Pillars: An Integrated CGRA Design Framework

Yijiang Guo, Guojie Luo
Center for Energy-efficient Computing and Applications, Peking University, Beijing, China
Email: {yijiang, gluo}@pku.edu.cn

***Abstract*—In this paper, we propose Pillars, an integrated CGRA design framework, to assist in design space exploration and hardware optimization of CGRA. Pillars allows an architect to describe a hierarchical CGRA design in a Scala-based language and produce an in-memory model for both behavior and structure. The model generates the RTL code and the structure for reconfiguration. This structure enables application mapping and context generation in a flattened representation generated from a hierarchical model. Thus, CAD tools in Pillars are able to map applications onto the architecture and produce contexts that enable cycle-accurate simulations. In the experimental evaluation, we demonstrate the capability of Pillars to model CGRA architectures by synthesizing variants of a widely known CGRA architecture, ADRES, into FPGA overlays.**

I. Introduction

Coarse-grained reconfigurable array (CGRA) is a class of reconfigurable architecture that provides word-level granularity in a reconfigurable array to overcome some of the disadvantages of FPGAs. CGRAs provide the capability for spatial, temporal and parallel computation, and hence can outperform common computing systems in many applications. CGRAs have been studied in academia for over a decade and a variety of CGRA architectures have been proposed [1].

There exist software tools [2] that the exploration of fine-grained FPGA architectures largely benefit from, while CGRA design and exploration tools remain in an embryonic period. Since design space for CGRAs is very large with many architectural decisions, there are increasing demands of a tool that permits the scientific exploration of CGRAs. Abstract architecture modeling, computer-aided design (CAD) algorithms, automatic RTL generator, and simulator should be integrated into a unified framework to adapt to the requirement of evaluating the area, speed, and power of designs over a set of applications in a specific domain.

CCF [3] is a CGRA compilation and simulation framework that is built on gem5 simulator [4], which does not simulate specific details like power and area. Stanford University proposed an open-source hardware/software tool chain for CGRA [5] that can rapidly create and validate alternative hardware implementations, but the immutable hardware template and the tediously long tool chain limit the adaptability for modern CGRAs with heterogeneous PEs, complex memory and interconnect. A recent framework CGRA-ME [6] permits the modeling and exploration of a wide variety of CGRA architectures and also facilitates research on CGRA mapping algorithms. The drawback of CGRA-ME is that the RTL generation rules written by experts are overmixed into the architecture interpreter, and therefore, the generator becomes brittle when developers iterate the logical implementation cycles after the feedback from physical design.

We propose Pillars[1], an open-source CGRA design framework, to assist in design space exploration and hardware optimization of CGRAs. Pillars provides a Scala-based architecture description language (ADL) for an architect to specify a CGRA architecture, which produces a unified, high-quality and synthesizable architectural abstraction. Auxiliary hardware modules and Verilog RTL are automatically generated according to the architectural abstraction, allowing physical implementation on an FPGA as an overlay. An integer linear programming (ILP) CAD tool can map data-flow graph (DFG) onto the specified CGRA, generating contexts for CGRA RTL-level simulation. Architecture designing, mapping, RTL generation and simulation are integrated in a unique framework, which benefits the division and cooperation of architects, CAD algorithm designers and hardware engineers.

II. Pillars

Taken integration into consideration, the major tools in Pillars are developed based on the Scala programming language [7], a widely used host language for developing embedded domain-specific language (eDSL) running on the Java virtual machine (JVM). Chisel [8], a Scala embedded hardware construction language that supports advanced hardware design using highly parameterized generators and layered domain-specific hardware languages, plays the role of Verilog RTL generator in our framework.

A. Overview

Fig. 1 illustrates the overall Pillars framework, where components and data-flow between them are shown. The components of the framework are numbered in the sequence of typical usage. The yellow portions represent tools or actions in our framework. The blue portions represent intermediate results during runtime. The grey portions represent inputs in a specific format.

The inputs to the framework are models in Scala-based ADL for the description of CGRA architectures ① and commonly accepted data-flow graphs (DFGs) [9] for the description of applications ⑦. The ADL of CGRA is parsed by an architecture interpreter ②, producing a hierarchical abstract model of the depicted CGRA architecture ③. In order to obtain a high-quality representation for mapping and reduce the complexity of RTL generation, the hierarchical abstract

[1]https://github.com/pku-dasys/pillars

model will be flattened ④. The flattened abstract model in device will produce corresponding basic Chisel modules ⑤ and modulo routing resource graph (MRRG) [10] to model CGRAs ⑥.

Mapper receives the DFG for a specific application as input, as well as the MRRG model of the CGRA architecture, to map the DFG onto the CGRA, and scheduler will reconstruct the schedule of mapping results ⑧. Together with the hierarchical abstract model, the products of mapper and scheduler can be translated into contexts that will be applied in simulation.

The auxiliary modules will be automatically generated depending on the regions of basic modules in the hierarchical abstract model to support cycle-accurate simulation, and interconnection will be realized ⑨. As a result, we will gain a Chisel top design ⑩ and thus the automatic generation of Verilog RTL ⑪ can be carried out.

We implement a component that aids simulator code generation ⑫. With the help of Chisel I/O tester and Verilator [11], a power RTL simulator used by RocketChip [12], we can obtain the result of cycle-accurate simulation for functional verification ⑬. In Section III, we demonstrate FPGA-overlay implementations of variants of the ADRES [13] CGRA architecture ⑭. Combining the performance, area and consumption of FPGA-overlay ⑮ with the mappability, throughput and runtime from mapper, we can evaluate the performance, power, and area of depicted designs of CGRA over a set of applications in a domain of interest ⑯.

B. Architecture Description

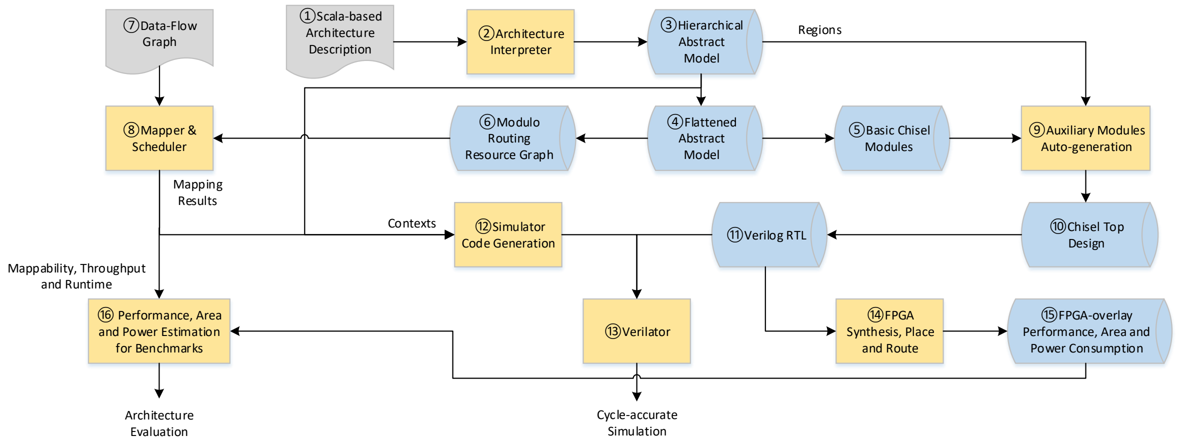
We employ a hierarchical design and flattened implementation methodology in our framework. The ADL for architecture description maintains its hierarchical heritage while all physical implementations are flattened. Only the basic elements of architecture are still corresponding to hardware modules while redundant nodes and layers will be optimized. Our methodology shields architects from complex detail of low-level hardware and enables hardware engineers to focus on the hardware generation of a few categories of fundamental modules, which separates the concerns of architects and hardware engineers.

The Pillars framework has the ability to model various CGRA architectures via Scala-based ADL, which inherits the syntax of Scala. Blocks and elements are fundamental components in our ADL. Blocks are able to represent the hierarchy, and each element shares a particular identification number with corresponding Chisel hardware implementation. A block can be composed of several sub-blocks and elements. There are five alternatives of predefined elements, multiplexer, const unit, arithmetic logical unit (ALU), load/store unit (LSU) and register files (RF).

Fig. 2 illustrates an example of architecture description. The block contains an ALU able to perform computation between the selected input and an immediate operand, and a subblock with 2 input ports and 1 output port. All blocks and elements are identified by names, and if they share a collective parent block, their name must be different. Each block can have any number of input and output ports through function calls, while an element should guarantee the same number of input and output ports with corresponding hardware, and names of them can be also specified. Connections between parent block, subblocks and elements can be added in a particular form. Elements have some parameters to define the hardware specifications. Since the block is declared as a configuration region, so all elements and elements in its subblocks share an auto-generated configuration controller, which is capable of storing and distributing configurations.

C. Mapper & Scheduler

The inputs of the mapper and scheduler are DFG and MRRG. A DFG is written in a dot graph format [14] that includes metadata, such as labeling inputs, outputs, operations, and operands within the computation. MRRG [10] has been used extensively in studies of CGRA due to its capability of modeling multiple contexts. The context repeats every



⑦Data-Flow Graph
①Scala-based Architecture Description
②Architecture Interpreter
③Hierarchical Abstract Model
Regions
⑧Mapper & Scheduler
⑥Modulo Routing Resource Graph
④Flattened Abstract Model
⑤Basic Chisel Modules
⑨Auxiliary Modules Auto-generation
Mapping Results
Contexts
⑫Simulator Code Generation
⑪Verilog RTL
⑩Chisel Top Design
Mappability, Throughput and Runtime
⑯ Performance, Area and Power Estimation for Benchmarks
⑬Verilator
⑭FPGA Synthesis, Place and Route
⑮FPGA-overlay Performance, Area and Power Consumption
Architecture Evaluation
Cycle-accurate Simulation

Fig. 1: Pillars framework overview showing the main components.

```
class BlockImmediate(name: String) extends BlockTrait {
  ......
  setConfigRegion()
  addInPorts(Array("in0", "in1"))
  addOutPorts(Array("out0"))

  // A multiplexer that can choose a data source
  // for the port "inputA" of the ALU.
  val mux0 = new ElementMux("mux0", muxParams)
  mux0.addInPorts(Array("input0", "input1"))
  mux0.addOutPorts(Array("out0"))
  addElement(mux0)

  // An ALU that can perform some operations.
  val alu0 = new ElementAlu("alu0", aluOpList,
                   supBypass = true, aluParams)
  alu0.addInPorts(Array("inputA", "inputB"))
  alu0.addOutPorts(Array("out0"))
  addElement(alu0)

  // A const unit connected to the port "inputB" of ALU.
  val const0 = new ElementConst("const0", constParams)
  const0.addOutPorts(Array("out0"))
  addElement(const0)

  // A black box with 2 input ports and 1 output port.
  val subBlock = new BlackBox("subBlock0")

  // Interconnection inside this block.
  addConnect(term("in0") -> mux0 / "input0")
  addConnect(term("in1") -> mux0 / "input1")
  addConnect(mux0 / "out0" -> alu0 / "inputA")
  addConnect(const0 / "out0" -> alu0 / "inputB")
  addConnect(term("in1") -> subBlock  / "input0")
  addConnect(alu0 / "out0" -> subBlock  / "input1")
  addConnect(subBlock / "out0" -> term("out0"))
}
```

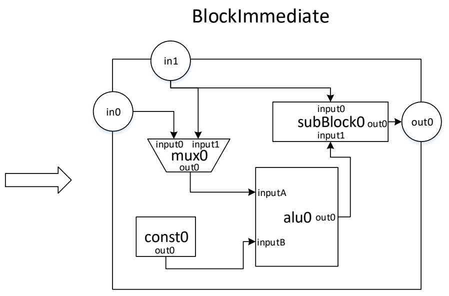


Fig. 2: An example of the Scala-based ADL. The settings of name and parameters are omitted.

```
/** A template tester.
 *
 * @param c              the top design
 * @param appTestHelper the class which is helpful
 *                        when creating testers
 */
class TemplateTester(c: TopModule,
   appTestHelper: AppTestHelper)
  extends ApplicationTester(c, appTestHelper) {

  val testII = appTestHelper.getTestII()

  //pre-process
  poke(c.io.en, 0)
  inputData()
  inputConfig(testII)

  //activating process
  poke(c.io.en, 1)
  checkPortOutsWithInput(testII)

  //post-process
  checkLSUData()

}
```

Fig. 3: A sample code of typical tester in Pillars.

II (initiation interval) cycles, with a new iteration of the application loop starting each repetition. The MRRG, which is the structure for reconfiguration, can be generated in Pillars according to the flattened abstract model and II.

The target of our mapper and scheduler is to determine where and when the operators in a DFG fire. We map each operator in DFG onto a functional node in MRRG with an ILP mapper. The ILP formulation of mapper is mainly based on Chin's approach [15]. The fire time and synchronization strategy of each operator are determined with a scheduler using topological search.

D. Hardware Generation

Basic Chisel modules, also called explicit modules, will be generated with the flattened abstract model at first. Auxiliary modules are automatically inferred to aid reconfiguration and running applications correctly. After the wires are connected, a Chisel top design is produced, which can generate the RTL code and enable cycle-accurate simulation.

Explicit modules corresponding to elements are the cornerstones of hardware generation. According to the parameters set up by users in the ADL, an explicit module can be generated with different data widths, sizes, logics and so on.

Pillars hides the generation process of auxiliary modules from architects while hardware engineers can improve the performance and quality of them in an arbitrary way. There are three kinds of auxiliary modules: configuration controllers, schedule controllers and synchronizers. Configuration controllers can repeat stored configurations every II cycles and distribute them to corresponding explicit modules. To control the cycle modules should fire, we employ the schedule controllers to fire ALUs and LSUs when operators are mapped onto them. Synchronizers can implement synchronous inputs for explicit modules with more than one input ports.

E. RTL-level Simulation

To simplify the simulator code generation, we define three processes of programming: pre-process, activating process and post-process. In the pre-process, the input data stream is transferred into LSUs through direct memory access (DMA), and contexts are read by the top-level CGRA module. The necessary contexts for the execution of an application are generated from the results of mapper and scheduler. After the top module is enabled, the activating process starts and auxiliary modules are fired. Explicit modules can perform routing or operations set by configuration controllers, if they have been fired by schedule controllers. In the post-process, we can get the output data stream from LSUs. The post-process is not necessary if there are no store operations in the targeted DFG.

As shown in Fig. 3, a few templates and tools in Pillars are useful to construct the simulation processes and produce classes in the specific format of Chisel testers using the Verilator backend. Thus, we can obtain the result of cycle-accurate simulation. The expected behaviors of CGRA can be verified at output ports of the top module during the activating process or the data obtained from LSUs during the post-process.

III. Experimental Study

A. Experimental Architectures

In our study, we model four CGRA architectures with two different PE designs (Fig. 6a & b), which are based on variants of the ADRES [13] architecture skeleton (Fig. 4). The complex PE (Fig. 6b) has two additional bypass multiplexers, which are also adopted in CGRA-ME [6]. The prototype of the full and reduced architecture skeletons in Fig. 4 are proposed in [16].

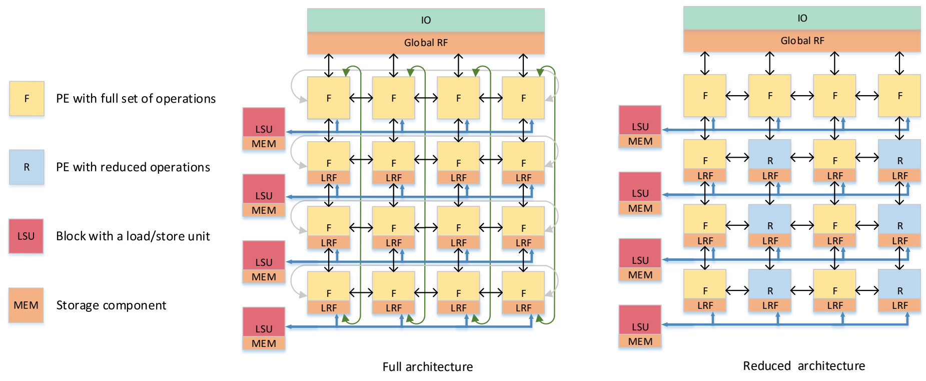

F
PE with full set of operations
R
PE with reduced operations
LSU
Block with a load/store unit
MEM
Storage component
IO
Global RF
LRF
Full architecture
Reduced architecture

Fig. 4: Variants of ADRES architecture skeleton.

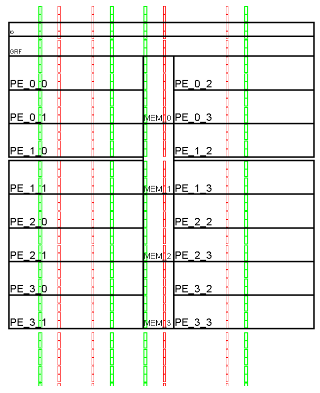
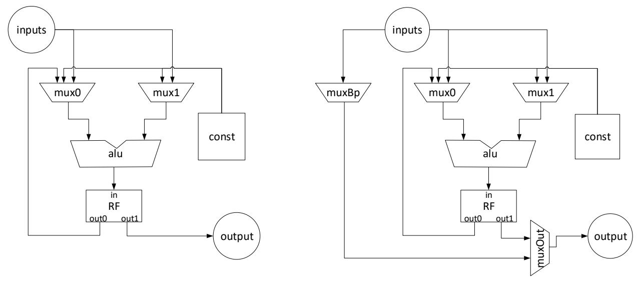


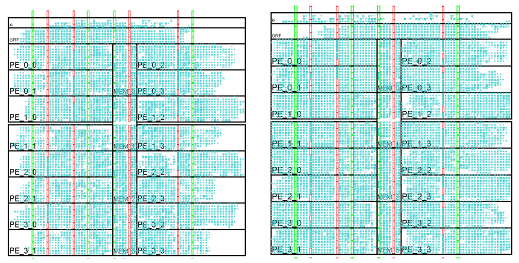
Fig. 5: Floorplan for Vivado place & route.



(a) Simple PE.

(b) Complex PE with two additional bypass multiplexers.

Fig. 6: Several architectures of PE with a local RF.



(a) Reduced-simple arch. generated from Pillars.

(b) Full arch. generated from Pillars.

Fig. 7: Layout of selected FPGA implementations.

The architectures in our experiments are denoted as Full, Full-Simple, Reduced, and Reduced-Simple, respectively.

In Fig. 4, ALUs in full PEs (yellow) can perform a full set of operations: `add`, `subtract`, `multiply`, `shifts`, `and`, `or`, and `xor`, while those in reduced PEs (blue) only have the capability of `add` and `subtract`. PEs in the same row share a block with a load/store unit. Similar to the original ADRES architecture, the PEs in the top row share a global RF instead of a local RF. The full architecture has torus connectivity between top and bottom rows, and between leftmost and rightmost columns. All the above architectures are implemented with a 32-bit data width.

B. Physical Implementation and Mapping Results

We target the Xilinx ZYNQ-7000 ZC706 evaluation board using Vivado 2019.2 for physical implementation. Fig. 5 presents the floorplan for all architectures under test by Verilog generated from Pillars. Considering the resource distribution, we interleave two neighboring PE columns to satisfy the resource requirement and improve the performance. The blocks with LSU are located at the center because of the communications with PEs.

Fig. 7 exhibits the layout of selected FPGA implementations in our experiments. Table I shows the maximum frequency and FPGA area breakdown of the implementations, as well as the success rate of mapping within 7200 seconds for benchmarks in [16] for 10 times over different random seeds. The resource usages of both explicit modules (on the left of "/") and auxiliary modules (on the right) are reported. For architectures with Reduced skeleton, benchmarks with many multiplication operations cannot find feasible mapping within the time limit, since there are only 10 multipliers and no toroid connections.

TABLE I: Physical implementation on ZC706 and mapping results for each architecture.

	Full Arch.	Reduced Arch.	Full-Simple Arch.	Reduced-Simple Arch.
Fmax[MHz]	32.2	36.8	86.2	87
LUT	13604 / 4902	11061 / 4646	11570 / 4488	9515 / 4376
FF	1656 / 8248	1656 / 8213	1656 / 8056	1656 / 8050
DSP	48 / 0	30 / 0	48 / 0	30 / 0
BRAM	2 / 0	2 / 0	2 / 0	2 / 0
Success rate	97.8%	55.6%	98.9%	55.6%

IV. Conclusion

We propose Pillars, a powerful and integrated CGRA design framework with consistent RTL and context generation. The capabilities of Pillars to model various architectures, generate RTL codes, map applications for context generation, and perform cycle-accurate simulation are demonstrated.

References

[1] M. Wijtvliet, L. Waeijen, and H. Corporaal, "Coarse grained reconfigurable architectures in the past 25 years: Overview and classification," in *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, 2016.

[2] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed *et al.*, "VTR 7.0: Next generation architecture and CAD system for FPGAs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 2, pp. 1–30, 2014.

[3] S. Dave and A. Shrivastava, "CCF: A CGRA compilation framework," https://github.com/MPSLab-ASU/ccf, 2018.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[5] A. H. C. Stanford, "Documentation for the entire CGRAFlow," https://github.com/StanfordAHA/CGRAFlowDoc, 2019.

[6] S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. Anderson, "CGRA-ME: A unified framework for CGRA modelling and exploration," in *International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2017.

[7] M. Odersky, L. Spoon, and B. Venners, "Programming in Scala: Updated for Scala 2.12," 2016.

[8] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a Scala embedded language," in *Design Automation Conference (DAC)*, 2012.

[9] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization (CGO)*, 2004.

[10] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "DRESC: A retargetable compiler for coarse-grained reconfigurable architectures," in *International Conference on Field-Programmable Technology (FPT)*, 2002.

[11] W. Snyder, "Verilator and SystemPerl," in *North American SystemC Users' Group (NASCUG) Meeting at Design Automation Conference*, 2004.

[12] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, "The rocket chip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[13] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2003.

[14] 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.

[15] S. A. Chin and J. H. Anderson, "An architecture-agnostic integer linear programming approach to CGRA mapping," in *Design Automation Conference (DAC)*, 2018.

[16] S. A. Chin, K. P. Niu, M. Walker, S. Yin, A. Mertens, J. Lee, and J. H. Anderson, "Architecture exploration of standard-cell and FPGA-overlay CGRAs using the open-source CGRA-ME framework," in *International Symposium on Physical Design (ISPD)*, 2018.