CGRA-ME: A Unified Framework for CGRA Modelling and Exploration

S. Alexander Chin*, Noriaki Sakamoto†, Allan Rui*, Jim Zhao*, Jin Hee Kim*,
Yuko Hara-Azumi† and Jason Anderson*
*Dept. of Electrical and Computer Engineering, University of Toronto
{xan, kimjin14, janders}@ece.utoronto.ca
†Dept. of Information and Communications Engineering, Tokyo Institute of Technology

***Abstract*—Coarse-grained reconfigurable arrays (CGRAs) are a style of programmable logic device situated between FPGAs and custom ASICs on the spectrum of programmability, performance, power and cost. CGRAs have been proposed by both academia and industry; however, prior works have been mainly self-contained without broad architectural exploration and comparisons with competing CGRAs. We present CGRA-ME – a unified CGRA framework that encompasses generic architecture description, architecture modelling, application mapping, and physical implementation. Within this framework, we discuss our architecture description language CGRA-ADL, a generic LLVM-based simulated annealing mapper, and a standard cell flow for physical implementation. An architecture exploration case study is presented, highlighting the capabilities of CGRA-ME by exploring a variety of architectures with varying functionality, interconnect, array size, and execution contexts through the mapping of application benchmarks and the production of standard cell designs.**

I. Introduction

Coarse-grained reconfigurable arrays (CGRAs) are a style of programmable logic device with two differing attributes from fine-grained field-programmable gate arrays (FPGAs): 1) the logic blocks in CGRAs are larger, possessing many inputs and outputs, and generally implementing ALU-like functionality, and 2) the interconnection fabric is datapath-oriented: busses of signals are routed together in tandem. The value proposition of CGRAs is improved speed, area-efficiency and power relative to FPGAs, owing to less overhead for programmability. Such benefits arise for applications whose computational requirements align closely with the available CGRA logic and interconnect. One can think of a CGRA as being midway between an FPGA and an ASIC, providing *some* of the programmability of an FPGA, while delivering superior power, performance, and cost characteristics for aligned applications. CGRAs are especially attractive when tailored to specific domains, where computational requirements are known, yet programmability is desired to accommodate an evolving technical specification.

CGRAs have been studied in academia for over a decade, and a variety of different architectures have been proposed [1]. Commercially, recent years have seen two CGRAs in the market, the Samsung Reconfigurable Processor [2] and the STP Reconfigurable Processor from Renesas Electronics [3]. To date, however, all prior works on CGRAs have been confined to their respective frameworks, wherein a research group or company proposes a single *style* of CGRA design and demonstrates its utility for a set of applications. The design space for CGRAs is very large with many architectural decisions for each component and this large design space is tightly coupled with CAD and compiler techniques for mapping applications to the CGRA. While there exist software tools that allow the modelling and evaluation of fine-grained FPGA architectures [4], to our knowledge, there is no analogous framework that permits the scientific exploration of CGRAs. There is a need, then, for a tool that permits an architect to model hypothetical CGRA architectures and implement CAD algorithms, to evaluate the area, speed, and power of such designs over a set of applications in a domain of interest. We describe such a CGRA architecture exploration framework, called *CGRA-ME*.

CGRA-ME allows an architect to specify a CGRA architecture in an XML-based language, and provides a flexible scheduling, mapping, placement and routing tool, built within the popular LLVM compiler framework [5], that maps an optimized C-language benchmark onto the specified CGRA. Automatically generated Verilog RTL for the CGRA device permits simulation with ModelSim for functional verification, and also allows synthesis of the CGRA into standard cells using an ASIC flow to assess area, performance and power. We demonstrate the capabilities of CGRA-ME by modelling a variety of candidate CGRA architectures, mapping benchmark applications to the architectures, and producing standard cell designs.

CGRA-ME aims to amalgamate aspects of prior CGRA work within a larger, and more complete CGRA architecture evaluation framework that encompasses architecture description, architecture modelling, application mapping, system simulation, and physical implementation. The framework is open-source and freely available to the research community [6]. We believe the framework provides a foundation to open up a wide variety of research opportunities for CGRA architecture, algorithms and applications.

II. Related Work

Space limitations preclude a detailed review of prior architectures and the interested reader is referred to excellent survey articles [7], [8], [9], [10], [11]. However, it is worth highlighting a few ways that CGRAs differ from one another. A first defining attribute relates to the logic functionality – the types of computations each CGRA block is able to perform. Many CGRAs in the literature contain blocks that can perform ALU-like computations, e.g. n-bit multiplication, or n-bit logical operations with blocks being homogeneous or heterogeneous. A second defining attribute is the CGRA interconnect

978-1-5090-4825-0/17/$31.00 ©2017 IEEE

architecture, which may range from a linear unidirectional style, to a two-dimensional mesh with nearest-neighbour connectivity between blocks. A third attribute concerns memory and whether it is contained within the CGRA, is external, or hybrid internal/external. A final attribute pertains to dynamic reconfigurability, reflecting whether or not the CGRA function units and interconnect have a static configuration, or whether functionality and routing can change on a cycle-by-cycle basis, allowing for multi-context CGRA operation.

Prior work has considered how applications can be mapped onto CGRAs (e.g. [12], [13], [8]). In most CGRA mapping tools, the input benchmark is represented as a data-flow graph (DFG). The CGRA target architecture, including blocks and their connectivity, is also represented as a graph. The mapping problem is to embed the data-flow graph onto the device graph. The key CAD tasks are scheduling, placement and routing. Scheduling involves determining the timestep for each operation; placement locates the operations from the data-flow graph onto specific locations within the device; routing selects the signal paths between placed operations. For CGRAs that support dynamic reconfigurability, the device model graph is usually replicated multiple times, where each replicant will represent the block functionality and routing in each execution context.

Several recent articles are related to sub-pieces of the CGRA-ME framework. Our mapping work bears similarity to the SPR tool from the University of Washington [12], which provides a generic mapper for CGRAs, able to adapt to a user-provided architecture specification. Other works [14], [15] describe simulators for CGRAs; however, these are disconnected from physical CGRA implementation and modelling. Another work [16] describes a CGRA architecture description language, but is disconnected from mapping of benchmarks.

III. CGRA-ME Framework Overview

Fig. 1 shows the overall CGRA-ME framework. Shaded portions represent implemented functionality described in this paper; unshaded portions represent planned future functionality. The pieces of the framework are numbered in the sequence of typical usage. Box ① represents the CGRA architecture description, specified by an architect in our custom XML-based language *CGRA-ADL*. At ②, the architecture description is parsed and used to construct a software-level architecture model of the CGRA. From this architecture model, a Device Model at ③ can be generated. This model represents the physical logic and interconnect resources of the CGRA and is used by the scheduling, placement and routing tool at box ④, which maps a benchmark data-flow graph ⑤ into the modelled CGRA. Prior to the mapping, the benchmark is subjected to standard compiler optimizations in LLVM at ⑥. Note that the input to the mapper is *both* a benchmark and an architecture description – the mapper is not tied to a specific CGRA device architecture. The output of the mapping step is a configuration bitstream for the CGRA, defining its logic and interconnect behaviour across clock cycles.

Synthesizable Verilog RTL can also be generated from the architecture model of the CGRA ⑧ and with this, and the configuration bitstream for the benchmark, the user is able to simulate the configured CGRA with ModelSim to verify correct logic functionality ⑦. The RTL can also be input to a standard cell ASIC flow ⑨ to permit a silicon implementation.

Future planned functionality includes the development of power, performance and cost models for the CGRA, derived from standard-cell ASIC implementation. We also envision that the architecture interpreter ② could generate a cycle-accurate SystemC model of the CGRA, which, when configured for a particular benchmark, would provide faster simulation and verification than ModelSim RTL simulation. The SystemC simulation results would be combined with the power, performance, and cost models, to provide estimates of overall CGRA performance.

IV. Architecture Model and Architecture Description Language

The CGRA-ME framework is able to model a wide range of CGRA architectures via an XML-based architectural language, *CGRA-ADL*. The language permits an architect to describe a CGRA in detail, permitting an in-memory device model of the complete CGRA to be constructed. The device model contains primitive components, such as routing multiplexers, registers, register files, and functional units, the connectivity between them, and configuration cells to program the array. An example of a module with primitive components is shown in Fig. 2. Each primitive within this module has its own parameters such as bit-width, or number of inputs. The in-memory device model is used to map applications onto the CGRA, and to generate synthesizeable Verilog RTL. The language also permits custom user-defined blocks to be integrated within architectures, as long as the user-defined block has its own predefined in-memory device model.

The CGRA description language is inspired by the language used in the VTR FPGA architecture evaluation framework [4]. We highlight key aspects here; the complete language specification can be found online [6]. The overall structure of an architecture description is shown in Fig. 3. The `<cgra>` tag on Line 1 opens the architecture description. Following this, the architect may make one or more definitions (Line 2), which are similar to `#define` macros in `C`. Next, the architect describes the one or more CGRA blocks that will comprise the architecture, including intra-block logic and routing (Lines 3–7). The later Lines 8–12 compose the overall array architecture by instantiating a pattern of the defined blocks and specifying the inter-block connectivity.

An example CGRA block along with its CGRA-ADL description is shown in Fig. 4. Lines 2–4 declare the module ports and their directions. Line 5 declares a wire. Line 6 instantiates a functional unit inside the CGRA block. `FuncUnit` is a primitive type in the language for a two-input single-output functional unit. Lines 7–10 form the connections between the ports, wire and the functional unit. Fig. 5 shows interconnection structures available for modelling intra-CGRA block connectivity. Four structures may be used: *direct*, *distributive*, *multiplexer* and *crossbar*. The multiplexer and crossbar interconnection on Lines 3 and 4, respectively, imply the presence of configuration cells.

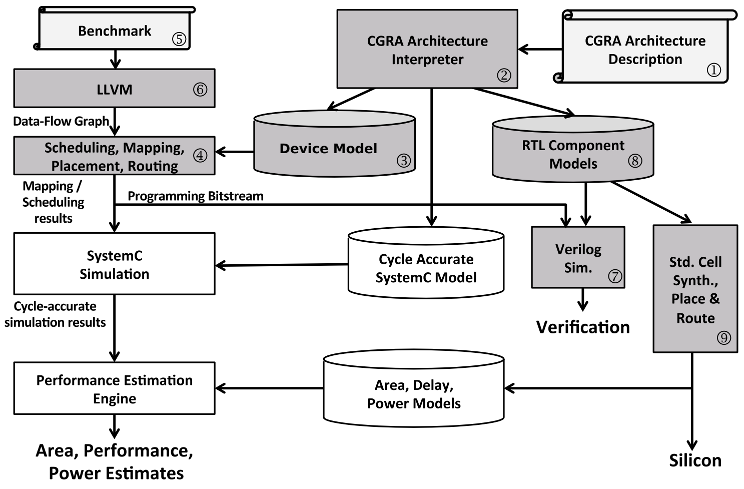

Benchmark ⑤
LLVM ⑥
Data-Flow Graph
Scheduling, Mapping, Placement, Routing ④
Mapping / Scheduling results
Programming Bitstream
SystemC Simulation
Cycle-accurate simulation results
Performance Estimation Engine
Area, Performance, Power Estimates
CGRA Architecture Interpreter ②
CGRA Architecture Description ①
Device Model ③
RTL Component Models ⑧
Cycle Accurate SystemC Model
Verilog Sim. ⑦
Verification
Std. Cell Synth., Place & Route ⑨
Area, Delay, Power Models
Silicon

Fig. 1: Framework vision and development status. Shaded portions represent implemented functionality described in this paper; unshaded portions represent planned future functionality.

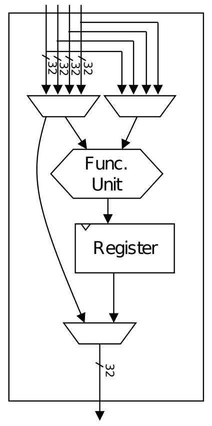
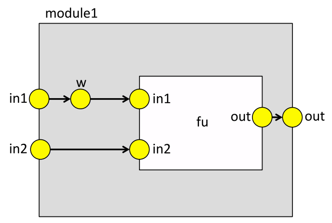


Fig. 2: Example CGRA processing block made from primitive components: three multiplexers, a functional unit and a register. Configuration cells (not shown) for the functional unit and routing multiplexers are automatically inferred within the framework.

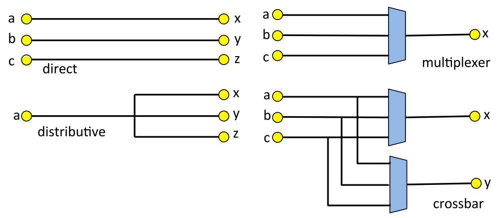
```
<cgra>
  <definition .../>
  <module name="block1">
    <input name="in"/>
    <inst module="fu"/>
    <connection .../> ...
  </module>
  <architecture row="4" col="4">
    <pattern>
      <block module="block1"/>
    </pattern> ...
  </architecture>
</cgra>
```

Fig. 3: CGRA-ADL architecture description example.



```
<module name="module1">
  <input name="in1"/>
  <input name="in2"/>
  <output name="out"/>
  <wire name="w"/>
  <inst name="fu" module="FuncUnit"/>
  <connection from="this.in1" to="w"/>
  <connection from="w" to="fu.in1"/>
  <connection from="this.in2" to="fu.in2"/>
  <connection from="fu.out" to="this.out"/>
</module>
```

Fig. 4: Using CGRA-ADL to describe a module (named `module1`) that contains a single functional unit and also to describe `module1`'s internal connections to the functional unit's input and output ports. An internal wire `w` is also modelled.



```
1: <connection from="a b c" to="x y z"/>
2: <connection from="a" distribute-to="x y z"/>
3: <connection select-from="a b c" to="x"/>
4: <connection select-from="a b c" to="x y"/>
```

Fig. 5: CGRA-ADL interconnect structures. Line 1 models direct connections; line 2 models distributive connections; line 3 models multiplexers, and line 4 models crossbars.

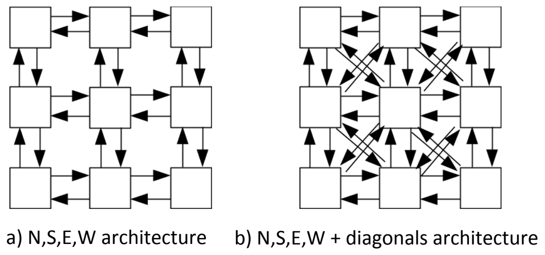


Fig. 6: CGRA interconnection architectures for which 'syntactic sugar' is provided in CGRA-ADL.

Additionally, we also provide a "syntactic sugar" shortcut for two common interconnect patterns, depicted in Fig. 6. The first (Fig. 6(a)) is an architecture with North, South, East, West connectivity; the second (Fig. 6(b)) adds the diagonal connections.

V. LLVM Front-end

LLVM is used to implement the front-end compiler support for the framework. Benchmarks written in `C` are parsed, optimized and translated into data-flow graphs (DFGs) [5]. The nodes in the DFG represent instructions from the LLVM intermediate representation (IR). The IR is based on a reduced/simple instruction set of basic logical and arithmetic operations, such as multiply, add, subtract, exclusive-OR, etc. LLVM's IR can be viewed as a machine-independent assembly code representation of an input program. These DFGs are written out into a dot graph format [17] that includes metadata, such as labeling inputs, outputs, operations, and operands within the operations. Each operation within the DFG corresponds to one LLVM instruction and the computational capabilities of a CGRA block are specified in terms of the types of LLVM instructions they are capable of supporting. In this way, the compiler's (i.e. LLVM's) representation of the computations and computational capabilities of the modelled CGRA architecture are aligned with each other.

VI. Mapper

Mapping applications to CGRA architectures involves associating each operation in the application's DFG with a physical functional unit, while finding and ensuring a route between inputs and outputs of each operation, with the correct latency and schedule. Here, we use the term *Mapper* to refer to Fig. 1 component ④. To model the physical routes and functional units within the architecture, a *Modulo Routing Resource Graph* (MRRG) [13] is constructed from the Device Model of the CGRA (Fig. 1 component ③). This graph, elaborated on in Section VI-A, represents routes and resources that can be used by the application DFG. The mapping problem then reduces to associating a DFG with an MRRG. This process of associating each operation in the DFG with a functional unit in the MRRG, while establishing a route between inputs and outputs of each operation, is quite difficult. In the current mapper, we take a simulated annealing approach for associating DFG operations to MRRG functional units [18]. Subsequently, a PathFinder-like [19] approach is used for routing connections between functional units. Since we map the DFG onto the MRRG, which already accounts for resource usage on a cycle-by-cycle basis, the scheduling of operations is implicit in the placement and routing. Similar approaches to this mapping problem have also been proposed previously in literature [12], [13].

A. Modulo Routing Resource Graph

The Modulo Routing Resource Graph (MRRG) is an abstract representation of the physical architecture of a CGRA [13]. This representation formulates the constraints of the modulo scheduling problem, operator placement, and value routing, within the graph itself and many recent architectures and tools rely on some form of this representation [20], [21], [12], [22], [23]. The graph contains, as vertices, all resources of the CGRA: the routes within the physical architecture, the storage elements, and the functional units that execute operations. The MRRG representation is especially useful for multiple-context architectures where routes and functional units can be shared to perform different operations on subsequent cycles in a repeating pattern. To model such multi-context architectures, the MRRG is composed of multiple instances of the CGRA device model – one for each context. A register in the underlying CGRA architecture, which stores a value across clock cycles, is translated into an edge between nodes in two subsequent device model instances. The mapping of DFG operations and edges to the MRRG reflects the computations each functional unit is performing in each dynamic context, and also the routing connectivity.

B. Simulated Annealing Mapper

The simulated-annealing mapper implemented within our framework allows for flexibility to map data-flow graphs to many architectures modelled in the CGRA-ADL XML language. Specifically, it handles a variety of inter-connectivity and varied functional units, further demonstrated in Section VIII.

We begin with initial placement. Every operation node in the DFG is placed onto a functional unit that is capable of performing the operation specified by the operation node. For example, a multiply operation will never be placed on a function unit that can only do additions. However, temporary overuse of functional units is permitted – multiple operation nodes may be placed on a single functional unit. Likewise, routes between operation nodes are allowed to be temporarily overused in a PathFinder-like fashion [19]. Overuse of resources and interconnect is penalized through costing during the annealing process and gradually removed.

The cost of a mapping is the summation of all used routing nodes and all used functional unit nodes within the MRRG. Along with this base cost, an additional penalty is applied for each routing or functional unit node that is overused. After initial placement, we enter the main simulated annealing loop and perform random operation swaps with necessary re-routing. Swaps are assessed using the cost function and accepted if cost decreases or probabilistically based on the current temperature. Temperature is decreased and the overuse penalty is increased periodically after a certain number of swaps. Mapping is considered complete as soon as a legal placement and routing is found.

VII. Physical Implementation

The framework is able to generate Verilog RTL for modelled CGRA architectures. The Verilog RTL can be simulated for functional verification. It can also be synthesized so that physical implementations of the CGRA can be produced and evaluated. Such physical implementations can inform area, performance and power models in an architecture evaluation context. An ASIC implementation can be produced by using a standard-cell toolchain or alternately, an FPGA *overlay* can be realized.

Our standard-cell synthesis flow uses Synopsys Design Compiler for technology mapping to the cell library, Cadence Encounter for placement and routing (with automatically generated floorplanning constraints), and Synopsys PrimeTime for timing analysis. It is worth mentioning that CGRAs with different area, performance, power trade-offs can be produced by simply changing the constraints supplied to the ASIC tools.

VIII. Architecture Exploration Case Study

The goal of this study is to demonstrate the viability of the CGRA-ME framework. The study accomplishes this through:

Benchmark	I/Os	Operations	Multiplier
add 10	10	10	0
add 14	14	14	0
add 16	16	16	0
2x2-p 1	5	5	1
2x2-f 2	6	6	1
cos 4	5	14	12
cosh 4	5	14	12
exponential 4	4	9	5
exponential 5	5	12	9
exponential 6	6	15	14
multiply 10	10	10	9
multiply 14	14	14	13
multiply 16	16	16	15
sinh 4	5	13	9
taylor series 4	5	10	6
weighted sum	16	16	8
mac	1	9	3

TABLE I: Benchmarks.

1) demonstrating CGRA-ME's capability to model a variety of CGRA architectures; 2) demonstrating the ability of the mapper to map applications onto various CGRA architectures. 3) demonstrating the framework's ability to produce Verilog RTL for a modelled CGRA, through a standard-cell implementation.

A variety of CGRA architectures are considered in this study, chosen to be representative of CGRAs proposed in the literature and different enough from one another to exhibit varying levels of mapping difficulty. Each architecture comprises an array of CGRA blocks with 32-bit-wide bus-based interconnect. The sample CGRA block we use is the same as in Fig. 2 with a few differences to the input multiplexers and functional unit capabilities. We consider two function block architectures and two interconnect architectures with varying array sizes. The architecture sizes range from 4×4 to 6×6. Additionally, we model these architectures with 1, 2 and 4 execution contexts.

The two function block architectures are called Homogeneous and Heterogeneous. In the former, all blocks can perform ALU-like operations: shift, logical, add, subtract, multiply. In the later, only half of the blocks can perform all operations; the other half can do all but multiply. In all designs, the output of every functional unit is registered. The two interconnect styles are called Orthogonal and Diagonal, reflecting the two routing architectures shown in Fig. 6, respectively. In the diagonal case, the routing multiplexers shown in Fig. 2 are widened to accommodate the additional diagonal connections. In all architectures, I/O blocks are placed on the perimeter. A single memory-access unit that can perform loads and stores is dedicated to each row within all architecture variants. The outputs of each block within a row is connected to the inputs of the access unit (address and data-in) while the output of the memory block (data-out) is connected to each of the blocks in the row as an additional input.

A. *Mapping & Architecture Evaluation*

The benchmarks are data-flow graphs having varying numbers of operation nodes and edges to reflect varying difficulties of mapping for the target architectures. Table I shows detailed characteristics of each benchmark.

Table II shows the mapping results for four different architectures, indicating whether the mapper was successful in finding an implementation for each benchmark in the four architectures. The table is divided into two sections; one for single context architectures and one for dual context. Within these divisions, there are the four architecture styles and within those are three different array sizes. Each cell indicates whether a benchmark was mapped (M) or unmapped (U). The architecture styles are arranged from least flexible (heterogeneous functional units with orthogonal routing) to most flexible (homogeneous functional units with diagonal routing). Broadly speaking, looking at the success-count in each column (last row), the results match with intuition: the largest array with the most flexible architecture can accommodate most of the benchmarks, whereas the smallest array with least flexible architecture can only accommodate a few due to limited compute ability and interconnect. Note that benchmarks using more than 8 multipliers (cf. Table I) are impossible to map within the single-context 4×4 heterogeneous architectures (as it has only 8 functional units that can perform the multiply operation).

The right side of Table II shows the mapping results for dual context architectures, where there are two configurations of the functional units and interconnect that alternate on a cycle-by-cycle basis. This effectively doubles the capacity of array though now, the throughput of the array is halved. An underlined cell in this half of the table indicates that the same benchmark was not able to be mapped with one context, but is now able to be mapped with two contexts. For these architectures, mapping success is much higher owing to the huge amount of flexibility granted by an additional execution context. The results for four context architectures are similar to the two context architectures but have been omitted due to space limitations.

We underscore that our intent here is not to draw architectural conclusions; rather, to demonstrate that CGRA-ME is capable of modelling and mapping to a variety of CGRA architectures having different degrees of functional and interconnection restrictiveness.

B. *Standard-Cell Implementation Results*

We synthesized standard-cell implementations for four 4×4 single context architectures evaluated, using the flow described in Section VII, targeting the TSMC $65\,\mathrm{nm}$ cell library. In this initial study, we direct the ASIC tools to produce area-optimized implementations without external memory or the memory port connections. Fig. 7 shows the floorplan and full metal layouts for two 4×4 CGRAs. The left two blocks show the floorplan and layout of the CGRA with homogeneous blocks and orthogonal connectivity between blocks; the right two blocks show the floorplan and layout of the heterogeneous CGRA with diagonal and orthogonal connectivity.

The area and timing analysis results for all four architectures evaluated are shown in Table III. The area results reflect the intuition that the homogeneous fabric with richer connectivity consumes the most area; the heterogeneous fabric with reduced connectivity consumes the least area. The critical path delay in all architectures is roughly the same, and reflects the delay through the multiplier ($\sim$ 200MHz for all architectures). The results in the table demonstrate that, from the architecture description in XML, CGRA-ME is able to generate a standard-

Benchmark	Single Context												Dual Context											
	Hetero. Orth.			Hetero. Diag.			Homo. Orth.			Homo. Diag.			Hetero. Orth.			Hetero. Diag.			Homo. Orth.			Homo. Diag.		
	4	5	6	4	5	6	4	5	6	4	5	6	4	5	6	4	5	6	4	5	6	4	5	6
add 10	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
add 14	U	U	M	M	M	M	U	U	M	M	M	M	$\underline{\text{M}}$	$\underline{\text{M}}$	M	M	M	M	$\underline{\text{M}}$	$\underline{\text{M}}$	M	M	M	M
add 16	U	U	M	U	M	M	U	U	M	U	M	M	$\underline{\text{M}}$	$\underline{\text{M}}$	M	$\underline{\text{M}}$	M	M	$\underline{\text{M}}$	$\underline{\text{M}}$	M	$\underline{\text{M}}$	M	M
2x2-f	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
2x2-p	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
cos 4	U	U	U	U	U	U	U	U	U	U	U	M	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	M
cosh 4	U	U	U	U	U	M	U	U	U	U	M	M	U	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	M	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	M	M
exponential 4	U	U	U	M	M	M	U	U	U	M	M	M	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	M	M	M	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	M	M	M
exponential 5	U	U	U	U	U	M	U	U	U	U	M	M	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	M	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	M	M
exponential 6	U	U	U	U	U	U	U	U	U	U	U	U	U	U	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	U	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$
multiply 10	U	U	M	U	M	M	M	M	M	M	M	M	$\underline{\text{M}}$	$\underline{\text{M}}$	M	$\underline{\text{M}}$	M	M	M	M	M	M	M	M
multiply 14	U	U	U	U	U	M	U	U	M	M	M	M	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	M	$\underline{\text{M}}$	$\underline{\text{M}}$	M	M	M	M
multiply 16	U	U	U	U	U	M	U	U	M	U	M	M	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	M	$\underline{\text{M}}$	$\underline{\text{M}}$	M	$\underline{\text{M}}$	M	M
sinh 4	U	U	U	U	U	U	U	U	U	U	U	U	U	U	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	U	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$
taylor series 4	U	U	U	M	M	M	U	U	U	M	M	M	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	M	M	M	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	M	M	M
weighted sum	U	U	M	M	M	M	U	U	M	U	M	M	$\underline{\text{M}}$	$\underline{\text{M}}$	M	M	M	M	$\underline{\text{M}}$	$\underline{\text{M}}$	M	$\underline{\text{M}}$	M	M
mac	U	U	U	U	M	M	U	M	M	M	M	M	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	$\underline{\text{M}}$	M	M	$\underline{\text{M}}$	M	M	M	M	M
# mapped	3	3	7	7	10	14	4	5	10	9	14	15	14	15	17	17	17	17	15	17	17	17	17	17

TABLE II: Benchmark mapping results for single and dual context architectures, four array styles and three array sizes. A Mapped benchmark is denoted by M and an Unmapped benchmark denoted by U. An $\underline{\text{M}}$ denotes a benchmark that was able to be mapped in two contexts but not in one context.

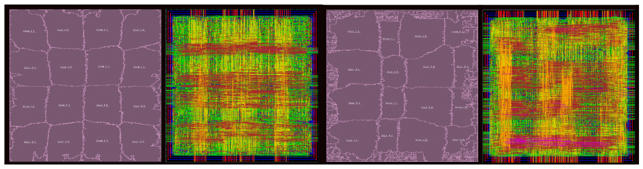


Fig. 7: Floorplans/layouts of 4x4 homogeneous orthogonal CGRA (left) and heterogeneous diagonal CGRA (right).

	Homo. Diagonal	Homo. Orthogonal	Hetero. Diagonal	Hetero. Orthogonal
Area (microns^2)	181,243	154,909	128,953	104,649
Delay (ns)	5.11	4.80	5.31	4.76

TABLE III: Area/timing results; single-context 4×4 archs.

cell implementation, that, combined with mapping results, permits the area/performance of hypothetical CGRA architectures to be evaluated for a set of benchmark applications.

IX. Conclusion & Future work

We have introduced CGRA-ME, a unifying CGRA framework that currently encompasses architecture description, architecture modelling, application mapping, and physical implementation. Through our architecture exploration case study, the CGRA-ADL language is demonstrated as a way for architects to describe a variety of CGRA architectures with varying components, sizes, and contexts. The generic simulated-annealing style mapper in CGRA-ME was shown to react to each architecture when mapping benchmarks – the more flexible the fabric, the higher the mapping success rate. Additionally, CGRA-ME was shown to generate Verilog RTL for the 4×4 arrays, from which we have generated a silicon design using TSMC 65 nm standard cells with a turnkey approach. The open-source CGRA-ME framework is freely available for the academic research community, providing a foundation for further CGRA research.

References

[1] H. Amano, "A Survey on Dynamically Reconfigurable Processors," *IEICE Transactions*, vol. 89-B, no. 12, pp. 3179–3187, 2006.

[2] C. Kim *et al.*, "ULP-SRP: Ultra Low-Power Samsung Reconfigurable Processor for Biomedical Applications," *ACM TRETS*, vol. 7, no. 3, pp. 22:1–22:15, 2014.

[3] T. Toi *et al.*, "Optimizing Time and Space Multiplexed Computation in a Dynamically Reconfigurable Processor," in *FPT*, 2013, pp. 106–111.

[4] J. Luu *et al.*, "VTR 7.0: Next Generation Architecture and CAD System for FPGAs," *ACM TRETS*, vol. 7, no. 2, pp. 6:1–6:30, 2014.

[5] C. Lattner and V. S. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *IEEE / ACM Intl. Symp. on Code Gen. and Opt.*, 2004, pp. 75–88.

[6] "CGRA-ME," 2017. [Online]. Available: http://cgra-me.ece.utoronto.ca

[7] R. Tessier, K. Pocek, and A. DeHon, "Reconfigurable Computing Architectures," *Proc. of the IEEE*, vol. 103, no. 3, pp. 332–354, 2015.

[8] B. De Sutter, P. Raghavan, and A. Lambrechts, *Coarse-Grained Reconfigurable Array Architectures*. Springer New York, 2013, pp. 553–592.

[9] R. Hartenstein, "Coarse Grain Reconfigurable Architectures," in *ASP-DAC*, 2001, pp. 564–569.

[10] H. Amano, "A Survey on Dynamically Reconfigurable Processors," *IEICE Transactions*, vol. 89-B, no. 12, pp. 3179–3187, 2006.

[11] V. Tehre and R. Kshirsagar, "Survey on Coarse Grained Reconfigurable Architectures," *Intl. Jrnl. of Comp. Appl.*, vol. 48, no. 16, pp. 1–7, 2012.

[12] S. Friedman *et al.*, "SPR: An Architecture-adaptive CGRA Mapping Tool," in *ACM FPGA*, 2009, pp. 191–200.

[13] B. Mei *et al.*, "DRESC: A Retargetable Compiler for Coarse-Grained Reconfigurable Architectures," in *IEEE FPT*, 2002, pp. 166–173.

[14] K. Patel, S. McGettrick, and C. J. Bleakley, "Rapid functional modelling and simulation of coarse grained reconfigurable array architectures," *Journal of Systems Architecture*, vol. 57, no. 4, pp. 383–391, 2011.

[15] A. Chattopadhyay and X. Chen, "A Timing Driven Cycle-Accurate Simulation for Coarse-Grained Reconfigurable Architectures," in *Intl. Symp. on Appl. Reconf. Comp.*, 2015, pp. 293–300.

[16] J. O. Filho *et al.*, "CGADL: An Architecture Description Language for Coarse-grained Reconfigurable Arrays," *IEEE Trans. VLSI Syst.*, vol. 17, no. 9, pp. 1247–1259, Sep. 2009.

[17] E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering," *Software: Practice and Experience*, vol. 30, no. 11, pp. 1203–1233, 2000.

[18] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi *et al.*, "Optimization by Simmulated Annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[19] L. McMurchie and C. Ebeling, "PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs," in *FPGA*, 1995, pp. 111–117.

[20] H. Park *et al.*, "Modulo Graph Embedding: Mapping Applications onto Coarse-Grained Reconfigurable Architectures," in *Proc. of the Intl. Conf. on Comp., Arch. and Synth. for Embed. Sys.* ACM, 2006, pp. 136–146.

[21] ——, "Edge-centric Modulo Scheduling for Coarse-Grained Reconfigurable Architectures," in *Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques*. ACM, 2008, pp. 166–176.

[22] L. Ma, W. Ge, and Z. Qi, "A Graph-Based Spatial Mapping Algorithm for a Coarse Grained Reconfigurable Architecture Template," *Inf. in Ctrl., Auto. and Robo.*, pp. 669–678, 2012.

[23] L. Chen and T. Mitra, "Graph Minor Approach for Application Mapping on CGRAs," *ACM TRETS*, vol. 7, no. 3, pp. 21:1–21:25, Sep. 2014.