A Run-time Hardware Routing Implementation for CGRA Overlays

Mateus Pinto da Silva[1] Maria Dalila Vieira[1] Ricardo S. Ferreira[2] José Augusto M. Nacif[1]
[1]Science and Technology Institute, Universidade Federal de Viçosa, Florestal, Minas Gerais, Brasil
[2]Informatics Departament, Universidade Federal de Viçosa, Viçosa, Minas Gerais, Brasil
{mateus.p.silva, maria.d.vieira, ricardo, jnacif}@ufv.br

***Abstract*—Accelerators became a wide-reaching solution for increasing computing systems' performance. However, they bring the trade-off between programming facility versus energy efficiency. FPGAs are highly energy-efficient accelerators, but complex to program. CGRA Overlays offers a more straightforward programming interface for FPGA and can use dataflows graphs as input. Mapping a dataflow graph requires three NP-Completes challenges: scheduling, placement, and routing. We present a greedy finite state machine hardware algorithm for CGRA Routing. Our goal is to produce a viable solution for a reliable FPGA use. Our routing approach reduces execution time and increases portability with a low area overhead. We achieve 3x speedup than a high-end commercial CPU, being able to route 86% of the graph edges on the CGRAME benchmarks.**

***Index Terms*—CGRA routing, reconfigurable architectures, FPGA accelerators**

I. Introduction

In recent years, with the rapid developments in society and technology, the demand for computer performance has continuously increased with the emergence of more complex challenges every day. In this context, heterogeneous platforms, consisting of one (or multiple) CPU(s) co-working with some accelerator has become a wide-reaching solution. However, there is a crucial trade-off in computer accelerators: programming facility versus energy efficiency [1].

Among computer accelerators the FPGA (Field-Programmable Gate Array) architecture has high energy efficiency, and it is run-time reconfigurable, which guarantees flexibility. On the other hand, it requires hardware knowledge of the programmer, and plenty of time to code, synthesize, and debug [2], [3]. CGRA (Coarse-Grained Reconfigurable Architecture) is a low granularity implementation of FPGAs, which decreases programming complexity and synthesize costs of the architecture, keeping most of the efficiency and the flexibility. However, there are few commercial chips available. CGRA can also be an overlay for FPGA, which reduces granularity [4], [5].

Mapping an algorithm into a CGRA requires three NP-complete challenges: scheduling, placement, and routing. Scheduling could be implemented by using time-multiplexing and/or fully pipelined [6]. Placement maps the data-flow nodes onto the architecture processing elements or functional units [7]. Routing is connecting data-flow nodes in the platform, with respect to the restrictions on the chosen CGRA layer. One example is the number of bypass connections in Processing Elements (PEs). Assuming that N is the gridline size, we can route 86% of the edges in linear time for a problem with an $O(2^N)$ complexity, using a replicable state machine that repeatedly tries to route the graph, marking the input as resolvable or non-resolvable in linear time.

We propose a hardware approach greedy finite state machine routing algorithm for CGRAs. Our state machine runs in linear time and can route most CGRA placements. Thus, we intend to try repeatedly routing until we find a solvable one in a set of finite Placement attempts, which makes our solution possible to replicate, using a strategy of division and conquest. Based on this, we improved the possibility of reconfiguring the CGRA in run-time without requiring any changes to the front-end compiler. We perform the entire routing process is inside the chip. The main contributions of this work are (1) Hardware-based and incremental architecture-independent CGRA routing; (2) Dynamic CGRA overlay improvement. Any FPGA system can include a CGRA overlay. In this case, it is possible to place the entire mapping algorithm on the chip, not just the routing. There will be an area overhead for the algorithm. However, the creation and transfer of the data flow configuration is faster, in addition to CPU independent, keeping all FPGA costs inside the chip.

We organize the remaining of the paper as follows: Section II explains the CGRA architectures and data-flow routing problem. Section III presents the related work. Section IV presents our routing approaches in more detail. In Section V, we present and discuss the results. Finally, in Section VI, we close our remarks and propose future work.

II. Background

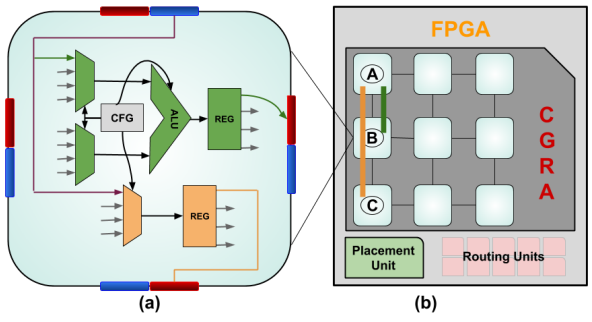

Fig. 1. (a) PE with B configuration; (b) Mesh CGRA.

A CGRA is an array of processing elements (PEs) interconnected by a network [2], [3]. These PEs are reconfigurable at the word level, while FPGA's standard cells are reconfigurable at the gate level. This decrease in granularity makes the mapping simpler. However, it sacrifices the flexibility of the chip. A PE consists of one Arithmetical Logic Unit (ALU), some bypasses, and configuration memory, as Figure 1(a) shows.

A processing element can communicate only with its neighbors. This local connection is shown in green in Figure 1. However, for several graphs, communication between non-neighboring PEs is necessary and can be achieved using the bypass structure, which passes a connection through the PE (s), as shown in orange in Figure 1. A bypass is a hardware structure that temporarily stores the input value of a neighboring PE and sends it to other adjacent PE (s), either to its ALU (s) or even bypass (es). The more bypass a PE has, the easier it is to route the CGRA, but the PE occupies more area of the FPGA, decreasing the number of PEs that the CGRA Overlay can have. In our example, we use one bypass structure. The difference between the order of the algorithm graph and the number of CGRA PEs results in unused ALUs, which can function as a bypass structure using a move function, facilitating routing. An architecture-independent algorithm for CGRA mapping must consider or be independent of the bypass structure per PE quantity.

All data transfers between the main memory and the chip occur in the CGRA borders, and the source and destination nodes must be there [2], [3]. It is possible, as well, to put constants on the PE registers before the algorithm execution, in both ALU and bypass registers. The CGRA can run the same algorithm with multiple data, similar to a Single Instruction, Multiple Data processor, and achieve higher performance working this way. It is possible to change the algorithm on-the-fly depending on mapping time. An FPGA system could consist of a complex CGRA Overlay, one or multiple placing units, and multiple routing units, as Figure 1(b) shows, to make the mapping faster and keeps it cost inside the FPGA.

III. Related work

Maurice Hanan *et al.* [6] aim to use spanning tree, Steiner tree, and special trees which satisfy a particular degree requirement. That approach uses Prism, Kruskal, and other algorithms to find routing for each graph node. It works on most architectures. However, it is implemented in software, because implementing in hardware is out of the scope of this paper. Our solution is simpler and suitable for hardware implementation, being a replicable low resource-consumption finite state machine.

Zhongyuan Zhao et al. [7] use a specific CGRA Overlay and does the mapping in the same operation, achieving efficiency in balanced algorithm graphs. Nevertheless, it is not an architecture-independent solution, working only to restrict CGRAs. Our algorithm works whatever is the target CGRA, just requiring two constants about the chip: the grid size and the max bypasses per PE.

Stephen Friedman et al. [8] employ the heuristic QuickRoute aside with PathFinder to solve CGRA routing. Besides, it is generic enough to be implemented as architecture-independent, although it is not simple for a hardware implementation [6]. Both works implement software-based approaches. Our solution is hardware-implemented, guaranteeing high speed because it is high frequency. We present a low resource-consumption high clock frequency finite state machine for CGRA routing that is architecture-independent implemented in hardware, working in all CGRA.

IV. Architecture-independent greedy routing approach

When using a CGRA, the compiler generates the data-flow graph from a programming language source code, as Figure 2(a) shows. After that, we send the graph to the placement unit inside the FPGA. So the placement unit performs one placement attempt. If the algorithm graph has an edge between two neighbors PEs, it will be routed in the placement, as presented in Figure 2(b). These edges are called trivial. Then, the routing unit tries to route in linear time. If this routing possible, we achieve the process end, as Figure 2(c) shows, so we have the CGRA network. Otherwise, we must generate another placement. The routing can be easier, harder, or even unsolvable based on placement results.

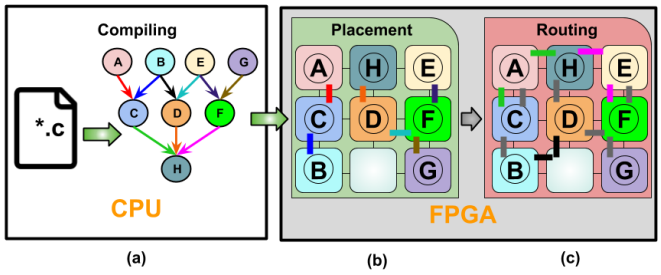

Fig. 2. (a) Graph generation; (b) Placement done; (c) Routing done.

We primarily developed our routing finite state machine in C Language, to make the debugging and testing easier. It is possible to count clock cycles in this implementation. However, the clock frequency and the FPGA area is only measurable using a hardware description language. We have written one hardware implementation in System Verilog HDL and generated two others from our version with imperative programming language using the High-Level Synthesis Tools Legup, a software that can convert C Language code to hardware description language code.

Routing a graph edge consists of finding a viable path through PEs bypasses, inputs, and outputs from source to the destination node. For every PE input, there must be a PE output connected to it. Because of that, a CGRA abstraction for routing can consider just one of them. We arbitrarily chose using PE's outputs. Our machine abstracts the CGRA grid in an array of elements with four booleans representing the four outputs from PEs and one integer for the number of used bypasses. The array size is the grid size, which is one of the

two constants required by our FSM (Finite State Machine). The other one is the max bypass quantity per PE, including the free ALUs to be used with the move function. The input is the graph edge list and the pre-filled grid, with information about used outputs from trivial edges and used ALUs by placement.

Algorithm 1 Pseudo-Code for the Routing Algorithm

```
0: function IsAdvancePossible(Orientation, Source)
0: if IsOutputFilled(Source) or (UsedBypass == MaxBypass and
   isBypassRequired()) then
0:     return False;
0: else
0:     return True;
0: end if
0: function Advance(Orientation, Source)
0: FillOutput(Source);
0: UsedBypass++;
0: function main()
0: Source, Destination ← GetNextEdge()
0: while NextEdge is not (0,0)(0,0)] do
0:     Source, Destination ← GetNextEdge()
0:     Modified ← False
0:     while Souce != Destination do
0:         while IsAdvancePossible(X, Source) do
0:             Advance(X, Source)
0:             Modified ← True
0:         end while
0:         while IsAdvancePossible(Y, Source) do
0:             Advance(Y, Source)
0:             Modified ← True
0:         end while
0:         if !Modified then
0:             EraseEdge(Source, Destination)
0:             Break
0:         end if
0:     end while
0: end while
0: =0
```

Algorithm 1 shows our routing pseudocode. The first outer loop represents the iteration through all graph edges waiting for routing. The Modified variable is just a flag for marking if the algorithm made any movement, started with the false value. First, the algorithm reads a graph edge. The second outer loop tests if the Source edge is already in the Destination. If it is not, the routing unit moves it. First, we arbitrarily chose to start moving the edge on the x-axis while it is possible.

We define this possibility by checking the occupation of the outputs and bypass structures. We cannot perform the movement if the desired output is not available, or the number of occupied bypasses is equal to the max. Otherwise, the movement can happen. The first and the last movements are the only ones that do not use bypasses because we perform the connection directly in PE's ALU. The algorithm does the same on the y-axis. In our debugging version, if the algorithm did not move from the source, that graph edge will be entirely removed from the CGRA, and the next edge will be requested.

We propose a sixteen states hardware approach greedy finite state machine to implement this algorithm, as shown in Figure 3. In a high-level, the NX State, the FSM request the next edge. If it is a reflexive one, the algorithm jumps to END, which means it finishes. The X state walks with the node in the x-axis while there are free bypasses and free outputs or if the connection is horizontally aligned. Y state

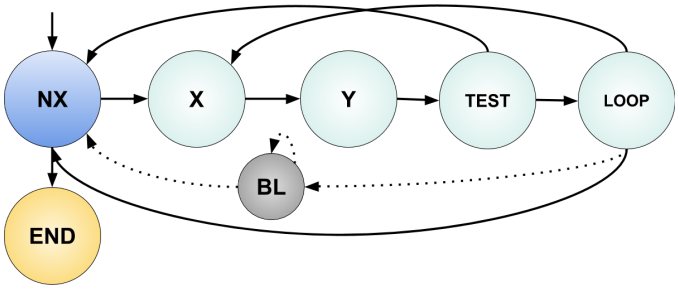

Fig. 3. A high-level description of our FSM.

does the same but on the y-axis. Test state checks if the current node is the destination one. If it is, the connection was successful. Otherwise, it proceeds to the Loop stage. The loop stage is responsible for stairway moves. The machine calls X state again the routing does any movement since the last X state. Otherwise, it will jump to the Blacklist (BL) state. The Blacklist state recursively clear all node attempts.

The Blacklist state is also just for debugging purposes. In our implementation, we used it for calculating non-routed edges. In a commercial version, there is no need for it, since when it is impossible to route an edge, the algorithm would abort and call for another placement, so the condition for calling Blacklist state would call End State. Also, it is possible to implement a Prefetch state before the NX state for caching the next input edge, costing some area in the chip, but making the algorithm execution faster by overlapping memory accesses.

V. Experimental Results

We tested our solution using our handwritten hardware description code and Legup output on the cutting-edge Intel Arria10 FPGA. Unfortunately, due to the lack of such an accelerator in the literature, we used the commercial Intel High-end CPU I7-7700HQ. We reached 200MHz on the FPGA using the handwritten code. The Legup code reaches the same frequency. However, the number of cycles was, on average, 10x higher, in addition to several errors in the compilation, making the tool practically unfeasible for our application. The hardware implementation achieves, in average, 3.8x the performance of the CPU time.

All of the runtimes tests were done using the benchmarks from CGRAME. The Table II shows the results. We produced great routed edges results for a very small machine. The more is the number of Empty ALUs, the algorithm can take more route possibilities because of the increased number of bypass structures (see Section II), observed in the increased number of routed edges. The clock cycle quantity is directly related to the number of non-trivial graph edges. This shows the importance of a precise Placement Algorithm.

TABLE I
FSM TIME RESULTS ON CGRAME BENCHMARKS.

Benchmark names	Clock cycles	CPU time (ms)	Verilog handwritten time (ms)	Legup time (ms)
accumulate	117,24	1,89	0,59	5,86
cap	204,80	2,50	1,02	10,24
conv2	103,50	1,40	0,52	5,18
conv3	171,00	2,10	0,86	8,55
mac	70,90	2,00	0,35	3,55
mac2	170,40	2,10	0,85	8,52
matrixmultiply	84,30	1,40	0,42	4,22
mults1	212,40	2,20	1,06	10,62
simple2	97,20	1,20	0,49	4,86
sum	30,80	1,50	0,15	1,54
twoloops1	109,20	1,40	0,55	5,46
twoloops2	80,20	2,50	0,40	4,01

TABLE II
FSM EDGE RESULTS ON CGRAME BENCHMARKS.

Benchmark Names	Empty ALUs	Edges				CGRA Usage
		Trivial		Routed		
		Y	N	Y	N	
Accumulate	7,00	12,10	9,90	87,27%	12,73%	22,60%
Cap	2,00	14,40	14,60	80,34%	18,97%	30,20%
Conv2	0,00	9,00	9,00	87,78%	12,22%	32,81%
Conv3	0,40	14,20	12,80	83,70%	16,30%	29,20%
Mac	5,00	6,10	6,90	90,77%	9,23%	20,78%
Mac2	1,00	14,70	15,30	79,33%	16,33%	27,20%
MatMultip	7,00	11,30	7,70	91,58%	8,42%	20,20%
Mults1	6,00	18,10	16,90	81,71%	18,29%	25,00%
Simple2	3,00	7,10	6,90	85,71%	14,29%	25,47%
Sum	2,20	4,10	3,90	95,00%	5,00%	19,44%
Twoloops1	0,00	8,50	9,50	81,11%	18,89%	25,31%
Twoloops2	7,00	14,20	6,80	90,48%	9,52%	21,00%

VI. CONCLUSION

We proposed a hardware approach greedy finite state machine routing algorithm for the CGRA mesh. We aim to implement placement, routing, scheduling ad the CGRA layer over the same FPGA. Thus, we can exploit the FPGA reconfigurability while we execute several instances of the mapping. So we can achieve solutions each time near to the optimal solution. Our solution is competitive because we achieve a high number of routed edges with a small resource-consuming finite state machine. We aim to replicate it for repeatedly try the routing until we find one solvable in a finite grid set. Besides, we stand out from the other works for implementing a low resource-consumption high clock frequency architecture-independent finite state machine CGRA routing, that require information about the grid size and the max bypasses per PE. Moreover, we offer a generic implementation that is architecture-independent, working on all Mesh CGRAs.

As future work, we aim to adjust our hardware to the N-hop architecture to be a more generic CGRA routing algorithm. This architecture provides a direct connection between all pairs of vertices with distance N+1 in the mesh. Furthermore, we should improve our solution to detect obstacles and provide alternative routes. Also, we intend to implement multicasting that consists of the reuse of paths with the same origin node, to use fewer CGRA resources, and route more graph edges. We also aim to work with QuickRoute and PathFinder heuristics, trying a simpler implementation of them to work on hardware, and accelerate even more the CGRA routing process.

ACKNOWLEDGMENTS

We thank LegUp Computing for providing the evaluation license for the commercial version of the tool. We also thank CNPq, CAPES and FAPEMIG for the financial support and the companies Intel and NVIDIA for access to their tools.

REFERENCES

[1] J. Cong, Z. Fang, M. Lo, H. Wang, J. Xu, and S. Zhang, "Understanding performance differences of fpgas and gpus," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018, pp. 93–96.

[2] L. Liu, J. Zhu, Z. Li, Y. Lu, Y. Deng, J. Han, S. Yin, and S. Wei, "A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications," *ACM Computing Surveys (CSUR)*, vol. 52, no. 6, pp. 1–39, 2019.

[3] A. Podobas, K. Sano, and S. Matsuoka, "A survey on coarse-grained reconfigurable architectures from a performance perspective," *arXiv preprint arXiv:2004.04509*, 2020.

[4] L. B. da Silva, F. Alves, J. A. Nacif, F. Passe, V. C. Vasconcelos, and R. Ferreira, "Cgra harp: Virtualization of a reconfigurable architecture on the intel harp platform."

[5] L. B. D. Silva, R. Ferreira, M. Canesche, M. M. Menezes, M. D. Vieira, J. Penha, P. Jamieson, and J. A. M. Nacif, "Ready: A fine-grained multithreading overlay framework for modern cpu-fpga dataflow applications," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–20, 2019.

[6] Z. Zhao, W. Sheng, Q. Wang, W. Yin, P. Ye, J. Li, and Z. Mao, "Towards higher performance and robust compilation for cgra modulo scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 9, pp. 2201–2219, 2020.

[7] M. Hanan and J. M. Kurtzberg, "A review of the placement and quadratic assignment problems," *Siam Review*, vol. 14, no. 2, pp. 324–342, 1972.

[8] S. Friedman, A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling, and S. Hauck, "Spr: an architecture-adaptive cgra mapping tool," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, 2009, pp. 191–200.