Compiled Acceleration of C Codes for FPGAs

Walid Najjar
Professor, Computer Science & Engineering
University of California Riverside
ROCCC

Riverside Optimizing Compiler for Configurable Computing

- A C/C++ to VHDL compiler
- Built on SUIF 2 and MachSUIF

Objective

- Code acceleration via mapping of circuits to FPGA
- Same speed as hand-written VHDL codes
- Improved productivity
  - Allows design and algorithm space exploration
- Keeps the user fully in control
  - We automate only what is very well understood
Motivation

- Bridge the *semantic* gap
  - Between algorithms and circuits
- Large scale parallelism on FPGAs
  - Exploiting it with HDLs can be labor intensive
- Bridge the *productivity* gap
  - Translating concise C codes to large scale circuits
Focus

Extensive compile time optimizations
- Maximize parallelism, speed and throughput
- Minimize area and memory accesses

Optimizations
- Loop level: fine grained parallelism
- Storage level: compiler configured storage for data reuse
- Circuit level: expression simplification, pipelining
Target Applications

Any application that can be accelerated on an FPGA

- Embedded domain
  - signal, image, video processing, communication, cryptography, pattern matching
- Biological sciences
  - Protein folding, DNA and RNA string matching
- Network processing
  - Virus signature detection, payload parsing
- Data mining
Features

Smart compiling, simple control
- Extensive compile time transformations and optimizations
- All under user control

Importing existing IP into C code
- Leverage the huge wealth of IP codes when possible

Not only a compiler
- A design space exploration tool
What ROCCC would not do

Compile arbitrary code
- Application codes optimized for sequential execution
- FPGA implementation requires other algorithms
- Code generation for FPGAs is hard enough, we cannot also solve the “dusty deck” problem too!

FPGA as an accelerator
- ROCCC is not intended to compile the whole code to FPGA
- Only compute intensive code segments, typically parallel loops

Automation: User stays in the loop
- We can automate what we understand very well
- So much that we do not yet know or understand, too early for full automation
ROCCC Overview – Current

- Loop level analysis & transformations
- Area, clock & throughput estimation

**C/C++** → **SUIF2**

**MachSUIF** → **VHDL Code Generation** → **VHDL**

**Intermediate C**
Machine generated C code with annotations for readability
Execution Model

A simplified model

- Decoupled memory access from datapath
- Parallel loop iterations
- Pipelined datapath

Diagram:
- Data memory (on or off chip)
- Data fetch
- Buffer
- Unrolled loop bodies
- Data store
- Data memory (on or off chip)
Outline

- Circuit Optimization
  - Same clock speed as hand written HDL code
  - Throughput of one always

- Storage Optimization
  - Minimize number of re-fetch from memory

- Loop Transformations
  - Maximize parallelism
  - Understand impact on area, clock and throughput
Compiled and Hand-written

Prior results
- A factor of 2x in speed between hand-coded HDL and compiler generated.
- Results from SA–C and StreamsC

Comparison
- Xilinx IP codes from the web site.
- Same codes, written in C and compiled.
- Criteria: Clock rate and Area
### Comparison – Clock Rates

<table>
<thead>
<tr>
<th>Code</th>
<th>Xilinx</th>
<th>ROCCC</th>
<th>%Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit_correlator</td>
<td>212</td>
<td>144</td>
<td>0.679</td>
</tr>
<tr>
<td>mul_acc</td>
<td>238</td>
<td>238</td>
<td>1.000</td>
</tr>
<tr>
<td>udiv</td>
<td>216</td>
<td>272</td>
<td>1.259</td>
</tr>
<tr>
<td>square root</td>
<td>167</td>
<td>220</td>
<td>1.317</td>
</tr>
<tr>
<td>cos</td>
<td>170</td>
<td>170</td>
<td>1.000</td>
</tr>
<tr>
<td>Arbitrary LUT</td>
<td>170</td>
<td>170</td>
<td>1.000</td>
</tr>
<tr>
<td>FIR</td>
<td>185</td>
<td>194</td>
<td>1.049</td>
</tr>
<tr>
<td>DCT</td>
<td>181</td>
<td>133</td>
<td>0.735</td>
</tr>
<tr>
<td>Wavelet*</td>
<td>104</td>
<td>101</td>
<td>0.971</td>
</tr>
</tbody>
</table>

(* hand written by us in VHDL)

Comparable clock rates
## Performance – Area

<table>
<thead>
<tr>
<th>Code</th>
<th>Xilinx IP</th>
<th>ROCCC</th>
<th>%Area (slice)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit_correlator</td>
<td>9</td>
<td>19</td>
<td>2.11</td>
</tr>
<tr>
<td>mul_acc</td>
<td>18</td>
<td>59</td>
<td>3.28</td>
</tr>
<tr>
<td>udiv</td>
<td>144</td>
<td>495</td>
<td>3.44</td>
</tr>
<tr>
<td>square root</td>
<td>585</td>
<td>1199</td>
<td>2.05</td>
</tr>
<tr>
<td>cos</td>
<td>150</td>
<td>150</td>
<td>1.00</td>
</tr>
<tr>
<td>Arbitrary LUT</td>
<td>549</td>
<td>549</td>
<td>1.00</td>
</tr>
<tr>
<td>FIR</td>
<td>270</td>
<td>293</td>
<td>1.09</td>
</tr>
<tr>
<td>DCT</td>
<td>412</td>
<td>724</td>
<td>1.76</td>
</tr>
<tr>
<td>Wavelet*</td>
<td>1464</td>
<td>2415</td>
<td>1.65</td>
</tr>
</tbody>
</table>

Average area factor: 2.5
Efficacy of Pipelining Scheme

Compared three schemes
- ROCCC (us)
- ImpulseC (LANL)
- Constraints solver (IRISA, France)

Benchmarks
- “Datapath” – a simple compute intensive datapath with feedback within the loop.
- “Control” – a CORDIC algorithm, a doubly nested control–flow–dominated loop body, with data–dependent branching within the loop.
# Pipelining – Results

<table>
<thead>
<tr>
<th></th>
<th>Stages</th>
<th>Rate</th>
<th>Memory</th>
<th>Slices</th>
<th>Freq.(MHz)</th>
<th>Samples/s</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DATAPATH – 8 bits</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Impulse</td>
<td>3</td>
<td>2</td>
<td>NA</td>
<td>336</td>
<td>59</td>
<td>29 M</td>
</tr>
<tr>
<td>ROCCC</td>
<td>1</td>
<td>1</td>
<td>NA</td>
<td>46</td>
<td>46</td>
<td>46 M</td>
</tr>
<tr>
<td>Solver</td>
<td>3</td>
<td>3</td>
<td>4 (2%)</td>
<td>110</td>
<td>161</td>
<td>36 M</td>
</tr>
<tr>
<td><strong>DATAPATH – 32 bits</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Impulse</td>
<td>4</td>
<td>2</td>
<td>NA</td>
<td>901</td>
<td>51</td>
<td>25 M</td>
</tr>
<tr>
<td>ROCCC</td>
<td>2</td>
<td>1</td>
<td>NA</td>
<td>125</td>
<td>27</td>
<td>27 M</td>
</tr>
<tr>
<td>Solver</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>304</td>
<td>80</td>
<td>26 M</td>
</tr>
<tr>
<td><strong>CONTROL – 32 bits</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>impulse</td>
<td>3</td>
<td>2</td>
<td>NA</td>
<td>157</td>
<td>117</td>
<td>58 M</td>
</tr>
<tr>
<td>ROCCC</td>
<td>37</td>
<td>1</td>
<td>NA</td>
<td>2234</td>
<td>79.5</td>
<td>79 M</td>
</tr>
<tr>
<td>Solver</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>196</td>
<td>147</td>
<td>73 M</td>
</tr>
</tbody>
</table>
Comments on the Pipeline

Clock
- ROCCC has the lowest clock cycle but the highest throughput.
- Both Datapath and Control

Area
- ROCCC has the smallest area on Datapath.
- The largest on Control.

Approach
- No separate controller.
- Control of the pipeline is integrated with the datapath.
Storage Optimizations

Objective

- Detect the reuse of data
- Structure on chip storage for that data
- Schedule the access for reuse
- De-allocate storage when data is not needed

All at compile time

Storage optimization reduces bandwidth pressure
Window operation: common in signal and image processing

A window operation: one iteration of a loop or loop nest.

Window sliding: movement in the iteration space.

High memory bandwidth pressure.
- Data reuse
- Separate reading/writing memories.
- Parallelism

Ref: Guo, Buyukkurt and Najjar, LCTES 2004
Smart Buffer

Definition

- In data-path storage (registers)
- Configured and scheduled by the compiler
- No register addressing: data is pushed by controller into the data path every cycle

Parameters

- Determined by the compiler based on
  - Window sizes in x and y, stride in x and y
  - Data bit width
  - Bus width to memory
Smart Buffer Components

**Managed set**: the set of elements covered by a window.
- All *live*: window available

**Kill set**: a set consists of the elements needed to clear their *live* signals after exporting this window.
Smart Buffer Code Generation

Compile time analysis

- Relies on window size, strides, data width and bus width.
- Generates *windows* and *sets* in the IR.

Resulting VHDL code

- Is not aware of the concepts of *sets* and *windows*.
- Only describes the logical and sequential relationship between signals/registers.

Automatic code generation

We shift run–time control burden to compiler
Smart Buffer Re-Read Factor

Before: each pixel needs to be read nine times except the image’s border.

After: only a small portion needs to be read twice:

$$\frac{\text{Window.y} - \text{stride.y}}{\text{SmartBuffer.y}} \times 100\%$$

$$\frac{3 - 1}{32} \times 100\% = 6.25\%$$

Re-read factor on MIPS: 9 times!
Compiler Transformations

Pre–Optimization Passes
- Control Flow Analysis (√)
- Data Flow Analysis (√)
- Dependence Analysis in Loops
- Alias Analysis

General Transforms
- Constant Propagation (√)
- Constant Folding & Identities (√)
- Copy Propagation (√)
- Dead Store Elimination (√)
- Common Sub Expression Elimination (√)
- Partial Redundancy Elimination (√)
- Unreachable Code Elimination (√)

Memory Transformations
- Scalar Replacement (√)

Loop Level Transformations
- Loop Independent Conditional Removal (√)
- Loop Peeling (√)
- Index Set Splitting
- Loop Unrolling – Full (√)
- Loop Unrolling – Partial (√)
- Loop Fusion (√)
- Loop Tiling
- Invariant Code Motion (√)
- Strength Reduction
Examples

- **FIR**
  - 5 tap, 8 bits

- **Discrete Wavelet Transform**
  - 5x3 (lossy) 8 bits

- **Smith–Waterman**
  - 2 bit data path: DNA
  - 5 bit data path: protein folding

- **Bloom Filter**
  - Probabilistic exact string matching
FIR C Code

FIR 5-tap

for (i=0; i<N; i=i+1) {
}
FIR 5-tap

Area: x4, throughput: x12
DWT C Code

for(i = 0; i<508; i = 1+i) {
    for(j = 0; j<510; j = 1+j {
        sum = (6*image[i][j])>> 3;
        sum = sum+(6* image[i][1+j])>> 3;
        sum = sum+(6* image[i][2+j])>> 3;
        sum = sum+(2* image[1+i][j])>> 3;
        sum = sum+(2* image[1+i][1+j])>> 3;
        sum = sum+(2* image[1+i][2+j])>> 3;
        sum = sum+(-1* image[2+i][j])>> 3;
        sum = sum+(-1* image[2+i][1+j])>> 3;
        sum = sum+(-1* image[2+i][2+j])>> 3;
        sum = sum+(8* image[3+i][j])>> 3;
        sum = sum+(8* image[3+i][1+j])>> 3;
        sum = sum+(8* image[3+i][2+j])>> 3;
        sum = sum+(-4* image[4+i][j])>> 3;
        sum = sum+(-4* image[4+i][1+j])>> 3;
        sum = sum+(-4* image[4+i][2+j])>> 3;
        output[i][j] = sum; }
    }
}
DWT

![Graph showing area and throughput vs. clock frequency](#)

- **Area**
- **Throughput**

- **56.3 MHz Clock Freq**

**Axes:**
- Y-axis: Area (slices)
- X-axis: Throughput (MegaPixel/second)
- **Legend:**
  - Pink square: Area
  - Blue circle: Throughput

**Data Points:**
- un1x1: Area 75.3, Throughput 79.3
- un2x2: Area 79.3, Throughput 56.3
- un4x4: Area 56.3, Throughput 44.4
- un8x8: Area 44.4

---

W. Najjar - UC Riverside

ARSC HPRC Workshop
Smith–Waterman Code

Dynamic Programming
- Used in protein modeling, bio-informatics, data mining ...
- A wave-front algorithm with two input strings
  \[ A[i,j] = F(A[i,j-1], A[i-1, j-1], A[i-1, j]) \]

  \[ F = CostMatrix(A[i,0], A[0,j]) \]

Our Approach
- “Chunk” the input strings in fixed sizes \( k \)
- Build a \( k \times k \) template hardware by compiling two nested loops (\( k \) each) and fully unrolling both.
- Host strip mines the two outer loops over this template.
S–W View

vertical input vector

horizontal input vector

MINMAX

MUX

CostMatrix

A[0,j+1]
A[i,j]
A[i+1,j]

A[i,j]
A[i,j+1]
A[i+1,j]

A[i+1,j+1]

A[0,j+1]
A[i,j]
A[i+1,j]

A[i,j+1]
A[i,j]
A[i+1,j]

A[i+1,j]
A[i+1,0]

A[i+1,j+1]

A[i+1,j]
A[i+1,0]

A[i+1,j+1]

S–W C Code

```c
int One_Cell(int a, int b, int c, int d, int e){
    int t1, t2, xy, sel;
    t1 = min3(a, b, c);
    t2 = max3(a, b, c);
    xy = bitcmb(d, e);
    sel = boollut(xy);
    return boolsel(t1, t2, sel);  }

int main(){
    int i, j, N =1024;
    int A[1024][1024];
    for(i=1; i<N; i=i+1)
        for(j=1; j<N; j=j+1)
            A[i][j] = One_Cell(A[i-1][j], A[i][j-1], A[i-1][j-1], BH[i-1], BV[j-1]);
}
```
S-W 2x2 Template

for(i = 1;(i< N);i = i+2)
    for(j = 1;(j< N);j = j+2)
        for(tmp0 = 0;(tmp0< 2);tmp0 = tmp0+1)
            for(tmp1 = 0;(tmp1< 2);tmp1 = tmp1+1) {
                int tmp00;
                t1 = min3(A[i+tmp0- 1][j+tmp1],
                            A[i+tmp0][j+tmp1- 1],
                            A[i+tmp0- 1][j+tmp1- 1]);
                t2 = max3(A[i+tmp0- 1][j+tmp1],
                            A[i+tmp0][j+tmp1- 1],
                            A[i+tmp0- 1][j+tmp1- 1]);
                xy = bitcmb(BH[i+tmp0- 1], BV[j+tmp1- 1]);
                sel = boollut(xy);
                tmp00 = boolsel(t1, t2, sel);
                A[i+tmp0][j+tmp1] =tmp00;
            }

COMPILER GENERATED
S–W 4x4 Tile Execution

1

2

3

4

5

6

7
## S–W Results

<table>
<thead>
<tr>
<th>Tile</th>
<th>Area (slices)</th>
<th>Area (%)</th>
<th>Clock (MHz)</th>
<th>Pipeline stages</th>
<th>GCUPS</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tile</td>
<td>Chip</td>
<td>P4</td>
<td>Tile</td>
<td>Chip</td>
<td></td>
</tr>
<tr>
<td>4x4</td>
<td>63</td>
<td>1%</td>
<td>126</td>
<td>3</td>
<td>0.67</td>
<td>189</td>
</tr>
<tr>
<td>8x8</td>
<td>286</td>
<td>1%</td>
<td>90</td>
<td>5</td>
<td>1.15</td>
<td>71.2</td>
</tr>
<tr>
<td>12x12</td>
<td>755</td>
<td>3%</td>
<td>97</td>
<td>8</td>
<td>1.75</td>
<td>41.1</td>
</tr>
<tr>
<td>16x16</td>
<td>1394</td>
<td>5%</td>
<td>108</td>
<td>11</td>
<td>2.51</td>
<td>31.9</td>
</tr>
</tbody>
</table>

### 2-bit data path

### 5-bit data path

<table>
<thead>
<tr>
<th>Tile</th>
<th>Area (slices)</th>
<th>Area (%)</th>
<th>Clock (MHz)</th>
<th>Pipeline stages</th>
<th>GCUPS</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tile</td>
<td>Chip</td>
<td>P4</td>
<td>Tile</td>
<td>Chip</td>
<td></td>
</tr>
<tr>
<td>4x4</td>
<td>817</td>
<td>3%</td>
<td>55</td>
<td>3</td>
<td>0.293</td>
<td>6.35</td>
</tr>
<tr>
<td>8x8</td>
<td>3604</td>
<td>15%</td>
<td>53</td>
<td>5</td>
<td>0.678</td>
<td>3.33</td>
</tr>
<tr>
<td>12x12</td>
<td>8344</td>
<td>35%</td>
<td>58</td>
<td>8</td>
<td>1.04</td>
<td>2.08</td>
</tr>
<tr>
<td>16x16</td>
<td>14883</td>
<td>63%</td>
<td>52</td>
<td>11</td>
<td>1.21</td>
<td>1.21</td>
</tr>
</tbody>
</table>
Bloom Filter

Work in progress

- A bloom filter is a space–efficient data structure used to test the set membership of an element.
- Adapted to detect virus signature bit patterns in packets.

Preliminary results

- 584 MB/sec on 1173 slices out of 46592 (2%)
Bloom Filter C Code

for(i=0;i<248;i++)
{
    for(j=0;j<7;j++)
    {
        value = input_stream[i+j];
        temp = value & 0x1;
        for(k=0; k<7; k++)
        {
            result_location1 = result_location1 ^ (hash_function1[k] & temp);
            result_location2 = result_location2 ^ (hash_function1[k] & temp);
            result_location3 = result_location3 ^ (hash_function1[k] & temp);
            result_location4 = result_location4 ^ (hash_function1[k] & temp);
            value = value >> 1;
        }
        found = bit_array[result_location1] & bit_array[result_location2] &
                bit_array[result_location3] & bit_array[result_location4];
    }
}

Compile time constant, folded

In data-path Table lookup
# Productivity “Speedup”

<table>
<thead>
<tr>
<th>Code</th>
<th>C</th>
<th>VHDL</th>
<th>Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIR</td>
<td>2</td>
<td>1,100</td>
<td>8x unrolled</td>
</tr>
<tr>
<td>DWT</td>
<td>18</td>
<td>16,500</td>
<td>8x8 unrolled</td>
</tr>
<tr>
<td>S–W</td>
<td>13</td>
<td>12,000</td>
<td>16x16 tile</td>
</tr>
<tr>
<td>B–F</td>
<td>11</td>
<td>3,400</td>
<td>8 bytes</td>
</tr>
</tbody>
</table>

**A ratio of ~ 1,000**
Current and Future Work

(More) Compiler transformations
- Multi-Loop fusion
- Pipelining of tiled code
- Smarter smart buffer

Backend IR for configurable computing
- Supports circuit optimization and generation
- Allow multiple front-ends and multiple targets
S–W Pipelined Tile

Increase throughput & speedup by (2k -1)
Smarter Buffer

Diagram showing the flow of data through various components such as input memory, smart buffers, address generators, data path sequencer, scalar data path, and output memory.
Conclusion

ROCCC can

- Extract and deliver large scale parallelism
  - Instruction and loop levels
- Optimize on-chip storage
- High throughput and speedup

www.cs.ucr.edu/roccc

Thank you