An FPGA-oriented target language for HLL compilation

Dave Bennett
Eric Dellinger
Jeff Mason
Prasanna Sundarajan
Xilinx Research Labs
Agenda

- Definition
- Motivation
- Implementation
- Advantages/Disadvantages
- Initial Results
- Conclusion
What is meant by “FPGA-oriented target language?”

A language that abstracts the architecture of the FPGA to something that is more friendly to HLL compilers:

- can be targeted by a high-level language compiler
- resembles to a degree possible within the context of an FPGA implementation, an ISA for a traditional Microprocessor
- takes advantage, as much as possible within the context of the other goals, of the unique capabilities of FPGAs
- can be rapidly converted into an FPGA implementation
- offers extensibility for architectural features and market-specific optimizations (e.g. MMX)

Note that some of these goals are conflicting.
## Motivation

### FPGA vs Microprocessor

<table>
<thead>
<tr>
<th></th>
<th>Microprocessor</th>
<th>FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Technology</strong></td>
<td>0.13 Micron</td>
<td>0.13 Micron</td>
</tr>
<tr>
<td><strong>Clock Speed</strong></td>
<td>1.6GHz</td>
<td>180MHz</td>
</tr>
<tr>
<td><strong>Internal Memory Bandwidth</strong></td>
<td>102 GBytes per Sec</td>
<td>7.5 TBytes per Sec</td>
</tr>
<tr>
<td><strong># Processing Units</strong></td>
<td>5 FPU(2MACs + 1FPU) + 6 MMU + 6 Integer Units</td>
<td>212 FPU or 300+ Integer Units or ..........</td>
</tr>
<tr>
<td><strong>Power Consumption</strong></td>
<td>130 WATTS</td>
<td>15 WATTS</td>
</tr>
<tr>
<td><strong>Peak Performance</strong></td>
<td>8 GFLOPs</td>
<td>38 GFLOPS</td>
</tr>
<tr>
<td><strong>Sustained Performance</strong></td>
<td>~2 GFLOPs</td>
<td>~19 GFLOPS</td>
</tr>
<tr>
<td><strong>I/O / External Memory Bandwidth</strong></td>
<td>6.4 GBytes/sec</td>
<td>67 GBytes/sec</td>
</tr>
</tbody>
</table>

*Slide courtesy Nallatech*
Agenda

- Definition
- Motivation
- **Implementation**
- Advantages/Disadvantages
- Initial Results
- Conclusion
How CHiMPS works

- Compiles C code (or, in the future, Fortran and other languages) into CHiMPS Target Language (CTL)
- Extracts dataflow graph from CTL
- Generates hardware from graph
- Additional outputs (such as a SystemC simulation or visual graph like the one shown here) can be generated for documentation and debug

```c
long f(long w, long x, long y, long z) {
    long u = x + y;
    long v = w - z;
    long s = z + v;
    long t = u & y;
    long q = t | s;
    return q;
}
```

Enter f; w, x, y, z  
reg u, v, s, t, q  
add x, y; u  
sub w, z; v  
add z, v; s  
and u, y; t  
or t, s; q  
exit f; q
Conceptual Architecture

• The CHiMPS Target Language is conceptually very similar to a microprocessor ISA with a dynamic number of registers
• Each instruction is converted to an instantiation of a predefined hardware module
• Registers become FIFOs that receive the result of one instruction and are then used as input to a subsequent instruction
• Instructions generally wait for all input FIFOs, then perform operation and send results to output FIFOs, which input to the next instruction(s), creating a pipeline
• Control flow (loops, conditionals) is handled with muxes and demuxes
Pipelining – our choice for parallelism

• Pipelining offers better parallelism than thread-level parallelism
  – All execution units can be executing on each cycle
  – A loop of “n” iterations, of “c” cycles, requires a total of $n \times c$ cycles to execute on a microprocessor architecture compared to $n+c$ cycles on a fully pipelined implementation

• Pipelining offers parallelism with order preserved which is key for traditional programming languages
  – Allows programmers, who think in a sequence of steps, to continue to think sequentially

• Multiple pipelined circuits may be explicitly specified for thread-level parallelism, while pipelining is inferred
Instruction Examples

- **Register creation**
  
  ```
  reg a:32u,b:64,c:13 // declare registers of various bit widths and sign
  ```

- **Arithmetic operations:**
  
  ```
  add a,b;c // Simple add of two numbers giving a third
  add a,b<<2,12;c // c = a+(b*4)+12
  multiply x,y;g // Multiplication
  ```

- **Floating point**
  
  ```
  fadd f,g;h // Floating add. Operand size determines precision
  ```

- **Memory access**
  
  ```
  read 0;0;4;addr;;data // Read four bytes of off-chip DRAM through cache 0 from addr
  write addr;data;width // Write four bytes of DRAM
  ```

- **Control Flow**
  
  ```
  cmp x,y;equal // Compare x and y
  demux m1;equal;br0;br1 // Demux data based on value of equal
  nfor f1;counter;niter // Start a loop of niter times
  ```
Preserving execution order

• There is a path that the PC of a microprocessor follows from beginning to end of a function, like a single ball going through a convoluted pipe
• CHiMPS allows many balls in the pipe at once
• The CTL must therefore preserve the execution order when there are data dependencies
  – Most instructions don’t need to pay any attention to this because the FIFOs guarantee execution order
  – Conditionals and loops require additional hardware to ensure order is maintained
  – Loop carry-forward dependencies require special treatment
"If" statement example

C source:

```c
long f(long w, long x, long y, long z) {
    long q, u, v, s, t;
    if (w*x) {
        u = x+y;
        v = w-z;
        s = z+v;
        t = u & y;
        q = t | s;
    } else {
        q = 0;
    }
    return q;
}
```

Enter f; w, x, y, z
reg qreg temp0,temp1:1 multiply w,x;temp0 cmp temp0,0;temp1 demux m0;temp1;b0;b1 branch b0
reg v,u,t,s
add x,y;u
add w,-z;v
add z,v;s
and u,y;t
or t,s;q
unbranch b0
branch b1
add 0;q
unbranch b1
mux m0
exit f;q

CHiMPS:

Enter f; w, x, y, z
reg q
reg temp0,temp1:1
multiply w,x;temp0
cmp temp0,0;temp1
demux m0;temp1;b0;b1
branch b0
reg v,u,t,s
add x,y;u
add w,-z;v
add z,v;s
and u,y;t
or t,s;q
unbranch b0
branch b1
add 0;q
unbranch b1
mux m0
exit f;q

DFG:
Loops are the key

- Everything that needs to be optimized is a loop
- There are three types of loop
  - Bounds are known at compile time
    ```c
    for (int i = 1; i < 10; i++)
    ```
  - Bounds are known when loop starts
    ```c
    for (int i = 1; i < N; i++)
    ```
  - Bounds aren’t known until exit condition is reached
    ```c
    while (*c)
    ```
- CHiMPS is able to pipeline all three types of loop
Random Access Off-chip memory

• On-chip memory block able to access external ram/disk and caches data inside FPGA
  – Specifically designed for each hardware configuration
• One or more CACHE blocks instantiated in BRAM
  – Each can handle 16 read instructions with up to 4 memory accesses per cycle
  – Cache miss reads into on-chip BRAM
  – Write-through to off-chip memory
  – Cache coherency maintained among multiple cache blocks/multiple FPGAs
• Compiler can optimize number and wiring of caches
  – Additional tool will be provided for manual optimization
static void chimps_dgemm (int M, int N, int K,
    const double ALPHA, const double* A, const int LDA,
    const double* B, const int LDB, const double BETA,
    double* C, const int LDC)
{
    register double      t0;
    int                 i, iail, iblj, icij, j, jal, jbj, jcj, l;

    for (j = 0; j < N; j++) {
        jbj = j * LDB; // LDB is the stride of the B array.
        jcj = j * LDC;

        /* Scale all rows with a column of C matrix */
        chimps_dscal( M, BETA, C+jcj, 1);

        for (l = 0; l < K; l++) {
            jal = l * LDA;
            iblj = l + jbj;

            /* Compute ALPHA * B */
            t0 = ALPHA * B[iblj];

            for( i = 0; i < M; i++)
            {
                iail = jal + i;
                icij = jcj + i;

                /* Compute ALPHA * B * A + C and assign it to C */
                C[icij] += A[iail] * t0;
            }
        }
    }
}
Agenda

• Definition
• Motivation
• Implementation
• Advantages/Disadvantages
• Initial Results
• Conclusion
Conceptual Advantages

*Emulating the x86 model*

- Defines a standard for FPGA implementation that can be public and open
- Solves difficult problems, for example concurrency and memory management, once so users don’t have to do so for every application
- Offers a more stable platform for tool developers since the CTL is likely to change little through many generations of FPGA
- Allows Xilinx to focus on what they do best while offering the opportunity to build around CTL
- Opportunities for FPGA architectural improvements to better support CTL
Specific advantages of CHiMPS Target Language

- Enables optimizing compiler technology to be applied to FPGAs
- Mapping to hardware is visible to coder
  - Makes it easy to see what types of code generate good hardware
  - Allows users to be productive quickly and learn on the fly
- Correlation between source code and hardware is maintained
  - Profile and debug with standard tools like GPROF and GDB
- Simulating at the CTL level can be extremely fast
- Ability to substantially improve implementation times
  - One of biggest complaints of FPGA computing users
- Gives compiler visibility into implementation
  - Area, speed and throughput can be determined statically without detailed knowledge of FPGA
- A “virtual hardware” capability that allows trading off performance and area
Disadvantages

Inherent:
• C has no native ability to describe concurrency
  – Limits what an optimizing compiler can really do
• High performance H/W requires thinking about critical H/W concepts
  – structure required to get maximum performance will be foreign
  – mindset for coding/refactoring accelerated functions unnatural

Implementation-specific:
• Front and back end division guarantees lost optimization opportunities
• Bad choices in CTL can impair optimization and implementation
  – Even the x86 wasn’t perfect

These are probably familiar issues to microprocessor architects
Agenda

• Definition
• Motivation
• Implementation
• Advantages/Disadvantages
• Initial Results
• Conclusion
Current status

- Successfully generated hardware from C code and run it on XUP board!
- All basic pieces in place
- Two months to implement compiler
- Viability of approach has been confirmed
- Performance of approach still being evaluated
- Three benchmarks simulated
  - Smith-Watermann
  - IDCT
  - DGEMM
- Currently working on optimizing compiler (ACE) and completing VHDL models targeting BEE2
C Language coverage

- Currently have a bare-bones compiler (LCC)
- Support a reasonable subset of ISO (ANSI) C
- Most unsupported features are due to compiler limitations
- A few are intrinsic to CHiMPS target language
- Methodology of spatial compilation could support all features, with some overhead

Limitations intrinsic to CHiMPS
1. Code that cannot be implemented using demux and loops. In C, can only generate these using GOTOs.
2. Function pointers (all functions are inlined)
3. Variable argument functions (no stack for arguments)
4. Recursion (inlining of functions and no stack)
Simulated Throughput

<table>
<thead>
<tr>
<th></th>
<th>Smith Watermann</th>
<th>IDCT8</th>
<th>IGEMM (48x48x48)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
<td>360,000,000</td>
<td>366,666,667</td>
<td>10,000</td>
</tr>
<tr>
<td>3GHz P4 Throughput</td>
<td>120,000,000</td>
<td>4,300,000</td>
<td>2,500</td>
</tr>
<tr>
<td>Device Size</td>
<td>4VLX40</td>
<td>4VLX40</td>
<td>4VLX40</td>
</tr>
<tr>
<td>Device Cost</td>
<td>$ 100.00</td>
<td>$ 100.00</td>
<td>$ 100.00</td>
</tr>
<tr>
<td>OPS/$ vs P4</td>
<td>3.60</td>
<td>102.33</td>
<td>4.80</td>
</tr>
</tbody>
</table>

- **Note on IGEMM**
  - Out of the box code with no optimizations done for CHiMPS
Conclusions

- Feasibility of an FPGA-oriented target language has been demonstrated
- For some set of benchmarks, cost/performance is better than 3GHz P4
- Promising avenue for further research
  - We will be continuing and submitting results in a paper for publication (hopefully) by end of the year