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
OUTLINE

- Background
- Introduction of Pillars
- Experimental study
- Q & A
What is coarse-grained reconfigurable array (CGRA)?

- The advantages of CGRA
  - Faster reconfiguration process due to word-level granularity.
  - A more effective tradeoff between efficiency and flexibility than FPGAs and custom ASICs.
  - The capability for spatial, temporal and parallel computation.
The motivation of Pillars

- CGRA design and exploration tools remain in an embryonic period. Two core requirements have not been completely satisfied:
  - Iterative optimization of hardware implementation.
  - Design space exploration.

- The main capabilities of existing frameworks and tools
  - CCF [3] can simulate acceleration of loops of general-purpose applications on CGRA.
  - CGRA-ME [6] permits the modeling and exploration of a wide variety of CGRA architectures, and also promote research on CGRA mapping algorithms.

- Key features of Pillars
  - Integration, flexibility and consistency.

- Open-source repository: https://github.com/pku-dasys/pillars
OUTLINE

- Background
- Introduction of Pillars
- Experimental study
- Q & A
The overview of Pillars

Mapping and scheduling

Mapping Results

Hardware generation

Regions

RTL-level simulation

Architecture Evaluation

1. Scala-based Architecture Description

2. Architecture Interpreter

3. Hierarchy Abstract Model

4. Flatten Abstract Model

5. Basic Chisel Modules

9. Auxiliary Modules Auto-generation

10. Chisel Top Design

11. Verilog RTL

12. Simulator Code Generation

13. Verilator

14. FPGA Synthesis, Place and Route

15. FPGA-overlay Performance, Area and Power Consumption

6. Data-Flow Graph

7. Mapper & Scheduler

8. Modulo Routing Resource Graph

16. Performance, Area and Power Estimation for Benchmarks

Contexts

Mappability, Throughput and Runtime

Description and abstraction of an architecture
Architecture description language (ADL)

- Hierarchy description and flattened implementations.

- Basic components:
  - Block: representing the design hierarchy of an architecture
  - Element: representing abstraction of Chisel hardware implementation.

- Predefined element: multiplexer, const unit, arithmetic logical unit (ALU), load/store unit (LSU) and register files (RF).

```scala
class BlockImmediate(name: String) extends BlockTrait {
  def addInPorts(ports: Array[String])
  def addOutPorts(ports: Array[String])

  // A multiplexer that can choose a data source
  def addInPorts(ports: Array[String], inputA: String, inputB: String)
  def addOutPorts(output: String)

  // An ALU that can perform some operations.
  def addInPorts(ports: Array[String], inputA: String, inputB: String)
  def addOutPorts(ports: Array[String], output: String)
  def addElement(element: String)

  // A const unit connected to the port "input0" of ALU.
  val const0 = new Const(value: value, constParams)
  const0.addOutPorts(Array("output0"))
  const0.addElement("const0")

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

  // Interconnection inside this block.
  addConnect(term("input0") -> mux0 / "input0")
  addConnect(term("input1") -> mux0 / "input1")
  addConnect(mux0 / "output0" -> alu0 / "inputA")
  addConnect(const0 / "output0" -> alu0 / "inputB")
  addConnect(term("input0") -> subBlock / "input")
  addConnect(alu0 / "output0" -> subBlock / "output")
  addConnect(subBlock / "output" -> term("output0"))
}
```
Hardware generation

- Explicit modules
  - Each explicit module corresponds to an element in the ADL.
  - 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.

- Auxiliary modules
  - Auxiliary modules assist the explicit modules to perform functions correctly.
  - Configuration controllers
    - Repeating stored configurations every initiation interval (II) cycles and distribute them to corresponding explicit modules.
  - Schedule controllers
    - Controlling the cycle modules should fire.
  - Synchronizers
    - Implementing synchronous inputs for explicit modules with more than one input ports.
Terminologies - data flow graph (DFG)

- Functional graph: data flow graph (DFG)
  - DFG: a graph representing the computational data flow
  - opNode: a node in DFG representing a computational operation
  - valNode: a node in DFG representing the data
  - DFG edge: for example, in \( c = a + b \), there are three edges connecting valNodes \( (a, b, c) \) and opNode \( (+) \)
Terminologies - modulo routing resource graph (MRRG)

- Physical graph: modulo routing resource graph (MRRG)
  - MRRG: a graph representing the hardware resources and structure
    - It is a "3D" hardware resource graph extended in the time domain
  - funcNode: a node in MRRG that implements an opNode in DFG
  - routingNode: a node in MRRG that transfers a valNode in DFG from a funcNode to another funcNode
  - MRRG edge: connecting funcNodes and routingNodes
Mapper & scheduler

- **Target:** producing contexts that guide reconfigurable modules to perform correct behavior during reconfiguration.

- **Mapper**
  - **Goal:** mapping every opNode in DFG to a specific funcNode in MRRG, and mapping every valNode in DFG to a connected sequence of routingNodes in MRRG.
  - **Method:** integer linear programming (ILP) solver based on [15].

- **Scheduler**
  - **Goal:** determining the fire time and synchronization method of each operator.
  - **Method:** topological search.
RTL-level simulation

- **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.

- **Activating process**
  - Explicit modules can perform routing or operations set by configuration controllers, if they have been fired by schedule controllers.

- **Post-process**
  - The output data stream can be obtained from LSUs.

```scala
/** 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.
Project Tree

```
.
  ├── app-mapping-results  // some pre-generated mapping results
  │ ├── build.sbt  // the library dependencies in sbt
  │ ├── doc  // documents of APIs in Pillars
  │ ├── dfg  // some DFGs in DOT format
  │ ├── fig  // some figures in README
  │ │ └── Makefile  // some MRRGs
  │ │ └── README.md  // the scalastyle file
  │ │ └── scalastyle-config.xml  // the scalastyle file
  │ │ └── scalastyle-test-config.xml  // the scalastyle file
  │ └── src
       └── main
            └── scala
                 └── tetriski
                      └── pillars
                           └── archlib  // the library of elements and blocks
                           └── core  // the core of Pillars
                           └── examples  // some examples showing how to use Pillars
                           └── hardware  // hardware implemented in Chisel
                           └── mapping  // mapping tools
                           └── Pillars.scala
                           └── testers  // testers in Pillars
                           └── util  // utility for realizing hardware
```
OUTLINE

- Background
- Introduction of Pillars
- Experimental study
- Q & A
Experimental architectures - architecture skeleton

operations: add, subtract, multiply, shifts, and, or and xor

operations: add, subtract

PE with full set of operations

PE with reduced operations

Block with a load/store unit

Storage component

Fig. 4: Variants of ADRES architecture skeleton.
Experimental architectures - processing element (PE)

- Simple PE
  - 2 multiplexers for selecting input data
  - 1 ALU for performing computation
  - 1 const unit for storing const value
  - 1 RF with 2 registers for temporarily holding arithmetic results
Experimental architectures - processing element (PE)

- **Complex PE**
  - 2 multiplexers for selecting input data
  - 1 ALU for performing computation
  - 1 const unit for storing const value
  - 1 RF with 2 registers for temporarily holding arithmetic results
  - 2 additional multiplexers for bypass
Target: Xilinx ZYNQ-7000 ZC706 evaluation board

Platform: Vivado 2019.2

We interleave two neighboring PE columns.
The blocks with LSU are located at the center.

Fig. 5: Floorplan for Vivado place & route.

Floorplan & layout

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

Fig. 7: Layout of selected FPGA implementations.
Physical implementation & mapping results

- Maximum frequency
- FPGA area breakdown of the implementations
- Success rate of mapping within 7200 seconds for benchmarks in [16]
- Explicit modules (on the left of “/”) and auxiliary modules (on the right)

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

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

- Background
- Introduction of Pillars
- Experimental study
- Q & A
"Can there be several connected ALUs in one PE?"
- Of course yes, this is one of the core strengths of Pillars.
- Users can design their PE blocks with arbitrary connection and hierarchy.

"What is the overhead of using Pillars?"
- The auxiliary modules may introduce overhead.
- The configuration controllers are necessary for reconfiguration, and have fixed and slight overhead according to the architecture.
- Users can determine the scale or abandoning of schedule controllers and synchronizers in a global config file. So the overhead of them can be controlled.
- Researches about a better tradeoff between the overhead and function of auxiliary modules are in our schedule.
Thanks for listening!