Optimizing Compiler for a CELL Processor

Critical Review by
Daniel Tracy
SPEs vs Multi-core

- **SPEs**
  - 256K
  - Explicit Communication

- **Multi-core**
  - Core
  - Main Memory
  - Hardware Coherence
  - Loads/Stores
### Vector Operations

<table>
<thead>
<tr>
<th>double</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>integer</td>
<td>integer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>h</th>
<th>h</th>
<th>h</th>
<th>h</th>
<th>h</th>
<th>h</th>
<th>h</th>
<th>h</th>
</tr>
</thead>
</table>

b b b b b b b b b b b b b b b
Vector Operations

128-bit Data Register

v0

128-bit Data Register

v1

128-bit Data Register

v2

.

.

128-bit Data Register

v127
Vector Operations

256K Local Memory

128-bit Data Register

v0

v1

v2

Slot 1

computation

Slot 2

vector permute

communication

branch
Goals of XL Compiler

• Compile C/Fortran code into parallel multi-SPE code

  • Utilize OpenMP pragmas

  • Insert run-time system to mimic shared memory

• Increase efficiency of SPE utilization

  • Utilize traditional vector compilation techniques

• Force scalar code to execute on vector units
Partition Manager

- Code and data must share a small footprint
- Traditional code overlay system employed
Data Locality: Software Caching

- Cache operations performed by vector unit!
- More than twofold common-case overhead
Memory Representation: Scalar vs Vector

Uniform Data Types
16-byte aligned

vs Arbitrary Types
(e.g., structs, stack, etc.)
Executing Scalar Code on Vector Units

Alignment: Operations between scalars

Scalar store: Masking Into Destination
Avoiding this Insanity?

- Only applies to code we fail to autovectorize
- Only applies to scalars we can’t register-allocate
- Allocate 128 bits for all variables
Miscellany

- Scheduling optimizations must account for limited dual-issue behavior
- Branch optimization involves injection of branch hint instructions for loops
- Autovectorization techniques utilized for SPE code generation

The techniques described for these problems are merely the application of previous work.
Experiments: Base Optimizations

Figure 6: SPE optimizations.
Experiments: Auto-vectorization

Figure 7: Simdization speedups.
Experiments: Multiprocessing

Figure 8: Speedups of parallelization.
Experiments: Multiprocessing

Figure 9: Speedups of parallelization with optimization.
Open Questions:

• What are the alternatives to this approach?
  • High speed library kernels
  • Assembly programming for SPE tasks
  • High-level array manipulation language?

• Could we achieve much higher speeds?
  • What was missed as a critical resource for maximum speed?
External Memory Bandwidth

- The most important resource on the CELL?