Design Exploration of Machine Learning Data-Flows onto Heterogeneous Reconfigurable Hardware

Westerley C. Oliveira, Michael Canesche, Lucas Reis, José Nacif, Ricardo Ferreira

[1]Departamento de Informática – Universidade Federal de Viçosa (UFV)
Avenida Peter Henry Rolfs – 36.570-900 – Viçosa – MG – Brazil

{westerley.oliveira,michael.canesche,lucas.t.reis,jnacif,ricardo}@ufv.br

Abstract. *Machine/Deep learning applications are currently the center of the attention of both industry and academia, turning these applications acceleration a very relevant research topic. Acceleration comes in different flavors, including parallelizing routines on a GPU, FPGA, or CGRA. In this work, we explore the placement and routing of Machine Learning applications dataflow graphs onto three heterogeneous CGRA architectures. We compare our results with the homogeneous case and with one of the state-of-the-art tools for placement and routing (P&R). Our algorithm executed, on average, 52% faster than Versatile Place&Routing (VPR) 8.1. Furthermore, a heterogeneous architecture reduces the cost without losing performance in 76% of the cases.*

1. Introduction

Machine learning and deep learning applications such as convolutional neural networks (CNN), matrix multiplications, and others are currently at the center of researchers' attention everywhere. They are used for the most diverse purposes, such as image classification, speech recognition, among others [Krizhevsky et al. 2012, Jo et al. 2017]. Different techniques are used to perform this task, such as parallelizing events on a GPU or an FPGA. Furthermore, domain-specific architectures as systolic arrays are very efficient in parallelizable architectures widely used to compute matrix multiplications [Liu et al. 2020]. However, systolic loses the flexibility to gain efficiency. A coarse-grained reconfigurable architecture (CGRA) loses effectiveness for specific tasks but provides flexibility to a broad range of applications.

This paper focuses on the data-flow graph (DFG) mapping onto CGRAs that, like FPGAs, provide the customization capability of low-level logic systems but with a more straightforward setup/design process. These advantages come from the fact that CGRAs are composed of more complex processing elements (PEs) that can be programmed at the word-level instead of bit-level like in FPGAs [Liu et al. 2019]. Besides all the flexibility of the CGRA, it can be costly depending on the chosen individual PE design or on the whole CGRA grid architecture. For example, PE's with more interconnection options are usually more expensive. Thus, architectures with less of those would end up cheaper. Most CGRA architectures are homogeneous, meaning that every PE can execute any instruction, which can be very expensive. In this work, we perform experiments on heterogeneous architectures to check whether they are suitable to achieve the same tasks with similar performances and lower costs than the homogeneous ones. Our contribution is to provide a design exploration of homogeneous and heterogeneous CGRA for machine learning data-flows, which are the biggest demand for accelerators nowadays. We

propose three heterogeneous architectures. The essential resources are multipliers. The homogeneous architecture is our baseline to evaluate the trade-offs: interconnections, mapping time, buffer resources, and multiplier placement. Our main question is whether a mapping algorithm and a reconfigurable heterogeneous architecture can efficiently implement typical machine learning data-flows. Furthermore, we would like to evaluate the scalability of the mapping for medium size data-flows.

The paper is organized as follows. Section 2 provides a brief description of the data-flow graphs, and Section 3 presents the CGRA architectures. In section 4, we discuss the placement algorithm and how we classified the solutions. Section 5 shows our results and provide a few examples. In Section 6 we compare with others works. And finally, in section 7, we discuss our findings, concluding the paper, and addressing directions for future work.

2. Background

This section provides the basic concepts, and it is organized as follows. First, Section 2.1 presents the matrix multiplication data-flow, which is the most common ML workload. Section 2.2 shows the convolution, which is also a critical ML workload. Section 2.3 introduces the systolic architectures and the trade-offs matrix size, performance, and memory bandwidth. Finally, Section 2.4 draws some insights on embedding tree patterns in mesh architectures.

2.1. The ML data-flow graphs (DFGs)

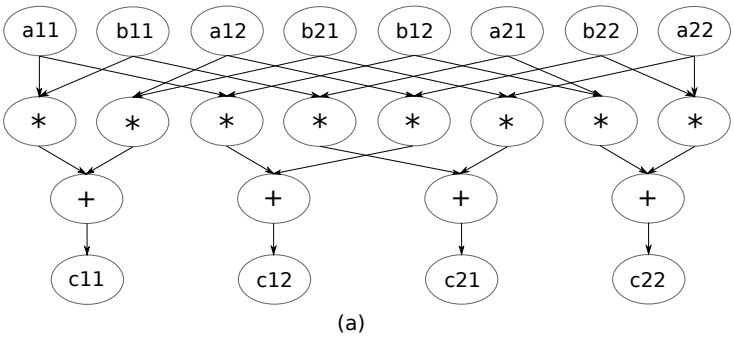
The graphs we worked with, as mentioned before, come from the most computationally intensive parts of machine learning applications.

Each node can be, for example, a multiplier, an adder, a multiplexer, among a few others. Figure 1 (a) shows an example of the data-flow graph for a 2x2 matrix multiplication $C = A \times B$. Every clock cycle, two new matrices A and B are set as the data-flow inputs, and a new result C is produced. The latency depends on the multiplier and adder latencies. Figure 1 (b) shows a table that summarizes the number of I/Os, multipliers, adders, maximum fan-out for an $n \times n$ matrix, where n ranges from 2 to 8. The number of inputs, outputs, multipliers, adders, and fan-out is $2n^2, n^2, n^3, (n^3 - n^2)$, and n, respectively. Therefore, the 4×4 NVidia tensor core has 64 and 48 adders, plus 16 adders to perform $D = A \times B + C$.

The data-flow graphs that compose our set of benchmarks were selected among algorithms kernels that are often used in machine learning applications, such as matrix multiplications and convolutions. Also, we selected binary trees since it is a typical reduction pattern in ML.

2.2. Convolutions

The convolution operation has been considered one of the essential operations in image processing systems and other applications that use convolution neural networks (CNN) [Jo et al. 2017, Krizhevsky et al. 2012]. However, the number of computations grows very fast according to the size of the convolution kernel. This has made researchers [Wei et al. 2017, Chen et al. 2014, Jo et al. 2018] strive to reduce the cost of these operations.



	I/Os	mults	adders	fan-out
2x2	12	8	4	2
3x3	27	27	18	3
4x4	48	64	48	4
5x5	75	125	100	5
6x6	108	216	180	6
7x7	147	343	294	7
8x8	192	512	448	8

(b)

Figure 1. (a)Dataflow graph for a full 2x2 matrix multiplication (b)Data summary for matrix multiplications up to 8x8

The main idea behind the convolution is shown on Figure 2(a). On the 3x3 convolution, a 3x3 filter is applied on the elements of the original matrix. The elements are multiplied element-wise and then summed up to return the result element that is present on the convolved matrix.

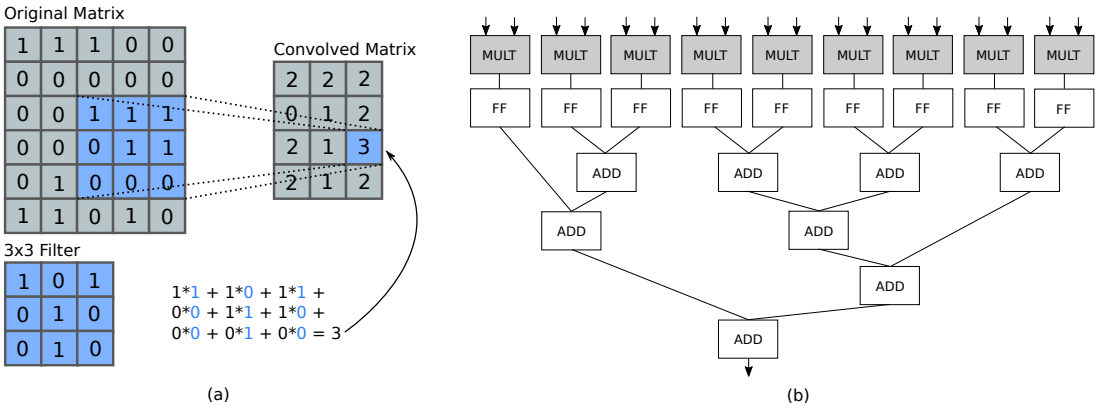


Figure 2. (a) Example of a 3x3 convolution (b) Convolution with K=3 data-flow graph

A convolution DFG is a simple reduction tree of multipliers and adders, as shown in Figure 2(b) for a 3×3 filter. The register layer on this example is present to balance the latency of the reduction tree and the multiplier layer, so the whole DFG returns 1 result after 2 clocks. Having these in mind, Section 2.4 focuses on how to embed trees in two-dimensional architectures. A reduction tree has small data reuse (factor 2) due to a large number of inputs compared to the total number of operations. However, there are opportunities for data reuse as a more significant number of convolution filters over the same input data are presented in most CNN designs.

2.3. Systolic Array

The systolic array is a parallel computing architecture designed to perform a specific task, such as matrix multiplications. It consists of a deeply pipelined network of PE's that, with a regular layout and local communication, can deliver a high throughput [Liu et al. 2018].

The data-flow of Figure 1 (a) can be implemented as a systolic array. However, it requires $O(n^3)$ multiplier/adder units and $O(n^2)$ I/O units to perform one matrix multiplication per cycle. Figure 3 (a) exemplifies a different approach for a 3x3 systolic matrix multiplication with $O(n^2)$ multiplier/adder units and $O(n)$ I/O units, which requires $O(n)$ cycles to compute a matrix multiplication. This approach reduces the I/O requirements. The PE's on the leftmost column receive the elements of each row of matrix A, which will be multiplied by matrix B, stored in each PE. After that, the result is added to the upper PE result and sent it to the next one. The bottom PEs will at each clock cycle release an element of the result matrix C. An FPGA implementation for this design was presented in [Wei et al. 2017].

In Figure 3 (b), we can see how the data-flow graph for this systolic implementation. The multipliers of the PE's on the left are colored in dark grey and represent the input nodes, while the adders of the bottom PE's are the output nodes. Our approach is more flexible, where both matrix multiplier designs (see Figure 1 and Figure 3) could be implemented. It is also interesting to perform sparse matrix multiplication. Recent work [Qin et al. 2020] shows that most matrices are irregular and sparse in deep learning workloads, which could lead to weak mappings on systolic architectures. Also, matrix multiply operations consume around 70% of the total runtime.

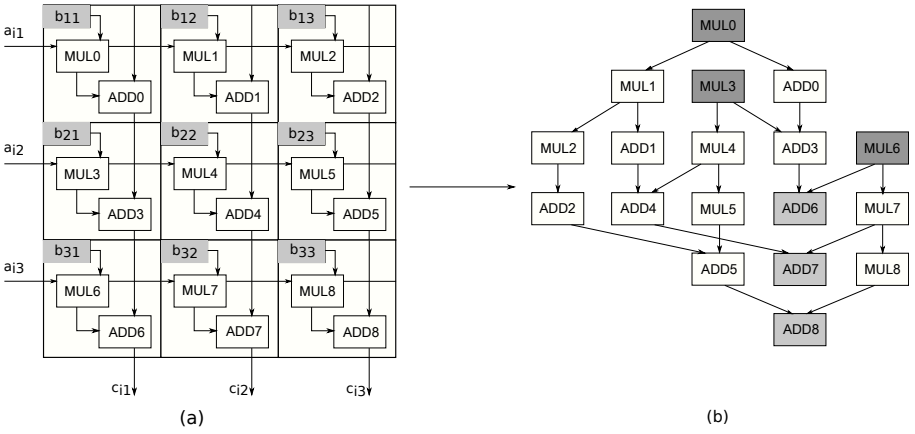


Figure 3. (a) Configuration of a systolic array for matrix multiplication (b) Representation as a data-flow graph

2.4. Trees

Data-flow graphs are not random graphs, and tree patterns are frequently found in parallel algorithms to implement the reduce steps of map-reduce algorithms. The question is how to embed a tree connection pattern in a fully-pipelined mesh architecture? In VLSI circuit research, a well-known approach to balance delay times is the H-tree, which is a self-similar fractal [Browning 1980], as shown in Figure 4(a). Although it has a symmetric layout, the occupancy rate is low, where there are many bypassing cells. For a tree with 16 leaves and 31 nodes in total, the total area is 49 cells, and there are 6 long wires ($cost = 1$ due a bypassing cell) shown in bold in Figure 4(a). As the long wires or bypassing cells are equally distributed, and the tree is fully balanced, there is no delay mismatch, and therefore, there is no need to include buffers to equalize the pipeline path lengths.

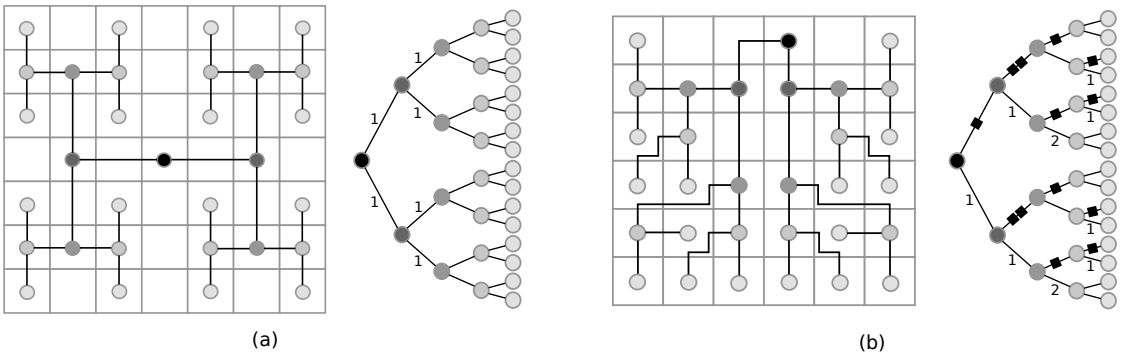


Figure 4. (a) 7x7 mesh and the H-tree configuration; (b) Lee/Choi tree in a 6x6 mesh.

Lee and Choi [Lee and Choi 1996] propose an algorithm to reduce the total area, as shown in Figure 4(b), where the same tree requires only 36 nodes in a 6×6 mesh. Nevertheless, the Lee/Choi tree has different path lengths from the leaves to the root, and it requires nine long routing wires, where two of them have $cost = 2$. When there is a long wire in one pipeline path that converges to one node, we should insert a buffer in the other path to equalize the delay. The black rectangular boxes at the bottom tree in Figure 4(b) represent these delay buffers. The worst case is a buffer of size 2, at the third level.

3. The CGRA

The CGRA is composed of an array of interconnected processing elements (PE's) in a mesh-like network. Unlike FPGAs, that work at a bit level, CGRAs work at a coarse grain, meaning they work at the word level. This fact brings benefits such as reduced mapping for more complex architecture, but on the other hand, very few commercial architectures and design toolchains are available[Liu et al. 2019].

We can categorize CGRA architectures into homogeneous and heterogeneous. In this case, homogeneous means that every PE present on the grid is identical to one another. Thus, each of those PEs can perform the same kind of operations regardless of the grid. In contrast, a heterogeneous configuration presents two or more different types of PE's that can also perform various types of operations.

It seems very straightforward that having a homogeneous configuration is the best way to go because it is less restricted, allowing better placements to be performed. However, this increases the whole architecture cost since every PE should be able to perform all operations. In this work, we propose a design exploration of three types of heterogeneous CGRA architectures and compare the results against the homogeneous case.

In our scenarios, the operation that causes a single PE to be more expensive is the multiplication [Luo and Lee 2000]. Therefore we divided our PE's into 2 groups, the ones that can perform multiplication and those that can not.

In Figure 5(a), all the PE's that are present are identical can perform multiplications, a grey-scale square represents those. Figure 5(b), however, shows multiplier PE's disposed as in a checkerboard, where nodes immediately adjacent to a special PE do not

have a multiplier. We also have a column-wise configuration presented in Figure 5(c) and one on which the multipliers are set to be on the borders of the grid Figure 5(d). At this point, it is worth noting that for all of these cases, the IO nodes of the data-flow graph must be placed on the grid's borders.

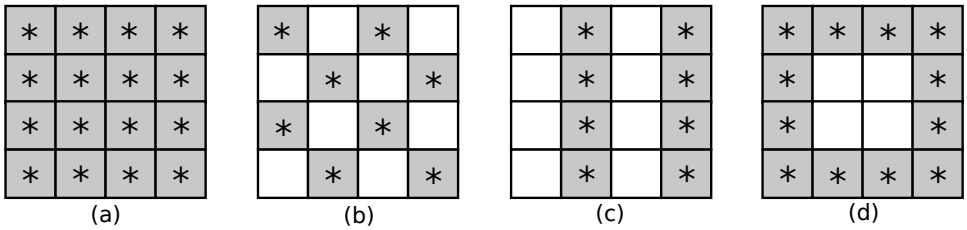


Figure 5. (a) a homogeneous grid; (b) checkerboard; (c) columns; (d) borders.

In addition to the heterogeneous setups proposed, we also perform a design exploration of three different interconnection architectures. Mesh, the most simple, in which each PE can communicate with its 4 immediate neighbors without extra wires. Mesh-plus can jump one cell in each direction, in addition to the four connections present in mesh. And on the hexagonal configuration, each PE's can access 6 other PE's around it, resembling a honeycomb pattern. Figure 6 illustrates these architectures.

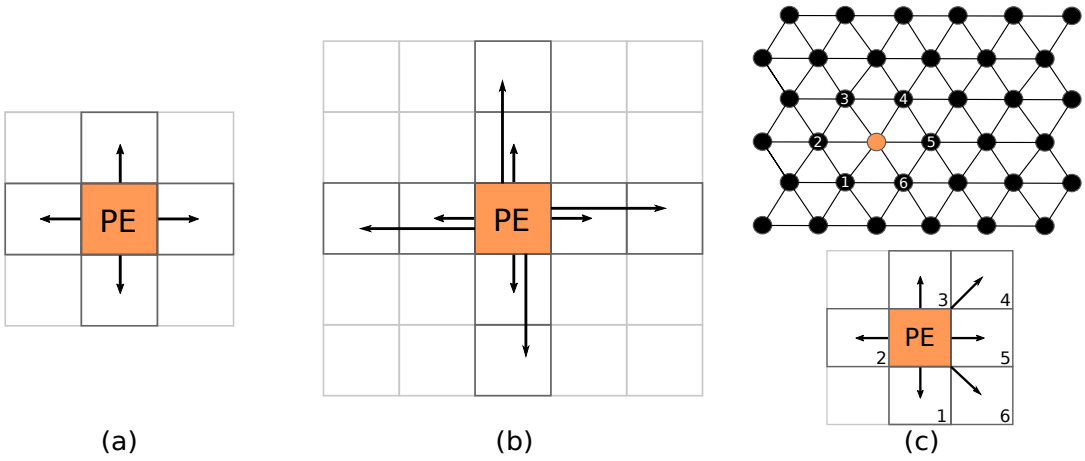


Figure 6. (a) Mesh; (b) Mesh-plus; (c) Hexagonal;

4. Mapping

We divided the mapping problem into two phases, as described in the following. First, to find a suitable placement solution for each proposed data-flow graph, we used a simulated annealing (SA) approach that aims to minimize the wire-length used, similar to previous approaches [Mei et al. 2003].

This was done by checking, at each iteration of the algorithm, if the cost of swapping two nodes would increase or decrease the total length cost of the solution, always keeping track of the positions of the graph's multipliers and IOs.

After that, we proceed to the routing phase, which aims to validate the solution by identifying a viable routing that demands the least amount of delay buffers per PE,

since that is crucial to the total cost of the architecture, even more than the total wire-length [Nowatzki et al. 2018].

For most data-flow edges, the routing is straightforward as the nodes are placed in adjacent cells. Therefore, only a small subset of edges require a routing strategy, where we implement the maze routing algorithm.

5. Experiments and Results

Table 1 summarizes the benchmarks we used in our experiments. A graph for a binary tree benchmark is denoted, for example, tree_n_X_t_X, where the number following n indicates the number of nodes on each connected component. The number following t is the number of connected components. So, following this notation, tree_n_31_t_3 is a graph with 3 binary trees of 31 nodes each. Next, we have systolic matrix multiplications and a classic matrix multiplication like the one shown in Figure 1. Last, we have convolutions ranging from 2×2 matrices to 5×5 matrices.

Table 1. List of benchmarks.

Benchmarks	Nodes	Mults	Benchmarks	Nodes	Mults
tree_n_15_t_1	15	8	matmul22	8	4
tree_n_31_t_1	31	16	matmul33	18	9
tree_n_63_t_1	63	32	matmul44	32	16
tree_n_127_t_1	127	64	matmul55	50	25
tree_n_15_t_2	30	16	matmul66	72	36
tree_n_31_t_2	62	32	matmul22normal	24	8
tree_n_63_t_2	126	64	convolution22	11	4
tree_n_15_t_3	45	24	convolution33	26	9
tree_n_31_t_3	93	48	convolution44	47	16
tree_n_15_t_4	60	32	convolution55	74	25
tree_n_31_t_4	124	64			

For all the benchmarks, the placement algorithm aimed to minimize the square area used. For instance, considering a homogeneous grid and a graph with 18 nodes, the smaller grid is a 5×5 architecture. The minimal size will be fixed by the minimum number of I/Os, multipliers, and/or general-purpose PE's for the heterogeneous ones.

Take for instance the heterogeneous borders model in Figure 5(d). If the side of the grid has size N, one could place at most 4*(N-1) multipliers on its borders. Thus, if a graph happened to have more multipliers than slots available, we had to nudge the grid's dimensions so it could fit. This happened, for example, on the bigger tree graphs where there are many leaves as inputs.

5.1. Trees

An interesting result of our mapping approach is the pattern that the graph tree_n_4_t_15 took after the placement onto the homogeneous grid. The trees followed the H-tree pattern starting from the root and then took a form almost like a puzzle piece. These "puzzle pieces" then arranged themselves so that they could fit tightly into the grid, using it with an occupation of 93.75% on an 8×8 grid against the occupation of 71.43% on a 7×12

grid for the H-tree configuration. Therefore, we reduce the area in 31.25% in comparison to the h-tree. For the same instance, the Lee/Choi [Lee and Choi 1996] requires 12.5% more area with buffer size 2. Our approach does not require buffers in this case. An illustration of this solution is shown in Figure 7.

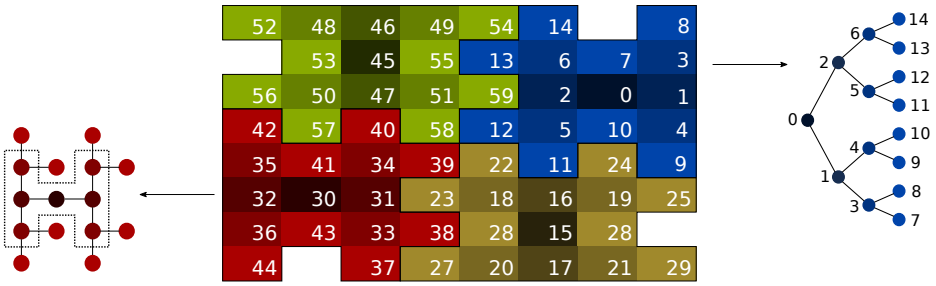


Figure 7. Puzzle-piece pattern for balanced trees on a homogeneous grid

5.2. Data-flows

As a reference to evaluate our simulated annealing implementation, we will compare to the VPR tool [Murray et al. 2020], the state-of-the-art on SA for FPGA. However, our target is a CGRA, although VPR was developed for FPGA, it still out-performs recent CGRA mapping as presented in the CGRA-ME design exploration tool [Chin et al. 2018]. VPR was executed 1000 times for each graph, each time using a different random seed. Other parameters were all set to the tool's standard. Among the 1000 results, the one with the lowest buffer size cost was chosen.

Figure 8 shows a comparison between the runtime (1000 executions) for our placement and the one performed on VPR. It is notable that for all benchmarks, our algorithm outperforms VPR. Especially for the larger graphs, such as tree_n_31_t_4, VPR ends up being very slow, while we were able to perform the placement four times faster. All the experiments were executed using the 16 cores, through openMP, of an Intel(R) Xeon(R) CPU E5-2630v3 2.4GHZ compiled with GCC 5.4.0 and the -O3 option.

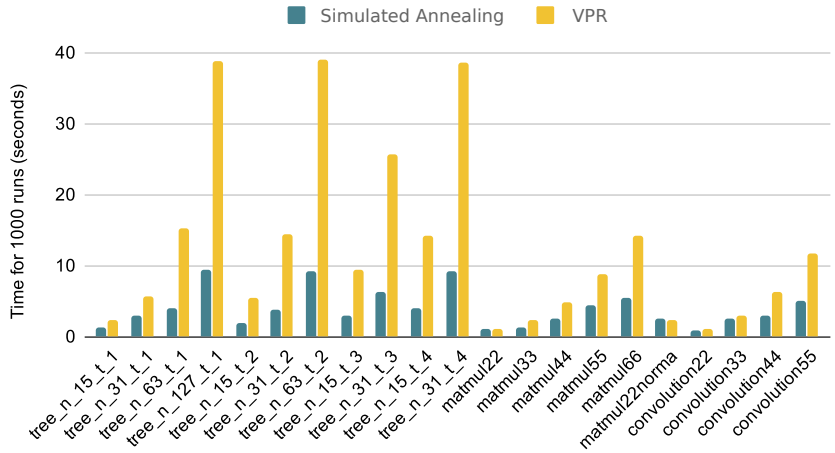
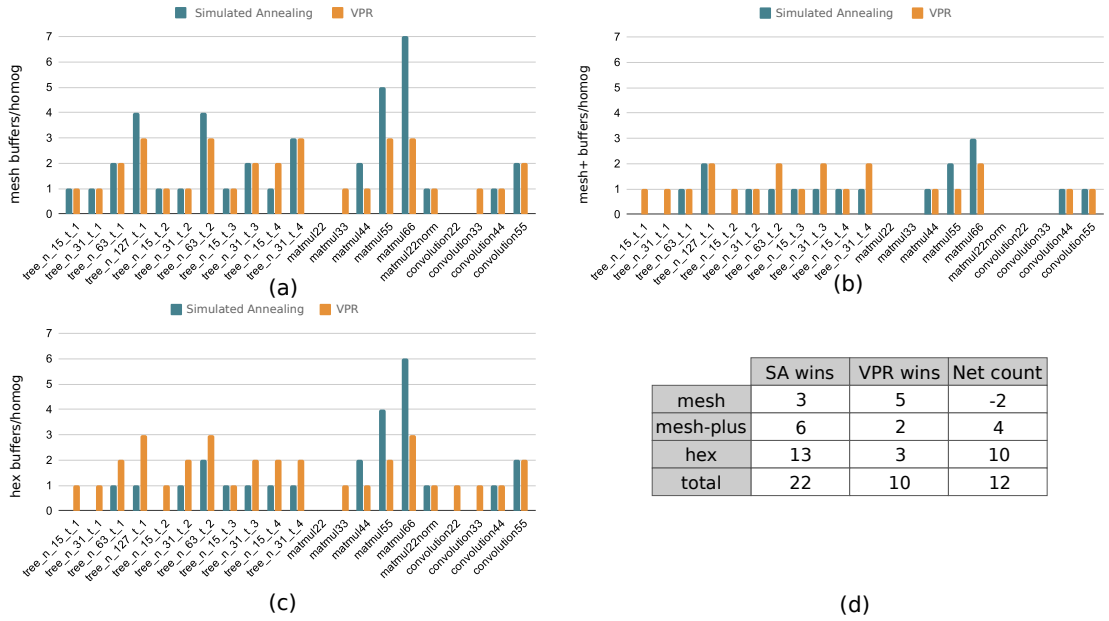


Figure 8. Runtime comparison between VPR and Simulated Annealing

The main goal of this work was to perform a design exploration of heterogeneous CGRAs. Considering the minimal grid size, Figure 9(a), Figure 9(b) and Figure 9(c) shows the minimum number of buffers needed for our algorithm and VPR (both on the homogeneous grid). Note that for most of the cases, our algorithm performed as good or better than VPR, except the mesh configuration. In Figure 9(d), we present a summary of how many times our simulated annealing or VPR won, alongside the net count of those results. For instance, considering the mesh-plus interconnection, our approach is better in 6 out of 21 graphs, VPR is better in 2 cases, and we reach the same buffer size in 13 graphs.



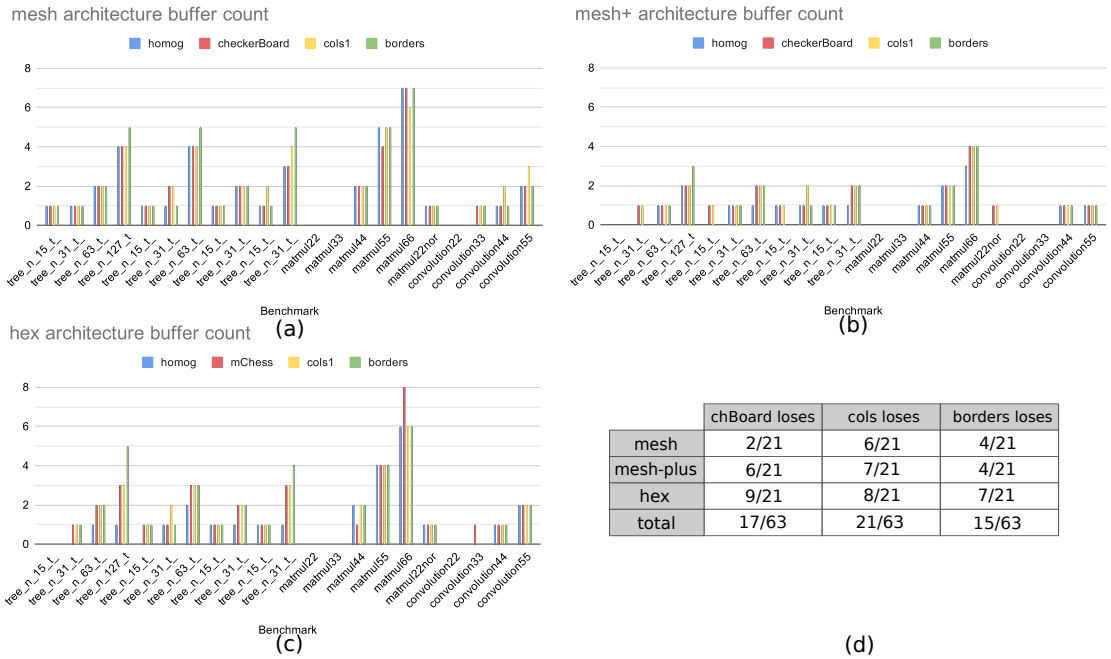
	SA wins	VPR wins	Net count
mesh	3	5	-2
mesh-plus	6	2	4
hex	13	3	10
total	22	10	12

Figure 9. Comparison between homogeneous Simulated Annealing approach and VPR8 [Murray et al. 2020].

Our first experiment was the baseline to evaluate heterogeneous architectures, as a homogeneous solution will be the lower bound. We compared the three heterogeneous architectures (checkerBoard, cols, borders). Firstly, we looked at the buffer size needed while using the three kinds of interconnections, as depicted in Figure 6. In this case, one can infer that most of the graphs performed similarly to the homogeneous case with just a few exceptions like tree_n_127_t_1, in which the borders architecture demanded one buffer more than the homogeneous.

Next, we evaluated the buffer size needed on the heterogeneous architectures for the three interconnection patterns. These results are displayed alongside the homogeneous case on Figure 10(a), Figure 10(b) and Figure 10(c). Note that for most of the benchmarks, there is a heterogeneous setup that performs as good as the homogeneous. In some cases, again, on the larger graphs, it is harder to outperform the homogeneous case cost, but we still got results that are reasonably close to it. A summary with the number of times that each heterogeneous configuration wasn't able to perform as good as the homogeneous is presented on Figure 10(d), where 2/21 for the mesh interconnection and the heterogeneous architecture checkerboard means that the homogeneous one is only better than heterogeneous in 2 out of the 21 graphs. The last line in Figure 10(d)

summarizes these totals for each heterogeneous architecture. The border-based and the checkerboard are more suitable for our benchmark set. This is an interesting result since most of FPGA use column patterns as a heterogeneous design to place the memory and DSPs.



	chBoard loses	cols loses	borders loses
mesh	2/21	6/21	4/21
mesh-plus	6/21	7/21	4/21
hex	9/21	8/21	7/21
total	17/63	21/63	15/63

Figure 10. Number of FIFO's on heterogeneous architectures

6. Related Work

Most CGRA mapping approaches focus on a processing element (PE) architecture shared by multiple instructions in a time-multiplexed fashion [Mei et al. 2003, Park et al. 2008] for homogeneous architectures. In general, the architecture size is small, where the most usual size is a 4×4 grid. Therefore, the maximum number of parallel operations is $4x4 = 16$. Furthermore, the PE should have the capability to decode the instructions from a local instruction memory, which consumes area and power. On the other hand, fully pipelined architectures are configured once before the execution, and the PE will be dedicated to evaluating a single operation, saving area, and power. Moreover, the architecture sizes are more significant than 16 PEs, and the grid size bounds the maximum number of parallel operations. Therefore, this work focus on fully pipelined architectures (FPAs).

The mapping problem for FPAs is a hard problem. A recent work [Nowatzki et al. 2018] shows that integer linear programming (ILP) and satisfiability modulo theory (SMT) approaches are costly in terms of computation time, which could take several minutes for small size graph (less than 20 nodes). These approaches solve the exact problem. Heuristics could be used. Even by using a hybrid approach (ILP plus heuristics) [Nowatzki et al. 2018, Weng et al. 2020] to solve the mapping in a reasonable time (a few seconds) for homogeneous FPAs, the FPA should have large buffers (more than 7 slots). Our approach shows that it is possible to saves

area by providing a design exploration of heterogeneous architecture and small buffers sizes (2 or 3).

A design exploration framework named CGRA-ME was presented in [Chin et al. 2018]. However, the main drawback is the maximum data-flow graph size (less than 25 nodes) and the mapping time (minutes to hours). The mapping strategies have two options: ILP and simulated annealing (SA). However, the SA implementation is inefficient in terms of execution time. VPR [Murray et al. 2020] is state-of-the-art for simulated annealing for FPGA. Our approach is also based on SA, and our results are competitive to VPR in terms of quality and execution time.

7. Conclusion

The results for the heterogeneous setups shows us that it is possible to find good enough placements, for the graphs tested in this work, that reduce the number of multiplier resources on the CGRA grid, therefore reducing its overall cost. A few other experiments that could be done and explored in future work consist of fixing some of the multipliers and/or I/Os on a good position, found on a pre-processing stage. This could be done on graphs that possess a regular and repetitive structure, such as the systolic array. This approach could guide the simulated annealing to a better or even an optimal solution. Future work could also include data-flow merging, which identifies graph similarities to generate a dedicated and straightforward reconfigurable architecture by minimizing the routing resources [Moreano et al. 2005]. We also plan to compare SA-based placement to genetic [Silva et al. 2006] and divide-and-conquer search [Fontes et al. 2018] approaches.

Acknowledgment

This work was carried out with the support of the Coordernação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Financing Code 001. Financial support from FAPEMIG, CNPq, and UFV.

References

Browning, S. A. (1980). The tree machine: A highly concurrent computing environment.

Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., Chen, Y., and Temam, O. (2014). Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGARCH Computer Architecture News*, 42(1):269–284.

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 cgras using the open-source cgra-me framework. In *International Symposium on Physical Design*.

Fontes, G., Silva, P., Nacif, J., Vilela, O., and Ferreira, R. (2018). Placement and routing by overlapping and merging qca gates. In *Int Symp on Circuits and Systems (ISCAS)*.

Jo, J., Cha, S., Rho, D., and Park, I.-C. (2017). Dsip: A scalable inference accelerator for convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 53(2):605–618.

Jo, J., Kim, S., and Park, I.-C. (2018). Energy-efficient convolution architecture based on rescheduled dataflow. *IEEE Transactions on Circuits and Systems I*, 65(12).

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*.

Lee, S.-K. and Choi, H.-A. (1996). Embedding of complete binary trees into meshes with row-column routing. *IEEE Trans on Parallel and Distributed Systems*, 7(5).

Liu, D., Yin, S., Luo, G., Shang, J., Liu, L., Wei, S., Feng, Y., and Zhou, S. (2018). Data-flow graph mapping optimization for cgra with deep reinforcement learning. *IEEE Trans on Computer-Aided Design of Integrated Circuits and Systems*, 38(12).

Liu, L., Zhu, J., Li, Z., Lu, Y., Deng, Y., Han, J., Yin, S., and Wei, S. (2019). A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications. *ACM Computing Surveys (CSUR)*, 52(6):1–39.

Liu, Z.-G., Whatmough, P. N., and Mattina, M. (2020). Systolic tensor array: An efficient structured-sparse gemm accelerator for mobile cnn inference. *IEEE Computer Architecture Letters*, 19(1):34–37.

Luo, Z. and Lee, R. B. (2000). Cost-effective multiplication with enhanced adders for multimedia applications. In *Int Symp on Circuits and Systems (ISCAS)*. IEEE.

Mei, B., Vernalde, S., Verkest, D., De Man, H., and Lauwereins, R. (2003). Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. In *International Conference on Field Programmable Logic and Applications*.

Moreano, N., Borin, E., De Souza, C., and Araujo, G. (2005). Efficient datapath merging for partially reconfigurable architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(7):969–980.

Murray, K. E., Petelin, O., Zhong, S., Wang, J. M., ElDafrawy, M., Legault, J.-P., Sha, E., Graham, A. G., Wu, J., Walker, M. J. P., Zeng, H., Patros, P., Luu, J., Kent, K. B., and Betz, V. (2020). Vtr 8: High performance cad and customizable fpga architecture modelling. *ACM Trans. Reconfigurable Technol. Syst.*

Nowatzki, T., Ardalani, N., Sankaralingam, K., and Weng, J. (2018). Hybrid optimization/heuristic instruction scheduling for programmable accelerator codesign. In *Int Conf on Parallel Architectures and Compilation Techniques (PACT)*.

Park, H., Fan, K., Mahlke, S. A., Oh, T., Kim, H., and Kim, H.-s. (2008). Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Int Conf on Parallel architectures and compilation techniques (PACT)*.

Qin, E., Samajdar, A., Kwon, H., Nadella, V., Srinivasan, S., Das, D., Kaul, B., and Krishna, T. (2020). Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *Int Symp on High Performance Computer Architecture (HPCA)*.

Silva, M., Ferreira, R., Garcia, A., and Cardoso, J. (2006). Mesh mapping exploration for coarse-grained reconfigurable array architectures. In *Int Conf on Reconfigurable Computing and FPGA's (ReConFig)*.

Wei, X., Yu, C. H., Zhang, P., Chen, Y., Wang, Y., Hu, H., Liang, Y., and Cong, J. (2017). Automated systolic array architecture synthesis for high throughput cnn inference on fpgas. In *Design Automation Conference (DAC)*.

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.