Generating Hardware-Software Co-Design From Dataflow Programs

Süleyman Savaş
2018-05-18
Outline

• Applications
• Dataflow Programming
• CAL Actor Language
• Compilation Framework
• Chisel – HDL
• Code Generation
• Case Studies & Results
• Conclusion
Applications

- Machine Learning
- Audio/Video Processing
- Wireless Communication
- Radar Signal Processing

- Massive data stream
- Continuous processing
- Chain of tasks
- Communication

- High performance
- Low power
Dataflow Programming Model

- Individual actors
- Parallel execution
- Data flowing through FIFO channels

- Arrows = FIFO channels
- Boxes = tasks/actors
CAL Actor Language

• Actors
  – Ports
  – Actions
    • code blocks
      – transform inputs into outputs

• Network
  – consists of actors and channels
  – connects actors to each other

• Schedule
Cal2Many

- **Actor Machine** translates the CAL scheduling into if-else statements and optimizes these statements.

- **Actor Execution Intermediate Representation (AEIR)** facilitates imperative code generation and creates abstract syntax tree.

- **Backends** generate target specific code.
Design Method

- CAL
- TURNUS
- CAL2Many
- C & Chisel
- #cores, NoC buffer size, memory size
- RISC-V Core
- Rocket chip generator
- 2D Mesh
- RISC-V gcc
Chisel

• HDL based on Scala
• Set of
  – Classes
  – Objects
  – Usage conventions
• Developed by Berkeley UC
Chisel cont.

- Everything is a node
- A node is:

```scala
class Node {
  // name assigned by user or from introspection
  var name: String = ""
  // incoming graph edges
  def inputs: ArrayBuffer[Node]
  // outgoing graph edges
  def consumers: ArrayBuffer[Node]
  // node specific width inference
  def inferWidth: Int
  // get width immediately inferrable
  def getWidth: Int
  // get first raw node
  def getRawNode: Node
  // convert to raw bits
  def toBits: Bits
  // convert to raw bits
  def fromBits(x: Bits): this.type
  // return lit value if inferrable else null
  def litOf: Lit
  // return value of lit if litOf is non null
  def litValue(def value: BigInt = BigInt(-1)): BigInt
}
```

- **Lit** – constants or literals
- **Op** – logical or arithmetic operations
- **Updateable** – conditionally updated nodes
- **Data** – typed wires or ports
- **Reg** – positive-edge-triggered registers
- **Mem** – memories
Chisel cont.

- Bit operations on **Bits**
- Arithmetic operations on **Num**

- **Bundles** – struct in C
- **Vecs** – indexable vector

```scala
val rom = Vec(UInt(3), UInt(7), UInt(4), UInt(0)) { UInt(width=3) }
val dout = rom(addr)
```

- **Mems** – random access memories

```scala
val rf = Mem(32, UInt(width = 64))
when (wen) { rf(waddr) := wdata }
val dout1 = rf(waddr1)
val dout2 = rf(waddr2)
```

- **Regs**

```scala
val r4 = Reg(UInt(width = 8))
val r5 = RegNext(r4)
val r6 = Reg(init = 0.U, next = r5)
```
Chisel cont.

• Modules
  – Similar to verilog modules

• Multiple clock domain

• BlackBox

• Tests in C++

• Verilog generation
class FullAdder extends Module {
  val io = new Bundle {
    val a = Input(UInt(width = 1.W))
    val b = Input(UInt(width = 1.W))
    val cin = Input(UInt(width = 1.W))
    val sum = Output(UInt(width = 1.W))
    val cout = Output(UInt(width = 1.W))
  }

  // Generate the sum
  val a_xor_b = io.a ^ io.b
  io.sum := a_xor_b ^ io.cin

  // Generate the carry
  val a_and_b = io.a & io.b
  val b_and_cin = io.b & io.cin
  val a_and_cin = io.a & io.cin
  io.cout := a_and_b | b_and_cin | a_and_cin
}
Code Generation

- CAL to C & Chisel
- Hot-spot action  
  - Chisel  
- Library of hw blocks  
  - Arithmetic operations  
- Rest of the application  
  - C & custom instructions
Code Generation

• Supporting
  – Single and multiple cores
  – Floating-point numbers
  – Complex numbers
  – Pipelining

• Not supporting
  – Arrays
  – Loops
  – Mixture of types
Code Generation

```java
static void singleQRD__acc__calculate_boundary () {
    
    // Copy two floating point numbers into 64 bit registers
    
    // Output registers - each will hold two outputs
    
    // Time to call the custom instructions
    ROCC_INSTRUCTION_NO_BLOCK (XCUSTOM_ACC, outputReg0, acc_input0, acc_input1, FUNCT_INV);
    ROCC_INSTRUCTION (XCUSTOM_ACC, outputReg1, acc_input2, 0, FUNCT_FIRE);
    
    // Save the results
    
    // Set the outputs
    
    _acc__calculate_boundary : action -->
    guard row_counter < COL_SIZE
    var
    float a,
    float b,
    float r_tmp,
    int index
    do
    if x_in[row_counter] = 0.0 then
        c := 1.0;
        s := 0.0;
    else
        index := row_counter * ROW_SIZE + col_counter;
        a := r[index] * r[index];
        b := x_in[col_counter] * x_in[col_counter];
        SquareRoot(a + b);
        r_tmp := SquareRoot_ret;
        c := r[index] / r_tmp;
        s := x_in[col_counter] / r_tmp;
        r[index] := r_tmp;
    end
    col_counter := col_counter + 1;
    end

    CAL

    val multiplier0 = Module (new FPMult (32))
    a_v1 := multiplier0.io.out
    multiplier0.io.in1 := x_v0
    multiplier0.io.in2 := x_v0

    val multiplier1 = Module (new FPMult (32))
    b_v1 := multiplier1.io.out
    multiplier1.io.in1 := x_in_v0
    multiplier1.io.in2 := x_in_v0

    val sqrt0 = Module (new fp32sqrt ()))
    val adder0 = Module (new FPA dd (32))
    sqrt0.io.in := adder0.io.out
    adder0.io.in1 := a_v1
    adder0.io.in2 := b_v1
    val SquareRoot_ret_v0 = sqrt0.io.out
    r_tmp_v1 := SquareRoot_ret_v0
```

C + custom instruction macros

Chisel
Code Generation - Pipelining

- Pipelined library blocks
- Keep track of latency on each data path
- Sync data paths at each binary operation
- Add latency of the block
- Sync output data paths
Case Studies

**QR Decomposition**

- 16 x 16 input matrix
- Floating-point

**Autofocus Criterion Calculation**

- 6 x 6 pixel kernel
- Complex numbers
Results

**QRD**

<table>
<thead>
<tr>
<th></th>
<th>Cycle Count</th>
<th>Clock Freq.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rocket Core</td>
<td>366 k</td>
<td>58 MHz</td>
</tr>
<tr>
<td>Rocket Core + Generated Accelerator</td>
<td>92 k</td>
<td>56 MHz</td>
</tr>
<tr>
<td>Rocket Core + Hand-written Accelerator</td>
<td>89 k</td>
<td>56 MHz</td>
</tr>
</tbody>
</table>

**Autofocus**

<table>
<thead>
<tr>
<th></th>
<th>Cycle Count</th>
<th>Clock Freq.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rocket Core</td>
<td>306 k cycles</td>
<td>58 MHz</td>
</tr>
<tr>
<td>Rocket Core + Hand-written Accelerator</td>
<td>108 k cycles</td>
<td>58 MHz</td>
</tr>
<tr>
<td>Rocket Core + Generated Accelerator</td>
<td>108 k cycles</td>
<td>58 MHz</td>
</tr>
</tbody>
</table>

**Performance results**

<table>
<thead>
<tr>
<th></th>
<th>LUT</th>
<th>FF</th>
<th>BRAM</th>
<th>DSP</th>
<th>Clock Freq.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rocket Core</td>
<td>39,843</td>
<td>16,512</td>
<td>12</td>
<td>24</td>
<td>58 MHz</td>
</tr>
<tr>
<td>Generated Accelerator</td>
<td>1165</td>
<td>914</td>
<td>2</td>
<td>25</td>
<td>104 MHz</td>
</tr>
<tr>
<td>Hand-written Accelerator</td>
<td>990</td>
<td>812</td>
<td>2</td>
<td>25</td>
<td>104 MHz</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>LUT</th>
<th>FF</th>
<th>DSP</th>
<th>BRAM</th>
<th>Clock Freq.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rocket Core</td>
<td>39,843</td>
<td>16,512</td>
<td>24</td>
<td>12</td>
<td>58 MHz</td>
</tr>
<tr>
<td>Hand-written Accelerator</td>
<td>5187</td>
<td>3732</td>
<td>42</td>
<td>0.5</td>
<td>131 MHz</td>
</tr>
<tr>
<td>Generated Accelerator</td>
<td>5239</td>
<td>3220</td>
<td>42</td>
<td>0.5</td>
<td>117 MHz</td>
</tr>
</tbody>
</table>

**Resource usage results**
Conclusions

• Generated code vs Hand-written
  – Performance
  – Resource usage

• Automation facilitates design space exploration
Future Work

• Chisel back-end optimization
• Library of basic hardware blocks
• NoC integration
• More case studies
  – Different types of cores
  – Different types of memories
Additional material