Custom Hardware State-Machines and Datapaths – Using LLVM to Generate FPGA Accelerators

Alan Baker
Altera Corporation
FPGAs are Awesome

- Fully Configurable Architecture
- Low-Power
- Customizable I/O

**Consumer**
- Automotive
- Entertainment
  - Broadband
  - Audio/video
  - Video display
- Automotive
  - Navigation
  - Entertainment

**Test, Measurement & Medical**
- Instrumentation
  - Medical
  - Test equipment
  - Manufacturing

**Communications & Broadcast**
- Wireless
  - Cellular
  - Base stations
  - Wireless LAN
- Networking
  - Switches
  - Routers
- Wireline
  - Optical
  - Metro
  - Access
- Broadcast
  - Studio
  - Satellite
  - Broadcasting

**Military & Industrial**
- Military
  - Secure comm.
  - Radar
  - Guidance and control
- Security & Energy Management
  - Card readers
  - Control systems
  - ATM

**Computer & Storage**
- Computers
  - Servers
  - Mainframe
- Storage
  - RAID
  - SAN
- Office Automation
  - Copiers
  - Printers
  - MFP
FPGA Design Hurdles

- **Traditional FPGA design entry done in hardware description languages (HDL)**
  - e.g. Verilog or VHDL
  - HDL describe the register transfer level (RTL)
  - Programmer is responsible for describing all the hardware and its behaviour in every clock cycle
  - The hardware to describe a relatively small program can take months to implement
  - Testing is difficult

- **Far fewer hardware designers than software designers**
Simpler Design Entry

- **Use a higher level of abstraction**
  - Easier to describe an algorithm in C than Verilog
  - Increases productivity
  - Simpler to test and verify
  - Increases the size of the developer pool

- **Sounds promising, but how can we map a higher level language to an FPGA?**
Our Vision

- Leverage the software community’s resources

- LLVM is a great compiler framework
  - Mature
  - Robust
  - Well architected
  - Easy to modify and extend
  - Same IR for different input languages

- We modify LLVM to generate Verilog
  - Implemented a custom backend target
OpenCL

- Our higher level language
- Hardware agnostic compute language
  - Invented by Apple
  - 2008 Specification Donated to Khronos Group and Khronos Compute Working Group was formed

What does OpenCL give us?
  - Industry standard programming model
  - Aimed at heterogeneous compute acceleration
  - Functional portability across platforms
OpenCL Conformance

- You must pass conformance to claim OpenCL support
  - Over **8000** tests
  - *Only one FPGA vendor has passed conformance*
The BIG Idea behind OpenCL

- **OpenCL execution model** ...
  - Define N-dimensional computation domain
  - Execute a kernel at each point in computation domain

**Traditional loops**

```c
void trad_mul(int n,
              const float *a,
              const float *b,
              float *c)
{
    int i;
    for (i=0; i<n; i++)
        c[i] = a[i] * b[i];
}
```

**Data Parallel OpenCL**

```c
kernel void dp_mul(global const float *a,
                   global const float *b,
                   global float *c)
{
    int id = get_global_id(0);
    c[id] = a[id] * b[id];
} // execute over “n” work-items
```
FPGAs vs CPUs

- FPGAs are dramatically different than CPUs

- Massive fine-grained parallelism
- Complete configurability
- Huge internal bandwidth
- No callstack
- No dynamic memory allocation
- Very different instruction costs
- No fixed number of program registers
- No fixed memory system
Targeting an Architecture

- In a CPU, the program is mapped to a fixed architecture
- In an FPGA, there is NO fixed architecture
- The program defines the architecture
- Instead of the architecture constraining the program, the program is constrained by the available resources
Datapath Architecture

FPGA datapath ~ Unrolled CPU hardware
A simple 3-address CPU
Load immediate value into register
Load memory value into register

**Instruction Flow**

1. **Fetch**
   - PC (Program Counter) to Load
   - Instruction to Load

2. **Load**
   - LdAddr from Load
   - LdData from Load

3. **ALU**
   - Aaddr from Registers
   - Baddr from Registers
   - Caddr from Registers
   - CWriteEnable to ALU

4. **ALU Output**
   - A from ALU
   - B from ALU
   - C from ALU

5. **Store**
   - StAddr from Store
   - StData from Store

**Registers**

- Aaddr
- Baddr
- Caddr
- CWriteEnable
- CData

**ALU**

- A
- B
- C

**Op**

- Load
- Store
Store register value into memory

**Diagram:**
- **PC** -> **Fetch**
- **Instruction** -> **Load**
- **LdAddr** -> **Load**
- **LdData** -> **Load**
- **StAddr** -> **Store**
- **StData** -> **Store**

**Registers:**
- **Aaddr**
- **Baddr**
- **Caddr**
- **CWriteEnable**
- **Op**
- **Val**

**ALU:**
- **A**
- **B**
- **C**
- **Op**
- **CData**

**Note:** The diagram illustrates the process of fetching an instruction, loading it into the registers, and then using the ALU to manipulate data before storing the result into memory.
Add two registers, store result in register
Multiply two registers, store result in register

**Instruction Flow:**
- **Fetch**:
  - PC → Fetch
  - Instruction

- **Load**:
  - Instruction → Registers
  - Aaddr → Load
  - LdData

- **ALU**:
  - Op
  - A
  - B
  - C
  - CWriteEnable
  - CData

- **Store**:
  - LdData → Load
  - StAddr → Store
  - StData

**Register Connections:**
- **Aaddr**
- **Baddr**
- **Caddr**
- **LdAddr**
- **StAddr**

**Operands:**
- **Op**
- **Val**

**ALU Operation:**
- Multiply two registers, store result in register
A simple program

- Mem[100] += 42 * Mem[101]

- CPU instructions:

  R0 ← Load Mem[100]
  R1 ← Load Mem[101]
  R2 ← Load #42
  R2 ← Mul R1, R2
  R0 ← Add R2, R0
  Store R0 → Mem[100]
CPU activity, step by step

R0 ← Load Mem[100]

R1 ← Load Mem[101]

R2 ← Load #42

R2 ← Mul R1, R2

R0 ← Add R2, R0

Store R0 → Mem[100]
Unroll the CPU hardware…

R0 ← Load Mem[100]

R1 ← Load Mem[101]

R2 ← Load #42

R2 ← Mul R1, R2

R0 ← Add R2, R0

Store R0 → Mem[100]
1. Instructions are fixed. Remove “Fetch”
... and specialize

1. Instructions are fixed. Remove “Fetch”
2. Remove unused ALU ops
... and specialize

1. Instructions are fixed. Remove “Fetch”
2. Remove unused ALU ops
3. Remove unused Load / Store

R0 ← Load Mem[100]
R1 ← Load Mem[101]
R2 ← Load #42
R2 ← Mul R1, R2
R0 ← Add R2, R0
Store R0 → Mem[100]
... and specialize

1. Instructions are fixed.
   Remove “Fetch”
2. Remove unused ALU ops
3. Remove unused Load / Store
4. Wire up registers properly!
   And propagate state.

R0 ← Load Mem[100]
R1 ← Load Mem[101]
R2 ← Load #42
R2 ← Mul R1, R2
R0 ← Add R2, R0
Store R0 → Mem[100]
... and specialize

1. Instructions are fixed.
   Remove “Fetch”
2. Remove unused ALU ops
3. Remove unused Load / Store
4. Wire up registers properly!
   And propagate state.
5. Remove dead data.

R0 ← Load Mem[100]
R1 ← Load Mem[101]
R2 ← Load #42
R2 ← Mul R1, R2
R0 ← Add R2, R0
Store R0 → Mem[100]
Instead of a register file, live data is carried through register stages like a pipelined CPU instruction.

Live ranges define the amount of data carried at each register stage.
Optimize the Datapath

1. Instructions are fixed. Remove “Fetch”
2. Remove unused ALU ops
3. Remove unused Load / Store
4. Wire up registers properly! And propagate state.
5. Remove dead data.
6. Reschedule!

R0 ← Load Mem[100]
R1 ← Load Mem[101]
R2 ← Load #42
R2 ← Mul R1, R2
R0 ← Add R2, R0

Store R0 → Mem[100]
FPGA datapath = Your algorithm, in silicon
Data parallel kernel

```c
__kernel void sum(__global const float *a, __global const float *b, __global float *answer)
{
    int xid = get_global_id(0);
    answer[xid] = a[xid] + b[xid];
}
```

float *a = [0, 1, 2, 3, 4, 5, 6, 7]
float *b = [7, 6, 5, 4, 3, 2, 1, 0]
float *answer = [7, 7, 7, 7, 7, 7, 7, 7]
Example Datapath for Vector Add

- On each cycle the portions of the datapath are processing different threads
- While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored
Example Datapath for Vector Add

- On each cycle the portions of the datapath are processing different threads
- While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored

Work item IDs

1 2 3 4 5 6 7
On each cycle the portions of the datapath are processing different threads

While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored
Example Datapath for Vector Add

- On each cycle the portions of the datapath are processing different threads
- While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored

8 work items for vector add example

Work item IDs

3 4 5 6 7
Example Datapath for Vector Add

On each cycle the portions of the datapath are processing different threads:

- While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored.

Silicon used efficiently at steady-state.
High Level Datapath Generation

Compiler Flow
Compiler Flow

Source Code

```c
kernel void sum(global float *a,
    global float *b,
    global float *c)
{
    int gid = get_global_id(0);
    c[gid] = a[gid] + b[gid];
}
```

Altera Offline Compiler

AOC

FPGA Programming File

Clang → OPT → LLC → Verilog Design File → QUARTUS II
kernel void
sum(global float *a,
global float *b,
global float *c)
{
    int gid = get_global_id(0);
    c[gid] = a[gid] + b[gid];
}
Compiler Flow

Source Code

```c
kernel void
sum(global float *a,
    global float *b,
    global float *c)
{
    int gid = get_global_id(0);
    c[gid] = a[gid] + b[gid];
}
```

Alterna Offline Compiler

FPGA Programming File

Clang → OPT → LLC → Verilog Design File

Middle end

Clang –O3 optimizations followed by numerous custom passes to target the FPGA architecture.
Source Code

```c
kernel void
sum(global float *a,
    global float *b,
    global float *c)
{
    int gid = get_global_id(0);
    c[gid] = a[gid] + b[gid];
}
```

Compiler Flow

**Frontend**
- Parses OpenCL extensions and intrinsics to produce LLVM IR

**Backend**
- Creates and schedules an elastic pipelined datapath and produces Verilog HDL

**Altera Offline Compiler**
- AOC

**FPGA Programming File**
- FPGA
- Verilog HDL
- QUARTUS™II

---

© 2014 Altera Corporation—Public
LLVM IR is used to describe a custom architecture specific to the program
Dealing with Resource Constraints

Branch Conversion
Branch Conversion Example

Branch

A: True

B: False

C
1. Determine control flow to conditionally executed basic blocks
Branch Conversion Example

1. Determine control flow to conditionally executed basic blocks
2. Predicate instructions
   - A is predicated if the branch was false and vice-versa
Branch Conversion Example

1. **Determine control flow to conditionally executed basic blocks**

2. **Predicate instructions**
   - A is predicated if the branch was false and vice-versa

3. **Combine A and B**
   - Branch is now unconditional
   - PHIs in C become select instructions
Branch Conversion Example

1. **Determine control flow to conditionally executed basic blocks**

2. **Predicate instructions**
   - A is predicated if the branch was false and vice-versa

3. **Combine A and B**
   - Branch is now unconditional
   - PHIs in C become select instructions

4. **Simplify the CFG**
   - Merges remaining blocks
Branch Conversion

- Squeezes the majority of the CFG into one basic block
- Saves significant amounts of area
- Increased instruction count in the basic block does not adversely affect performance
Improving Performance of Individual Threads

Loop Pipelining
OpenCL Task

- Kernel operates on a single thread
- Data for each iteration depends on the previous iteration
- Loop carried dependency bottleneck performance

```
__kernel void accumulate(__global float *a,
                      __global float *b,
                      int n)
{
    for (int i=1; i<n; ++i)
        b[i] = b[i-1] + a[i];
}
```
Loop Carried Dependencies

- Loop-carried dependency: one iteration of the loop depends upon the results of another iteration of the loop

```
kernl void state_machine(ulong n)
{
    t_state_vector state = initial_state();
    for (ulong i=0; i<n; i++) {
        state = next_state(state);
        unit y = process(state);
        // more work...
    }
}
```

- The value of `state` in iteration 1 depends on the value from iteration 0
- Similarly, iteration 2 depends on the value from iteration 1, etc
To achieve acceleration, we can pipeline each iteration of a loop containing loop carried dependencies

- Analyze any dependencies between iterations
- Schedule these operations
- Launch the next iteration as soon as possible

kernel void state_machine(ulong n)
{
    t_state_vector state = initial_state();
    for (ulong i=0; i<n; i++) {
        state = next_state( state );
        unit y = process( state );
        // more work...
    }
}
Loop Pipelining Example

- **No Loop Pipelining**
  - No overlap of iterations
  - Finishes faster because iterations are overlapped

- **With Loop Pipelining**
  - Looks almost like ND-range thread execution!
So what’s the difference?

- **Pipelined Threads** launch 1 thread per clock cycle in pipelined fashion.
- **Loop Pipelining** may not resolve loop dependencies in 1 clock cycle.

Loop Pipelining enables Pipeline Parallelism AND the communication of state information between iterations.
A new iteration can be launched each cycle

Each iteration still takes multiple cycles to complete, but subsequent iterations are not bottlenecked
A new iteration can be launched each cycle
Each iteration still takes multiple cycles to complete, but subsequent iterations are bottlenecked

```
__kernel void accumulate(__global float *a,
    __global float *b,
    int n)
{
    for (int i=1; i<n; ++i)
        b[i] = b[i-1] + a[i];
}
```
A new iteration can be launched each cycle
Each iteration still takes multiple cycles to complete, but subsequent iterations are bottlenecked
Accumulator Datapath

- A new iteration can be launched each cycle
- Each iteration still takes multiple cycles to complete, but subsequent iterations are bottlenecked
Dependence Analysis

- **Has profound effect on Loop Pipelining**
  - Can lead to difference in performance of more than 100x

- **Significant effort spent to improve dependence analysis**
  - Especially loop-carried dependence analysis

- **Added complex range analysis to help**

- **Uses knowledge of our specialized hardware and programming model**

- **Never good enough!**
LLVM Issues/Wishlist
LLVM Issues

- **Intrinsics don’t support structs**
  - We extended CallInst for our intrinsics

- **Module pass managers running every analysis on every function when only requesting a single function**

- **On-the-fly pass manager not inheriting analyses**

- **Ran into several scaling problems with LLVM passes**
  - Often due to significant loop unrolling and inlining

- **Loop representation**
  - Well formed loops are extremely important to us
  - Some optimizations introduce extra loops
  - while(1) with no return is useful to us
LLVM Wishlist

- Conditional preservation of analyses
- Windows debug support
- Improved dependence analysis
Thank You
References

- Altera OpenCL Example Designs
  http://www.altera.com/support/examples/opencl/opencl.html

- Altera OpenCL Best Practices Guide

- Stratix V Overview
  http://www.altera.com/devices/fpga/stratix-fpgas/stratix-v/stxv-index.jsp

- Cyclone V Overview
  http://www.altera.com/devices/fpga/cyclone-v-fpgas/cyv-index.jsp

- Stratix V ALM
  www.altera.com/literature/hb/stratix-v/stx5_51002.pdf