An Architecture-Independent CGRA Compiler enabling OpenMP Applications

Takuya Kojima†, Carlos Cesar Cortes Torres‡, Boma Adhi‡, Yiyu Tan‡, Kentaro Sano‡
† The University of Tokyo, Japan
‡ RIKEN, Japan
Coarse-grained reconfigurable architectures

A data flow graph as an application

General structure of the CGRAs

- Coarse-Grained Reconfigurable Architecture (CGRA)
  - Composed of an array of Processing Elements (PEs)
  - Providing a word-level reconfigurability (e.g., 32-bit)
    - Smaller energy-overhead than FPGAs (bit-level)
  - Generally used as an accelerator

Comparison with other architectures[2]

Purpose & Proposal

- Trend in CGRA research: design space exploration framework
  - Highly customizability
  - Possibilities of domain-specific architecture

- Challenge
  - No general-purpose & architecture-independent compiler frontend for CGRAs
  - Needs of abstracting hardware layer for software programmers

- Our proposal
  - A compiler framework enabling OpenMP offloading to CGRAs
  - A case study: implementation for RIKEN CGRA
Wide Variety of Design Choices

- Characteristics of CGRA Design
  - Reconfiguration style
  - PE array size
  - Interconnection topology
  - Operational capabilities in PE
  - Ability to handle control flows
Wide Variety of Design Choices

- Characteristics of CGRA Design
  - Reconfiguration style
    - Time-Multiplexing manner
    - Spatial one
  - PE array size
  - Interconnection topology
  - Operational capabilities in PE
  - Ability to handle control flows
Wide Variety of Design Choices

- Characteristics of CGRA Design
  - Reconfiguration style
  - PE array size
    - Ranges 10-10^4
  - Interconnection topology
  - Operational capabilities in PE
  - Ability to handle control flows

![Distribution of the PE array size](image-url)
Wide Variety of Design Choices

- Characteristics of CGRA Design
  - Reconfiguration style
  - PE array size
  - Interconnection topology
    - Mesh
    - Meshplus
    - Torus, etc
  - Operational capabilities in PE
  - Ability to handle control flows
Wide Variety of Design Choices

- Characteristics of CGRA Design
  - Reconfiguration style
  - PE array size
  - Interconnection topology
  - Operational capabilities in PE
    - Bit-width
    - Floating point
    - SIMD
    - Custom instruction (e.g., ReLU, sigmoid for ML)
  - Ability to handle control flows

32bit? 64bit? int? float? fixed?

Data to neighbors/Data memory

32bit? 64bit? int? float? fixed?

Data from neighbors/Data memory

Output Register

Register file

ALU

SEL

SEL
Wide Variety of Design Choices

- Characteristics of CGRA Design
  - Reconfiguration style
  - PE array size
  - Interconnection topology
  - Operational capabilities in PE
  - Ability to handle control flows
    - Conditionals are supported or not?
    - Loop-carried dependence is allowed or not?

Loop containing conditional parts
DFG with Partial Predication
Partial predication on CGRAs
A case of CGRA: RIKEN CGRA [5]

- Design template for design space exploration
  - Implemented with SystemVerilog
- Two types of tiles
  1. LS (Load/Store)
     - Data access according to loop control info.
  2. PE
     - Computation
- Reconfiguration style: Spatial
  - FIFO buffers allow operands to arrive at different times
Limitations of Existing Compilers
CGRA vs FPGA in compilation flow

For FPGAs:
- C/C++ etc. → High-Level Synthesis
  - Logic Synthesis
    - Gate level netlist
    - Technology mapping
      - LUT level netlist
    - Clustering
      - LB level netlist
    - Place&Route
      - Configuration data

For CGRAs:
- C/C++ etc. → HDL
  - DFG extraction
    - Loop kernel
  - Mapping
    - Place&Route (+Scheduling)
    - Configuration data

Coarse granularity mitigates compilation complexity

Sequential part for CPU

For FPGAs:
- Gate level netlist

For CGRAs:
- Configuration data

The First International Workshop on Coarse-Grained Reconfigurable Architectures for High-Performance Computing (CGRA4HPC)
# Existing compilers for CGRAs

<table>
<thead>
<tr>
<th>Methods</th>
<th>Frontend</th>
<th>Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Musketeer (STP Tool) [9]</td>
<td>C</td>
<td>Renesas STP (DRP)</td>
</tr>
<tr>
<td>PipeRench Compiler [10]</td>
<td>C-like DSL (DIL)</td>
<td>PipeRench</td>
</tr>
<tr>
<td>SambaFlow™ [12]</td>
<td>PyTorch, TensorFlow, etc</td>
<td>SambaNova RDA</td>
</tr>
<tr>
<td>BlackDiamond [13]</td>
<td>C-like DSL</td>
<td>Parameterized</td>
</tr>
<tr>
<td>CCF [14]</td>
<td>C (pragma)</td>
<td>ADRES[16]-like</td>
</tr>
<tr>
<td>Kim, Hee-Seok, et al [15]</td>
<td>OpenCL</td>
<td>SRP</td>
</tr>
<tr>
<td>MENTAI [16]</td>
<td>C</td>
<td>Cool Mega Array</td>
</tr>
<tr>
<td>CGRA-ME [3]</td>
<td>C</td>
<td>Parameterized</td>
</tr>
<tr>
<td>DSAGEN [6]</td>
<td>C (pragma)</td>
<td>Parameterized</td>
</tr>
</tbody>
</table>

**Commercial products**

TM: Time-Multiplexing, SP: Spatial
### Existing compilers for CGRAs

<table>
<thead>
<tr>
<th>Methods</th>
<th>Frontend</th>
<th>Targets</th>
<th>Reconf.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Musketeer (STP Tool) [9]</td>
<td>C</td>
<td>Renesas STP (DRP)</td>
<td>TM</td>
</tr>
<tr>
<td>PipeRench Compiler [10]</td>
<td>C-like DSL (DIL)</td>
<td>PipeRench</td>
<td>SP (TM)</td>
</tr>
<tr>
<td>SambaFlow™ [12]</td>
<td>PyTorch, TensorFlow, etc</td>
<td>SambaNova RDA</td>
<td>SP</td>
</tr>
<tr>
<td>BlackDiamond [13]</td>
<td>C-like DSL</td>
<td>Parameterized</td>
<td>TM/SP</td>
</tr>
<tr>
<td>CCF [14]</td>
<td>C (pragma)</td>
<td>ADRES[16]-like</td>
<td>TM</td>
</tr>
<tr>
<td>MENTAI [16]</td>
<td>C</td>
<td>Cool Mega Array</td>
<td>SP</td>
</tr>
<tr>
<td>CGRA-ME [3]</td>
<td>C</td>
<td>Parameterized</td>
<td>TM/SP</td>
</tr>
<tr>
<td>DSAGEN [6]</td>
<td>C (pragma)</td>
<td>Parameterized</td>
<td>SP</td>
</tr>
<tr>
<td><strong>This work</strong></td>
<td><strong>OpenMP</strong></td>
<td><strong>Parameterized</strong></td>
<td><strong>TM/SP</strong></td>
</tr>
</tbody>
</table>

**Commercial products**

TM: Time-Multiplexing, SP: Spatial
## Existing compilers for CGRAs

<table>
<thead>
<tr>
<th>Methods</th>
<th>Frontend</th>
<th>Targets</th>
<th>Reconf.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Musketeer (STP Tool) [9]</td>
<td>C</td>
<td>Renesas STP (DRP)</td>
<td>TM</td>
</tr>
<tr>
<td>PipeRench Compiler [10]</td>
<td>C-like DSL (DIL)</td>
<td>PipeRench</td>
<td>SP (TM)</td>
</tr>
<tr>
<td>SambaFlow™ [12]</td>
<td>PyTorch, TensorFlow, etc</td>
<td>SambaNova RDA</td>
<td>SP</td>
</tr>
<tr>
<td>DSAGEN [6]</td>
<td>C (pragma)</td>
<td>Parameterized</td>
<td>SP</td>
</tr>
<tr>
<td>This work</td>
<td>OpenMP</td>
<td>Parameterized</td>
<td>TM/SP</td>
</tr>
</tbody>
</table>

**Design space exploration**
Fair comparison between various types of CGRAs

**Reuse of source codes**
Minimizing efforts to modify the codes

**Easy to compare other architectures**
Comparing to GPU, many-core CPU with the same kernel

---

TM: Time-Multiplexing,  SP: Spatial
Our Proposal: CGRA OpenMP
Target directive

- Accelerator offloading features
  - Added since OpenMP 4.0
  - Similar concept to OpenACC
  - Mainly supporting GPU offloading
  - Explicit data transfer between hosts and device (map clause)

```c
#pragma omp target map(to: v1, v2) map(from: p)
#pragma omp parallel for private(i)
for (i = 0; i < N; i++) {
    p[i] = v1[i] * v2[i];
}
```

Code snippet with target directive [17]
Implementation based on LLVM

- LLVM: An open-source compiler framework
  - LLVM-IR: target-independent intermediate representation
  - Common optimization and analysis algorithms (Pass)
  - Official sub-projects
    - C frontend Clang, Fortran frontend Flang, OpenMP, etc

C/C++ codes \(\rightarrow\) clang \(\rightarrow\) LLVM IR \(\rightarrow\) Pass 1, Pass 2, ..., Pass N \(\rightarrow\) Optimized IR \(\rightarrow\) Backend Pass

Fortran codes \(\rightarrow\) flang

Backend Pass

\(\rightarrow\) X86 asm
\(\rightarrow\) ARM asm
\(\rightarrow\) RISC-V asm

The First International Workshop on Coarse-Grained Reconfigurable Architectures for High-Performance Computing (CGRA4HPC)
Compilation flow in CGRA OpenMP

- Compiler-driver can automate the compilation
Compilation flow in CGRA OpenMP

- Compiler-driver can automate the compilation

(1) Dividing into host and device (CGRA) codes

- clang-offload-bundler
  - A utility tool for heterogeneous single source programming languages.

The First International Workshop on Coarse-Grained Reconfigurable Architectures for High-Performance Computing (CGRA4HPC)
Compilation flow in CGRA OpenMP

- Compiler-driver can automate the compilation

- CGRAOmpPass
  - Code verification
  - DFG extraction
  - Runtime insertion

The First International Workshop on Coarse-Grained Reconfigurable Architectures for High-Performance Computing (CGRA4HPC)
Compilation flow in CGRA OpenMP

- Compiler-driver can automate the compilation

```
OpenMP Code → clang → LLVM IR → Unbundle → Host LLVM IR → opt → Bundle → Control IR → opt → CGRA LLVM IR → CGRAOmp Pass → CGRA Model → Mapping → Data Flow Graph → Generate Executable for host → Host Object → Linker
```

- Template for runtime routine
  - Data transfer
  - Configuration
Compilation flow in CGRA OpenMP

- Compiler-driver can automate the compilation

Diagram:

1. OpenMP Code → clang → LLVM IR
2. LLVM IR → Unbundle → Host LLVM IR
3. CGRA LLVM IR → opt → CGRAOmp Pass
4. CGRAOmp Pass → opt → Control IR
5. Control IR → Bundle
6. Bundle → CGRA Runtime Lib → linker
7. Host LLVM IR → opt
8. CGRA Model → Mapping
9. Data Flow Graph → (4) Mapping (PnR)

- Generated DFG independent of mapping algorithm
- DOT format
CGRA Model Description
The model defines CGRA execution style, etc (JSON)

```json
{
  "category": "decoupled",
  "address_generator": {
    "control": "affine",
    "max_nested_level": 3
  },
  "conditional": {
    "allowed": false
  },
  "inter-loop-dependency": {
    "allowed": false
  },
  "custom_instructions": ["fexp", "fsin", "fcos"],
  "generic_instructions": ["add", "sub", "mul", "udiv", "sdiv", "and", "or", "xor", "fadd", "fsub", "fmul", "fdiv"],
  "instruction_map": [
    {"inst": "xor", "rhs": {"ConstantInt": -1}, "map": "not"},
    {"inst": "xor", "map": "xor"}
  ]
}
```

An example: the case of RIKEN CGRA
The model defines CGRA execution style, etc (JSON)

```json
{
  "category": "decoupled",
  "address_generator": {
    "control": "affine",
    "max_nested_level": 3
  },
  "conditional": {
    "allowed": false
  },
  "inter-loop-dependency": {
    "allowed": false
  },
  "custom_instructions": [ "fexp", "fsin", "fcos" ],
  "generic_instructions": [ "add", "sub", "mul", "udiv", "sdiv", "and", "or", "xor", "fadd", "fsub", "fmul", "fdiv" ],
  "instruction_map": [
    { "inst": "xor", "rhs": {"ConstantInt": -1}, "map": "not" },
    { "inst": "xor", "map": "xor" }
  ]
}
```

Classification of CGRAs

Decoupled

An execution model decoupling memory access and computation [6]

An example: the case of RIKEN CGRA
The model defines CGRA execution style, etc (JSON)

```json
{
  "category": "decoupled",
  "address_generator": {
    "control": "affine",
    "max_nested_level": 3
  },
  "conditional": {
    "allowed": false
  },
  "inter-loop-dependency": {
    "allowed": false
  },
  "custom_instructions": ["fexp", "fsin", "fcos"],
  "generic_instructions": ["add", "sub", "mul", "udiv", "sdiv", "and", "or", "xor", "fadd", "fsub", "fmul", "fdiv"],
  "instruction_map": [
    {"inst": "xor", "rhs": {"ConstantInt": -1}, "map": "not"},
    {"inst": "xor", "map": "xor"}
  ]
}
```

- Ability to memory access control
  - Only affine access is allowed
  - Up-to 3-nested loops
  - i.e., $C_0 + C_1v_1 + C_2v_2 + C_3v_3$

An example: the case of RIKEN CGRA
CGRA model description

- The model defines CGRA execution style, etc (JSON)

```json
{
  "category": "decoupled",
  "address_generator": {
    "control": "affine",
    "max_nested_level": 3
  },
  "conditional": {
    "allowed": false
  },
  "inter-loop-dependency": {
    "allowed": false
  },
  "custom_instructions": ["fexp", "fsin", "fcos"],
  "generic_instructions": ["add", "sub", "mul", "udiv", "sdiv", "and", "or", "xor", "fadd", "fsub", "fmul", "fdiv"],
  "instruction_map": [
    {"inst": "xor", "map": "not"},
    {"inst": "xor", "rhs": {"ConstantInt": -1}, "map": "not"},
    {"inst": "xor", "map": "xor"}
  ]
}
```

- Ability to handle control flow
- In this example of the CGRA
- Both conditional and loop-carried dependencies are not supported

An example: the case of RIKEN CGRA
The model defines CGRA execution style, etc (JSON)

```json
{
    "category": "decoupled",
    "address_generator": {
        "control": "affine",
        "max_nested_level": 3
    },
    "conditional": {
        "allowed": false
    },
    "inter-loop-dependency": {
        "allowed": false
    },
    "custom_instructions": ["fexp", "fsin", "fcos"],
    "generic_instructions": ["add", "sub", "mul", "udiv", "sdiv", "and", "or", "xor", "fadd", "fsup", "fmul", "fdiv"],
    "instruction_map": [
        {
            "inst": "xor", "rhs": {"ConstantInt": -1}, "map": "not"
        },
        {
            "inst": "xor", "map": "xor"
        }
    ]
}
```

An example: the case of RIKEN CGRA

- What kind of instructions are supported in ALU
  - `custom_instructions`: instructions not in LLVM-IR
    - The Same function name should be used in codes
  - `generic_instructions`: Corresponding LLVM IR instructions
  - `instruction_map`: mapping LLVM IR instr. to ALU opcode
    - Some mapping conditions are available
CGRA model description

- The model defines CGRA execution style, etc (JSON)

```json
{
  "category": "decoupled",
  "address_generator": {
    "control": "affine",
    "max_nested_level": 3
  },
  "conditional": {
    "allowed": false
  },
  "inter-loop-dependency": {
    "allowed": false
  },
  "custom_instructions": ["fexp", "fsin", "fcos"],
  "generic_instructions": ["add", "sub", "mul", "udiv", "sdiv", "and", "or", "xor", "fadd", "fsub", "fmul", "fdiv"],
  "instruction_map": [
    {
      "inst": "xor",
      "rhs": {
        "ConstantInt": -1
      },
      "map": "not"
    },
    {
      "inst": "xor",
      "map": "xor"
    }
  ]
}
```

- What kind of instructions are supported in ALU
  - `custom_instructions`: instructions not in LLVM-IR
    - The Same function name should be used in codes
  - `generic_instructions`: Corresponding LLVM IR instructions
  - `instruction_map`: mapping LLVM IR instr. to ALU opcode
    - Some mapping conditions are available

An example: the case of RIKEN CGRA

```c
CGRAOMP_CUSTOM_INST float FMA(float x, float y, float z) {
  return x * y + z;
}
```

function declaration in source codes for the custom instruction
Flow of CGRAOmpPass

The First International Workshop on Coarse-Grained Reconfigurable Architectures for High-Performance Computing (CGRA4HPC)
Flow of CGRAOmpPass

- **OpenMP CGRA target IR**
  - CGRA Model JSON
  - Model Manager Pass
  - Verification Pass
  - DFG Pass
  - Insert Runtime Pass

Verifies if the kernel can be executed on the target CGRA:
- Compatibility of operations
- Memory access pattern
- Loop structure, etc.
Flow of CGRAOmpPass

- The First International Workshop on Coarse-Grained Reconfigurable Architectures for High-Performance Computing (CGRA4HPC)
Code example: 3x3 convolution

- convolution-2d.c from PolyBench-ACC

```c
#pragma omp target parallel for private(i,j) map(to:A[:,0:]) map(from:B[:,0:])
for (i = 1; i < _PB_NI - 1; ++i)
{
    for (j = 1; j < _PB_NJ - 1; ++j)
    {
        B[i][j] = 0.2f * A[i-1][j-1] + 0.5f * A[i-1][j] + -0.8f * A[i-1][j+1] + -0.3f * A[i][j-1] + 0.6f * A[i][j] + -0.9f * A[i][j+1] + 0.4f * A[i+1][j-1] + 0.7f * A[i+1][j] + 0.1f * A[i+1][j+1];
    }
}
```

← Only this pragma is inserted
Demonstration of the compiler driver

bash $ cgraomp-cc convolution-2d.c --cgra-config=presets/decoupled_affine_AG.json --save-temps --enable-cgraomp-debug -Xclang="-I../..//utilities"

Clang front-end : [ OK ]
OpenMP target unbundling : [ OK ]
Optimization of host code : [ OK ]
1th Pre-Optimization of CGRA kernel code : [ OK ]
Verify kernel, extract DFG, and insert runtime : [ OK ]

[INFO]: Start verification
[INFO]: Instantiating CGRModel
[INFO]: Searching for OpenMP kernels
[INFO]: Found offloading function: __omp_offloading_fd04_24208e4_kernel_conv2d_l98
[INFO]: Verifying a kernel for decoupled CGRA: .omp_outlined.
[INFO]: Detected perfectly nested loop in 2 nested loop kernel: for.body Nested level 1
[INFO]: Verifying Affine AG compatibility of a loop: for.body
[INFO]: Saving DFG: ./convolution-2d__.omp_outlined._for.body.dot

bash $
DFG Optimization after extraction
Generated DFG

- Data dependencies in LLVM-IR cause unbalanced DFG
DFG-level optimization: Tree-Height-Reduction

- An important optimization for LSI design and High-level synthesis [18]
  - Graph transformation based on commutativity & associativity of operators
    - e.g., addition (+), multiplication (*)
  - This work integrates Huffman code-based algorithm [19] as a built-in pass
Applying Tree-Height-Reduction

- Easy to custom pass pipeline for DFG optimization

```bash
Clang front-end : [ OK ]
OpenMP target unbundling : [ OK ]
Optimization of host code : [ OK ]
1th Pre-Optimization of CGRA kernel code : [ OK ]
Verify kernel, extract DFG, and insert runtime : [ OK ]
[INFO]: Start verification
[INFO]: Instantiating CGRAModel
[INFO]: Searching for OpenMP kernels
[INFO]: Found offloading function: __omp_offloading_fd04_24208e4_kernel_conv2d_l98
[INFO]: Verifying a Kernel for decoupled CGRA: .omp_outlined.
[INFO]: Detected perfectly nested loop in 2 nested loop kernel: for.body Nested level 1
[INFO]: Verifying Affine AG compatibility of a loop: for.body
[INFO]: applying CGRAOmp::BalanceTree
[INFO]: Saving DFG: ./convolution-2d_.omp_outlined._for.body.dot
```
DFG after optimization

Computational node
Constant nodes
Mem. access nodes
DFG Pass Plugin

- Easy to create and enable your own custom pass in similar manner in LLVM

```c
bool HelloDFGPass::run(CGRADFG &G, Loop &L, FunctionAnalysisManager &FAM,
                        LoopAnalysisManager &LAM,
                        LoopStandardAnalysisResults &LAR)
{
    llvm::errs() << "My DFG Pass is called: Hello World\n";
    return false;
}

extern "C" ::=CGRAOMP::::DFGPassPluginLibraryInfo getDFGPassPluginInfo()
{
    return { "A sample of DFG Pass",
            [](DFGPassBuilder &GP)
               {
                   PB.RegisterPipelineParsingCallback(
                       [](StringRef Name, DFGPassManager &PM)
                           {
                               if (Name == "hello"){
                                   PM.addPass(HelloDFGPass());
                                   return true;
                               }
                           });
               };
    return false;
}
```

Pass function

Call back function
Evaluation
Experimental setup

- LLVM version 12.0.1
- CGRA design
  - RIKEN CGRA
  - 8x10 array (8x8 PE tiles + 8+8LS tiles)
- Benchmark: 3x3 convolution
- Backend (mapping algorithm)
  - GenMap[20] currently supports RIKEN CGRA
  - Genetic algorithm-based mapping

GenMap

Application Mapping Framework for spatially mapping CGRAs using Genetic Algorithm

Python

https://github.com/hungalab/GenMap
### Mapping results

<table>
<thead>
<tr>
<th>priority</th>
<th>Total wire length</th>
<th>Map area</th>
<th>Latency diff.</th>
</tr>
</thead>
<tbody>
<tr>
<td>naïve DFG area</td>
<td>40.38</td>
<td>40 (5 x 8)</td>
<td>6</td>
</tr>
<tr>
<td>naïve DFG latency balance</td>
<td>50.56</td>
<td>56 (7 x 8)</td>
<td>2</td>
</tr>
<tr>
<td>Optimized DFG</td>
<td>46.21</td>
<td>40 (5 x 8)</td>
<td>0</td>
</tr>
</tbody>
</table>

**Diagram:**
- ** naïve DFG 5x8 mapping**
- **Optimized DFG 5x8 mapping**

- Longer lat. 3
- Shorter lat. 2
- Diff.: 3-2

The First International Workshop on Coarse-Grained Reconfigurable Architectures for High-Performance Computing (CGRA4HPC)
Conclusion & Future work

This work
- Proposes a CGRA compiler designated to handle the same source code regardless of the target architecture
- Uses OpenMP offloading

Future work
- To extend verification and analysis for other types of CGRAs
- To implement runtime insertion
- To make it work together with CGRA simulators or FPGA overlays
References


References


References


