Canal: A Flexible Interconnect Generator for Coarse-Grained Reconfigurable Arrays

Jackson Melchert, Keyi Zhang, Yuchen Mei, Mark Horowitz, Christopher Torng, and Priyanka Raina

***Abstract*—The architecture of a coarse-grained reconfigurable array (CGRA) interconnect has a significant effect on not only the flexibility of the resulting accelerator, but also its power, performance, and area. Design decisions that have complex trade-offs need to be explored to maintain efficiency and performance across a variety of evolving applications. This paper presents Canal, a Python-embedded domain-specific language (eDSL) and compiler for specifying and generating reconfigurable interconnects for CGRAs. Canal uses a graph-based intermediate representation (IR) that allows for easy hardware generation and tight integration with place and route tools. We evaluate Canal by constructing both a fully static interconnect and a hybrid interconnect with ready-valid signaling, and by conducting design space exploration of the interconnect architecture by modifying the switch box topology, the number of routing tracks, and the interconnect tile connections. Through the use of a graph-based IR for CGRA interconnects, the eDSL, and the interconnect generation system, Canal enables fast design space exploration and creation of CGRA interconnects.**

***Index Terms*—Accelerator architectures, data flow computing, hardware acceleration, high performance computing, reconfigurable architectures.**

I. Introduction

COARSE-GRAINED reconfigurable arrays (CGRAs) have been studied heavily in recent years as a promising configurable accelerator architecture [4], [7], [8], [13]. The end of Moore's law necessitates the creation of specialized hardware accelerators to enable running increasingly complex image processing and machine learning applications. While a variety of hardware accelerator architectures exist, CGRAs have emerged as an interesting midpoint between the flexibility of an FPGA and the performance and energy-efficiency of an application-specific accelerator. A CGRA can achieve high energy-efficiency and performance due to word-level arithmetic operations and interconnect, while maintaining enough flexibility to run a variety of applications that evolve over time [2].

CGRAs, as well as other spatial accelerator architectures, often have hundreds of compute cores and memory cores. These compute cores (called processing elements or PEs) and memory (MEM) cores are laid out spatially in a grid of tiles and are connected through a configurable interconnect. An example is shown in Fig. 1. The reconfigurable interconnect contains switch boxes (SBs), which connect the PE/MEM outputs to the tracks in the interconnect, and connection boxes (CBs), which connect the interconnect tracks to the inputs of the cores. While having a large number of compute cores enables very high performance, the reconfigurable interconnect connecting these cores can constitute over 50% of the CGRA area and 25% of the CGRA energy [13]. Design space exploration of the interconnect is necessary to achieve high performance with lower energy and area costs.

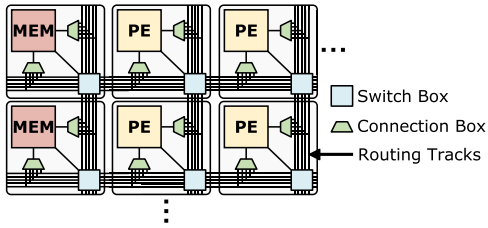

Fig. 1. Architecture of a CGRA with PE tiles, memory tiles, connection boxes, and switch boxes.

There are many interconnect design choices that directly impact the power, performance, and area of the resulting accelerator, including the number and bitwidth of tracks in the interconnect, how the processing elements and memories are arranged in the array, how those elements are connected, and how the interconnect is configured. An agile approach for specifying and generating the interconnect is needed for efficient design space exploration.

In this paper, we present Canal, a Python-embedded domain-specific language (eDSL) and compiler for specifying and generating reconfigurable interconnects for CGRAs using a graph-based intermediate representation. The major contributions of our paper are:

1) We describe a graph-based intermediate representation (IR) for CGRA interconnects that is capable of representing and generating a variety of topologies.
2) We propose an embedded domain-specific language (eDSL) called Canal that can compile an interconnect architecture specification into the graph-based IR.
3) We propose an interconnect generator system that can take the IR and automatically produce hardware, place and route collateral, and a bitstream generator.
4) We explore various design space choices using Canal and demonstrate its effectiveness in generating an efficient CGRA design.

II. Related Work

Previous attempts have been made to create an interconnect generator for CGRAs, but new demands in CGRA design necessitate a more flexible and powerful system. VPR is one of the most established FPGA architecture research tools [1]. It allows users to adjust various design aspects of the FPGA and observe the effects on final application performance, such as timing and area usage. However, VPR does not offer an RTL generator and users have to design their FPGA independently and hand-write the VPR architecture file accordingly.

CGRA-ME [3] is a CGRA architecture research tool similar to Canal in that it also offers integrated RTL generation and place and route

Manuscript received 21 December 2022; revised 14 February 2023; accepted 4 April 2023. Date of publication 19 April 2023; date of current version 19 May 2023. This work was supported in part by the DSSoC DARPA grant, the Stanford AHA Agile Hardware Center and Affiliates Program, Intel's Science and Technology Center (ISTC), the Stanford SystemX Alliance, SRC JUMP 2.0 PRISM Center, and NSF CAREER under Grant 2238006. *(Jackson Melchert and Keyi Zhang contributed equally to this work.) (Corresponding author: Jackson Melchert.)*

The authors are with the Stanford University, Stanford, CA 94305 USA (e-mail: melchert@stanford.edu; keyi@stanford.edu; yuchenm@stanford.edu; horowitz@ee.stanford.edu; ctorng@stanford.edu; praina@stanford.edu).

Digital Object Identifier 10.1109/LCA.2023.3268126

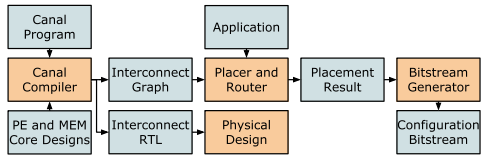

Fig. 2. The canal interconnect generator system. It takes an interconnect specification written as a program in the Canal eDSL and produces the interconnect RTL implementation. Canal also takes an application and places and routes it on a CGRA with the specified interconnect and generates a configuration bitstream.

Fig. 3. Left: Hardware interconnect. Right: Directed graph based intermediate representation of the configurable interconnect. Not all connections between the PE and SBs are shown for simplicity.

tools. One of the major differences is the architecture specification. CGRA-ME opts for a more rigid XML-based input whereas Canal takes in a Python eDSL program, which is more flexible and readable.

DSAGEN [14] uses a graph-based representation for the connectivity information within an accelerator design, however Canal offers much finer-grained representation of the interconnect. While DSAGEN provides a switch node that can be use within a CGRA architecture, Canal provides the ability to construct switch boxes of different topologies using individual connections between nodes.

FastCGRA [17] and OpenCGRA [10] are similar CGRA architecture exploration tools that use eDSLs to construct the hardware. However, users explicitly construct multiplexers and switches, from which RTL is generated. Canal abstracts away the notion of hardware primitives and lets the compiler backend choose how to generate the hardware. In summary, Canal supports more tools, allows for higher flexibility, and enables easier design space exploration than previous attempts at an interconnect generator.

III. System Design

In this section, we introduce the design of the Canal system, including the graph-based IR for representing interconnects, the Canal eDSL, the static interconnect generation used to translate the IR into hardware, and finally how that IR interfaces with the application place and route (PnR) algorithms. Fig. 2 summarizes how the Canal interconnect generator interfaces with the PE and memory core designs, application PnR, RTL generation, and bitstream generation.

A. Graph-Based Intermediate Representation

The primitives in Canal's intermediate representation (IR) are nodes, which represent anything that can be connected in the underlying hardware, and edges, which are wires connecting the nodes together. An example of the IR for a switch box is shown in Fig. 3.

All edges are unidirectional so the IR represents a directed graph. Nodes in the graph can have multiple incoming edges which, when translated into hardware, transform into multiplexers. Each node also has attributes that provide additional information for type checking and hardware generation.

This intermediate representation is flexible enough to represent a wide variety of interconnect topologies and handle an arbitrarily complex set of CGRA cores. We will discuss a few design space exploration experiments that exploit this flexibility in Section IV.

B. The Canal Language

The Canal language is a Python-embedded domain-specific language (eDSL) that constructs the interconnect intermediate representation described in the previous section. The Canal language translates the Python description of an interconnect into this IR, so at the lowest level, a designer could instantiate nodes in the Canal language and wire them together.

As Canal is embedded in Python, we have also built a layer on top of the basic primitives of Canal, which simplifies the IR construction. For instance, for creating a uniform interconnect (all switch boxes have the same topology) with no diagonal connections, we provide a simple helper function that produces different interconnect topologies by varying function parameters such as height and width of the array, switch box topology, number of tracks, bit width of tracks, and density of pipeline registers.

The Canal eDSL allows designers to easily conduct design space exploration by varying the parameters in the helper functions, or by generating an entirely new interconnect. Canal can easily be integrated with other DSLs. For example, one could use a DSL for specifying individual CGRA tiles and then integrate them together using Canal to generate the interconnect.

C. Generating Interconnect Hardware

Because Canal's IR only describes the connectivity among the different nodes, it is up to the hardware compiler backend to decide how to lower the IR. We implemented two different hardware compiler backends that lower the IR into (1) a static mesh interconnect and (2) a statically configured network-on-chip (NoC). This NoC has data channels along with a ready-valid interface and routing is configured statically before the application runs. We use magma [11] as our hardware implementation language, but this could be extended to any hardware generator framework.

To generate a static mesh interconnect, we adopt the following principles to generate hardware:

1) Nodes with hardware attributes (e.g., a processing element core) instantiate the specified hardware.
2) Directed edges are translated into wires.
3) Nodes with multiple incoming edges generate multiplexers.

We also use attributes associated with each node to lower the node to different hardware components. For instance, a register node will be lowered into a physical register. A port node will be lowered to a CB (with an internal multiplexer) where the output of the CB connects to the port of the core.

Reusing the same IR to generate a statically configured NoC has several challenges. First, the application graph may have fanouts, that is, one output port of a node is connected to multiple input ports. While this is simple to handle in a static interconnect, we now need an area-efficient way to handle control signals. Since valid signals flow in the same direction as the data, generating hardware for valid channels follows the same strategy as that used for the data channels. However, since ready signals flow in a direction opposite to that of the data channels, we need a way to merge ready signals at the fan-in point. We *and* the appropriate ready signals at every fan-in point, reusing the routing configuration to minimize the overhead.

Another challenge is that a ready-valid NoC needs FIFOs present in the interconnect to buffer data when a downstream tile is not yet ready. While we can easily generate a fixed-size FIFO for a register IR node, the area cost of those FIFOs can be quite high, as shown in Fig. 4. To reduce the area overhead introduced by FIFOs while maintaining backward compatibility with a static interconnect, we realize that we can combine two registers from adjacent tiles into a single size-two FIFO. We call this a split FIFO. The first register's FIFO control signals are passed from the first tile into the second tile and its register.

After the graph is translated into an RTL description, Canal verifies structural correctness by comparing the connectivity of the hardware with that of the IR by parsing the generated RTL. In addition, Canal

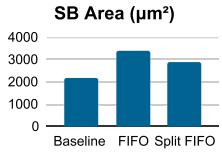

Fig. 4. Area comparison of a baseline fully static switch box, a switch box that includes FIFOs for ready/valid applications, and an optimized switch box with a split FIFO.

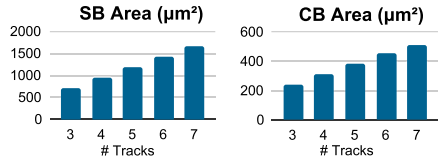

Fig. 5. Left: Area of a switch box as the number of tracks increases. Right: Area of a connection box as the number of tracks increases.

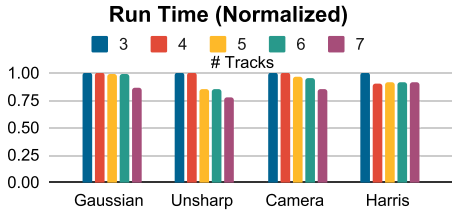

Fig. 6. Application run time comparison on CGRAs with switch boxes that have different number of tracks.

also has a built in configuration sweep test suite that exhaustively tests every possible connection in IR on the CGRA. This ensures correctness of the design.

The methodology described here also applies to generating dynamic NoCs. Instead of lowering a node into a configurable multiplexer to select among incoming data tracks, we can generate a router whose routing table is computed based on the same connectivity information.

D. Place and Route Using Canal Interconnect

This section describes how the Canal system integrates with a PnR backend (see Fig. 2) to enable running applications on a given interconnect. During the translation from a Canal program into the directed graph representation, information regarding important hardware characteristics, like core or wire delays, can be embedded into the graph. The Canal system then executes PnR in three stages: packing, placement, and routing. The remainder of this section describes the PnR backend we use in our results.

During the packing stage, constants and registers in the application are placed directly into tiles when available. After packing, the placement tool places the tiles in the application onto the interconnect in two stages: global and detailed placement. Global placement uses an analytical algorithm that leverages the standard conjugate gradient method on the summation of the cost of each net [5]. The cost of a net is the combination of its half-perimeter wire length (HPWL) and a legalization term for memory tiles.

After global placement, we perform detailed placement based on simulated annealing [12]. The cost function for simulated annealing is the total wirelength of the application, calculated by summing the HPWL cost for each net, and an additional term for penalizing pass-through tiles (those that are only used for routing).

After placement, we route using an iteration-based routing algorithm [9]. During each iteration, we compute the slack on a net and determine how critical it is given global timing information. Then we route using the A* algorithm on the weighted graph. The weights for each edge are based on historical usage, net slack, and current congestion. This allows us to balance both routing congestion and timing criticality.

IV. Evaluation

We evaluate the Canal system by first exploring the optimizations of interconnect FIFOs described in Section III-C and then by using the Canal system to conduct design space exploration of a CGRA interconnect.

A. Interconnect FIFO Optimizations

We evaluate the effect of introducing FIFOs on switch box area. As described in Section III-C, we need to include FIFOs in the configurable routing when running applications with ready-valid signaling.

As a baseline, we compare against a fully static interconnect with five 16-bit routing tracks containing PEs with two outputs and four inputs, synthesized in GlobalFoundries 12 nm technology. As shown in Fig. 4, adding these depth two FIFOs to the baseline design introduces a 54% area overhead. Splitting the FIFO between multiple switch boxes results in 32% area overhead over the baseline. This optimization allows for a much more efficient implementation of an interconnect that supports ready-valid signaling.

B. Interconnect Design Space Exploration

We use Canal to explore three important design space axes of a configurable interconnect: switch box topology, number of routing tracks, and number of switch box and connection box port connections. We find that Canal's automation greatly simplifies the procedure to explore each option in the following subsections.

1) Exploring Switch Box Topologies and Routing Tracks: The switch box topology defines how the tracks on each side of the switch box connect to the tracks on the remaining sides of the switch box. The topology affects how easily nets can be routed on the interconnect. High routability corresponds to short routes and short critical paths in applications, allowing the CGRA to be run at higher frequencies which decreases application run time. For these experiments we investigate two different switch box topologies: Wilton [16] and Disjoint [15]. These switch box topologies have the same area, they both connect each input to the other sides once.

We found that the Wilton topology performs much better than the Disjoint topology, which failed to route in all of our test cases. The Disjoint topology is worse for routability because every incoming connection on track i has a connection only to track i on the three other sides of the SB. This imposes a restriction that if you want to route a wire from any point on the array to any other point on the array starting from a certain track number, you must only use that track number. In comparison, the Wilton topology does not have this restriction resulting in many more choices for the routing algorithm, and therefore much higher routability [6].

We also vary the number of routing tracks in the interconnect. This directly affects the size of both the connection box and the switch box and the amount of routing congestion. For these experiments we measure the area of the connection box and switch box as well as the run time of applications running on the CGRA.

In this experiment we use an interconnect with five 16-bit tracks and PE tiles that have 4 inputs and 2 outputs. As shown in Fig. 5, the area of both the switch box and connection box scale with the number of tracks. From Fig. 6, we can see that the run time of the applications generally decreases as the number of tracks increases, although the benefits are less than 25%.

Canal allows designers to easily perform this type of switch box topology design space exploration and determine the trade-offs between the number of tracks in the interconnect and the resulting run time of the applications.

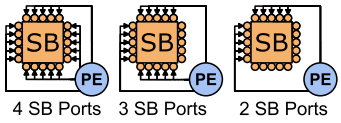

Fig. 7. Reducing the number of connections from the outputs of the PE to the outgoing ports of the switch box.

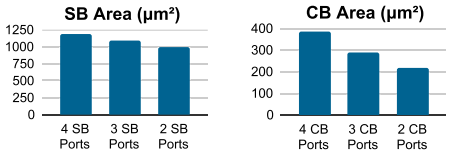

Fig. 8. Area comparison of a switch box and a connection box that have varying number of connections with the four sides of the tile.

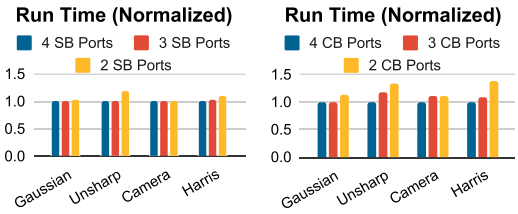

Fig. 9. Run time comparison of a switch box and connection box that have varying connections from the four sides of the tile.

2) Exploring Switch Box and Connection Box Port Connections: Finally, we explore how varying the number of switch box and connection box port connections affects the area of the interconnect and the run time of applications executing on the CGRA. In Canal, we have the ability to specify how many of the incoming tracks from each side of the tile are connected to the inputs/outputs of the PE/MEM cores. Decreasing these connections should reduce the area of the interconnect, but may decrease the number of options that the routing algorithm has. For these experiments, we vary the number of connections from the incoming routing tracks through the connection box to the inputs of the PE/MEM core, and vary the number of connections from the outputs of the core to the outgoing ports of the switch box. At maximum, we can have 4 SB sides, with connections from the core output to the four sides of the switch box. We then decrease this by removing the connections facing east for a total of three sides with connections, and finally we also remove the connections facing south for a total of two sides with connections. This is shown in Fig. 7. We do the same for the connection box.

As shown in Fig. 8, as the number of connections from the core to the switch box decreases, we see a decrease in switch box area. From Fig. 9, we can see that this generally has a small negative effect on the run time of the applications. In this case, a designer could choose to trade some performance for a decrease in switch box area. We see the same trends with the connection boxes as well.

V. Conclusion

We have developed Canal, a domain-specific language and interconnect generator for CGRAs. The Canal language allows a designer to easily specify a complex configurable interconnect, while maintaining control over the low-level connections. The hardware generator, placer and router, and bitstream generator help to facilitate design space exploration of CGRA interconnects. We demonstrate the flexibility of Canal by creating a hybrid ready-valid interconnect and demonstrate the design space exploration capabilities of Canal by evaluating different switch box topologies, number of interconnect routing tracks, and number of SB and CB port connections. The power and flexibility that Canal provides will enable more designers to create and explore diverse and interesting CGRA architectures.

References

[1] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *Proc. Int. Workshop Field Programmable Log. Appl.*, Springer, 1997, pp. 213–222.
[2] A. Carsello et al., "Amber: A 367 GOPS, 538 GOPS/W 16 nm SoC with a coarse-grained reconfigurable array for flexible acceleration of dense linear algebra," in *Proc. IEEE Symp. VLSI Technol. Circuits*, 2022, pp. 70–71.
[3] S. A. Chin et al., "CGRA-ME: A unified framework for CGRA modelling and exploration," in *Proc. IEEE 28th Int. Conf. Appl.-Specific Syst. Architectures Processors*, 2017, pp. 184–189.
[4] V. Govindaraju et al., "DySER: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, vol. 32, no. 5, pp. 38–51, Sep./Oct. 2012.
[5] A. B. Kahng, S. Reda, and Q. Wang, "APlace: A general analytic placement framework," in *Proc. Int. Symp. Phys. Des.*, 2005, pp. 233–235.
[6] M. I. Masud, "FPGA routing structures: A novel switch block and depopulated interconnect matrix architectures," Masters Thesis, Dept. Elect. Comput. Eng., Dalhousie Univ., Halifax, Canada, 1998.
[7] B. Mei, M. Berekovic, and J.-Y. Mignolet, "ADRES & DRESC: Architecture and compiler for coarse-grain reconfigurable processors," in *Fine- and Coarse-Grain Reconfigurable Computing*. Berlin, Germany: Springer, 2007.
[8] R. Prabhakar et al., "Plasticine: A reconfigurable architecture for parallel patterns," in *Proc. IEEE/ACM 44th Annu. Int. Symp. Comput. Architecture*, 2017, pp. 389–402.
[9] J. S. Swartz, V. Betz, and J. Rose, "A fast routability-driven router for FPGAs," in *Proc. ACM/SIGDA 6th Int. Symp. Field Programmable Gate Arrays*, 1998, pp. 140–149.
[10] 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 *Proc. IEEE 38th Int. Conf. Comput. Des.*, 2020, pp. 381–388.
[11] L. Truong and P. Hanrahan, "A golden age of hardware description languages: Applying programming language techniques to improve design productivity," in *Proc. 3rd Summit Adv. Program. Lang.*, 2019, pp. 7:1–7:21.
[12] P. J. Van Laarhoven and E. H. Aarts, "Simulated annealing," in *Simulated Annealing: Theory and Applications*. Berlin, Germany: Springer, 1987.
[13] A. Vasilyev, N. Bhagdikar, A. Pedram, S. Richardson, S. Kvatinsky, and M. Horowitz, "Evaluating programmable architectures for imaging and vision applications," in *Proc. IEEE/ACM Int. Symp. Microarchitecture*, 2016, pp. 1–13.
[14] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki, "DSAGEN: Synthesizing programmable spatial accelerators," in *Proc. IEEE/ACM 47th Annu. Int. Symp. Comput. Architecture*, 2020, pp. 268–281.
[15] N. H. E. Weste and K. Eshraghian, *Principles of CMOS VLSI Design: A Systems Perspective*. Reading, MA, USA: Addison-Wesley, 1993.
[16] S. J. E. Wilton, "Architecture and algorithms for field programmable gate arrays with embedded memory," PhD Thesis, Dept. Elect. Comput. Eng., Univ. Toronto, Toronto, Canada, 1977.
[17] S. Zheng, K. Zhang, Y. Tian, W. Yin, L. Wang, and X. Zhou, "FastCGRA: A modeling, evaluation, and exploration platform for large-scale coarse-grained reconfigurable arrays," in *Proc. Int. Conf. Field-Programmable Technol.*, 2021, pp. 1–10.