Trident Compiler : A Compiler for Floating Point Applications

Justin L. Tripp
and Trident Compiler Team
Los Alamos National Laboratory

July 2006
Introduction

- Scientific Computing pushes the boundaries of computing
- Acceleration of code through co-processors
- Examples Floating-Point co-processors, GPUs, I/O controllers

- FPGAs provide a way to provide custom hardware
- FPGAs are normally programmed using VHDL or Verilog (HDL Languages)
- However, HDL Languages are very low-level and difficult
Scientific Applications

- Scientific Computing is typically floating point (singles and doubles)
- Scientific Computing requires support of legacy software
- Much of the software has high-level parallelism already extracted (e.g., MPI, etc.)
- Need a compiler to work with high-level languages that provides acceleration for Scientific Computing.
Trident Compiler

FPGAs can perform high-performance floating-point (FP) operations. However, traditional FPGA development is difficult and few tools exist to specifically aid FP hardware development.

Trident Goals:

- Accept C input with double and float data types.
- Automatically extract available parallelism.
- Automatic pipelining of loops.
- Allow the selection from different FP libraries.
- Allow user developed FP libraries.
Comparison of Compilers

<table>
<thead>
<tr>
<th></th>
<th>Trident</th>
<th>Handel-C</th>
<th>ImpulseC&lt;sup&gt;tm&lt;/sup&gt;</th>
<th>Carte&lt;sup&gt;tm&lt;/sup&gt;C</th>
<th>Mitrion-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSI C</td>
<td>Subset</td>
<td>No</td>
<td>Subset</td>
<td>Subset</td>
<td>No</td>
</tr>
<tr>
<td>Native FP</td>
<td>Yes</td>
<td>No</td>
<td>Imminent</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>IEEE-754</td>
<td>Yes</td>
<td>Yes</td>
<td>No*</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Target new HW</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Ext. FP Lib.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Trident was developed as a test-bed for floating-point compilation. It provides us a platform to explore trade-offs at all levels as well as explore novel optimizations. It also provides a way to understand the impact of different code selections.
Big Picture

Profiled Code

Inner Loop

Code

Send Inputs

Read Results

Inner Loop

TCC

.c

VHDL

Board Model

Synthesizer

EDIF

BitStream

Xilinx ISE

Read Results

Send Inputs

Profiled Code
Four principal phases of compilation:

- LLVM front-end
- Trident IR Transformation
- Scheduling
- Synthesis

LLVM - Low Level Virtual Machine (www.llvm.org)
IR - Intermediate Representation
LLVM: How Trident Uses it

• GCC-based C and C++ front-end provided by LLVM produces LLVM bytecode (optimizations and linking are disabled).

• Note: C programs should not contain print statements, recursion, malloc or free calls, function arguments or returned values, calls to functions with variable length argument lists or arrays without a declared size.

• LLVM Trident pass optimizes LLVM bytecode (constant propagation, small function inlining, loop invariant hoisting, tail call elimination, small loop unrolling, CSE and others) and generates modified form of LLVM language

• Trident parses LLVM language into Trident IR (Java)
LLVM Language

- RISC-like three address code ($a = \text{add float } b, c$)
- SSA Static Single Assignment form (infinite virtual register set and explicit dataflow)
- Simple low-level control flow constructs create explicit control-flow graph (ret, br, switch)
- Load/store instructions with typed pointers
- Explicit language-independent type information (void, bool, int, float, double, ushort, ..., pointer, array, structure, function)
- Explicit typed pointer arithmetic (getelementptr takes a pointer and returns element address)
Trident Intermediate Representation (IR)

Control Flow Graph

IR is used for:
Optimization, Resource allocation, Scheduling, Operation Mapping.

Operations also include:
• Start and Stop times
• Operand Type
• HW Reuse set
• Operator Class Information

HyperBlock

Hyperblock

Operator
Operands
Predicates

fpadd %tmp_24 %tmp_21 %tmp_23 %b1

Los Alamos National Laboratory
Est. 1943
Hardware Analysis and Instruction Scheduling

There is always a limited possible communication bandwidth with memory and space on any given FPGA chip. Ensuring successful implementation of the circuit on the target chip and achieving maximum execution speed requires analysis of the hardware.

- **Hardware Analysis**
  - Preliminary schedule to determine times of memory reads and writes
  - Array to memory allocation
  - Logic space requirements analysis

- **Instruction Scheduling (schedule type chosen by user)**
  - non-loop code - ASAP, ALAP, Force-Directed
  - loop code - Modulo scheduling
Array Allocation

- No memory allocation provided on FPGAs - Where should it go?

**Strategy:**

- Pre-allocate to memories to understand read and write latencies
- Preliminary schedule operations to know where the reads and writes are
- Choose a memory allocation
- Evaluate memory allocation on # of cycles required
Choosing a Memory Allocation

- Check to see if there memory is available
- Iterate through each memory and each array
- Check the cost in # of cycles for this array in this memory
- Optimize through minimization in the change in the overall allocation cost

While sufficient space in mem. and not all arrays have been allocated

For each array

Attempt to allocate this array to this memory

If cannot allocate increase the number of attempts to allocate this array
Modulo Scheduling is an algorithm that schedules a loop as just shown (with a body having a depth of II cycles), while simultaneously finding a working II and scheduling the instructions within the body of the loop such that hardware conflicts and interloop dependency problems can be avoided.

**II - Initiation Interval**
Modulo Scheduling: Results

These results are for the main loop of the code:

<table>
<thead>
<tr>
<th></th>
<th>No Modulo</th>
<th></th>
<th>Modulo</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FD</td>
<td>ASAP</td>
<td>ALAP</td>
<td>FD</td>
</tr>
<tr>
<td>simple pi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average Ops/Cycle</td>
<td>0.168</td>
<td>0.168</td>
<td>0.170</td>
<td>1.76</td>
</tr>
<tr>
<td>Max Ops/Cycle</td>
<td>2.0</td>
<td>4.0</td>
<td>2.0</td>
<td>25.0</td>
</tr>
<tr>
<td>Cycle Count</td>
<td>89</td>
<td>89</td>
<td>88</td>
<td>29</td>
</tr>
<tr>
<td>simple float loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average Ops/Cycle</td>
<td>2.12</td>
<td>2.12</td>
<td>2.83</td>
<td>29.25</td>
</tr>
<tr>
<td>Max Ops/Cycle</td>
<td>4.0</td>
<td>4.0</td>
<td>4.0</td>
<td>50.0</td>
</tr>
<tr>
<td>Cycle Count</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>4</td>
</tr>
</tbody>
</table>
Synthesis

- Datapath generation
- FSM and Multiple-block Control
- Arrays
- FP Library Integration
  - Quixilica
  - Arénaire
  - Trident (local library)
- HDL Generation
  - Abstract Circuit
  - VHDL Backend
  - Board Interface
Synthesis – Structure

Top Level

Control Module

Memory Bus

Block 1
State Machine
Datapath

Register File

Block N
State Machine
Datapath

Control

Memory Bus
Synthesis Results

Here are the results for a few benchmarks. These were obtained for the Cray XD1 using the ISE 6.3p3 tools with the Quixilica floating-point library. The overhead for interfacing to the Cray FPGA Board is about 10-15%.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Clk (MHz)</th>
<th>Slice Count</th>
<th>%Area</th>
<th>Blocks</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>Photon</td>
<td>187</td>
<td>12,109</td>
<td>51</td>
<td>1</td>
<td>112</td>
</tr>
<tr>
<td>Photon-hand</td>
<td>98</td>
<td>8,819</td>
<td>20</td>
<td>1</td>
<td>98</td>
</tr>
<tr>
<td>Euclid</td>
<td>200</td>
<td>7,039</td>
<td>19</td>
<td>1</td>
<td>71</td>
</tr>
</tbody>
</table>

*Photon-hand* is an engineer generated design for the Radiative Heat transfer application (Photon). The numbers are just for the design pipeline and do not include any overhead required to interface with a particular board.
Current Status

Trident provides an open framework for FP computation exploration.

- Available under the GPL at http://sourceforge.net/projects/trident
- Support for Single and Double floating point operations.
- Support for fixed size arrays.
- Limited applications can be synthesized, simulated and executed on the Cray XD1.
- Two different FP Libraries can be selected (Quixilica, Arénaire).
- Loop Pipelining via Modulo Scheduling is supported.
Future Directions

- Add support for streaming data
- More aggressive memory bandwidth allocation
- Dynamic Arrays
- Support for other HW platforms
- Support for Execution Estimation
Questions?

Web address:

http://sourceforge.net/projects/trident

E-mail:

jtripp@lanl.gov