Lecture #25 – High-Level Compilation

Quick Points

<table>
<thead>
<tr>
<th>Sunday</th>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
<th>Thursday</th>
<th>Friday</th>
<th>Saturday</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Lect-25</td>
<td></td>
<td>Lect-26</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Dead</td>
<td></td>
<td></td>
<td>Project</td>
<td></td>
<td>Project</td>
<td></td>
</tr>
<tr>
<td>Week</td>
<td></td>
<td></td>
<td>Seminars</td>
<td></td>
<td>Seminars</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(EDE')</td>
<td></td>
<td>(Others)</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>Finals</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Write-ups</td>
<td>Deadline</td>
</tr>
<tr>
<td>Week</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>Electronic</td>
<td></td>
<td></td>
<td>Project</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Grades</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>December</td>
<td></td>
<td></td>
<td>Due</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Project Deliverables

- Final presentation [15-25 min]
  - Aim for 80-100% project completeness
  - Outline it as an extension of your report:
    - Motivation and related work
    - Analysis and approach taken
    - Experimental results and summary of findings
    - Conclusions / next steps
  - Consider details that will be interesting / relevant for the expected audience
- Final report [8-12 pages]
  - More thorough analysis of related work
  - Minimal focus on project goals and organization
  - Implementation details and results
  - See proceedings of FCCM/FPGA/FPL for inspiration

Recap – Reconfigurable Coprocessing

- Processors efficient at sequential codes, regular arithmetic operations
- FPGA efficient at fine-grained parallelism, unusual bit-level operations
- Tight-coupling important: allows sharing of data/control
- Efficiency is an issue:
  - Context-switches
  - Memory coherency
  - Synchronization

Instruction Augmentation

- Processor can only describe a small number of basic computations in a cycle
  - $1$ bits -> $2^i$ operations
- Many operations could be performed on $2^W$-bit words
- ALU implementations restrict execution of some simple operations
  - e.g. bit reversal

Recap – PRISC [RazSmi94A]

- Architecture:
  - couple into register file as “superscalar” functional unit
  - flow-through array (no state)
Recap – Chimaera Architecture

- Live copy of register file values feed into array
- Each row of array may compute from register of intermediates
- Tag on array to indicate RFUOP

PipeRench Architecture

- Many application are primarily linear
  - Audio processing
  - Modified video processing
  - Filtering
- Consider a “striped” architecture which can be very heavily pipelined
  - Each stripe contains LUTs and flip flops
  - Datapath is bit-sliced
  - Similar to Garpi/Chimaera but standalone
- Compiler initially converts dataflow application into a series of stripes
  - Run-time dynamic reconfiguration of stripes if application is too big to fit in available hardware

PipeRench Internals

- Only multi-bit functional units used
- Very limited resources for interconnect to neighboring programming elements
- Place and route greatly simplified

PipeRench Place-and-Route

- Since no loops and linear data flow used, first step is to perform topological sort
- Attempt to minimize critical paths by limiting NO-OP steps
- If too many trips needed, temporally as well as spatially pipeline

PipeRench Prototypes

- 3.6M transistors
- Implemented in a commercial 0.18µ, 6 metal layer technology
- 125 MHz core speed (limited by control logic)
- 66 MHz I/O Speed
- 1.5V core, 3.3V I/O

Parallel Computation

- What would it take to let the processor and FPGA run in parallel?

Modern Processors

Deal with:
- Variable data delays
- Dependencies with data
- Multiple heterogeneous functional units
Via:
- Register scoreboarding
- Runtime data flow (Tomasulo)
OneChip

- Want array to have direct memory→memory operations
- Want to fit into programming model/ISA
  - Without forcing exclusive processor/FPGA operation
  - Allowing decoupled processor/array execution
- Key Idea:
  - FPGA operates on memory→memory regions
  - Make regions explicit to processor issue
  - Scoreboard memory blocks

OneChip Pipeline

- Basic Operation is:
  - FPGA MEM[Rsource]→MEM[Rdst]
    - block sizes powers of 2
  - Supports 14 "loaded" functions
    - DPGA/contexts so 4 can be cached
  - Fits well into soft-core processor model

OneChip Instructions

- Basic Operation is: FPGA MEM[Rsource]→MEM[Rdst]
- block sizes powers of 2
- Supports 14 "loaded" functions
- DPGA/contexts so 4 can be cached
- Fits well into soft-core processor model

OneChip (cont.)

- Basic op is: FPGA MEM→MEM
- No state between these ops
- Coherence is that ops appear sequential
- Could have multiple/parallel FPGA compute units
  - Scoreboard with processor and each other
  - Single source operations?
  - Can’t chain FPGA operations?

OneChip Extensions

- FPGA operates on certain memory regions only
- Makes regions explicit to processor issue
- Scoreboard memory blocks

Shadow Registers

- Reconfigurable functional units require tight integration with register file
- Many reconfigurable operations require more than two operands at a time
Multi-Operand Operations

- What’s the best speedup that could be achieved?
  - Provides upper bound
- Assumes all operands available when needed

Additional Register File Access

- Dedicated link – move data as needed
  - Requires latency
- Extra register port – consumes resources
  - May not be used often
- Replicate whole (or most) of register file
  - Can be wasteful

Shadow Register Approach

- Small number of registers needed (3 or 4)
- Use extra bits in each instruction
- Can be scaled for necessary port size

Shadow Register Approach (cont.)

- Approach comes within 89% of ideal for 3-input functions
- Paper also shows supporting algorithms [Con99A]

Summary

- Many different models for co-processor implementation
  - Functional unit
  - Stand-alone co-processor
- Programming models for these systems is a key
- Recent compiler advancements open the door for future development
- Need tie in with applications

Outline

- Recap
- High-Level FPGA Compilation
  - Issues
  - Handel-C
  - DeepC
  - Bit-width Analysis
Overview

- High-level language to FPGA an important research area
- Many challenges
- Commercial and academic projects
  - Celoxica
  - DeepC
  - Stream-C
- Efficiency still an issue
- Most designers prefer to get better performance and reduced cost
  - Includes incremental compile and hardware/software codesign

Languages

- Standard FPGA tools operate on Verilog/VHDL
- Programmers want to write in C

Compilation Time

- Traditional FPGA synthesis often takes hours/days
- Need compilation time closer to compiling for conventional computers

Programmable-Reconfigurable Processors

- Compiler needs to divide computation between programmable and reconfigurable resources
- Non-uniform target architecture
  - Much more variance between reconfigurable architectures than current programmable ones

Issues

- Languages
- Standard FPGA tools operate on Verilog/VHDL
- Programmers want to write in C

Compilation Time

- Traditional FPGA synthesis often takes hours/days
- Need compilation time closer to compiling for conventional computers

Programmable-Reconfigurable Processors

- Compiler needs to divide computation between programmable and reconfigurable resources
- Non-uniform target architecture
  - Much more variance between reconfigurable architectures than current programmable ones

Why Compiling C is Hard

- General language
- Not designed for describing hardware
- Features that make analysis hard
  - Pointers
  - Subroutines
  - Linear code
- C has no direct concept of time
- C (and most procedural languages) are inherently sequential
  - Most people think sequentially
  - Opportunities primarily lie in parallel data

Notable Platforms

- Celoxica – Handel-C
  - Commercial product targeted at FPGA community
  - Requires designer to isolate parallelism
  - Straightforward vision of scheduling
- DeepC
  - Completely automated – no special actions by designer
  - Ideal for data parallel operation
  - Fits well with scalable FPGA model
- Stream-C
  - Computation model assumes communicating processes
  - Stream based communication
  - Designer isolates streams for high bandwidth

Celoxica Handel-C

- Handel-C adds constructs to ANSI-C to enable hardware implementation
- Synthesizable HW programming language based on C
- Implements C algorithm direct to optimized FPGA or RTL

Fundamentals

- Language extensions for hardware implementation as part of a system level design methodology
  - Software libraries needed for verification
  - Extensions enable optimization of timing and area performance
  - Systems described in ANSI-C can be implemented in software and hardware using language extensions defined in Handel-C to describe hardware
  - Extensions focused towards areas of parallelism and communication
Variables

- Handel-C has one basic type - integer
  - May be **signed** or **unsigned**
  - Can be any width, not limited to 8, 16, 32 etc.

Variables are mapped to hardware registers

```c
void main(void) {
    unsigned 6 a;
a = 45;
}
```

Variables are mapped to hardware registers

- Handel-C has one basic type - integer
  - May be **signed** or **unsigned**
  - Can be any width, not limited to 8, 16, 32 etc.

Variables are mapped to hardware registers

```c
void main(void) {
    unsigned 6 a;
a = 45;
}
```

Variables are mapped to hardware registers

- Handel-C has one basic type - integer
  - May be **signed** or **unsigned**
  - Can be any width, not limited to 8, 16, 32 etc.

Variables are mapped to hardware registers

```c
void main(void) {
    unsigned 6 a;
a = 45;
}
```

Variables are mapped to hardware registers

- Handel-C has one basic type - integer
  - May be **signed** or **unsigned**
  - Can be any width, not limited to 8, 16, 32 etc.

Variables are mapped to hardware registers

```c
void main(void) {
    unsigned 6 a;
a = 45;
}
```

Variables are mapped to hardware registers

- Handel-C has one basic type - integer
  - May be **signed** or **unsigned**
  - Can be any width, not limited to 8, 16, 32 etc.

Variables are mapped to hardware registers

```c
void main(void) {
    unsigned 6 a;
a = 45;
}
```

Variables are mapped to hardware registers

- Handel-C has one basic type - integer
  - May be **signed** or **unsigned**
  - Can be any width, not limited to 8, 16, 32 etc.

Variables are mapped to hardware registers

```c
void main(void) {
    unsigned 6 a;
a = 45;
}
```

Variables are mapped to hardware registers

- Handel-C has one basic type - integer
  - May be **signed** or **unsigned**
  - Can be any width, not limited to 8, 16, 32 etc.

Variables are mapped to hardware registers

```c
void main(void) {
    unsigned 6 a;
a = 45;
}
```

Variables are mapped to hardware registers

- Handel-C has one basic type - integer
  - May be **signed** or **unsigned**
  - Can be any width, not limited to 8, 16, 32 etc.

Variables are mapped to hardware registers

```c
void main(void) {
    unsigned 6 a;
a = 45;
}
```

Variables are mapped to hardware registers

- Handel-C has one basic type - integer
  - May be **signed** or **unsigned**
  - Can be any width, not limited to 8, 16, 32 etc.

Variables are mapped to hardware registers

```c
void main(void) {
    unsigned 6 a;
a = 45;
}
```

Variables are mapped to hardware registers

- Handel-C has one basic type - integer
  - May be **signed** or **unsigned**
  - Can be any width, not limited to 8, 16, 32 etc.

Variables are mapped to hardware registers

```c
void main(void) {
    unsigned 6 a;
a = 45;
}
```

Variables are mapped to hardware registers

- Handel-C has one basic type - integer
  - May be **signed** or **unsigned**
  - Can be any width, not limited to 8, 16, 32 etc.

Variables are mapped to hardware registers

```c
void main(void) {
    unsigned 6 a;
a = 45;
}
```

Variables are mapped to hardware registers

- Handel-C has one basic type - integer
  - May be **signed** or **unsigned**
  - Can be any width, not limited to 8, 16, 32 etc.

Variables are mapped to hardware registers

```c
void main(void) {
    unsigned 6 a;
```
DeepC Compiler

- Consider loop based computation to be memory limited
- Computation partitioned across small memories to form tiles
- Inter-tile communication is scheduled
- RTL synthesis performed on resulting computation and communication hardware

DeepC Compiler (cont.)

- Parallelizes compilation across multiple tiles
- Orchestrates communication between tiles
- Some dynamic (data dependent) routing possible

Control FSM

- Result for each tile is a datapath, state machine, and memory block

Bit-width Analysis

- Higher Language Abstraction
  - Reconfigurable fabrics benefit from specialization
  - One opportunity is bitwidth optimization
- During C to FPGA conversion consider operand widths
  - Requires checking data dependencies
  - Must take worst case into account
  - Opportunity for significant gains for Booleans and loop indices
  - Focus here is on specialization

Arithmetic Analysis

- Example
  ```c
  int a;
  unsigned b;
  a = random();
  b = random();
  a = a / 2;
  b = b >> 4;
  a = random() & 0xff;
  ```
  - a: 32 bits, b: 32 bits
  - a: 31 bits, b: 32 bits
  - a: 31 bits, b: 28 bits
  - a: 8 bits, b: 28 bits

Loop Induction Variable Bounding

- Applicable to for loop induction variables.
- Example
  ```c
  int i;
  for (i = 0; i < 6; i++) {
     ...
  }
  ```
  - i: 32 bits
  - i: 3 bits
Clamping Optimization

- Multimedia codes often simulate saturating instructions
- Example

```c
int valpred
```  
```c
valpred: 32 bits
```  
```c
if (valpred > 32767)
valpred = 32767
else if (valpred < -32768)
valpred = -32768
```  
```c
valpred: 16 bits
```  

Solving the Linear Sequence

```plaintext
a = 0 <0,0>
for i = 1 to 10
    a = a + 1 <1,460>
for j = 1 to 10
    a = a + 2 <3,480>
for k = 1 to 10
    a = a + 3 <24,510>
...= a + 4 <510,510>
```

- Sum all the contributions together, and take the data-range union with the initial value
- Can easily find conservative range of <0,510>

FPGA Area Savings

<table>
<thead>
<tr>
<th>Without bitwise</th>
<th>With bitwise</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm (8)</td>
<td>adpcm (8)</td>
</tr>
<tr>
<td>bubblesort (32)</td>
<td>bubblesort (32)</td>
</tr>
<tr>
<td>convolve (16)</td>
<td>convolve (16)</td>
</tr>
<tr>
<td>histogram (16)</td>
<td>histogram (16)</td>
</tr>
<tr>
<td>intfir (32)</td>
<td>intfir (32)</td>
</tr>
<tr>
<td>intmatmul (16)</td>
<td>intmatmul (16)</td>
</tr>
<tr>
<td>jacobi (8)</td>
<td>jacobi (8)</td>
</tr>
<tr>
<td>life (1)</td>
<td>life (1)</td>
</tr>
<tr>
<td>median (32)</td>
<td>median (32)</td>
</tr>
<tr>
<td>mpegcorr (16)</td>
<td>mpegcorr (16)</td>
</tr>
<tr>
<td>newlife (1)</td>
<td>newlife (1)</td>
</tr>
<tr>
<td>parity (32)</td>
<td>parity (32)</td>
</tr>
<tr>
<td>pmatch (32)</td>
<td>pmatch (32)</td>
</tr>
<tr>
<td>sor (32)</td>
<td>sor (32)</td>
</tr>
</tbody>
</table>

Summary

- High-level is still not well understood for reconfigurable computing
- Difficult issue is parallel specification and verification
- Designers efficiency in RTL specification is quite high. Do we really need better high-level compilation?