Architecture Exploration of Standard-Cell and FPGA-Overlay CGRAs Using the Open-Source CGRA-ME Framework

S. Alexander Chin
Dept. of ECE, University of Toronto
Toronto, Ontario, Canada
xan@ece.utoronto.ca

Kuang Ping Niu
Dept. of ECE, University of Toronto
Toronto, Ontario, Canada
kuangping.niu@mail.utoronto.ca

Matthew Walker
Dept. of ECE, University of Toronto
Toronto, Ontario, Canada
matthewjp.walker@mail.utoronto.ca

Shizhang Yin
Dept. of ECE, University of Toronto
Toronto, Ontario, Canada

Alexander Mertens
Dept. of ECE, University of Toronto
Toronto, Ontario, Canada

Jongeun Lee
School of ECE, UNIST
Ulsan, Korea
jlee@unist.ac.kr

Jason H. Anderson
Dept. of ECE, University of Toronto
Toronto, Ontario, Canada
janders@eecg.toronto.edu

ABSTRACT

We describe an open-source software framework, *CGRA-ME*, for the modeling and exploration of coarse-grained reconfigurable architectures (CGRAs). CGRAs are programmable hardware devices having large ALU-like logic blocks, and datapath bus-style interconnect. CGRAs are positioned between fine-grained FPGAs and standard-cell ASICs on the spectrum of programmability – they are less flexible than FPGAs, yet are more flexible than ASICs. With CGRA-ME, an architect can describe a CGRA architecture in an XML-based language. The framework also allows the architect to map benchmarks onto the architecture and provides automatic generation of Verilog RTL for the modeled architecture. This allows the architect to simulate for verification purposes, and perform synthesis to either an ASIC or FPGA-overlay implementation of the CGRA, assessing performance, area, and power consumption. In an experimental study, we use CGRA-ME to model, map benchmarks onto, and evaluate several variants of a widely known CGRA [24], considering both standard-cell and FPGA-overlay physical realizations of the CGRA.

CCS CONCEPTS

• **Hardware → Reconfigurable logic and FPGAs**;

KEYWORDS

Reconfigurable architectures, coarse-grained reconfigurable architectures, CGRAs, FPGA overlays

ISPD '18, March 25–28, 2018, Monterey, CA

ACM ISBN 978-1-4503-5626-8/18/03…$15.00
https://doi.org/10.1145/3177540.3177553

ACM Reference Format:
S. Alexander Chin, Kuang Ping Niu, Matthew Walker, Shizhang Yin, Alexander Mertens, Jongeun Lee, and Jason H. Anderson. 2018. Architecture Exploration of Standard-Cell and FPGA-Overlay CGRAs Using the Open-Source CGRA-ME Framework. In *ISPD '18: 2018 International Symposium on Physical Design, March 25–28, 2018, Monterey, CA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3177540.3177553

1 INTRODUCTION

Coarse-grained reconfigurable arrays (CGRAs) are programmable hardware architectures with coarse-grained logic blocks, often resembling ALUs, and datapath-style interconnect, where buses of signals are routed together as a group. These attributes stand in contrast to fine-grained field-programmable gate arrays (FPGAs), which contain a mix of fine and coarse-grained blocks for implementing logic (e.g. look-up-tables and hardened DSP blocks), and where individual logic signals are routed in a singular manner. CGRAs are thus less flexible than FPGAs, affecting both the ease of application mapping, as well as the power/performance/cost, elaborated upon below. In this paper, we describe an open-source framework under active development at the University of Toronto called *CGRA-ME* [9] that permits the modeling and exploration of a wide variety of CGRA architectures, and also facilitates research on CGRA mapping algorithms. Here, we also detail two physical implementations of CGRAs and show the capabilities of the framework's mapper, demonstrating the effectiveness of CGRA-ME.

A CGRA can be physically realized in a number of ways: 1) as a custom chip, with manual layout, 2) as a standard-cell ASIC, or 3) on an FPGA as an *overlay*. We believe the first option to be prohibitively expensive, and therefore, we focus on the second and third options in this paper, which we refer to as a *standard-cell CGRA* and an *FPGA-overlay CGRA*, respectively. A standard-cell CGRA will have comparatively less silicon area dedicated to programmability than an FPGA, therefore offering better power/performance than an FPGA for applications whose computational

and communication needs are aligned with the CGRA architecture's capabilities. Standard-cell CGRAs have been proposed by both academia (e.g. [15, 24]), and more recently, in industry [17, 29].

On the other hand, because an FPGA-overlay CGRA is a programmable platform implemented *on top* of a programmable platform (the underlying FPGA), the power/performance benefits versus the direct use of the underlying FPGA are less clear. If the FPGA-overlay CGRA is carefully implemented to make best-possible use of the underlying FPGA resources, then power/performance parity may be achievable. Despite this apparent weakness, we believe an important future usage scenario for FPGA-overlay CGRAs will be in the cloud/datacenter context, where FPGAs are deployed alongside standard processors, to be used as hardware application accelerators. Particularly, we envision the existence of libraries of pre-compiled FPGA-overlay CGRA architectures, where a given architecture is tailored towards a specific application domain (e.g. machine learning or finance), with an easy-to-use software API. A user would select the pre-compiled FPGA-overlay CGRA that best meets their application needs, and aim to achieve application acceleration over a standard CPU/GPU implementation.

Regardless of the physical manifestation, a key benefit of CGRAs versus FPGAs is in application mapping runtime. Owing to their reduced flexibility, CAD tools for CGRAs need to make fewer decisions, reducing CAD complexity significantly. This brings runtimes closer to software compilation, as opposed to the hours or days that may be required for application mapping to FPGAs. Moreover, applications targeted to CGRAs are typically specified at a higher abstraction level than traditional hardware design. Applications are specified as data-flow graphs or in software, as opposed to Verilog/VHDL RTL.

While open-source modeling and evaluation frameworks have long existed for traditional processors (e.g. [4]), GPUs [2], and fine-grain FPGAs [3], there is no analogous framework for CGRAs. With CGRA-ME, a wide variety of CGRA architectures can be modeled, by a human architect using an XML-based language. The CGRA-ME framework accepts the architecture description as input, as well as an application benchmark to be mapped into the CGRA. An in-memory device model of the CGRA is constructed and the application is mapped into the CGRA. Verilog RTL for the CGRA is produced automatically, along with a bitstream configuring the device for the given benchmark. The generated Verilog can be synthesized to a standard-cell ASIC or an FPGA-overlay CGRA implementation to assess power/performance of the architecture. The RTL and configuration bitstream can also be simulated for functional correctness. CGRA-ME is a thus powerful platform for CGRA architecture and CAD research that we believe will be useful to both the academic and industrial communities.

2 FRAMEWORK OVERVIEW

Figure 1 depicts the CGRA-ME framework, its internal components, and data-flow. Circled labels are shown on each component. The inputs to the framework are a C-language application benchmark ①, as well as a textual description of the CGRA architecture ②. The XML-based CGRA-ME architecture description language, *CGRA-ADL* is elaborated upon in Section 2.1. The CGRA-ADL is parsed by an architecture interpreter ③, producing an in-memory device model of the CGRA ④. The benchmark is parsed and then optimized by the LLVM compiler [18]. One or more data-flow graphs are extracted from the input benchmark and input to the CGRA Mapper ⑥. The specific portions of the program for which data-flow graphs are extracted are based on user annotations to the source code. The CGRA mapper ⑥ receives the application data-flow graph as input, as well as uses the in-memory CGRA device model to map the application onto the CGRA. The mapping algorithm is described in more detail in Section 2.2.

The CGRA architecture interpreter ③ is also capable of generating Verilog RTL for the specified CGRA ⑦. The generated Verilog has several utilities. First, the Verilog, along with a configuration bitstream produced by the mapper, can be simulated to verify CGRA functionality ⑧. Second, the Verilog can be input into a standard-cell ASIC design flow ⑨, to realize a physical implementation, which can then inform performance/power/area models Ⓑ. Or third, the RTL can be input into an FPGA design flow Ⓒ to realize an FPGA-overlay CGRA implementation, and associated performance/power/area models Ⓒ. In Section 4, we demonstrate standard-cell and FPGA-overlay implementations of variants of the ADRES [24] CGRA architecture. Finally, physical performance/power/area models can be combined with the mapping results to analyze the specific input benchmark implemented on the modeled CGRA Ⓓ.

2.1 CGRA-ME Architecture Description Language

CGRA-ADL allows the textual specification of a CGRA architecture in a concise XML-based language. The complete language definition is available on the CGRA-ME website [8]; here, we overview a few aspects of the language at a high level.

Figure 2 illustrates the pieces of an architecture description at an abstract level. Line 1 opens the description. At line 2, an architect may optionally use the `definition` tag to define one or more constants. The `module` tag at line 3 opens the description of a CGRA module, which may have inputs (line 4) and outputs (not shown). A module may contain sub-modules (line 5), which are interconnected with one another (line 6), as elaborated upon below. Within the `cgra` tags, a module may be described in a hierarchical manner, composed of sub-modules, which may themselves contain sub-modules.

The definition of the CGRA is opened with the `architecture` tag (line 9, Figure 2). Here, the example shows a 4×4 CGRA architecture. Lines 10-13 define a pattern of modules that populate the 4×4 array. The language is flexible in that it allows both homogeneous and heterogeneous CGRA architectures to be described, with a variety of interconnect styles for interconnecting the blocks, as described below.

Figure 3 shows an example CGRA module, `moduleA`, which contains within a sub-module as well as programmable interconnect that permits selection from three top-level inputs to two internal sub-module inputs. The CGRA-ADL specification is given in Figure 4. Line 1 opens the description. Lines 2-5 define the module I/Os. Line 6 instantiates a sub-module, of type `FuncUnit` having an instance name `fu_inst`. Lines 7-10 define the multiplexed interconnect to the sub-module. Implicit in this description is the presence of

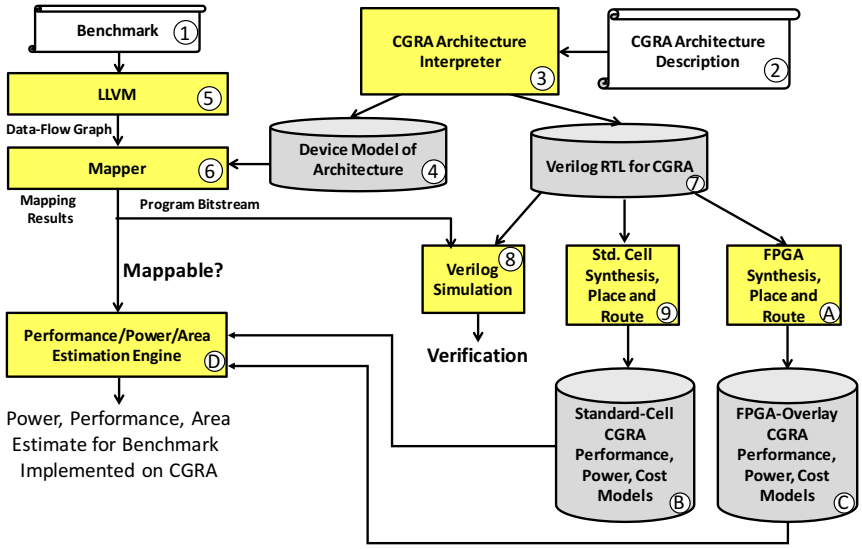

Figure 1: CGRA-ME framework overview showing the main components.

```
1  <cgra>
2     <definition .../>
3     <module name="blockA">
4        <input name="in"/>
5        <inst  module="fu"/>
6        <connection ... /> ...
7     </module>
8
9     <architecture row="4" col="4">
10       <pattern>
11          <block module="blockA"/>
12       </pattern> ...
13    </architecture>
14 </cgra>
```

Figure 2: High-level structure of a CGRA architecture description.

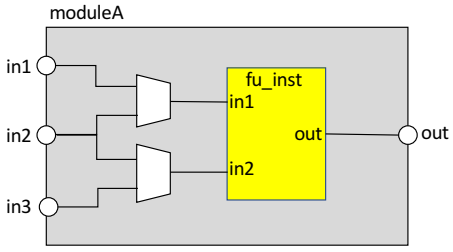

Figure 3: Example module within a CGRA.

```
1  <module name="moduleA">
2     <input  name="in1"/>
3     <input  name="in2"/>
4     <input  name="in3"/>
5     <output name="out"/>
6     <inst   name="fu_inst" module="FuncUnit"/>
7     <connection select-from="this.in1 this.in2"
8        to="fu_inst.in1"/>
9     <connection select-from="this.in2 this.in3"
10       to="fu_inst.in2"/>
11    <connection from="fu_inst.out"
12       to="this.out"/>
13 </module>
```

Figure 4: CGRA-ADL module definition for `moduleA` depicted in Figure 3.

```
1  <architecture rows="8" cols="8">
2   <pattern row-range="0 7" col-range="0 7"
3     row="2" col="2">
4      <block module="A"/> <block module="B"/>
5         <!-- row break -->
6      <block module="C"/> <block module="D"/>
7   </pattern>
8  </architecture>
```

Figure 5: CGRA-ADL for a heterogeneous CGRA with four block types.

SRAM configuration cells that drive the multiplexer-select signals. Lines 11-12 defines the interconnect from the sub-module output to the higher-level output. Line 13 closes the description.

With the `architecture` tag in Figure 2, a generic CGRA architectures can be composed, with varied functional unit architectures (homogeneous or heterogeneous) and a variety of different inter-block connectivity. Consider the example shown in Figure 5. Line 1 specifies an architecture with 8 rows and 8 columns. The `pattern` tag on line 2 opens a 2×2 pattern definition that applies repeatedly to the entire 8×8 CGRA. Lines 4-6 specify that the 2×2 repeating pattern comprises 4 modules, *A*, *B*, *C*, and *D*, where *A* and *B* are on the first row; *C* and *D* are on the second row. Hence, the architecture is heterogeneous with 4 types of blocks, in an iterated 2×2 pattern. Although not shown in the example, the CGRA-ADL also contains syntactic sugar for common styles of inter-block CGRA interconnect, namely, North-South-East-West (NSEW) block connectivity, as well as NSEW+diagonal connectivity.

2.2 Mapping

2.2.1 Background. The process of implementing a benchmark onto a CGRA is called *mapping*. This process involves, at the highest level, a translation of a benchmark from a high-level software language such as C to a *data-flow graph* (DFG). A data-flow graph is an directed graph that contains a number of operations such as *add* or *mul* as nodes. Edges within the graph represent the flow of data from outputs to inputs of operations. This data-flow graph is then implemented on the CGRA through association of operations to functional units and ensuring routes for data between them, while accounting for registers and obeying a correct operation schedule.

To aid this association of operations with the physical CGRA, a device model of the CGRA is created. The Modulo Routing Resource Graph (MRRG) [23] is a representation that is commonly used to model CGRAs. This representation is used as the device model shown as ④ in Figure 1. This directed graph contains representations of functional units and routing resources as vertices with edges showing valid paths between resources. The MRRG is created by querying each module instance within the CGRA hierarchy to produce a graph sub-graphs of the MRRG, building the MRRG in a piece-meal fashion. The mapping problem is the finding an association of operation nodes in the DFG to functional units in the MRRG while ensuring values produced by operations are associated with a path over a number of routing resource nodes to the destination operation.

2.2.2 ILP Mapper. While other mappers have employed annealer-based [14, 23] or graph-based [7, 22] mapping techniques, in this work we use a constraint-based technique by formulating the graph association problem as an integer linear programming (ILP) problem. While our ILP mapper is not the first formulation to be applied to CGRAs [19, 25, 30, 31], the main difference with CGRA-ME's ILP formulation [8] is that the formulation is valid over any architecture that an MRRG can be generated, while other works are constrained to specific architectures or architectural templates.

Here we provide a brief discussion of the formulation. We refer to four sets of items: $FuncUnits$: every execution slot of every function unit within the architecture or one function node within the MRRG; $RouteRes$: every routing resource (wire or register) within the architecture or one routing node within the MRRG; Ops: every operation defined in the DFG that is to be mapped; $Vals$: every value output from an Operation in the DFG.

And we define two sets of binary variables. The first set of variables define the placement of operations onto functional units – a mapping from Ops to $FuncUnits$.

- $F_{p,q}$: functional-unit node p in the MRRG is used for implementing operation q in the DFG.

The second set of variables define the use of routing resources by values (a mapping from $Vals$ to $RouteRes$).

- $R_{i,j}$: routing node i is used for routing value j.

A number of constraints are applied, some of which are outlined next.

Operation Placement: This ensures every operation in the DFG is placed on exactly one functional unit within the MRRG.

$$\sum_{p \in FuncUnits} F_{p,q} = 1, \forall q \in Ops \tag{1}$$

Functional Unit Exclusivity: This ensures each functional unit slot (represented by $FuncUnits$) is occupied by at most one DFG Operation (i.e. there is no multiple usage of a single functional unit slot).

$$\sum_{q \in Ops} F_{p,q} \leq 1, \forall p \in FuncUnits \tag{2}$$

Functional Unit Legality: This ensures that operations are only placed on functional units that can implement the operation (applies to reduced architectures).

$$\begin{aligned} &F_{p,q} = 0 \\ &\forall p \in FuncUnits, q \in Ops \\ &\text{where: } q \notin SupportedOps(p) \end{aligned} \tag{3}$$

where $SupportedOps(p)$ is the set of operations that are supported by functional unit p.

Route Exclusivity: This ensures that each routing resource is occupied by *at most* one value (i.e. multiple DFG Values cannot be routed onto the same routing resource).

$$\sum_{j \in Vals} R_{i,j} \leq 1, \forall i \in RouteRes \tag{4}$$

Another set of constraints ensures that values between operations are routed from the source operations to the destination operations but a full discussion is omitted due to space limitations.

According to the aforementioned constraints, the following objective function is minimized to find the minimum number of routing resources used by the mapping:

$$Minimize \sum_{\forall i \in RouteRes, \forall j \in Vals} R_{i,j} \tag{5}$$

2.3 Verilog & Bitstream Generation

The in-memory objects representing modules are organized in a hierarchical manner. Each of these objects represent a logical hardware unit, and encapsulates its connections, ports, sub-modules, and configuration sub-modules. Each MRRG node also links to its module instance, enabling easy reverse mapping from node to module. A Verilog implementation of the architecture is automatically generated from the module tree: for each unique module type, a parameterized Verilog file is emitted with ports, sub-module declarations, and a generic implementation including connecting wires. Sub-modules containing configuration registers are also included, hooked up to the module they configure, and automatically linked together to form an inter-module scan chain.

A successful mapping associates all *Ops* and *Vals* with one or more MRRG nodes. The MRRG is iterated over and a map from module instance to a list of associated *Op*s, *Val*s and MRRG nodes is created. The configuration sub-modules are then walked in scan chain order, and the connected module instance is presented with its corresponding data stored in the map. The module then generates the configuration bits required by pattern matching and inspection

of the *Op*s, *Val*s, and MRRG nodes. A Verilog file is produced, containing a module that will use the generated bitstream to program an instance of a top-level CGRA module, which was generated separately. The configuration generation and programming has been tested with ModelSim [10], and expected behaviour was verified at outputs.

3 PHYSICAL IMPLEMENTATION METHODOLOGY

3.1 Standard-Cell

In this work, we leverage the FreePDK45 [26] standard-cell library, an open-source 45 nm standard-cell library, for the experimental study using standard-cells. The tool-generated Verilog of the targeted architectures is used for physical implementation and is put through the following flow: Synopsys' Design Compiler is selected to perform technology mapping. Following technology mapping, we analyze area breakdown of targeted architectures. Architectures are mapped for minimum area and minimum delay. With the mapped design netlist as input, all architectures are pushed through standard-cell place and route, with Cadence's Innovus, which then produce accurate standard-cell placement, parasitic information (SPEF file), as well as total chip area. Given the design netlist and SPEF file, we perform static timing analysis with Synopsys' PrimeTime and determine designs' critical path. In Section 4.3.1, we demonstrate how user-specified architectures can be assessed for their design feasibility, at various stages of design flow: standard-cell technology mapping, place and route, and static timing analysis.

3.2 FPGA-Overlay

We also target the 45 nm Altera Stratix IV FPGA (same process node as the standard-cell implementation) using Altera/Intel's Quartus Prime ver. 16.1. A challenge that arose in the FPGA implementation pertains to floorplanning in the presence of CGRA blocks containing multipliers. It is desirable if the rows/columns of the CGRA's physical realization form a regular grid coherent with the CGRA's logical representation (e.g. CGRA blocks in column 2 are placed to the right of blocks in column 1, and so on). In the targeted Stratix IV FPGA, the DSP blocks containing hardened multipliers are arranged in columns that are relatively far apart. Interspersed among the DSP columns are comparatively numerous columns of standard LUT-based logic blocks, as well as occasional RAM-block columns. The floorplans of the CGRAs considered in Section 4.3.2 are thus defined primarily in relation to the DSP columns, with the LUT-based blocks between the DSPs being underutilized.

4 EXPERIMENTAL STUDY

4.1 Experimental Architectures

In this study we model two architecture variants resembling ADRES [24]. Figure 6 shows a high-level view of both architectures. The two targeted variants of ADRES are the "Full" and "Reduced" architecture, both are 4×4 grid of functional units; the Full architecture consists of functional units with a full set of operations: add, subtract, multiply, shifts, and, or, and xor, while also having torus connectivity between top and bottom rows, and between leftmost and rightmost columns; the Reduced architecture contains processing elements with only add and subtract capabilities in odd columns, and there are no torus connections.

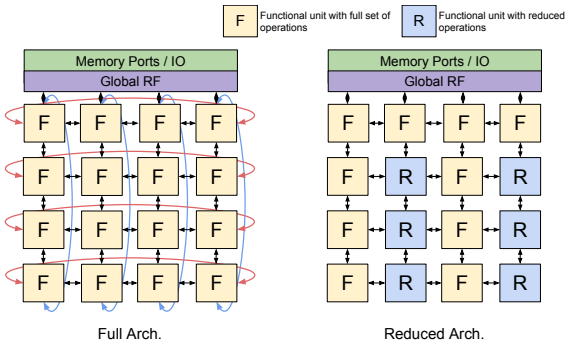

Figure 6: "Full" and "Reduced" variants of ADRES architecture

4.2 Benchmarks & Mapping Results

The ILP mapper within CGRA-ME was used to map a combined set of synthetic and real benchmarks to the two architecture variants. Table 1 shows the benchmark set mapped to our two architecture variants. All benchmarks were able to be mapped into the Full architecture that contains 16 multipliers (one in each functional unit) and toroid connections. In the Reduced architecture, there are only 10 multipliers and no toroid connections. Not all benchmarks map to this architecture, especially ones with many multiplication operations. The benchmarks *cap*, *mac*2, *mults*1, and *mults*2 are unable to be mapped to the reduced architecture. Though they all have less than 10 multiply operations, no feasible mapping exists as shown by the ILP mapper – due to constrained routing.

All benchmarks finished mapping in less than ~12 mins. The longest runtime for mapping is for *cap* on the Full architecture, followed by *mults*1 and *mults*2. In these benchmarks, between 12 and 14 of the total 16 functional units are used. This high usage imposes a harder constraint problem to solve, leading to longer runtimes.

4.3 Physical Implementation

4.3.1 Standard-Cell. For the standard-cell implementation, both architecture variants are synthesized targeting minimum area, as well as minimum delay, producing four sets of results. Soft constraint floorplanning shown in Figure 8 is applied to generate more regular designs. All four implementations are given the same aspect ratio. After place and route, we are able to extract the total chip area of all four designs shown in Table 2. Figure 9 shows the chip layout of all four designs, while Figure 10 demonstrates where each sub-module is placed, which can be compared against the soft floorplan constraints in Figure 8.

Figure 7 presents the area breakdown of the four variants. Within the breakdown, the following categorization is used: multiplexers of any size are routing resources, registers storing configuration bits are configuration memory, data registers and register files are data memory, and functional units are computation resources.

	Architecture Mappability		Mapping Runtime [s]		Benchmark Statistics		
	Full Arch.	Reduced Arch.	Full Arch.	Reduced Arch.	Ops	Mults	I/O
accum	M	M	21.9	21.8	8	4	5
cap	M	U	727.6	1.8	12	9	4
conv2	M	M	3.4	3.4	7	5	3
conv3	M	M	21.4	109.9	11	7	4
mac	M	M	2.4	2.9	5	3	3
mac2	M	U	60.6	211.3	12	8	6
matmult	M	M	5.3	9.3	9	5	3
mults1	M	U	406.2	145.9	15	8	5
mults2	M	U	397.8	212.0	13	8	5
sum	M	M	0.3	0.3	3	1	2

Table 1: Mapping results for both Full and Reduced architectures. M indicates the benchmark was able to be mapped on the architecture. U indicates that the benchmark was unable to be mapped on the architecture. Mapping runtimes for each architecture and benchmark are also reported. Statistics for the benchmarks are also included listing the number of compute operations, number of multiplication operations, and the number of I/Os. Ops is the total number of operations including multiplications.

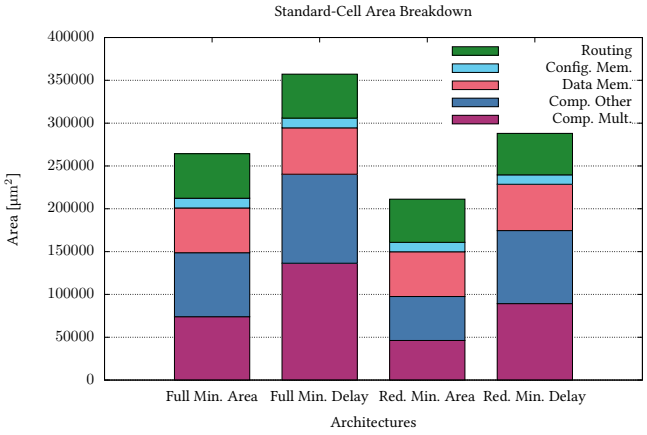

Figure 7: Standard-cell area breakdown of Full and Reduced architectures, when targeting minimum area and minimum delay.

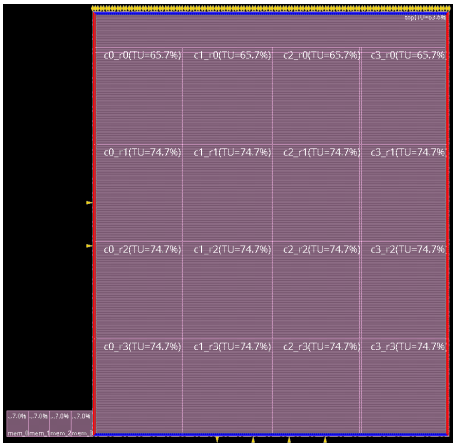

Figure 8: Floorplan for Cadence's Innovus place and route

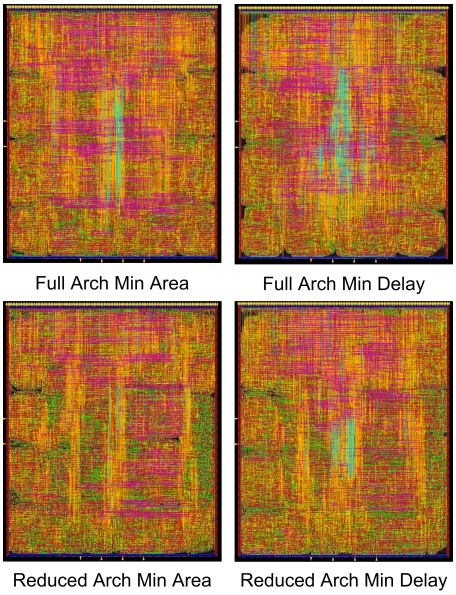

Figure 9: Standard-cell layout for all four designs in physical view

From Figure 7, we observe that routing, configuration memory, and data memory are nearly constant across the architecture and optimization variants. Also, increasing computation capabilities within the processing elements results in the largest impact to the overall area required. Additionally, between the minimum area and minimum delay architectures, the multiplier area sees the largest increase.

Static timing analysis (STA) is performed to determine the design's critical path using the SPEF file and netlist output from the place-and-route tool. Although the FMax generated using FreePDK45 may not be representative of an optimal design, comparisons of between architecture variants can still provide relative conclusions.

	Full Arch Min Area	Full Arch Min Delay	Reduced Arch Min Area	Reduced Arch Min Delay
Area [μm^2]	317927.0	403753.2	253239.8	327189.2
FMax [MHz]	140	164	142	192

Table 2: Standard-cell post-place-and-route area and performance of all four designs.

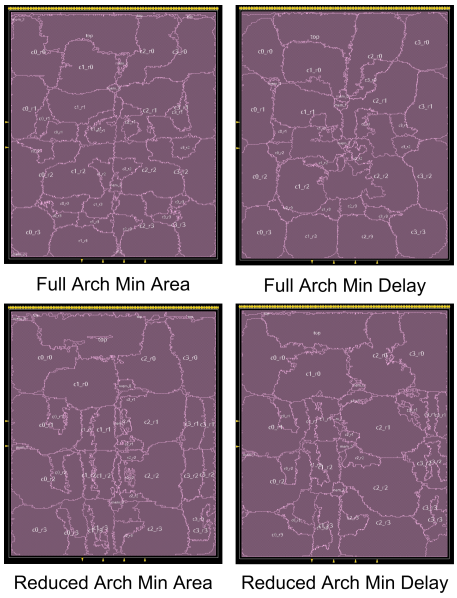

Figure 10: Standard-cell layout for all four designs in amoeba view

	Full Arch	Reduced Arch
Area [ALM]	8803	7192
FMax [MHz]	89	103

Table 3: FPGA area and performance for both architectures on Stratix IV.

From the STA results, the inter-processing-element routing multiplexers and the multipliers are the bottleneck of the designs. The Full architecture's torus routing multiplexers contribute to its lower FMax, compared to the Reduced architecture. Maximum operating frequencies of each design are shown in Table 2.

4.3.2 FPGA-Overlay. Figure 11 shows the layout of the two architectures implemented in Stratix IV. Area and performance results are reported in Table 3. Area numbers reflect Stratix IV ALMs (adaptive logic modules), each of which contains a dual-output fracturable 6-input LUT, carry-chain circuitry and two flip-flops. Logic blocks with multipliers use Stratix IV DSP blocks (not shown in the table).

Figure 12 shows an area breakdown of ALMs used on the FPGA. Here only the data memory and configuration memory stay constant and we see an increase in both the computation area as well as routing area between the Reduced architecture and Full architecture.

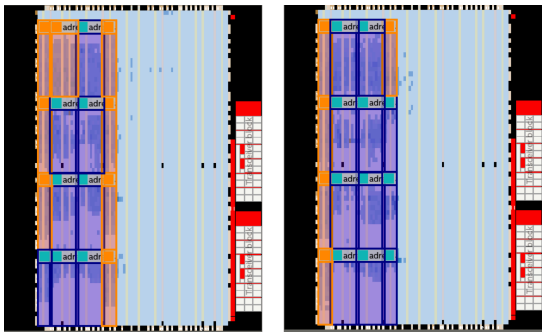

Figure 11: Floorplanned Stratix IV implementations of homogeneous CGRA with torus connections (left), and heterogeneous CGRA without torus connections (right).

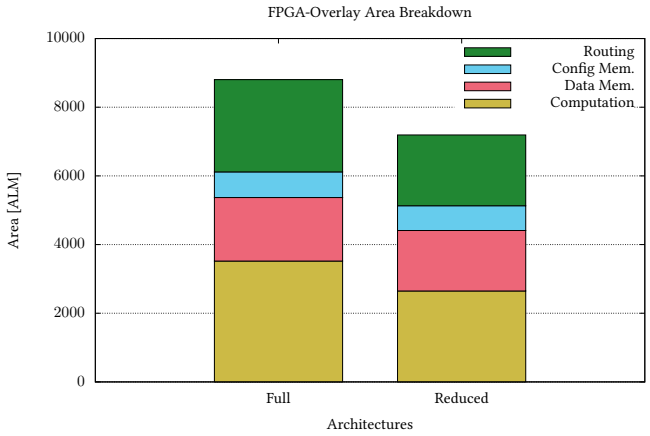

Figure 12: FPGA area breakdown for both architecture variants.

5 RELATED WORK

There are many different CGRA designs and mapping flows proposed in the past, the summary of which can be found in excellent survey articles [1, 11, 16, 27, 28]. One differentiating goal of the CGRA-ME framework is that it is able to generically model, map applications onto, and generate RTL designs for a wide range of architectures within a single framework, allowing for a 'level playing field'. Having all components including mapper and bitstream and RTL generator is essential to getting realistic performance numbers for specific benchmark applications.

Recent works have developed subsets of the CGRA-ME framework. CGADL [13] provides a modeling language to describe CGRA architectures but does not implement a way for application or benchmark mapping nor RTL output. Other works are focused on

mappers such as DRESC [23], SPR [14] and others [7, 20, 22] but are not integrated into a larger generic architecture modeling framework. For physical design, there are a number of related works but they were not produced within a larger unifying framework, which again makes it difficult to compare different architectures in a holistic way. Other works also leverage standard-cells for implementation [12, 15, 21, 24]. And for FPGA overlays, Capalija et al. [5, 6] produced prototype designs with a custom mapping flow.

6 CONCLUSIONS AND FUTURE WORK

In this work we have demonstrated the capabilities of the CGRA-ME framework, encompassing architecture modeling, application mapping, and physical implementation. The CGRA-ADL language was shown to be capable of modeling complex patterns for our test architecture and from the use of the RTL generation through CGRA-ME, two different physical implementations were generated – one in standard-cells and as an FPGA-overlay. Area breakdowns and FMax for all implementations were reported, highlighting some of the design tradeoffs that can be made. The ILP based mapper within CGRA-ME was shown to quickly generate mappings for both architecture variants, reacting to the reduced routing and functional unit resources.

The CGRA-ME framework is open-source [8] and freely available for use by the academic research community – providing a basis for future CGRA research.

7 ACKNOWLEDGMENTS

The authors thank Dr. Qiang Wang and Taneem Ahmed from Huawei for their comments and suggestions on this research.

REFERENCES

[1] Hideharu Amano. 2006. A Survey on Dynamically Reconfigurable Processors. *IEICE Transactions* 89-B, 12 (2006), 3179–3187.

[2] Ali Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE Int'l Symposium on*. IEEE, Washington, DC, USA, 163–174. https://doi.org/10.1109/ISPASS.2009.4919648

[3] Vaughn Betz and J. Rose. 1997. VPR: A new packing, placement and routing tool for FPGA research. In *Int'l Workshop on Field Programmable Logic and Applications*. Springer, Berlin, Germany, 213–222.

[4] Nathan L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. S. B. Altaf, N. Vaish, M. D. Hill, and D. A. Wood. 2011. The gem5 simulator. *SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[5] Davor Capalija and T. S. Abdelrahman. 2013. A high-performance overlay architecture for pipelined execution of data flow graphs. In *Field Programmable Logic and Applications, 2013 23rd Int'l Conference on*. IEEE, Washington, DC, USA, 1–8. https://doi.org/10.1109/FPL.2013.6645515

[6] Davor Capalija and T. S. Abdelrahman. 2014. Tile-based bottom-up compilation of custom mesh-of-functional-units FPGA overlays. In *Field Programmable Logic and Applications, 2014 24th Int'l Conference on*. IEEE, Washington, DC, USA, 1–8. https://doi.org/10.1109/FPL.2014.6927456

[7] Liang Chen and T. Mitra. 2014. Graph Minor Approach for Application Mapping on CGRAs. *ACM Trans. Reconfigurable Technol. Syst.* 7, 3, Article 21 (Sept. 2014), 25 pages. https://doi.org/10.1145/2655242

[8] S. Alexander Chin, S. Niu, S. Yin, A. Mertens, M. Walker, and J. Anderson. 2018. CGRA-ME. (2018). http://cgra-me.ece.utoronto.ca

[9] S Alexander Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. Anderson. 2017. CGRA-ME: A unified framework for CGRA modelling and exploration. In *IEEE 28th Int'l Conference on Application-specific Systems, Architectures and Processors 2017*. IEEE, Washington, DC, USA, 184–189. https://doi.org/10.1109/ASAP.2017.7995277

[10] Mentor Graphics Corporation. 2018. Modelsim. (2018). https://www.mentor.com/products/fpga/verification-simulation/modelsim/

[11] Bjorn De Sutter, P. Raghavan, and A. Lambrechts. 2013. *Coarse-Grained Reconfigurable Array Architectures*. Springer New York, New York, NY, 553–592. https://doi.org/10.1007/978-1-4614-6859-2_18

[12] Amin Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim. 2015. NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st Int'l Symposium on*. IEEE, Washington, DC, USA, 283–295. https://doi.org/10.1109/HPCA.2015.7056040

[13] Julio Oliveira Filho et al. 2009. CGADL: An Architecture Description Language for Coarse-grained Reconfigurable Arrays. *IEEE Trans. VLSI Syst.* 17, 9 (Sept. 2009), 1247–1259.

[14] Stephen Friedman, A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling, and S. Hauck. 2009. SPR: An Architecture-adaptive CGRA Mapping Tool. In *Proc. of the ACM/SIGDA Int'l Symposium on FPGAs (FPGA '09)*. ACM, New York, NY, USA, 191–200. https://doi.org/10.1145/1508128.1508158

[15] Seth Copen Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer. 1999. PipeRench: A Co/Processor for Streaming Multimedia Acceleration. *SIGARCH Comput. Archit. News* 27, 2 (May 1999), 28–39. https://doi.org/10.1145/307338.300982

[16] Reiner Hartenstein. 2001. Coarse grain reconfigurable architectures. In *Proc. of the ASP-DAC 2001*. IEEE, Washington, DC, USA, 564–569. https://doi.org/10.1109/ASPDAC.2001.913368

[17] Changmoo Kim, M. Chung, Y. Cho, M. Konijnenburg, S. Ryu, and J. Kim. 2014. ULP-SRP: Ultra Low-Power Samsung Reconfigurable Processor for Biomedical Applications. *TRETS* 7, 3 (2014), 22:1–22:15.

[18] Chris Lattner and V. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of the Int'l Symposium on Code Generation and Optimization (CGO '04)*. IEEE, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[19] Ganghee Lee, K. Choi, and N. D. Dutt. 2011. Mapping Multi-Domain Applications Onto Coarse-Grained Reconfigurable Architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 5 (May 2011), 637–650. https://doi.org/10.1109/TCAD.2010.2098571

[20] Hongsik Lee, D. Nguyen, and J. Lee. 2015. Optimizing Stream Program Performance on CGRA-based Systems. In *Proc. of the 52Nd Annual Design Automation Conference (DAC '15)*. ACM, New York, NY, USA, Article 110, 6 pages. https://doi.org/10.1145/2744769.2744884

[21] Cao Liang and X. Huang. 2008. SmartCell: A power-efficient reconfigurable architecture for data streaming applications. In *2008 IEEE Workshop on Signal Processing Systems*. IEEE, Washington, DC, USA, 257–262. https://doi.org/10.1109/SIPS.2008.4671772

[22] Lu Ma, W. Ge, and Z. Qi. 2012. *A Graph-Based Spatial Mapping Algorithm for a Coarse Grained Reconfigurable Architecture Template*. Springer Berlin Heidelberg, Berlin, Heidelberg, 669–678. https://doi.org/10.1007/978-3-642-25992-0_89

[23] Bingfeng Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. 2002. DRESC: A retargetable compiler for coarse-grained reconfigurable architectures. In *IEEE Int'l Conference on Field-Programmable Technology, 2002. Proceedings*. IEEE, Washington, DC, USA, 166–173. https://doi.org/10.1109/FPT.2002.1188678

[24] Bingfeng Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. 2003. *ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix*. Springer Berlin Heidelberg, Heidelberg, Germany, 61–70. https://doi.org/10.1007/978-3-540-45234-8_7

[25] Tony Nowatzki, M. Sartin-Tarm, L. De Carli, K. Sankaralingam, C. Estan, and B. Robatmili. 2013. A General Constraint-centric Scheduling Framework for Spatial Architectures. *SIGPLAN Not.* 48, 6 (June 2013), 495–506. https://doi.org/10.1145/2499370.2462163

[26] James E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, and R. Jenkal. 2007. FreePDK: An Open-Source Variation-Aware Design Kit. In *Proc. of the 2007 IEEE Int'l Conference on Microelectronic Systems Education (MSE '07)*. IEEE, Washington, DC, USA, 173–174. https://doi.org/10.1109/MSE.2007.44

[27] Vaishali Tehre and R. Kshirsagar. 2012. Survey on Coarse Grained Reconfigurable Architectures. *Intl. Jrnl. of Comp. Appl.* 48, 16 (2012), 1–7.

[28] Russell Tessier, K. Pocek, and A. DeHon. 2015. Reconfigurable Computing Architectures. *Proc. of the IEEE* 103, 3 (2015), 332–354.

[29] Takao Toi, N. Nakamura, T. Fujii, T. Kitaoka, K. Togawa, K. Furuta, and T. Awashima. 2013. Optimizing time and space multiplexed computation in a dynamically reconfigurable processor. In *Field-Programmable Technology (FPT), 2013 Int'l Conference on*. IEEE, Washington, DC, USA, 106–111.

[30] Jonghee W. Yoon, A. Shrivastava, S. Park, M. Ahn, R. Jeyapaul, and Y. Paek. 2008. SPKM: A Novel Graph Drawing Based Algorithm for Application Mapping Onto Coarse-grained Reconfigurable Architectures. In *Proc. of the 2008 Asia and South Pacific Design Automation Conference*. IEEE Computer Society Press, Los Alamitos, CA, USA, 776–782. http://dl.acm.org/citation.cfm?id=1356802.1356988

[31] Jonghee W. Yoon, A. Shrivastava, S. Park, M. Ahn, and Y. Paek. 2009. A Graph Drawing Based Spatial Mapping Algorithm for Coarse-Grained Reconfigurable Architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17, 11 (Nov 2009), 1565–1578. https://doi.org/10.1109/TVLSI.2008.2001746