

HPCGRA - An Orthogonal Designed CGRA Generator for High Performance Spatial Accelerators

Lucas Bragança Silva¹, Michael Canesche¹,
Ricardo Ferreira¹, José Augusto M. Nacif¹

¹Departamento de Informática – Universidade Federal de Viçosa (UFV)
Avenida Peter Henry Rolfs – Minas Gerais – MG – Brazil

{lucas.braganca,michael.canesche,ricardo,jnacif}@ufv.br

Abstract. Recently, with the increasing accession of domain-specific architectures to execute kernels with a high computing density and the exploration sparse architectures using Systolic Arrays brought the opportunity to work with Coarse-grained reconfigurable architectures (CGRAs). Unlike Systolic Array, CGRA can run different kernel sets and keep a good balance between energy consumption and performance. In this work, we present the HPCGRA, an orthogonal designed CGRA generator for high-performance spatial accelerators. Our tool does not require any expertise in Verilog design. In our approach, the CGRA is designed and implemented in an orthogonal fashion, through wrapping, with the main axis: the functional units, the interconnection patterns, the routing, and elastic buffer capabilities, and the configuration words and memories. It optimizes and simplifies the process of creating CGRAs architectures using a portable description (JSON file) and generating a generic, scalable, and efficient Verilog RTL code with Veriloggen. The tool automatically generates CGRA with up to 46x66 functional units, which could reach 1.2 Tera ops/s.

1. Introduction

With the increasing adhesion of domain-specific architectures such as Systolic Arrays to execute kernels with a high computation density, it is necessary to develop new tools to facilitate the exploration of project space (DSE) of domain/application-specific architectures. Recent work has focused on the generation of Systolic Arrays from kernels implemented at a high level [Weng et al. 2020, Jia et al. 2020]. Other works have focused a high-performance implementation of a basic operation such as a matrix multiplication unit by using a systolic array. Then, applications in Deep Learning such as deep neural networks (DNN) and convolutional neural networks (CNN) takes profit of these high performance specialized low layer, where more than 70% of the execution time requires matrix multiplications [Asgari et al. 2019, Zhang et al. 2019]. These previous approaches focus on producing optimized architectures for only a specific set of kernels to be accelerated.

Unlike Systolic Array, coarse grain reconfigurable architectures (CGRA) can run different kernel sets. However, there is a lack of tools available to help developers to generate CGRA models. The first challenge is how to describe architecture. For instance, how to define each processing element and its basic operations? What should be the interconnection network to provide high performance and low cost in the processing elements

communications [Chin et al. 2017]. In this way, we propose a novel tool capable of generating RTL code in portable Verilog for FPGA and/or ASICs that automatically synthesizes CGRAs from a high-level description. Our approach helps the hardware developers to carry out analyzes of various types of architectures, including a design exploration by using an orthogonal approach, to obtain data on the consumption of hardware resources, energy consumption, maximum operating frequency speed, and performance.

In this work we present HPCGRA, an optimized tool to generate a high performance CGRA from a single high level description for FPGA's and ASIC's platforms. From an portable JSON configuration file, an architecture is automatically built. We provide orthogonal features to design, interconnect and configure each processing elements (PEs). Also, the tool generates generic Verilog RTL code without the developer worrying about which FPGA to use or for which FPGA chip vendor. Furthermore, a ASIC design could be also generated. Therefore, our tool contributes to reduce and to improve the design cycle, and it provides a high level abstraction layer where the developers will focus in how modeling the main architecture requirements.

The contributions of our work are as follows:

- A simple method of creating generic parallel architectures from a single portable description;
- Orthogonal designed methodology;
- Portable and efficient Verilog RTL code;
- An intermediate assembly code to simplify the programming and code generation for CGRAs.

The paper will be organized as follows. Section 2 provides a brief description of the proposed tool. Section 3 show how our approach compares to related work. Section 4 evaluate our tool by generating CGRA designs. Finally, Section 5 concludes by presenting the main results and future works.

2. HPCGRA Tool

HPCGRA is a tool that allows the detailed description of a CGRA architecture by using a simple high level description in JavaScript object notation (JSON) [JSON 2020]. From this description, it automatically generates all RTL code of the architecture in the Verilog language. Our generator is an open source project based on the Veriloggen library [Takamaeda-Yamazaki 2015], which is written in Python to provide high level facilities to produce Verilog code. Most of recent related work to design specific or reconfigurable architectures are based on the language Chisel [Bachrach et al. 2012] built on top of the Scala language. We have chose Veriloggen because Python is a more widespread language today, allowing better adhesion of the tool and possible collaborations from the community.

First, we specify a default format to perform the JSON code transformation into a CGRA architecture. It is important to highlight that our format is also orthogonal to define the main architecture features. In the proposed format, it is possible to describe each processing element (PE) individually to design heterogeneous architectures. In case of homogeneous ones, high level generators could be easily created to generate our input format. This approach simplifies the elaboration of any format of the CGRA architecture. It is possible to define the set of operations for each PE. In addition, we also define which

are its neighbors. Section 2.1 details a simple, generic and orthogonal description format. By using our tool, we also create architectures that can be generated directly from a single command line, without the need for an input JSON file, where the developers only define the number of PEs with a predefined interconnection pattern set. Several CGRA architectures in the literature are based on these interconnection patterns. Table 1 depicts the main interconnection models provided by our tool. As already mentioned, this facility could be used to fast prototype homogeneous designs. However, our generator allow DSE of heterogeneous architectures.

Name	Description
Mesh	Each PE communicates with its adjacent neighbors.
One Hop	Each PE communicates with its adjacent neighbors and with its neighbors' adjacent neighbors.
Diagonal	Same as the mesh with the addition of the diagonals neighbors.
Hexagonal	Each PE has 6 neighbors, forming a pattern similar to a beehive.

Table 1. Interconnection patterns for the high level HPCGRA generator.

2.1. Format Specification

In this section, we introduce the propose JSON format to describe a CGRA architecture. We create a representation with few attributes to simplify the architecture description. The three main fields are: (1) the “shape”, that represents the number of CGRA rows and columns; (2) the “data_width” field is an integer that defines the bit width of the architecture’s processing data, and (3) “PEs” field is an array of objects that describes the processing elements.

In the proposed format, each PE is individually specified. This definition facilitates the construction of heterogeneous architectures in an orthogonal fashion by using wrappers. Each PE object contains the following attributes “id”, “type”, “neighbors”, “route_type”, “elastic_queue” and “isa”.

The “id” attribute is the unique identifier of each PE. The “type” attribute defines whether the PE receives data from/to outside as an input and/or output node, or it only communicates with other internal PEs. The values for the “type” field are: “input”, “output”, and “basic”, respectively. The “neighbors” attribute define the list of IDs in the neighboring of the current PE. This approach allows the generation of any interconnect network architecture between PEs.

The attribute “route_type” determines what type of internal routing of the PE. In the current version of our tool, there are three routing options as follows:

- “no_routing”: The PE has no routing capability. The function unit (FU or ALU) receives the input data from its neighbors and sends the result to any PE output ports.
- “one_routing”: In addition to perform the internal operation inside the functional units, the PE has the capability to route a single signal from any neighbor PE to any/all, as shown in Figure 1.a.
- “full_routing”: The PE could route any neighbor’ PE to any neighbor’ PE as it implements an internal crossbar network, as shown in Figure 1.b.

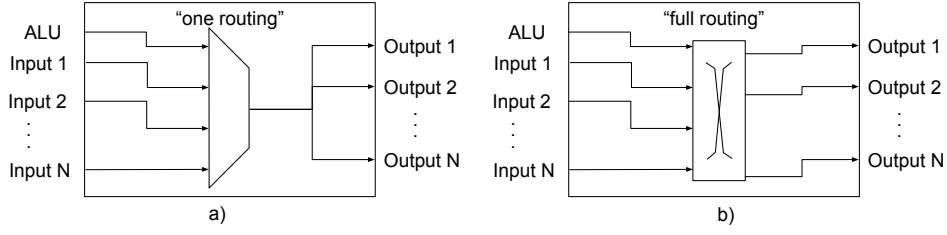


Figure 1. PE routing mechanism. a) "one routing", b) "full routing".

A common problem for two-dimension spatial architectures like CGRAs in pipelined design is to avoid delay mismatching after the placement and routing (P&R) steps [Nowatzki et al. 2018]. This problem occurs when two or more paths has different arrival times when reach a determined PE, due the P&R decisions. This problem could be fixed by adding elastic queues at the PE entrances [Nowatzki et al. 2018]. These queues are programmed resource where it is possible to set its size, and hence we are able to create data delays, and solve the delay mismatches generated by during the "P&R" steps. The "elastic_queue" attribute specifies the maximum size of the PE queues, which can be 0 if not required.

The "isa" field corresponds to the PE instruction set. A PE performs computations with data from neighboring PEs, external input data for "input" PEs, and an internal register. In the current version of our tool, we add a set of instructions with the primary basic operations. These operations are specified by using a list inside the "isa" field. Thus, it is easily to choose which operations each PE is capable of performing. We have already implemented arithmetic and logical operations. The operations are labeled as "add", "sub", "mul", "and", "or", "not", "madd", "addadd", "subsub", "addsub", "mux", "pass"; where we can highlight that there are some ternary operations, such as multiplication and sum (madd), a common operation is many applications such as matrix multiplications.

To better illustrate our input format used to generate a CGRA architecture, an example of a 2x2 architecture is shown in Figure 2. This code generates a CGRA with 4 PEs and mesh network. The PEs in the first column are input PEs, and the PEs of the last column is "output" type.

2.2. A CGRA Instruction Set Architecture

In this section, we present a CGRA instruction set or an assembly code for configuring any CGRA generated architecture. The assembly code simplifies the code generation targeting our CGRA. We use this assembly as an intermediate format. It allows applications written in different languages to compile for this format, which makes this tool more versatile. The fields of the language instructions vary according to the definition of the PE, which can have several neighbors and perform computation on all data from all neighbors. The basic operations showed in Section 2.1 are used in the description of the assembly instruction with the addition of the routing instruction called "route". This instruction defines which PE input will route to PE output. We use the IDs of PE to pass to the instruction. Figure 3 depicts our instruction format.

The ID fields in the format A of the instruction source operands also allow the use of the reserved word "load" or a constant numeric value. For PEs that have external data

```

1 {
2   "shape": [ 2, 2], "data_width": 16, "pe":
3   [
4     {"id":0,"type":"input","neighbors":[1, 2],"route_type":"
       one_routing","elastic_queue":0,"isa":["sub","add"]},
5     {"id":1,"type":"output","neighbors":[0, 3],"route_type":
       "full_routing","elastic_queue":0,"isa":["or","and"]},
6     {"id":2,"type":"input","neighbors":[0, 3],"route_type":"
       no_routing","elastic_queue":2,"isa":["madd"]},
7     {"id":3,"type":"output","neighbors":[1, 2],"route_type":"
       one_routing","elastic_queue":2,"isa":["mux","not"]}
8   ]
9 }

```

Figure 2. Sample JSON description for CGRA mesh 2x2.

Format A: <operation> \$<pe id dst> #<delay> \$<pe id src 1> #<delay> \$<pe id src 2>... #<delay> \$<pe id src N>
Format B: <operation=route> \$<pe id src 1> \$<pe id src 2>

Figure 3. Assembly Format.

entry, we use the word "load". The numeric value operates through a constant register inside the PE, loaded at configuration time. The Format B performs an internal routing of the PE using the "route" operation, and the fields of the sources of the operand 1 and 2 are the IDs of the neighboring PEs that will be locally routed. To perform the routing of the output from the functional unit or ALU, the reserved word "alu" will place as the source field ID 1, and for output PEs, the reserved word "store" can be used in the source field 2. For PEs that have an elastic queue to balance the ALU inputs, it is possible to add a delay (specified by the number of clock cycles) next to the "#" symbol in front of the source fields. To better illustrate what application code is like in assembly, Figure 4 presents an application for performing vector sum. In Figure 4.a the C code is presented, in Figure 4.b the assembly code is presented and in Figure 4.c a possible mapping of the assembly code is shown in a 2x2 CGRA.

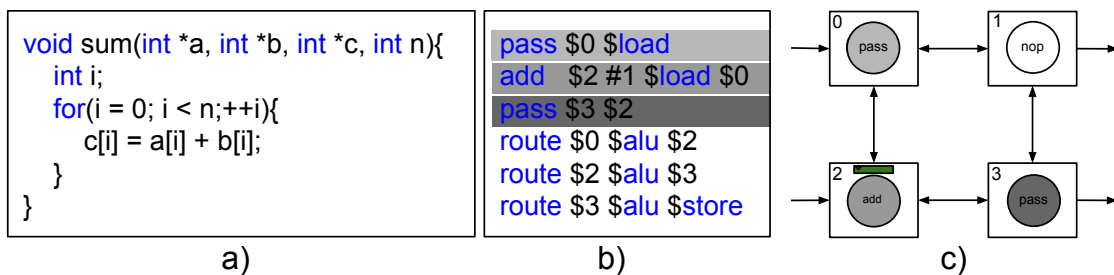


Figure 4. Assembly code for vector sum.

2.3. Orthogonal design approach

For the generation of a CGRA architecture, we developed an orthogonal approach, where the axis of *functionality*, *reconfiguration*, and *connection* between the PEs are independent. In this way, architectures with different numbers of PEs perform the arrangement in the same way. For this, a PE entity is a wrapper, that the generator automatically adds the necessary resources for routing, configuration, and computation.

Figure 5 presents a representation of the dimensions of the generator. ALU ① is the core of architecture, as it performs computations. From the ALU, other components are added, according to the architecture description. The elastic queue ② components are connected to the ALU input ports. The routing ③ dimension is responsible for routing the PE inputs in the elastic queue and the ALU output to the PE outputs. The next dimension is interconnection ④, where each PE is connected to its neighbors according to the description. And finally, the configuration ⑤ dimension that is linked to all components.

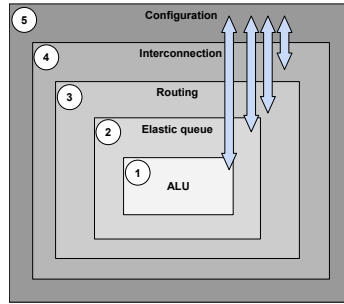


Figure 5. Representation of the dimensions of the generator.

We have developed a partial reconfiguration mechanism for each CGRA component. This mechanism allows efficiently and quickly reconfiguration. A configuration bus is automatically generated for any CGRA, it bypasses by all the PEs of the array, where rows and columns are traversed simultaneously. The bus uses registers for each PE throughout the route to shorten the critical path. Figure 6 shows the mechanism. It is straightforward to calculate the maximum time to configure the entire array since the configuration is sent in the pipeline at each clock cycle to the PEs of the row and column, where the configuration time (tc) calculation as follows: $tc = (1/F_{max}) * (L + C)$, where, F_{max} is clock frequency, and L is the number of rows in the array and C is the number of columns.

3. Related Work

In this section, we present recent works for the generation of reconfigurable architectures. The current works have no focus on the production of generic architectures such as CGRAs. These works have focused on generating specific architectures for kernels that demand high computational density, unlike our tool that allows the user to generate a generic architecture from a JSON description.

In [Chin et al. 2018], the CGRA-ME framework is presented. Unlike our tool that uses the JSON format, CGRA-ME uses the XML format to describe CGRA architectures. For that, several standards were created, creating a specific language, with several TAGs that allow us to describe the detailed functioning of all the components of a CGRA. Our

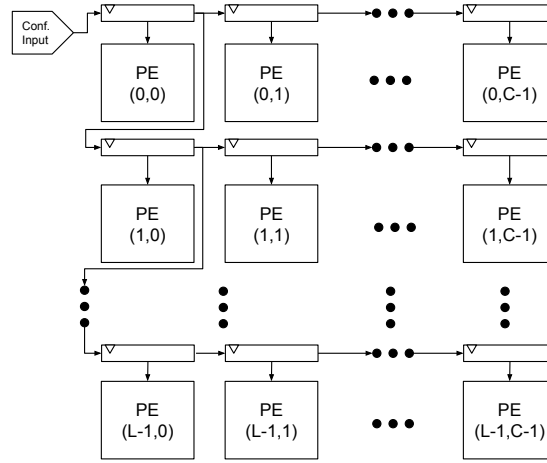


Figure 6. CGRA reconfiguration model.

tool simplifies the description by creating components with greater granularity. Thus, the user only focuses on describing the CGRA directly.

The tool described in work [Jia et al. 2020], presents a generator of Systolic Arrays to perform vector computation. The tool uses a compiler [Genc 2020] as frontend for the tool and uses the Chisel [Bachrach et al. 2012] infrastructure to generate the architecture’s Verilog code. Unlike our tool, only architectures for specific computing are generated, which makes it unfeasible for systems that need to speed up different computations.

The work [Weng et al. 2020] presents a framework for DSA (design specific architecture). The DSA framework receives as input C kernel codes annotated with pragmas and generate a specific spatial architecture. The framework is capable of generating an architecture that minimizes the tradeoff between performance, energy efficiency, and area. For this, the compiler has several optimization phases, such as decouple memory from computing, to leave the ideal code to be described in the form of a data flow graph. The work generates architecture through small blocks that together are capable of generating any functional architecture. These blocks are PEs, switches, memory, FIFOs, and controllers, which allow the construction of an architecture graph (ADG). For a set of kernels, an ADG generates that is iteratively optimized, adding components and removing components at random and evaluated at each iteration using an objective function. The objective function evaluates performance per square millimeter until a target value, or the graph stays stable.

4. Evaluation

In this section, we present the evaluations performed with the HPCGRA tool. The criterion adopted to evaluate the tool was the quality of the RTL code generated in terms of performance and use of resources for different types of CGRA architectures. For this, we generate CGRA with sizes 9x9, 18x18, 36x36. For each size of CGRA, we generate four types of interconnection between the PEs, the *Mesh*, *One-hop*, *Diagonal*, and *Hexagonal* model. We synthesize all architecture using the Intel/Altera infrastructure. For this, we adopted the FPGA Arria 10AX115U3F45E2SGE3 to synthesize CGRA as overlays. Arria 10 has 427200 logic blocks (ALM) and 1518 digital signal processing blocks (DSPs).

We set the target clock for all syntheses to 400 Mhz.

To validate the generator’s scalability, we synthesized a heterogeneous CGRA with an array of 46x66 PEs and with 4 bits of processing word. Since the FPGA Arria 10 has only 1518 DSPs, only half of the PEs perform multiplication. CGRA used 39% of ALMs and 100% of DSP and obtained a maximum frequency of 403 MHz. The theoretical performance of this architecture is 1.2 TOPs, showing that the generated code is scalable.

Table 2 presents a comparison of three generated CGRA sizes, where each size has three different types of internal routing. It is important to highlight that we use ADRES and HyCube columns, where the CGRAs have been generated by using the work presented in [Taras and Anderson 2019]. The ADRES architecture is compatible with our “one routing” routing model shown in Figure 1.a, where our approach uses 32% less ALMs per PE in comparison to the best result presented in the previous work [Taras and Anderson 2019]. The HyCube model is compatible with our “full routing” implementation shown in Figure 1.b, where our approach is approximately 50% more resource-efficient than [Taras and Anderson 2019]. Our generator is scalable and allows the developers to increase the number of elements, maintaining the same average resource usage per PE, as shown in Table 2.

	ADRES	HyCube	Our NR	Our OR	Our FR	Our NR	Our OR	Our FR	Our NR	Our OR	Our FR
Array Size	4x4	4x4	9x9	9x9	9x9	18x18	18x18	18x18	36x36	36x36	36x36
ALM	5051	5664	6400	11954	13827	26801	49589	58111	108709	202643	237248
DSP	32	32	81	81	81	324	324	324	1296	1296	1296
AVG (ALM/PE)	216	330	79	147	170	82	153	179	83	156	183

Table 2. Resource usage for 16-bit CGRAs. ADRES and HyCube using FPGA STRATIX 10 [Taras and Anderson 2019] and ours using FPGA ARRIA 10. NR denotes “no_routing”, OR denotes “one_routing” and “FR” denotes “full_routing”.

We also perform a design exploration of several configurations by ranging the interconnection patterns, routing resources and buffer resources. Figure 7 shows the use of resources of the Mesh and Diagonal architectures. Figures 7 a, b and c are of “no routing”, “one routing” and “full routing” models, respectively. Each CGRA has 0, 2, and 4 sizes of elastic queues at ALU entrances. PEs that have elastic queues greater than zero has a higher latency in their data path, decreasing the critical path and improving the performance of the architecture against the grain and increasing the use of hardware resources. We can notice in the graph of Figure 7.c that the frequency increases for CGRA have elastic row sizes 2 and 4 but decreases for those that do not, with size 0.

Figure 8 shows the use of resources of the One-hop and Diagonal architectures. Figures 8 a, b and c are of “no routing”, “one routing” and “full routing” models, respectively. Each CGRA has 0, 2, and 4 sizes of elastic queues at ALU entrances. PEs that have elastic queues greater than zero has a higher latency in their data path, decreasing the critical path and improving the performance of the architecture against the grain and increasing the use of hardware resources. We can notice in the graph of Figure 8.c that the frequency increases for CGRA have elastic row sizes 2 and 4 but decreases for those that do not, with size 0. the One hop model communicates with two more neighbors than

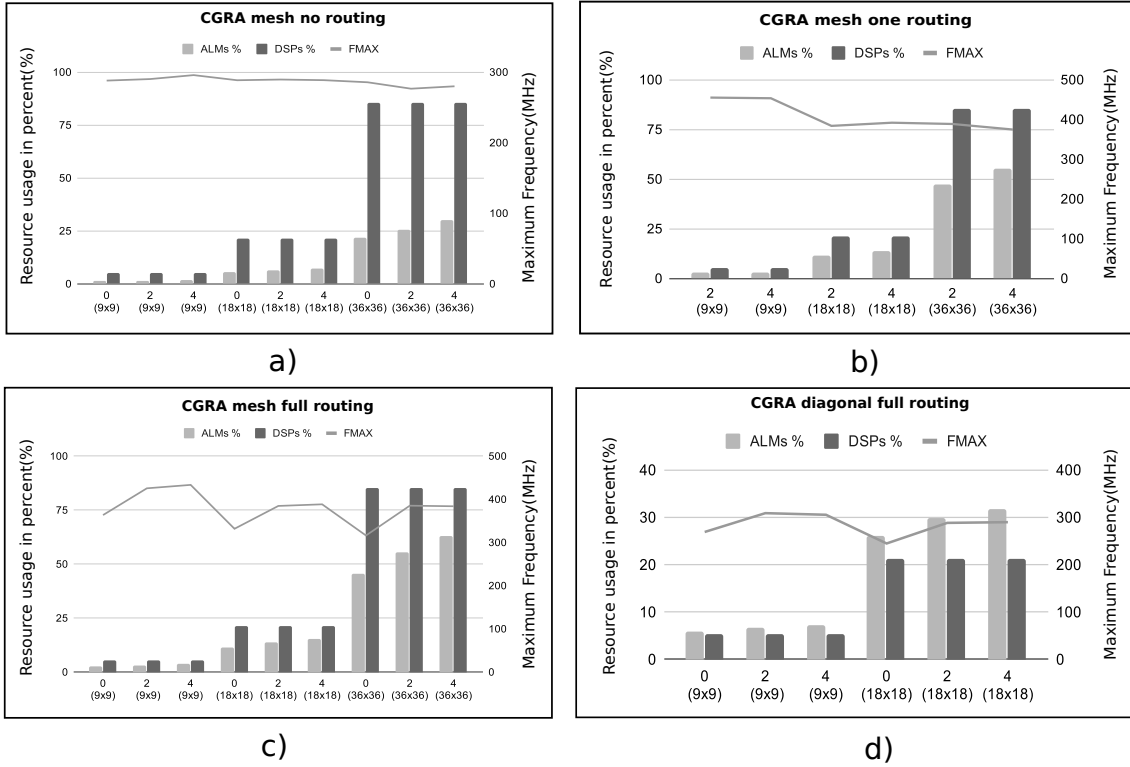


Figure 7. Graph of resource usage and a maximum frequency. Graph a, b and c refers to the Mesh interconnection model; a) without internal routing in the PE; b) with only one entry routing; c) with all routing possibilities; d) Diagonal interconnection model with all routing possibilities.

the Mesh model, it uses more hardware resources per PE, so the model with full routing Figure 8.c was able to synthesize only the sizes 9x9 and 18x18 with the elastic rows of size 0, 2 and 4.

Figure 7.d and Figure 8.d show the use of resources for the Diagonal model. In this model, each PE has eight neighbors. This model has more rout ability than the Mesh model, but uses more resources, for the version "full routing" only the sizes 9x9 and 18x18 fit in the FPGA. The version without routing "no routing" showed a drop in frequency to the extent 36x36. Figures 9 a and b show the results for Hexagonal architecture, in this architecture, each PE has six neighbors and the One hop model. However, the connection pattern between the neighbors is similar to a beehive. CGRAs with this connection pattern has a frequency identical to the One hop pattern, but with fewer variations when increasing the size of the array and the amount of latency of the elastic queues.

5. Conclusion

In this work, we present the HPCGRA tool. A simple way to generate Verilog code for CGRA architectures from a simple JSON description. Our tool is open source and is available on Github¹. The Verilog code generated by the tool is generic and synthesizable for different FPGA platforms such as Intel/Altera or Xilinx. The synthesis results show that the frequency of the architectures is stable even with an increase in the number of

¹<https://github.com/lesc-ufv/hpcgra>

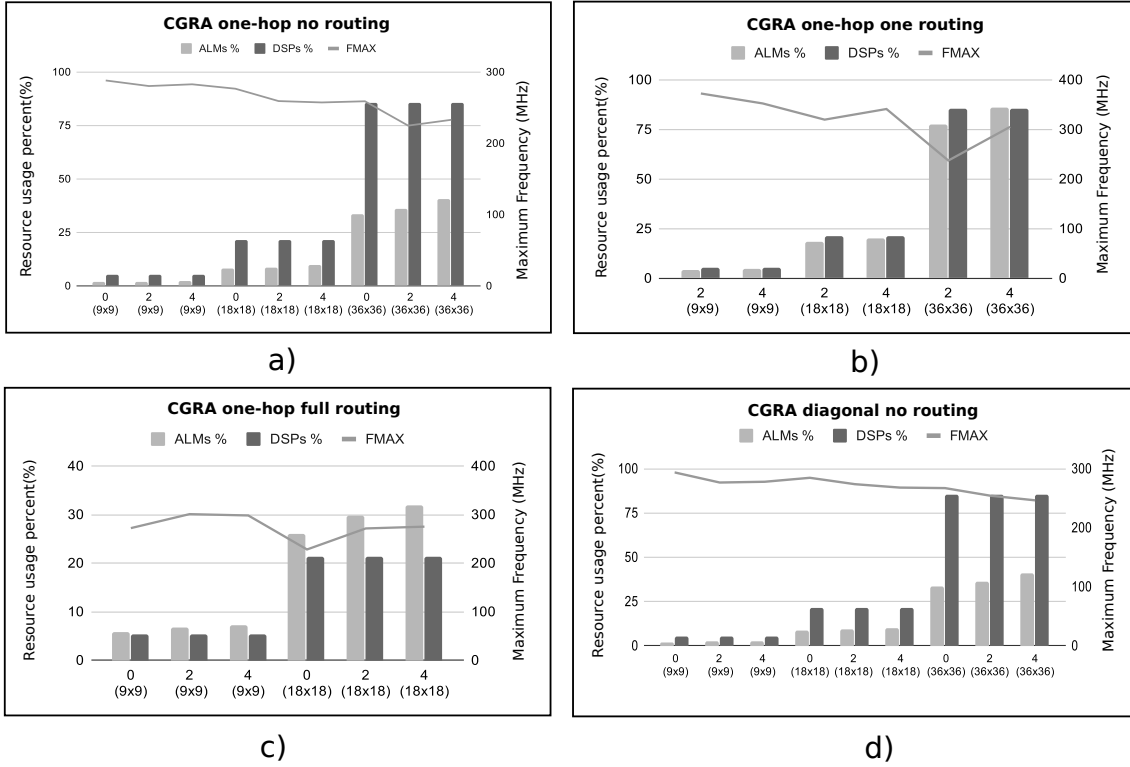


Figure 8. Graph of resource usage and a maximum frequency. Graph a, b and c refers to the One-hop interconnection model; a) without internal routing in the PE; b) with only one entry routing; c) with all routing possibilities; c) Diagonal interconnection model without internal routing in the PE.

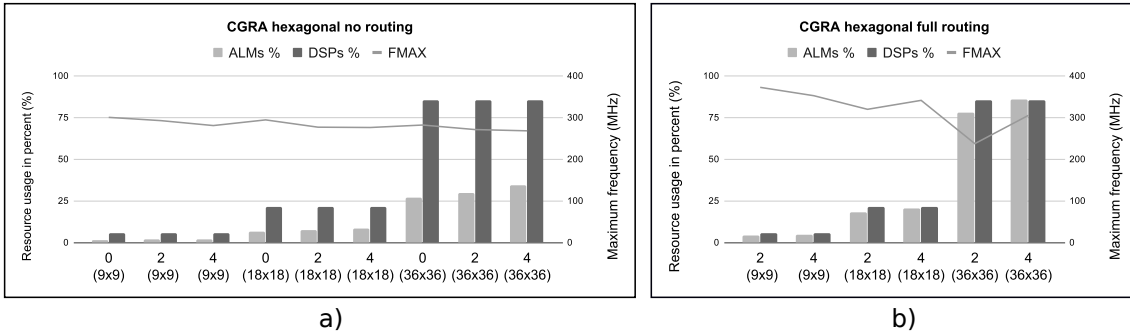


Figure 9. Graph of resource usage and a maximum frequency for CGRA Hexagonal interconnection model; a) without internal routing in the PE; b) with all routing possibilities.

PE of the CGRAs. The generator is scalable, even increasing the size of the array, the ALM / PE ratio remains stable. It was possible to synthesize architectures with 3036 PEs, with a theoretical performance of 1.2 TOPs. As future work, we will add a memory access interface through the AXI protocol facilitating the coupling of the architectures in heterogeneous high-performance computing platforms, such as Intel HARPv2 and AWS F1. Another improvement is the addition of specific functions in place of a PE ALU. For the creation of CGRA specialized in image processing algorithms, we can create a specific PE for applying filters.

References

- Asgari, B., Hadidi, R., Kim, H., and Yalamanchili, S. (2019). Eridanus: Efficiently running inference of dnns using systolic arrays. *IEEE Micro*, 39(5):46–54.
- Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., and Asanović, K. (2012). Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221. IEEE.
- Chin, S. A., Niu, K. P., Walker, M., Yin, S., Mertens, A., Lee, J., and Anderson, J. H. (2018). Architecture exploration of standard-cell and fpga-overlay cgas using the open-source cgra-me framework. In *Proceedings of the 2018 International Symposium on Physical Design*, pages 48–55.
- Chin, S. A., Sakamoto, N., Rui, A., Zhao, J., Kim, J. H., Hara-Azumi, Y., and Anderson, J. (2017). Cgra-me: A unified framework for cgra modelling and exploration. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 184–189. IEEE.
- Genc, H. (2020). A dsl for systolic arrays. <https://github.com/hngenc/systolic-array>. Accessed em: 2020-08-11.
- Jia, L., Lu, L., Wei, X., and Liang, Y. (2020). Generating systolic array accelerators with reusable blocks. *IEEE Micro*, 40(4):85–92.
- JSON (2020). Introducing json. <https://www.json.org/json-en.html>. Accessed em: 2020-07-25.
- Nowatzki, T., Ardalani, N., Sankaralingam, K., and Weng, J. (2018). Hybrid optimization/heuristic instruction scheduling for programmable accelerator codesign. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pages 1–15.
- Takamaeda-Yamazaki, S. (2015). Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In *International Symposium on Applied Reconfigurable Computing*, pages 451–460. Springer.
- Taras, I. and Anderson, J. H. (2019). Impact of fpga architecture on area and performance of cgra overlays. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 87–95. IEEE.
- Weng, J., Liu, S., Dadu, V., Wang, Z., Shah, P., and Nowatzki, T. (2020). Dsagen: Synthesizing programmable spatial accelerators. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 268–281. IEEE.
- Zhang, J., Zhang, W., Luo, G., Wei, X., Liang, Y., and Cong, J. (2019). Frequency improvement of systolic array-based cnns on fpgas. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4. IEEE.