Compiled Code Acceleration Of NAMD On FPGAs

Jason Villarreal, John Cortes, and Walid A. Najjar
Department of Computer Science and Engineering
University of California, Riverside
Common Approaches to Utilizing Reconfigurable Platforms
Introduction - ROCCC

- Riverside Optimizing Compiler for Configurable Circuits
- Converts a subset of C to VHDL
  - Language is C with restrictions
  - No explicit parallel or hw statements required
  - Can compile our code to software or hardware
- Designed not for entire programs, but just the computational kernels
  - Loop nests
  - Streams of data in and streams out
ROCCC Specifics

- **Goals**
  - Maximize the parallelism and clock rate
  - Minimize area and memory accesses

- **Approach - Extensive Compiler Optimizations**
  - Loop level: fine grained parallelism
  - Storage level: compiler configured storage for data reuse
  - Circuit level: expression simplification, pipelining
## Optimizations Supported

<table>
<thead>
<tr>
<th>Loop</th>
<th>Procedure</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Normalization</td>
<td>• Code hoisting</td>
<td>• Scalar replacement</td>
</tr>
<tr>
<td>• Invariant code motion</td>
<td>• Code sinking</td>
<td>• Array RAW/WAW elimination</td>
</tr>
<tr>
<td>• Peeling</td>
<td>• Constant propagation</td>
<td>• Array renaming</td>
</tr>
<tr>
<td>• Unrolling</td>
<td>• Algebraic identities simplification</td>
<td>• Constant array value propagation</td>
</tr>
<tr>
<td>• Fusion</td>
<td>• Constant folding</td>
<td>• Feedback reference elimination</td>
</tr>
<tr>
<td>• Tiling (blocking)</td>
<td>• Copy propagation</td>
<td></td>
</tr>
<tr>
<td>• Strip mining</td>
<td>• Dead code elimination</td>
<td></td>
</tr>
<tr>
<td>• Interchange</td>
<td>• Unreachable code elimination</td>
<td></td>
</tr>
<tr>
<td>• Un-switching</td>
<td>• Scalar renaming</td>
<td></td>
</tr>
<tr>
<td>• Skewing</td>
<td>• Reduction parallelization</td>
<td></td>
</tr>
<tr>
<td>• Induction variable substitution</td>
<td>• Division/multiplication by constant approximation</td>
<td></td>
</tr>
<tr>
<td>• Forward substitution</td>
<td>• If conversion</td>
<td></td>
</tr>
</tbody>
</table>
High Level Code Restrictions

- No low level knowledge required
- Mark the area going to hardware with empty function calls (begin_hw() and end_hw())
- Only perfectly nested loop nests are allowed
- Recursion and pointers are disallowed
  - Only recurrence relations in 1-D or 2-D arrays are allowed
- No structs or unions
- Small syntactic restrictions
for (j = 1; j < N; ++j) {
}
Example ROCCC code

```c
begin_hw();
for (k = 0 ; k < numAtoms * 3 ; k+=3)
{
    t2 = p_i_x - inputArray[k + 0] ;
    r2 = t2 * t2 + r2_delta ;
    t2 = p_i_y - inputArray[k + 1] ;
    r2 += t2 * t2 ;
    t2 = p_i_z - inputArray[k + 2] ;
    r2 += t2 * t2 ;
    if (r2 <= cutoff2_delta)
        outputArray[k] = 1 ;
    else
        outputArray[k] = 0 ;
}
end_hw() ;
```
Application On NAMD

- NAnoscale Molecular Dynamics
  - One of the most popular molecular dynamics programs
  - Optimized for supercomputing systems
- Researchers want millions of atoms at timescales of seconds
  - Current processors allow for microsecond timescales
  - Time step is in femtoseconds
Main loop

for each timestep
  for every atom I in system
    for each other atom J in system
      compute the forces exerted by atom J on atom I
      sum all the forces
      compute its next position

• Of course, not all forces contribute much to the total force
• More complex calculations on the boundaries
  • One loop body with 60 variants!
// Find atoms in range 1
Loop 1:
{
    // Both fast and slow calculations
}

// Find atoms in range 2
Loop 2:
{
    slow_d = kqq;
    val = diffa * slow_d;
}

// Find atoms in range 3
Loop 3:
{
    fast_d += vdw_d;
    val = diffa * fast_d;
}
Most Critical Region

- All 60 loop instances count for 82% of total execution time
- One particular loop instance counted for ~80% of all loop instance computations
  - Total of >65.6% of all execution in one loop
  - That loop contained 52 floating point operations
    - 29 additions/subtractions
    - 17 multiplications
    - 6 divisions
      - All by constants, automatically transformed by ROCCC
for(k=0; k<npairi; ++k)
{
    const int j = pairlisti[k] ;
    register const CompAtom* p_j = p_1 + j ;
    const double* lj_pars = vdwtypearray[j] ;
    const double A = scaling * lj_pars->A ;
    // Etc.
}
Code Transformations - Linearization

Pairlist

Atom List

Reordered List
Example of Linearization

Original Code

for(k=0; k<npairi; ++k)
{
    const int j = pairlist[i][k] ;
    register const CompAtom* p_j = p_1 + j ;
    const double* l_j_pars = vdwtypearray[j] ;
    const double A = scaling * l_j_pars->A ;
    // Etc.
}

Modified Code

for(k=0; k<npairi; ++k)
{
    // Done in Software
    A[k] = vdwtypearray[pairlist[k]]->A ;
}

for(k=0; k<npairi; ++k)
{
    // Done in Hardware
    // Use A[k] instead of A
}

for(k=0; k<npairi; ++k)
{
    // Done in Software
    // Store A[k] if necessary
}
A simplified model

- *Decoupled* memory access from datapath
- Parallel loop iterations
- Pipelined datapath
- Data is pushed by the buffer into the datapath
Interface Mechanism

- Platform dependant
  - Currently ported and supporting the SGI-RASC blade on the SGI Altix 4700 system
  - Interface can be generated by ROCCC, but we are currently only supporting what we have
- Smart buffer reads/writes from the SRAMs located on the SGI-RASC blade
  - RASCLib API is called from C code to pass data back and forth, start the algorithm, initialize registers
SGI RASC RC100 Blade

- Connected to the SGI Altix 4700
  - Single system image
  - 16 to 512 Intel Itanium 2 processors
  - 2 Virtex 4 LX200 chips
- NUMALink architecture
  - Supports 6.4 GB/s/FPGA
Results

- Compiled the most critical loop into hardware
  - Compared iterations of the loop in software and hardware
  - First step in an end-to-end approach
- Single precision implementation fit on LX200
- Double precision implementation overmapped
  - We split double precision into three vectors (x, y, and z) and each individual vector fits on an FPGA
Memory Bandwidth Issues

- Memory is the bottleneck
  - Single precision required 48 bytes per cycle
  - Double precision vectors required 96 bytes per cycle
  - SGI-RASC can feed Single precision once per cycle
    - We need 5.856 GB/s for single precision
  - Double precision needs two cycles to collect data, before datapath can begin
### Results Of One Loop Iteration

<table>
<thead>
<tr>
<th></th>
<th>Vertex 4LX200</th>
<th>1.6 GHz Itanium2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Speed</td>
<td>Slices</td>
</tr>
<tr>
<td>Single Precision</td>
<td>149 MHz</td>
<td>44%</td>
</tr>
<tr>
<td>Double Precision X</td>
<td>168 MHz</td>
<td>63%</td>
</tr>
</tbody>
</table>
Union of Ranges and a New Approach

- Not the ideal way to implement
- All ranges’ calculations combined into one hardware chunk
  - Overmapped - >250 stages in the pipe
- Each range’s calculations combined with distance calculation
  - Some in hw, some in sw possible
- Compiler approach allows us to change this rapidly and try other things
A New Approach

Incoming Data (j atoms)

Distance Calculation
If (within cutoff)
calculate forces

Distance Calculation
If (within cutoff)
calculate forces

Distance Calculation
If (within cutoff)
calculate forces

Summation
Conclusions

- We have run ROCCC on a substantial benchmark (NAMD) and generated floating point and double precision hardware
- We have shown significant speedup of the most critical loop over software
- Compiler approach allows us to try different approaches and versions of the program
Thank you!
Problems

- Our code simulates correctly, but we are currently working with SGI to resolve some issues with the hardware on the actual system.
The original NAMD code had 6 divisions, but they were all divisions by constants:
  - fast_d / 2 + fast_c / 4 + fast_b / 6

ROCCC detected these and replaced them with either a multiplication of the reciprocal or a subtract operation on the exponent:
  - This was performed in hardware only
  - Automatically detected - no low level knowledge necessary
Introduction
Spatial Computing

Data Driven

Parallel execution

```
for (I = 0 ; I < N; ++I)
{
}
```

- Image processing
- Cryptography
- Dynamic programming
- Molecular dynamics
Introduction
Reconfigurable Computing

- FPGAs provide flexible implementations of hardware
- Circuits on FPGAs coupled with microprocessors
  - Allows for both hardware and software interacting

Software designers are used to programming sequentially
- Few languages support spatial computing