Mapping Linear System Solver Algorithms to FPGA on the XD1 Using High Level Toolsets
Joseph Fernando, Dennis Dalessandro, Ananth Devulapalli, Eric Stahlberg Kevin Wohlever
(fernando,dennis,ananth,eric,kevin@osc.edu)

July 12, 2006
Outline

- Our Approach
- High Level Performance Model Development
- Problems selected
- Overall FIR architectures
- Automatic high level (C to VHDL) translation toolsets
  - Targeting integer and single precision floating point
- Implementation Results
- Lessons learned & Future work
Our Approach

- Select problems that are extensively used in the HPC community
- Use algorithms that can be optimally mapped to processor arrays
  - Target systolic arrays specifically
- Use automatic tools to determine the performance of various (limited) design choices
  - Clock frequency and latency after mapping to FPGA
  - Resources required (slices, Block RAM)
High Level Performance Model Development

- Targeted Design flow
- ISE 8.1 SP3 with IP updates for Impulse C (for floating point)
- ISE 7.1 for Dime C
High Level Performance Model Development (Cont.)

- Advantages
  - Can describe algorithm using familiar language
    - Useful to describe legacy algorithms
  - Develop, test and debug using standard compiler environments
    - Shorter design and development cycle improves productivity
  - Verify and Validate results quickly using known legacy data sets
  - Target user’s with Software Engineering background
High Level Performance Model Development (Cont.)

Disadvantage

- Limited capabilities to describe algorithm
  - No explicit support for systolic array design
- Complex control flow mechanism independent of the algorithm
- Serial language used to describe concurrent tasks
  - Mitiron C is concurrent
- Non-optimal design for area, throughput, latency
  - Final design cannot be biased for specific optimal parameter
  - Scheduling of processor arrays cannot be specified
- Non-standard use of C language constructs
- Problem size limited when processor arrays are used
Problems Selected

- **FIR Filter (32-bit integer)**
  - Develop a 8 tap FIR filter using Dime C and Impulse C
  - Integrated the generated code with the OSC developed DMA engine on the XD1

- **LU Decomposition**
  - Developed and Synthesized for a 256x256 matrix using Impulse C
  - Simulated circuit using Dime C generated code

- **LU Solve**
  - Developed and Synthesized for a 256x256 matrix using Impulse C
  - Simulated circuit using Dime C generated code
Overall FIR Architecture - Dime C

- Interfaced with a XD1 DMA engine (not the targeted platform)
- Can read and write data simultaneously up to 200 M Quad words/sec
- Data transfer is the controlled by the DMA engine
- FIR circuit generates one output sample every clock cycle.
- Capable of operating at 196 MHz.
FIR Architecture - Dime C (Cont.)

- The operations of the FIR filter is organized as shown.
- An output is generated every clock cycle when there is data.
- Can specify the computation clock cycle.
FIR Architecture Impulse C

- This architecture is used in all implementations
  - LU Decomposition and LU Solve
- Loop Unrolling and Pipelining pragmas are used for hardware circuits
- Targets the XD1 specifically
  - Does not use the DMA engine to transfer data
Automatic high Level (C to VHDL) Toolsets

- Impulse C
  - Streams based paradigm used for I/O
  - Different processes communicate using streams
  - Support read and write to a stream from the host and the FPGA
  - Memory based I/O not supported on the XD1
Automatic high Level (C to VHDL) Toolsets (Cont.)

- Dime C
  - Pipes, channels and memory based I/O to communicate between functions
  - Supports many ANSI-C language constructs
  - Limitations on pointer use and single dimensional array currently
  - Limited support for concurrency
  - Does not target the XD1
Implementation Results

- **FIR8 Filter**
  - Dime C generated circuit operates at 196 MHz
  - Produces a single result every clock cycle as given by the simulation results
  - Impulse C generated circuit operates at 124 MHz
Implementation Results (Cont.)

- LU Decomposition (256x256 matrix)
  - Takes 925 ms (Impulse C) Vs 72 ms on the host
    - Lower I/O bandwidth contributes to higher latency

<table>
<thead>
<tr>
<th></th>
<th>Impulse C</th>
<th>Dime C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating clock</td>
<td>91</td>
<td>71</td>
</tr>
<tr>
<td>Frequency (MHz)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slices</td>
<td>3207(13%)</td>
<td>9295(39%)</td>
</tr>
<tr>
<td>BRAM</td>
<td>138(59%)</td>
<td>157(67%)</td>
</tr>
</tbody>
</table>
### Implementation Results (Cont.)

- LU Solve (256x256 matrix)

<table>
<thead>
<tr>
<th></th>
<th>Impulse C</th>
<th>Dime C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating clock</td>
<td>102</td>
<td>97</td>
</tr>
<tr>
<td>Frequency (MHz)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slices</td>
<td>3256(13%)</td>
<td>8258(34%)</td>
</tr>
<tr>
<td>BRAM</td>
<td>138(59%)</td>
<td>157 (67%)</td>
</tr>
</tbody>
</table>
Lessons Learned

- **Impulse C**
  - Highly productive environment for algorithm implementation
  - Can manually investigate various design options/algorithms with little effort
  - I/O bandwidth can be a limiter of performance
  - Using DMA engine for I/O will improve overall performance

- **Dime C**
  - Good results for integer based circuits when coupled with fast DMA based I/O
  - Higher resource utilization and lower speed for floating point computation algorithms
Challenges & Future Work

- Users need tools that support systolic array design
- Optimal designs for multiple processor array algorithms must be the targeted
  - Specify the weights for a given cost function
- Tools need support for the automatic extraction of parallelism
  - Explicit as well as implicit
- Standard interfaces between tools
  - Reduce/eliminate the glue logic
Challenges & Future Work (Cont.)

- Need mechanisms to control the timing of the computations
- Various levels of pipelined operators
- Facilities to control the schedule of execution
- Support for Partitioning of processor arrays
  - To enable very large sized problems to be mapped to FPGA