Automatic Compilation for Domain Specific Accelerators

Ross Daly
Caleb Donovick
Jackson Melchert
Golden Age of Computer Architecture!
Golden Age of Computer Architecture!

- Architecture Specifications change frequently
Golden Age of Computer Architecture!

• Architecture Specifications change frequently
• Compiler is the (often overlooked) key component!
Golden Age of Computer Architecture!

- Architecture Specifications change frequently
- Compiler is the (often overlooked) key component!
- Waterfall methodology:
Golden Age of Computer Architecture!

- Architecture Specifications change frequently
- Compiler is the (often overlooked) key component!
- Agile methodology:
Golden Age of Computer Architecture!

- Architecture Specifications change frequently
- Compiler is the (often overlooked) key component!
- Agile methodology:
  - Automatically generate compiler for every spec change
**CGRA/FPGA**

- Compile to IR (CoreIR)
- Common Optimizations
- Mapping
- Packing
- Placement
- Routing
- Bitfile generation

**CPU**

- Compile to IR (LLVM)
- Common Optimizations
- Instruction Selection
- Peephole Optimization
- Instruction Scheduling
- Register Allocation
- Assembly
CGRA/FPGA

- Compile to IR (CoreIR)
- Common Optimizations
- Mapping
- Packing
- Placement
- Routing
- Bitfile generation

CPU

- Compile to IR (LLVM)
- Common Optimizations
- Instruction Selection
- Peephole Optimization
- Instruction Scheduling
- Register Allocation
- Assembly
CGRA Mapping

- Application Halide Program
- Lower
- CoreIR Graph
- Map PE and Memory
- Mapped CoreIR Graph
- CGRA Bitstream
Our DSL-based Hardware Generation and Software Compilation Flow

PEak Program (PE spec)

PEak Compiler

PE HW in Magma

Magma Compiler

CGRA Verilog

Compiler Collateral

Halide Compiler

CoreIR Graph

PE and MEM Mapper

Mapped CoreIR Graph

Place & Route Engine

CGRA Bitstream
Our DSL-based Hardware Generation and Software Compilation Flow

PEak Program (PE spec) → PEak Compiler → PE HW in Magma

Lake Program (MEM spec) → Lake Compiler → MEM HW in Magma

Magma Compiler → CGRA Verilog

Compiler Collateral

Application Halide Program

Halide Compiler → CoreIR Graph

PE and MEM Mapper → Mapped CoreIR Graph

Place & Route Engine → CGRA Bitstream
Output of Halide Compiler

CoreIR Graph

From Global Buffer

Unified Buffer \rightarrow Computation Kernel \rightarrow Unified Buffer \rightarrow Computation Kernel

To Global Buffer
Desired Output of Mapper

From Global Buffer

Lake-Specified Mem Tile

PEak-Specified PE Tile

To Global Buffer

Mapped CoreIR Graph
Kernels are composed of CoreIR Primitives

From Buffer/IO: add, mul, sub, ashr

Computational Kernel: add, mul, sub, div

To Buffer/IO: add, mul, sub, ashr, div
CoreIR has SMT QF BitVector Semantics

\[ \text{Out} = \text{In0} - \text{In1} \]
Mapping

Kernel

Mapped Kernel

CoreIR
Primitives

PEak-Specified
PE Tile
Instruction Selection

Rewrite Rule Table

- Rewrite Rule 1
  - CoreIR: add
  - PEak-Specified: ...

- Rewrite Rule 2
  - CoreIR: add
  - PEak-Specified: ...

- Rewrite Rule 3
  - CoreIR: div
  - PEak-Specified: ...

- Rewrite Rule 4
  - CoreIR: ashr
  - PEak-Specified: ...

...
Instruction Selection

Rewrite Rule Table

<table>
<thead>
<tr>
<th>Rule</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub</td>
<td>1.2</td>
</tr>
<tr>
<td>mul, add</td>
<td>3.1</td>
</tr>
<tr>
<td>div</td>
<td>6.0</td>
</tr>
<tr>
<td>shr, add</td>
<td>4.3</td>
</tr>
</tbody>
</table>

CoreIR Primitives

PEak-Specified PE Tile

Kernel

Mapped Kernel

Instruction Selection Algorithm
Peak Compiler generates a table of Rewrite Rules
PE ISA Specification

PE Functional Specification
PE Functional Specification

class PE(Peak):
    def __call__(self, inst: Const(Instruction), A: Word, B: Word, C: Word) -> {"res":Word, "flag":Bit}:
PE Functional Specification

class PE(Peak):
    def __call__(self, inst: Const(Instruction), A: Word, B: Word, C: Word) -> {"res":Word, "flag":Bit}:
        if inst.invert_A:
            A = ~A
        if inst.op == Opcode.Add:
            res, c_out = A.add(B, inst.c_in)
            flag = c_out
        elif inst.op == Opcode.Mul:
            res = A * B
            flag = (res == 0)
        return res, flag

Specific types (or composition of types) for operands and instructions
PEak: PE DSL

PE ISA Specification

```python
class Opcode(Enum):
    Add = 0
    Mul = 1
...
# Define Instruction
class Instruction(Product):
    op = Opcode
    invert_A = Bit
    c_in = Bit
```

PE Functional Specification

```python
class PE(Peak):
    def __call__(self, inst: Const(Instruction), A: Word, B: Word, C: Word) -> {'res':Word, 'flag':Bit}:
```

Specific **types** (or composition of types) for operands and instructions
class Opcode(Enum):
    Add = 0
    Mul = 1
...

# Define Instruction
class Instruction(Product):
    op = Opcode
    invert_A = Bit
    c_in = Bit

# Define Word
Word = UnsignedBitVector[16]

Specific types (or composition of types) for operands and instructions
PEak: PE DSL

PE ISA Specification

class Opcode(Enum):
    Add = 0
    Mul = 1
    ...

# Define Instruction
class Instruction(Product):
    op = Opcode
    invert_A = Bit
    c_in = Bit

# Define Word
Word = UnsignedBitVector[16]

Specific *types* (or composition of types) for operands and instructions

PE Functional Specification

class PE(Peak):
    def __call__(self, inst: Const(Instruction), A: Word, B: Word, C: Word) -> {"res":Word, "flag":Bit}:
        if inst.invert_A:
            A = ~A

        if inst.op == Opcode.Add:
            res, c_out = A.add(B, inst.c_in)
            flag = c_out
        elif inst.op == Opcode.Mul:
            res = A * B
            flag = (res == 0)
        elif ...
        ...
        return res, flag
Subtract?

PE Functional Specification

class PE(Peak):
    def __call__(self, inst: Instruction, A: Word, B: Word, C: Word) -> {"res":Word, "flag":Bit}:
        if inst.invert_A:
            A = ~A

        if inst.op == Opcode.Add:
            res, c_out = A.add(B, inst.c_in)
            flag = c_out
        elif inst.op == Opcode.Mul:
            res = A * B
            flag = (res == 0)
        elif ...
        ...
        return res, flag
class PE(Peak):
    def __call__(self, inst: Instruction, A: Word, B: Word, C: Word) -> {"res":Word, "flag":Bit}:
        inst = Instruction(op=Add, invert_A=1, c_in=1)

        if inst.invert_A:
            A = ~A

        if inst.op == Opcode.Add:
            res, c_out = A.add(B, inst.c_in)
            flag = c_out
        elif inst.op == Opcode.Mul:
            res = A * B
            flag = (res == 0)
        elif ... :
            ...
        return res, flag
res = \sim A + B + 1
PE Functional Specification

```python
class PE(Peak):
    def __call__(self, inst: Instruction, A: Word, B: Word, C: Word) -> {"res":Word, "flag":Bit}:
        inst = Instruction(
            op=Add, invert_A=1, c_in=1)

        if inst.invert_A:
            A = ~A

        if inst.op == Opcode.Add:
            res, c_out = A.add(B, inst.c_in)
            flag = c_out
        elif inst.op == Opcode.Mul:
            res = A * B
            flag = (res == 0)
        elif ...

        return res, flag
```

res = \(\sim A + B + 1 = B - A\)
RiscV Peak Specification

class RISCV(Peak):
    def __init__(self):
        self.rf = RegisterFile(32, Word)
        self.PC = Register(Data)
class RISCV(Peak):
    def __init__(self):
        self.rf = RegisterFile(32, Word)
        self.PC = Register(Data)

    def __call__(self, inst: Instruction) -> {'next_PC': Word}:
        # ID
        rs1_idx, rs2_idx, rd_idx, ... = decode(inst)
        rs1_val, rs2_val = self.rf.read(rs1_idx, rs2_idx)

        # EX
        ...

        # MEM
        ...

        # WB
        self.rf.write(rd_val)
# RiscV ISA Specification with Algebraic Data Types

## 32-bit RISC-V Instruction Formats

| Format        | Bit       | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---------------|-----------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Register/register | funct7  | rs2 | rs1 | funct3 | rd | opcode |
| Immediate       | imm[11:0] | rs1 | funct3 | rd | opcode |
| Upper immediate  | imm[31:12] |     |      |      |      |

- **opcode**: (7 bits): Partially specifies which of the 6 types of instruction formats.
- **funct7, and funct3**: (10 bits): These two fields, further than the opcode field, specify the operation to be performed.
- **rs1**: (6 bits): specifies the first operand (i.e., source register).
- **rs2**: (6 bits): specifies the second operand register.
- **rd**: (6 bits): specifies the destination register to which the computation result will be directed.
RiscV ISA Specification with Algebraic Data Types

```python
class Register(Product):
    funct7 = Funct7Enum
    rs2 = BitVector[5]
    rs1 = BitVector[5]
    funct3 = Funct3Enum
    rd = BitVector[5]
    opcode= Opcode

class Immediate(Product):
    ...

class UImmediate(Product):
    ...

class Store(Product):
    ...

class Branch(Product):
    ...

class Jump(Product):
    ...
```

Instruction = \textbf{Sum}[Register, Immediate, UImmediate, Store, Branch, Jump]

### 32-bit RISC-V instruction formats

| Format         | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----------------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Register/register | funct7 | rs2 | rs1 | funct3 | rd | opcode |
| Immediate       | imm[11:0] | rs1 | funct3 | rd | opcode |
| Upper immediate | imm[31:12] | rd | opcode |

- **opcode** (7 bits): Partially specifies which of the 6 types of instruction formats.
- **funct7**, and **funct3** (19 bits): These two fields, further than the opcode field, specify the operation to be performed.
- **rs1** (6 bits): Specifies, by index, the register containing first operand (i.e., source register).
- **rs2** (6 bits): Specifies the second operand register.
- **rd** (6 bits): Specifies the destination register to which the computation result will be directed.
Multiple Interpretations of PEak Specification

- PEak program uses abstract types provided by the PEak DSL such as Bit, BitVector etc.
- Each component of the PEak compiler provides a separate concrete implementation of these abstract types
- Multiple interpretations of a PEak specification in different contexts
Multiple Interpretations of PEak Specification

- PEak program uses abstract types provided by the PEak DSL such as `Bit`, `BitVector` etc.
- Each component of the PEak compiler provides a separate concrete implementation of these abstract types
- Multiple interpretations of a PEak specification in different contexts
Discovering a Rewrite Rule

CoreIR.Sub

In0  In1

Out

PE

A  B  C  inst

res  flag
Input/Output Bindings
Input/Output Bindings

CoreIR.Sub \[\text{In0} \rightarrow \text{A} \rightarrow \text{PE} \rightarrow \text{inst} \rightarrow \text{Out}\]

CoreIR.Sub \[\text{In1} \rightarrow \text{B} \rightarrow \text{PE} \rightarrow \text{inst} \rightarrow \text{Out}\]

PE \[\text{A} \rightarrow \text{res} \rightarrow \text{PE} \rightarrow \text{B} \rightarrow \text{flag} \rightarrow \text{PE} \rightarrow \text{C} \rightarrow \text{PE} \rightarrow \text{inst} \rightarrow \text{PE} \rightarrow \text{Out}\]
Input/Output Bindings

CoreIR.Sub → PE

- In0, In1 → A, B, C, inst
- Out → res, flag

Constant
Setting Constants

CoreIR.Sub

\[ \text{inst} = \text{Instruction}( \text{op=Add}, \text{invert}_A=1, \text{c_in}=1) \]
CoreIR.Sub(in0, in1) == PE(inst, input_binding(in0, in1))
\[ \exists (\text{input\_binding}, \text{inst}) \quad \text{st} \quad \forall (\text{in0}, \text{in1}): \]

CoreIR.Sub(\text{in0}, \text{in1}) == PE(\text{inst}, \text{input\_binding}(\text{in0}, \text{in1}))
\[ \exists (\text{input\_binding, inst}) \quad \text{st} \quad \forall (\text{in0, in1}): \]

CoreIR.Sub(\text{in0, in1}) = PE(\text{inst, input\_binding(\text{in0, in1})})[\text{‘res’}]
\[ \exists (\text{input\_binding}, \text{inst}) \quad \text{st} \quad \forall (\text{in0}, \text{in1}, \text{other}): \]

\[ \text{CoreIR.Sub}(\text{in0}, \text{in1}) = \text{PE}(\text{inst}, \text{input\_binding}(\text{in0}, \text{in1}, \text{other}))[\text{‘res’}] \]
How to Handle State?
How to Handle State?

Transform
Floating Point?
Performance of Rewrite Rule Generator

- Problem: Universally Quantified SMT queries can take a long time
- Solutions:
  - It is okay to be slightly slow (unless doing DSE!)
  - Different ways to encode the final formula
  - Different techniques for solving Quantified Expression
- Recent results:
  - ~1 minute to solve 20 rewrite rules on our current CGRA.
What patterns to use in the rewrite rule table?

PEak Program (PE spec) ➔ PEak Compiler ➔ Rewrite Rule Table ➔ PE and MEM Mapper ➔ Mapped CoreIR Graph ➔ Place & Route Engine ➔ CGRA Bitstream

Rewrite Rules:
- Rewrite Rule 1
- Rewrite Rule 2
- Rewrite Rule 3
- Rewrite Rule 4
- ...

Operations:
- sub
- mul
- div
- add
- ashr
Which Patterns?

- Enumerate all possible patterns up to a size
  - Lots of uncommon patterns
  - Bloated Rewrite Rule Table
    - Slower instruction selection
- Analyze target domain’s applications for common subgraphs
  - Approach used for our upcoming DSE paper
- Only very basic patterns
  - Use peephole optimization/packing after instruction selection
CPU Instruction Selection
CGRA Compilation

CoreIR Graph

From Global Buffer

Unified Buffer ➔ Computation Kernel ➔ Unified Buffer ➔ Computation Kernel ➔ To Global Buffer
Control Flow Graph

Basic Block
(Machine independent)

R2 ← Sub(R0, R1)
R3 ← M[R2]
M[R3] ← R1
R4 ← Add(R1, 0x50)
...

Basic Block
Basic Block
Basic Block
Compiling WebAssembly to RiscV?
Transform RiscV to remove Register File

Before:

RISCV

Transform:

RISCV

Register File

Register File

rs1

rs2

rd
Discovering Subtract

Out ← Sub(In0, In1)

RISCV
Branch/Memory Instructions?

RISCV

inst rs1 rs2 PC MemRead

rd Next PC Mem Addr Mem Write
The Future

• Goal: Fully Automatic compiler generation for Accelerator Architectures
Thank You