

A Framework for Automated CGRA Design Space Exploration with Genetic Algorithm Optimization

Maryam Katebzadeh, Daniel Vazquez, Andres Otero, Alfonso Rodriguez

Centro de Electrónica Industrial

Universidad Politécnica de Madrid

Madrid, Spain

{m.katebzadeh, daniel.vazquez, joseandres.otero, alfonso.rodriguez}@upm.es

Abstract—The rapid growth of compute-intensive applications has created a pressing need for computing architectures that effectively balance flexibility, efficiency, and performance. While Field-Programmable Gate Arrays (FPGAs) offer a good level of flexibility, they suffer from high configuration overhead and energy consumption. Coarse-Grained Reconfigurable Architectures (CGRAs) provide a more energy-efficient alternative with lower configuration costs. They can be customized for domain-specific applications by modifying their coarse-grained processing elements to execute particular sequences of operations. In fact, their domain-specific nature can be used to further improve their energy efficiency and reduce their area overhead by exploiting computing fabric specialization. This can be achieved by replacing homogeneous processing elements with a subset of heterogeneous, more optimized ones that are specifically suited to the target application domain.

However, achieving an optimal CGRA configuration requires extensive design space exploration (DSE), which involves evaluating many architectural possibilities. Existing CGRA frameworks struggle with slow and inefficient exploration due to long runtimes and constrained customization options. These issues make it hard to find the best configurations rapidly. To tackle these challenges, this paper presents Genetic Algorithm-based CGRA Generator (GA-CG), a framework that enhances DSE in the CGRA design process. GA-CG uses a genetic algorithm to discover an efficient structural configuration, thereby improving resource utilization and reducing power consumption.

Index Terms—CGRAs, Heterogeneous Computing, Design Space Exploration, Genetic Algorithm Optimization

I. INTRODUCTION

As Moore’s Law slows, future processor chips will face limitations not from transistor resources but energy efficiency. Reconfigurable architectures provide a more energy-efficient alternative to general-purpose Central Processing Units (CPUs) by using custom hardware optimized for specific applications. While FPGAs offer high flexibility, their bit-level programmability leads to significant configuration overhead [1]. In contrast, CGRAs operate at the word-level, minimizing configuration overhead and enhancing energy efficiency. CGRAs have become a preferred solution for accelerating compute-intensive workloads in domain-specific applications. For instance, security applications that require support for cryptographic algorithms demand high-performance and energy-efficient hardware [2]. With their balanced combination of flexibility, performance, and power efficiency, CGRAs are

ideal for such tasks. Additionally, CGRAs are well-suited for accelerating machine learning models due to their reconfigurable architecture, which provides the flexibility to efficiently adapt to various workloads that fixed-function accelerators often cannot [3]. To meet the diverse needs of these domain-specific applications, CGRA frameworks typically rely on DSE mechanisms to build custom hardware architectures and evaluate a wide range of configurations and optimizations for specific requirements. They can also include compiler support, simulation tools, and performance analysis tools, providing comprehensive assistance for efficient design, testing, and optimization. However, current CGRA frameworks face several limitations. For instance, they struggle to support flexible architectural design effectively because they mostly rely on traditional Hardware Description Languages (HDLs), such as Verilog or VHDL. These HDLs have limited support for high-level customization and parameterization, making it difficult to manage complex designs. In addition, designers have mainly chosen homogeneous CGRAs (i.e., using the same Processing Element (PE) design in the architecture) due to reduced hardware design complexity. However, this uniformity leads to unused resources. Consequently, there is a growing need to introduce heterogeneity (i.e., using different, specialized PE designs with an application-set-driven distribution) in CGRA generation to enhance resource utilization and optimize power efficiency. In addition, some of the existing frameworks suffer from long runtimes, making them unsuitable for rapid DSE. The core challenge is the absence of a flexible, high-level CGRA design framework that can efficiently support heterogeneous architectures and enable fast, automated DSE.

To overcome these limitations, we present GA-CG, an open-source framework designed for fast and flexible CGRA architecture generation using genetic algorithm optimization.¹ The main contributions include:

- GA-CG, a CGRA design framework that enables rapid DSE and fast architecture generation.
- The use of Chisel, a hardware construction language embedded in Scala, which in turn provides GA-CG with a flexible and highly customizable infrastructure, simplifying the integration of heterogeneous components.

¹Source code available at https://github.com/des-cei/ga_cg_gen.

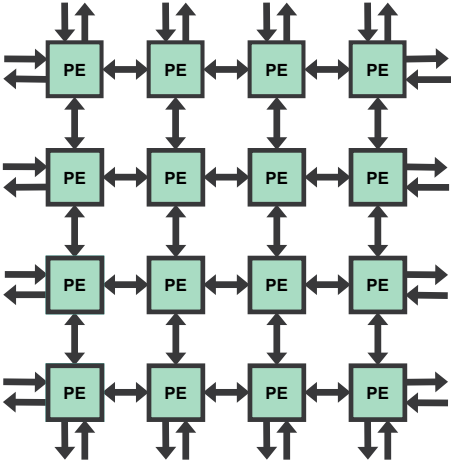


Fig. 1: Sample CGRA configuration featuring N2N topology

- An automatic DSE methodology driven by a genetic algorithm, which efficiently searches large and complex design spaces to find resource-efficient architectures for CGRA designs.

This paper is organized as follows: Section II provides background information on key concepts used in this work, beginning with an introduction to CGRAs and prior CGRA frameworks, followed by a discussion of the differences between homogeneous and heterogeneous architectures, and an overview of Chisel. It also highlights the challenges addressed by our proposed solution. Section III introduces the GA-CG architecture and its components and features. Section IV describes the proposed optimization methodology for automated DSE. Section V outlines the experimental setup of the study and showcases the results obtained. Finally, Section VI concludes the paper.

II. BACKGROUND AND RELATED WORK

This section begins with an overview of the background of CGRAs. A typical CGRA consists of a two-dimensional array of PEs, which can be interconnected according to a specific topology. A PE can be as simple as an Arithmetic Logic Unit (ALU) or as complex as a CPU, depending on the intended application. Common interconnection topologies include neighbor-to-neighbor (N2N), ring, tree structures, etc. CGRAs vary widely and are generally classified into spatially distributed and time-multiplexed designs. In spatially-distributed CGRAs, users need to place and route the application’s Data Flow Graph (DFG) onto the architecture. In contrast, in time-multiplexed CGRAs, users must employ a more complex method to manage instruction scheduling within each PE. To better understand the architecture, Figure 1 illustrates the CGRA structure in this work, which is a spatially distributed CGRA based on a N2N topology.

A. Frameworks for DSE

Given the wide and complex design space of CGRAs, we now focus on existing frameworks that support CGRA DSE,

as these frameworks enable effective navigation of this vast space. This examination also provides important context for the challenges discussed in the introduction. CGRA-ME [4] is a framework that enables high-level modeling and evaluation of CGRA architectures. It supports architectural specification and application mapping, allowing users to analyze performance metrics. However, while CGRA-ME [4] provides a flexible framework for mapping, it does not inherently generate candidate solutions for DSE; rather, it is up to the designer to develop them. The Open-Source Elastic CGRA Generator [5] features a scalable microarchitecture, supports flexible PE configurations, and provides key performance metrics essential for efficient hardware design. Instead of directly modifying the SystemVerilog source code, the generator employs a Python-based parser to handle configuration, which then outputs the corresponding SystemVerilog. This approach adds a layer of abstraction, which can simplify high-level configuration but may make low-level modifications less straightforward. In contrast, many other CGRA frameworks are implemented directly in traditional HDLs, offering more direct hardware control.

B. HDLs for DSE

While traditional HDLs offer precise control, they often result in long development times and costly DSE due to poor support for high-level parameterization and limited abstraction capabilities. To address these issues, newer frameworks are adopting embedded HDLs—hardware description languages built within general-purpose programming languages such as Scala, Python, or Haskell—for greater flexibility, modularity, and productivity in hardware design. For instance, OpenCGRA [6] employs a Python-based HDL, offering more flexibility. However, it is constrained by a pre-defined PE template, which limits architectural customization. These limitations highlight the need for a more adaptable and efficient CGRA development framework. Despite the growing availability of CGRA design tools, there remains a significant gap in frameworks that offer both high-level architectural flexibility and efficient DSE. Their challenges are particularly restrictive for researchers and developers aiming to design application-specific heterogeneous CGRAs. As a result, there is a pressing need for a framework that empowers hardware architects to prototype and explore complex CGRA designs rapidly. To this end, this research proposes a new Chisel-based CGRA framework. Central to this proposed framework is the choice of Chisel as the development language, due to its ability to support the design goals of flexibility and scalability, resulting in less resource utilization.

Chisel [7] is a modern hardware construction language developed at UC Berkeley that brings software engineering principles to digital hardware design. Built on Scala, Chisel enables agile, flexible, and efficient development of complex, application-specific accelerators. Unlike previous frameworks that primarily targeted homogeneous CGRAs, Chisel’s support for parameterized hardware makes it suitable for designing heterogeneous architectures.

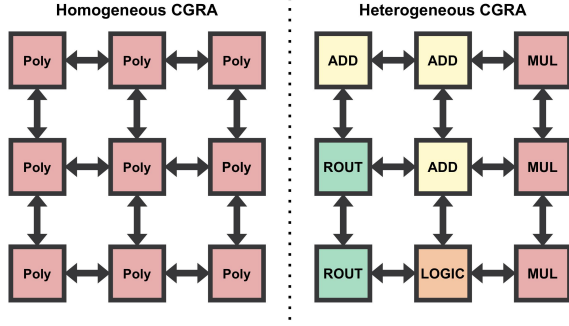


Fig. 2: Homogeneous vs Heterogeneous CGRA

C. Homogeneous vs. Heterogeneous CGRAs

In a homogeneous CGRA (as shown in Figure 2), all PEs are identical in both structure and functionality. In the figure, each PE is of the polyvalent type, programmable, and can execute multiple types of operations with the same set of resources. While this uniformity simplifies design and programming, it often leads to inefficient area utilization and increased energy consumption, as many PEs may carry capabilities that go unused for a given task. In contrast, a heterogeneous CGRA (illustrated in Figure 2) features a diverse array of specialized PEs, each optimized for specific functions. For example, some of them only support addition, while some of them only support multiplication. By deploying only the necessary types of PEs for a given workload, heterogeneous CGRAs minimize redundant resources, resulting in more efficient area usage and lower energy consumption. Since most CGRA frameworks are built using traditional HDLs, exploring and implementing heterogeneous CGRAs becomes a challenging and time-consuming task. This limitation motivates the use of more flexible hardware construction languages such as Chisel, which enable rapid prototyping, better parameterization, and easier DSE for complex architectures. In the following section, we will introduce the GA-CG framework, where DSE of heterogeneous CGRAs is simplified and more efficient due to Chisel's capabilities combined with the proposed infrastructure for heterogeneous architectures.

III. OVERVIEW OF GA-CG

The GA-CG architecture keeps the architecture of the Open-Source Elastic CGRA Generator [5], but uses Chisel instead of SystemVerilog for better and more flexible parameterization.

A. Hardware Structure

GA-CG is spatially distributed and features a N2N topology, as shown in Figure 1. In this CGRA, each PE contains a Functional Unit (FU) capable of executing arithmetic operations such as addition, subtraction, multiplication, shift, and logical operations. PEs can retrieve data either from neighboring PEs or the input nodes. GA-CG implements a valid-ready handshake protocol for PE communication, making the architecture elastic, capable of handling variable data latencies.

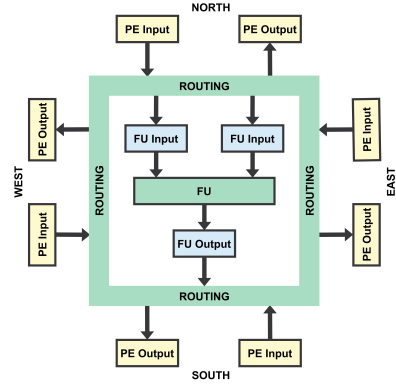


Fig. 3: PE structure

Each PE has input ports from the north, south, east, and west, connected to Elastic Buffers that store both data and control signals. A Fork Sender module ensures the designated output ports are ready before forwarding data. Meanwhile, the FU performs computations using a Join module for synchronizing data streams and an ALU for basic operations. The architecture is parameterizable, allowing designers to define the number of rows and columns of PEs and specify the locations of the CGRA input and output ports. Each PE can either forward data to its FU for computation or bypass it to neighboring PEs, depending on its configuration (see Figure 3). Each PE is configured using three 32-bit configuration words that define routing, operation, and control behavior. An additional 32-bit word provides a unique identifier for each PE, totaling four configuration words per PE.

The CGRA architecture is first modeled in Chisel. Based on this model, heterogeneity is introduced by exploiting Chisel's advanced hardware generation capabilities to only include the specific functions each PE is intended to perform. We call this method **PE specialization**. It uses a configuration matrix to organize PEs into specialized categories—such as addition/subtraction (**A**), multiplication (**M**), shift and logical operations (**L**), routing (**R**), and polyvalent (**P**)—allowing each PE to focus on its designated task. The setup is straightforward to implement in Chisel. As an example, Listing 1 shows the configuration matrix written in Chisel for the 3×3 heterogeneous CGRA illustrated in Figure 2. This matrix defines the different PEs types assigned across the architecture, reflecting the architectural heterogeneity of the CGRA.

```
1 new WithCGRAOpGroups(Array(
2   Array("A", "A", "M"),
3   Array("R", "A", "M"),
4   Array("R", "L", "M")))
```

Listing 1: Example CGRA PE specialization in GA-CG

B. Application Mapping

With the configuration matrix and functional assignment of PEs in the heterogeneous CGRA system established, the next step involves defining how computation is performed

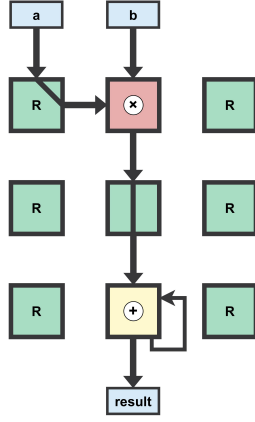


Fig. 4: Mapping of MAC kernel on CGRA

using this PE specialization. To make effective use of the architecture, kernels (representing core computational tasks) must be selected and mapped appropriately. These kernels perform essential operations or algorithms and must be carefully designed to align with the capabilities and configuration of the CGRA. Listing 2 shows the C code for the Multiply-Accumulate (MAC) kernel, and Figure 4 illustrates its optimal mapping onto the CGRA. The figure shows that multiplication is performed in only one PE, since the kernel contains only one multiplication operation. The addition operation also occurs once, while the remaining operations are handled by routing, which uses the least resources. Notably, the datapath exhibits no irregular turns, indicating optimal routing and minimal latency, an important metric considered in the subsequent genetic algorithm evaluation.

The next section explains how a genetic algorithm helps the system explore and determine the best configuration for a given set of kernels.

```

1 void mac(int *a, int *b, int *result) {
2     for (int i = 0; i < 10; i++) {
3         *result += a[i] * b[i];
4     }
5 }

```

Listing 2: MAC kernel

IV. STRUCTURE AND EXECUTION PATH TUNING WITH GENETIC ALGORITHM

This section builds upon the previous discussion on PE specialization to introduce an approach for optimizing the CGRA architecture for a set of computational kernels. This optimization process is based on CGRA-ME [4], which has been extended to include a model of the CGRA generator proposed in this work to facilitate the extraction of both the PE specialization of the CGRA and the configuration bitstreams of PEs necessary for executing each mapped kernel. This method, CGRA-ME [4] modified with the genetic algorithm, enhances resource utilization by optimizing the mapping of a kernel onto the CGRA, considering both the cost of PE

resources and the efficiency of the execution paths chosen for that kernel. In this work, we present a genetic algorithm-based approach to optimize the CGRA architecture by tuning both the PE specialization and the overall data flow path cost: while PE cost typically dominates resource consumption, the length (i.e., the number of hops between CGRA inputs and outputs, including computing and routing-only PEs) of the data flow paths can impact resource utilization and execution latency. To accurately reflect the latter, we track the number of PEs involved across all data flow paths during the kernel mapping. The genetic algorithm searches for CGRA configurations that minimize a combined cost function balancing PE type allocation and data path cost, while ensuring all target computational kernels can be successfully mapped.

The cost function guiding the evolutionary search combines PE cost and path cost using a weighted formula:

$$\text{Cost} = W_{\text{PE}} \times C_{\text{PE}} + W_{\text{PATH}} \times C_{\text{PATH}} \quad (1)$$

where W_{PE} and W_{PATH} are weighting factors. W_{PE} is higher because PE cost includes resource-intensive hardware blocks (e.g., ALUs, registers), whereas path cost only accounts for less demanding interconnections.

Table I presents the post-synthesis resource utilization and the corresponding normalized weights, which represent the PE cost for various PE types. To ensure accurate estimation of individual hardware costs, each PE was synthesized independently using Vivado, thereby eliminating cross-optimization effects.

The weight for each PE type is calculated using the following equation, where LUTs, Flip-Flops, and DSP blocks:

$$\text{Weight} = \alpha \times \text{LUTs} + \beta \times \text{Flip-Flops} + \gamma \times \text{DSPs} \quad (2)$$

with weighting coefficients $\alpha = 1$, $\beta = 1$, and $\gamma = 100$ (LUTs and Flip-Flops are considered lightweight, while costly DSP blocks receive a higher weight). The weights are normalized to a maximum of 20 for easier comparison across PE implementations. The total resource cost across the entire CGRA, denoted C_{PE} , is computed as the sum of the normalized weights of all instantiated PEs in the generated CGRA. Now we define the path cost, which is calculated as 1 for each PE used along the path. For example, the C_{PATH} in Figure 4 is 4, since the path traverses 4 PEs. The optimization process continues until no further improvements are observed, such as no improvement over 20 consecutive generations, or a maximum number of generations is reached. The final output is a heterogeneous CGRA optimized for the target kernel set, balancing PE and path costs.

PE Type	LUTs	Flip-Flops	DSPs	Normalized Weight
R	400	552	0	5
M	950	960	10	15
A	983	960	0	10
L	1660	960	0	13
P	1935	962	10	20

TABLE I: Resource usage of each PE type

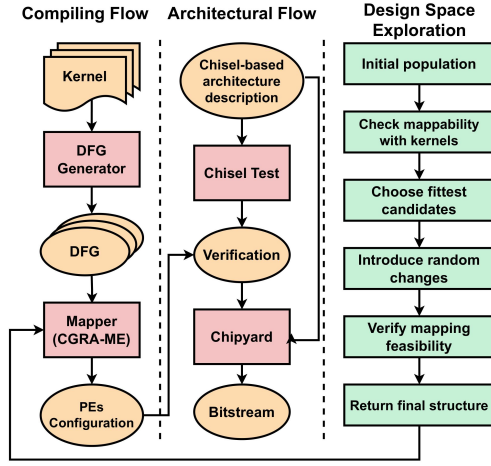


Fig. 5: Proposed DSE methodology

Figure 5 presents the proposed DSE methodology in this work and Algorithm 1 outlines the evolutionary optimization process used to refine CGRA configurations. The genetic algorithm begins with a generated population of homogeneous and polyvalent CGRA configurations (as shown in Figure 2) and refines them iteratively through mutation and selection across generations. Starting with homogeneous, polyvalent CGRAs ensures that applications remain mappable within the given CGRA size; if a larger CGRA is required to support the mappings, this can be identified early in the optimization process.

V. EXPERIMENTAL SETUP AND RESULTS

This section outlines the experimental setup used to validate the proposed framework and discusses the results of resource utilization based on the implemented architecture.

The CGRA generator described in section III is implemented in Chisel and integrated into the Chipyard framework (version 1.10.0) [8] as a submodule. Our CGRA generator model is implemented in CGRA-ME [4] version 2.0 and is based on the foundational work from [9]. We extended this implementation in the present work to support heterogeneous architectures. The Gurobi Optimizer [10] is used in CGRA-ME [4] to optimize mapping efficiently and reduce runtime. The Chisel-based CGRA is instantiated in an IP block named overlay within the Chipyard environment. This overlay block is connected to the Rocket core via the RISC-V Custom Coprocessor (RoCC) interface, allowing the CGRA to function as a coprocessor via custom ISA extensions. The CGRA structure, as produced by the genetic algorithm, is used to configure and control the CGRA instance deployed in the Chipyard platform.

The effectiveness of the proposed framework is evaluated using a set of computational kernels that demonstrate the GA-CG’s ability to efficiently map applications onto the CGRA. These kernels represent cases where applications ei-

Algorithm 1 CGRA Evolution Algorithm

```

1: Input: DFG of the kernels
2: Output: Optimized heterogeneous CGRA
3: Initialize population of homogeneous CGRAs
4:  $best\_cost \leftarrow \infty$ 
5:  $best\_cgra \leftarrow []$ 
6: for  $generation = 1, 2, \dots, G$  do
7:    $offspring \leftarrow []$ 
8:   for each  $individual \in population$  do
9:      $new\_cgra \leftarrow clone(individual)$ 
10:     $mutated \leftarrow mutate(new\_cgra, mutation\_rate)$ 
11:    append  $mutated$  to  $offspring$ 
12:   end for
13:    $combined \leftarrow population \cup offspring$ 
14:    $new\_population \leftarrow []$ 
15:   for each  $cgra \in combined$  do
16:     if  $is\_mappable(cgra)$  then
17:        $cost \leftarrow cost\_function(cgra)$ 
18:       if  $cost < best\_cost$  then
19:          $best\_cost \leftarrow cost$ 
20:          $best\_cgra \leftarrow cgra$ 
21:       end if
22:       append  $cgra$  to  $new\_population$ 
23:     end if
24:   end for
25:   Sort  $new\_population$  by cost
26:    $population \leftarrow$  top 10 of  $new\_population$ 
27: end for
28: if  $best\_cgra \neq None$  then
29:   return  $best\_cgra$ 
30: end if

```

ther fully or partially fit within the CGRA. Table II summarizes the kernels selected for the DSE.

To evaluate the efficiency of the proposed architecture, this part presents and analyzes the resource utilization results for a 4×4 CGRA, which features four input ports on the north edge and four output ports on the south edge. The kernels listed in Table II were evaluated under three scenarios: a baseline homogeneous CGRA made of polyvalent PEs (which was not processed by the genetic algorithm), and two cases where the genetic algorithm was applied and evolved for 20 and 100 generations, respectively. For all three architectures, RTL code was generated using Chipyard and synthesized into bitstreams for the VCU118 FPGA using Vivado. Subsequently, resource utilization was measured and compared. Figure 6 shows the total PE cost of the generated CGRA after 20 and 100 generations of optimization, highlighting that longer evolution leads to improved PE cost efficiency. The results demonstrate that the genetic algorithm improves hardware efficiency, saving significant resources compared to the baseline homogeneous CGRA. A detailed comparison of these results is provided in Table III.

The results in Table III highlight that compared to the baseline homogeneous configuration, LUT utilization drops by

Kernel	Description
mvt	Matrix-Vector Multiplication
bicg	Bi-Conjugate Gradient
gesummv	Matrix sum and Matrix-Vector Multiplication
syrk	Symmetric Rank-K Update
syr2k	Symmetric Rank-2K Update
trmm	Triangular Matrix-Matrix Multiplication
gemm_mul	Multiplication Component in GEMM
gemm_add	Addition Component in GEMM
matmul	Standard Matrix Multiplication
simple_addition	Basic Element-wise Addition
accumulate	Sequential Value Accumulation
cap	Complex Arithmetic Pattern
conv2	2D Convolution
conv3	3D Convolution
mac	Multiply-Accumulate Operation
mults1	Basic Multiplications (Set 1)
mults2	Basic Multiplications (Set 2)
conv2_unrolled	Unrolled 2D Convolution

TABLE II: Benchmark kernels used to assess the proposed DSE methodology

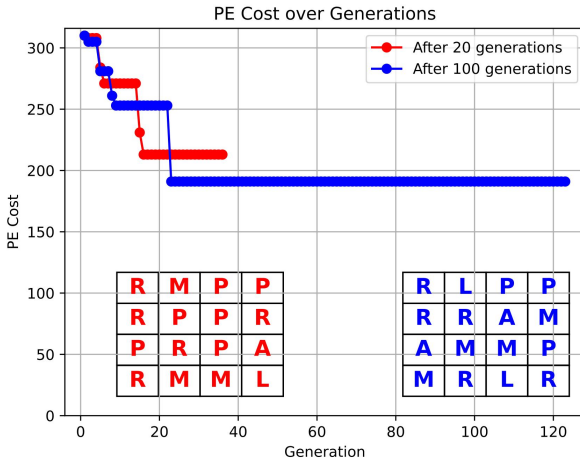


Fig. 6: Evolution after 20 and 100 generations

40.23% after 20 generations of genetic algorithm evolution, and by 47.45% after 100 generations. Flip-Flop usage also declines, with a reduction of 15.01% after 20 generations and 15.05% after 100 generations. The most significant savings occur in DSP blocks utilization: the number of DSP blocks required is reduced from 160 to 90—a decrease of 43.75%—after 20 generations, and down to 70—a 56.25% reduction—after 100 generations. These reductions suggest that the genetic algorithm effectively guides the evolution toward a more heterogeneous application-specific CGRA architecture, thereby reducing overall resource utilization.

Configuration	LUTs	Flip-Flops	DSPs
Polyvalent-type	28853	13739	160
Optimized after 20 generations	17243	11676	90
Optimized after 100 generations	15160	11670	70

TABLE III: Resource utilization of GA-CG generated CGRAs

VI. CONCLUSION

This paper has presented GA-CG, a framework for automated and flexible DSE that employs a genetic algorithm to identify optimal CGRA configurations for a given application set. It facilitates the creation of heterogeneous CGRA architectures, leading to reduced resource utilization while at the same time still proposing feasible solutions. The use of Chisel as the base language for hardware generation makes parameterization and customization easier, allowing the architecture to be more flexible for diverse application needs. Overall, the integration of a genetic algorithm with a Chisel-based design flow accelerates exploration and enhances customization, making GA-CG particularly effective for addressing the challenges of domain-specific computing on reconfigurable architectures.

ACKNOWLEDGMENT

This work has been supported by the A-IQ Ready project, which receives funding within the Chips Joint Undertaking (Chips JU) – the Public-Private Partnership for research, development and innovation under Horizon Europe – and National Authorities (MICIU/AEI/10.13039/501100011033, reference PCI2022-135077-2) under grant agreement no. 101096658.

REFERENCES

- [1] J. Cong, H. Huang, C. Ma, B. Xiao, and P. Zhou, “A fully pipelined and dynamically composable architecture of cgra,” in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2014, pp. 9–16.
- [2] V. T. D. Le, H. L. Pham, T. H. Tran, V. D. Tran, T. H. Vu, and Y. Nakashima, “Ctfe: A high-efficient heterogeneous cryptographic cgra for diverse security applications,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2024.
- [3] Y. Luo, C. Tan, N. B. Agostini, A. Li, A. Tumeo, N. Dave, and T. Geng, “MI-cgra: An integrated compilation framework to enable efficient machine learning acceleration on cgras,” in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, 2023, pp. 1–6.
- [4] J. Anderson, R. Beidas, V. Chacko, H. Hsiao, X. Ling, O. Ragheb, X. Wang, and T. Yu, “Cgra-me: An open-source framework for cgra architecture and cad research : (invited paper),” in *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2021, pp. 156–162.
- [5] D. Vázquez, A. Rodríguez, and A. Otero, “Open-source elastic cgra generator,” in *Proceedings of the 21st ACM International Conference on Computing Frontiers: Workshops and Special Sessions*, ser. CF '24 Companion. New York, NY, USA: Association for Computing Machinery, 2024, p. 83–86. [Online]. Available: <https://doi.org/10.1145/3637543.3652876>
- [6] C. Tan, C. Xie, A. Li, K. J. Barker, and A. Tumeo, “Opencgra: An open-source unified framework for modeling, testing, and evaluating cgras,” in *2020 IEEE 38th International Conference on Computer Design (ICCD)*, 2020, pp. 381–388.
- [7] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzyniak, and K. Asanović, “Chisel: Constructing hardware in a scala embedded language,” in *DAC Design Automation Conference 2012*, 2012, pp. 1212–1221.
- [8] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić, “Chippyard: Integrated design, simulation, and implementation framework for custom socs,” *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.
- [9] R. M. Zamacola Alcalde, “Design methodologies and architectures for just-in-time hardware composition of multi grain reconfigurable accelerators,” July 2022, unpublished. [Online]. Available: <https://oa.upm.es/71045/>
- [10] Gurobi Optimization, LLC, *Gurobi Optimizer Reference Manual*, 2025, <https://www.gurobi.com>.