Software and Firmware Co-development using High-level Synthesis

Nikhil Pratap Ghanathe*
Dr. Herman Lam*
Dr. Alan D George*

Dr. Darin Acosta**
Dr. Ivan Furic**
Alexander Madorsky**

* NSF Center for High-Performance Reconfigurable Computing (CHREC)
** The Institute for High Energy Physics and Astrophysics (IHEPA), supported by Fermi National Lab
Trigger primitive conversion into polar coordinates

Zone image formation

Zone hit extenders

Phi pattern detectors

Best Phi pattern selectors

Patterns to primitives matching

Delta phi and delta theta calculation

Best tracks selector

Pt assignment

Best tracks coordinates and Pt
Motivation

**Code development**
- Current version of code developed by UF team
- Verilog implementation took years

**Maintenance**
- CMS upgrades hardware/algorithm at regular intervals
- Code & development complexity rapidly increasing
  - Lack of flexibility, lengthy development time

**Verification**
- C++ code written manually and painfully made to be consistent with Verilog
  - Important for scientists to verify code in C++ (not Verilog)
- C++ code becoming inconsistent with added (Verilog) code complexity

2016-09-27 Nikhil Pratap Ghanathe
Goal:
Explore use of high-level synthesis languages and tools for next-generation CMS code for

- Parallel development of firmware and C++ model
  - Single source code
- CMSSW compatibility (g++ compatibility)
- Increase flexibility in code development
  - Decrease in development time
- Consistent high-level (C++) verification
**Tool Requirements**
- Tight latency control and optimal resource usage
- g++ compatibility for future inclusion into CMSSW
- Good C++ code performance

**Tool exploration and selection**
- Explored OpenCL, Vivado HLS, BlueSpec

**Rationale for Vivado HLS**
- Directives-driven, architecture-aware compiler with best possible QoR
  - Mature support for Xilinx
- C/RTL co-simulation
- Easy integration into RTL-based design flow
- Compatibility with g++ compiler
Vivado HLS Design Flow

Functional Specification

C Design

Synthesis

RTL Design

Verification

C Wrapper

Packaging

Vivado IP Packager

Architectural Verification

- IP Integrator
- System Generator
- RTL

Vivado HLS Design Flow
HLS Productivity
**Challenge:** Parallel execution of “for” loop

**Optimization:** Loop Unrolling

- Multiple iterations executing in parallel instead of sequential execution
  - Latency improvement
  - HLS automatically synchronizes multiple iterations
**Challenge**: Memory contention (parallel access to LUTs)

**Optimization**: Array Partitioning

- Memory contention resolved
  - Numerous accesses to LUT at the same instant
  - Latency minimized significantly
- “N” parallel access done in 1 clock cycle instead of N clock cycles
**Challenge:** Parallel execution and persistence

- Multiple instances of function executing in parallel
- Each instance has to have an array which is persistent

**Optimization:** Object-oriented Approach

```c
static test inst[5];
#pragma HLS ARRAY_PARTITION variable=inst complete dim=1
//create 5 objects

// unroll loop to have 5 instances running
multiple_inst_label1:for(int i=0;i<5;i++)
#pragma HLS UNROLL
    inst[i].test_func(a[i],b[i],index[i],&c[i]);
```

Declare “Array of objects as Static”

Partition array of objects completely

Unroll loop for parallel execution
HLS Fine-grained Control

YOU ARE IN CONTROL!
**Challenge**: Emulate “always” block

- Use while(1) loop to emulate
  - HLS infers no fanin/fanout

**Optimization**

- Use While(en==1) loop
  - HLS ignorant of value of “en” signal

- Manipulate HLS into synthesizing an “always” block
  - Demonstrates the amount of control user has on synthesized design
**Challenge:** Undesired FSM extracted for purely combinational design

- HLS establishes false dependencies; hence latency = 3 clock cycles

**Optimization:** Array Reshaping

- HLS treats all operations as one and latency = 0 clock cycles

```c
#pragma HLS ARRAY_RESHAPE variable=ph_zone complete dim=0

ph_zone[0][2] = 0;
if (phzvl[2][0] & 0x1) ph_zone[0][2](1, 1+ph_hit_w20-1) = ph_zone[0][2](1, 1+ph_hit_w20-1) | ph_hit[2][0];
if (phzvl[2][1] & 0x1) ph_zone[0][2](39, 39+ph_hit_w20-1) = ph_zone[0][2](39, 39+ph_hit_w20-1) | ph_hit[2][1];
if (phzvl[2][2] & 0x1) ph_zone[0][2](76, 76+ph_hit_w20-1) = ph_zone[0][2](76, 76+ph_hit_w20-1) | ph_hit[2][2];
```

Control how HLS treats a set of operations

- Control RTL level constructs from HLS level
- Latency minimized from 3 clock cycles to 0 clock cycles
Challenge: Save 1 clock cycle from Sorter module

Optimization: Inline & Latency Directive, Code structure manipulation
- HLS tries to fit everything into 1 clock cycle

Challenge: Critical path delay > Clock period

Optimization: Explicit Pipelining

```
sort( a, winner0, &winid[0], ret_a);
sort( ret_a, winner1, &winid[1], ret_a1);
sort( ret_a1, winner2, &winid[2], ret_a2);
```

Extra register inserted for first sorter

HLS optimizes code as it pleases when function is INLINED

Solution: DON'T ALLOW HLS to optimize first sorter

```
create new version of sorter function - "SORT_1"
inline "SORT_1"
DO NOT INLINE "SORT"
```
**Challenge:** HLS optimizes shift register

**Optimization:** **Code Structure**

```c
void test ( ap_uint <4> in, ap_uint<4> out[3]) {

volatile ap_uint<4> temp[5];

temp[4]=in;

#pragma HLS unroll
test_label8: for(i=4; i>0; i--){

temp[i-1] = temp[i];
}

out[2]=temp[0];
out[1]=temp[1];
out[0]=temp[2];
}
```

- **Define as VOLATILE to avoid optimizations**
- **Explicitly create shift register**
- **Assign outputs**
## Resource Usage Statistics

<table>
<thead>
<tr>
<th>Module Name</th>
<th>HLS (% of LUTs)</th>
<th>Verilog (% of LUTs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive Converter</td>
<td>12%</td>
<td>6%</td>
</tr>
<tr>
<td>Zone Image Formation</td>
<td>1%</td>
<td>1%</td>
</tr>
<tr>
<td>Zone hit Extender</td>
<td>1%</td>
<td>1%</td>
</tr>
<tr>
<td>Phi-Pattern Detector</td>
<td>11%</td>
<td>16%</td>
</tr>
<tr>
<td>Sorter</td>
<td>3%</td>
<td>3%</td>
</tr>
<tr>
<td>Co-ordinate Delay</td>
<td>0</td>
<td>2%</td>
</tr>
<tr>
<td>Patterns to primitive matching</td>
<td>10%</td>
<td>16%</td>
</tr>
<tr>
<td>Delta phi and theta calculation</td>
<td>2%</td>
<td>2%</td>
</tr>
</tbody>
</table>

HLS resource Usage <= Verilog Resource Usage
Compatibility in CMSSW Environment

**Goal:** Automation

Compile HLS code in C++ using ‘g++’ compiler

```c
uint16 hstrip, uint8 clctpat, uint24 *ph, uint16 *th,
```

**Arbitrary precision data-types a challenge**

- C-based arbitrary precision data-types not supported by standard C compilers (gcc)
- Does not reflect bit-accurate behaviour of the code
- Vivado HLS uses own-built ‘apcc’ compiler for C-designs

**Solution:** C++ based design

- C++ uses arbitrary precision data-types defined in SystemC standard
Performance Benchmarking

- Primitive Converter module benchmarked on CMSSW
  - HLS (for functional simulation) slower by factor of 2 relative to manually-written C++ code
  - Tolerable factor, hence a good result (preliminary result)

Manually-written C++

HLS code

*CMSSW - (CMS-Software)
Progress:

- Translated and verified all modules of the EMUTF using HLS
- Successfully tested “Primitive Converter” and “Zone image formation module” on Virtex-7 FPGA for 1000 track stubs
- Hardware output of HLS generated code matches output of baseline Verilog impl.

Conclusions:

- Performance and latency constraints met for all modules
  - Sorter module re-worked to save 1 clock cycle of CSC Track-finder
- Resource usage comparable
  - Observed to be better than Verilog impl. for majority of cases
- Compatibility in CMSSW environment for verification in C++
- Lessons learned
  - HLS optimization techniques documented for future use
THANK YOU

Questions?