.1109/PES.2006.1709227.

[3] Daniel Ruiz et al. "Open-Source Shared Memory implementation of the HPCG benchmark: analysis, improvements and evaluation on Cavium ThunderX2". In: *2019 International Conference on High Performance Computing Simulation (HPCS)*. 2019, pp. 225–232. DOI: 10.1109/HPCS48598.2019.9188103.

[4] Jing Zeng, Jun Lin, and Zhongfeng Wang. "An Improved Gauss-Seidel Algorithm and Its Efficient Architecture for Massive MIMO Systems". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 65.9 (2018), pp. 1194–1198. DOI: 10.1109/TCSII.2018.2801867.