Compiling for Heterogeneous
Multi-core CELL Processor

Yuan Zhao
Department of Computer Science
Rice University

Comp635
September 10, 2007
- 1 PPE (AltiVec), 8 SPE (vector) each with 256KB LS
- PPE responsible for fork-and-joining SPE threads
- Each SPE has its own address space and can only access data from LS, data transfer explicitly controlled through DMA
- When running at 3.2GHz, about 200GFlops for single precision floating points, 20GFlops for double precision

**Compilation challenges**
- Increasing on-chip parallelism: coarse and fine level
- Heterogeneity: strength of different components
- Explicitly managed memory hierarchy: LS utilization
Compilation Model

- Dependence-based automatic approach
  - Loop nests in regular applications
  - No parallelism directives/pragmas

- Source-to-source compiler, procedure outlining
  - Fortran programs, array syntax supported
  - Rewrite in C to utilize parallelization libraries, vector data types and intrinsics

- Parallelization, vectorization (PPE and SPE), data movement, data alignment, data reuse, synchronization
Outline

- Parallelism hierarchy: parallelization and vectorization
- Memory hierarchy: data movement
- Experimental results
- Conclusion
- Future work
Related Work on Parallel Code Generation

- Parallel programming languages and models
  - HPF, OpenMP, CoArray Fortran, UPC, MPI, HPCS languages

- Related work: user specified parallelism directives/pragmas
  - Eichenberger et al, IBM research compiler, OpenMP
    - CellSs, RapidMind, PeakStream, Sequoia, Charm++

- Related work: automatic parallelization and vectorization

- Our approach: dependence-based automatic code generation strategy
  - Loop nests in regular applications
  - No parallelism directives
Dependence-based Code Generation

- Parallelization and vectorization
  - A loop is parallelizable if it doesn't carry dependences
  - The innermost loop is short vectorizable if it doesn't carry dependence cycles and array references are memory contiguous
    - An anti-dependence on a statement itself is allowed
  - A short vectorizable loop can always be parallelized
    - Anti-dependence to be preserved by post-store and synchronization
  - Algorithm to identify a parallel loop and a short vectorizable loop
    - Dependences are carried at different levels
    - Search from the outermost loop towards the innermost loop
      - If the loop carries no dependence, mark parallel
      - Otherwise, make it sequential, remove the loop and all dependences it carries, repeat the search process
    - Check the innermost loop for short vectorizability
**Dependence-based Code Generation**

- **Example**
  
  ```
  DO K = 1, L
    DO J = 1, M
      DO I = 1, N
        A(I,J,K) = A(I,J,K+1) + A(I+1,J,K) + A(I+1,J+1,K)
      ENDDO
    ENDDO
  ENDDO
  ```

  - **K,J** sequential, with barrier synchronization
  - **I** parallel
    - iterations partitioned across PEs
    - anti-dependence: post-store and uni-directional synchronization
  - **I** vectorizable

  
  
  **K**  |  **J**  |  **I**  
  ---:|---:|---:|---:|
  <1  | =0  | =0  | anti
  0  | =0  | <1  | anti
  0  | <1  | <1  | anti
  
  **Step 0**

  
  **Step 1**

  
  **Step 2**

  
  - **K,J** sequential, with barrier synchronization
  - **I** parallel
    - iterations partitioned across PEs
    - anti-dependence: post-store and uni-directional synchronization
  - **I** vectorizable
Parallelizing Loop with Anti-dependence

- **Failsafe method**
  - easy CELLization
  - extra data transfer

- **Temporary allocation reduction**
  - Post-store, synchronization
  - Privatize temporary for each PE

SPE's copy can reside in LS only

\[
A(2:N-1) = B(1:N-2) + A(3:N)
\]

```fortran
DO I = 1, N-2     T(I) = B(I) + A(I+2)     ENDDO
DO I = 1, N-2     A(I+1) = T(I)               ENDDO
A(2:N-1) = B(1:N-2) + A(3:N)
```

```fortran
DMA_GET(b2, b3);
uni-directional_synch_notifynext;
for i = my_lb, my_ub
  b1(i)=b2(i)+b3(i);
bt(1:1) = b1(my_lb:my_lb)
dma_put(b1(my_lb+1:my_ub));
uni-directional_synch_waitprev;
dma_put(bt(1:1));
```
Dependence-based Code Generation

- **Synchronization**
  - Implemented with mailbox mechanism

- **Migrating to use direct SPE-to-SPE communication**
  - Designate one SPE as work group leader
Vector Instruction Set on PPE and SPE

- AltiVec on PPE, new SIMD on SPE
  - Limited vector length (16 bytes)
  - Contiguous memory access
  - Data alignment constraint

- Compilation challenges
  - Vectorization
    - Have as many array references aligned as possible

- Vectorization
  - Basic block based
    - S. Larsen and S. Amarasinghe
  - Loop based
    - Intel, IBM, VAST/AltiVec, GNU compiler
    - many work on conventional vector machines
  - Our approach: loop based
Vector Instruction Set on PPE and SPE.

- Data alignment
  - Related work: Loop peeling
    
    \[
    \text{DO } I = 1, N \\
    A(I) \\
    \text{ENDDO}
    \]
    
    \[
    \begin{array}{cccccccc}
    1 & 2 & 3 & 4 & 1 & 2 & 3 & 4 \\
    \end{array}
    \]
    
    \[
    \begin{array}{c}
    A(1) \\
    A(\text{PEEL})
    \end{array}
    \]
  
  - Related work: pipelined vector load and store
    - A. Eichenberger, P. Wu and K. O'Brien
  
  - Related work: vectorized scalar replacement
    - J. Shin, J. Chame and M. Hall
  
  - Our approach: additional method of loop alignment

PEEL = AlignSize(A(1))

DO I = 1, PEEL
  \[
  \begin{array}{c}
  A(I) \\
  \end{array}
  \]
  \text{ENDDO}

DO I = PEEL+1, N-VL+1, VL
  \[
  \begin{array}{c}
  A(\text{I:}\text{I+VL-1}) \\
  \end{array}
  \]
  \text{ENDDO}
Vector Instruction Set on PPE and SPE ..

- Loop alignment to expose more opportunities
  - Data alignment pivot
    - the array reference we pick to make aligned by loop peeling (the rest are either aligned or unaligned, accordingly)
    - \( \text{AlignSize}(A(I)) \): the number of iterations need to be peeled
      \[ 0 \leq \text{AlignSize}(A(I)) \leq VL-1 \]
    - Array references partitioned into alignment equivalent classes
      
      ```
      DO I = 1, N
      = A(I) + A(I+1) + A(I+VL)
      ENDDO
      ```

  - write pivot: pick one write reference in the class with most references
  - read pivot: pick one read reference in the class with most references
    - considering vectorized scalar replacement
Vector Instruction Set on PPE and SPE ...

- Conform write pivot and read pivot
  - if AlignSize(write pivot) ≠ AlignSize(read pivot), loop peeling fails
  - increase loop alignment distance to make two AlignSizes equal, note that original alignment distance is the minimum distance required to preserve the array syntax semantics
  - Example: \( A(2:N-1) = A(1:N-2) + A(3:N) \)

\[
\begin{align*}
T1 &= A(1) + A(3) \\
DO \ I &= 1, \ N-3 \\
T2 &= A(I+1) + A(I+3) \\
A(I+1) &= T1 \\
T1 &= T2 \\
ENDDO \\
A(N-1) &= T1
\end{align*}
\]

\[
\begin{align*}
(T1,T2,T3) &= A(1:3) + A(3:5) \\
DO \ I &= 1, \ N-3 \\
T4 &= A(I+3) + A(I+5) \\
A(I+1) &= T1 \\
T1 &= T2 \\
T2 &= T3 \\
T3 &= T4 \\
ENDDO \\
A(N-3:N-1) &= (T1,T2,T3)
\end{align*}
\]
Outline

- Parallelism hierarchy: parallelization and vectorization
- **Memory hierarchy: data movement**
- Experimental results
- Conclusion
- Future work
Related Work on Data Movement

- Explicit data transfer for SPE's LS
- Related work: software cache
  - Eichenberger et al, IBM compiler, cache lookup code before each reference, 64KB, 4-way, 128bytes cache line
- Related work: loop transformations
  - Eichenberger et al, loop blocking, prefetching
- Related work: array copying
  - Lam et al, reduce cache conflict misses in blocked loop nests
  - Temam et al, cost-benefit models for array copying
  - Yi, general algorithm with heuristics for cost-benefits analysis
  - ATLAS
- Related work: software prefetching
  - Callahan et al, Mowry et al, prefetch analysis, prefetch placement
Data Movement

- Multi-buffering to hide the latency of DMA transfers
  - All loop invariants are passed to SPE once at the beginning
  - Remaining array references are partitioned into reference groups
    - For each group leader, insert DMA data transfers at the innermost loop that the reference is variant, strip-mine the loop properly
    - For other references, perform vectorized scalar replacement to get value from the group leader
Data Movement

- Data alignment
  - Vector offset: last 4 bits of source and destination address be same
  - Cache line alignment
  - Naturally aligned
    - Get: over-fetch
    - Put: loop peeling on PPE

<table>
<thead>
<tr>
<th>0</th>
<th>16</th>
<th>32</th>
<th>48</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
</tr>
<tr>
<td>I</td>
<td>J</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DMA(&A, 4 bytes)
DMA(&B, 8 bytes)
DMA(&D, 16 bytes)
DMA(&H, 8 bytes)
DMA(&J, 4 bytes)
Data Movement..

- **Data alignment**
  - Vector offset: last 4 bits of source and destination address be same
  - Cache line alignment
  - Naturally aligned
    - Get: over-fetch
    - Put: loop peeling on PPE

```plaintext
DO I = 1, N
  A(I) = ...
ENDDO

PEEL = AlignSize(A(1))
DO I = 1, PEEL
  A(I) = ...
ENDDO
DO I = PEEL+1, N-VL+1, VL
  A(I:I+VL-1) = ...
ENDDO
```

DMA(&A, 4 bytes)
DMA(&B, 8 bytes)
DMA(&D, 16 bytes)
DMA(&H, 8 bytes)
DMA(&J, 4 bytes)
Outline

- Parallelism hierarchy: parallelization and vectorization
- Memory hierarchy: data movement
- Experimental results
- Conclusion
- Future work
### Experiments

- **3.2GHz CELL, 1GB memory, xlc compiler, f2c tool, SDK1.x, SDK2.0**

- **Stencils**

<table>
<thead>
<tr>
<th>1d:</th>
<th>( B(2:N-1) = (A(1:N-2) + A(3:N) + C(2:N-1)) \times 0.34 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2d:</td>
<td>( A(2:N-1,2:M-1) = A(2:N-1,2:M-1) + (B(1:N-2,2:M-1) + B(3:N,2:M-1)) + B(2:N-1,1:M-2) + B(2:N-1,3:M)) \times 0.25 )</td>
</tr>
<tr>
<td>anti:</td>
<td>( A(2:N-1) = (A(3:N) + B(2:N-1) + C(2:N-1)) \times 0.34 )</td>
</tr>
</tbody>
</table>
  | antitmp: | \( T(1:N-2) = (A(3:N) + B(2:N-1) + C(2:N-1)) \times 0.34 \)  
  | | \( A(2:N-1) = T(1:N-2) \) |

- **Swimfloat: swim in SPEC benchmark with float type**
  - 10 loop nests: \( L0...9 \), \{\( L0,1,2,7 \) once\}, \{\( L4,6,9 \), 1d boundary copy\}
  - \( L3 \): 2 level loop, 4 stencil statements; \( L5 \): 2 level loop, 3 stencil statements
  - \( L8 \): 2 level loop, 3 stencil statements, 3 array copy statements

- **Mgridfloat: mgrid in SPEC benchmark with float type**

- **Optimizations:** vectorized scalar replacement, software pipelined vector load/store, loop peeling on PPE, work load on PPE
Experiments.

- SPE faster than PPE (pipeline length, OS)
- Small vs big problem size (thread cost in SDK1.1, enough work on each PE)
- Loop peeling for DMA data alignment

Small: $N=7555333$  Big: $N=44222111$

Small: $M=N=2955$  Big: $M=N=7255$
Experiments..

- Reduced temporary array allocation leads to reduced data transfer.
Experiments ...

Swim problem size: 1335x1335, 1200 iterations (L3,5,8 collected with 20 iterations)

- Individual tuning of each loop nest
Reducing Thread Creation Cost

- Thread creation cost (SDK 1.x)
  - microseconds, SPE return immediately
  
<table>
<thead>
<tr>
<th>SPEs</th>
<th>1SPE</th>
<th>2SPEs</th>
<th>3SPEs</th>
<th>4SPEs</th>
<th>5SPEs</th>
<th>6SPEs</th>
<th>7SPEs</th>
<th>8SPEs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4259</td>
<td>7914</td>
<td>11428</td>
<td>14964</td>
<td>18915</td>
<td>22836</td>
<td>26642</td>
<td>30324</td>
</tr>
</tbody>
</table>

- Load all SPE function codes to SPE's LS at the beginning
  - SPE runs a dispatcher function that waits for a number indicating which function to execute
  - PPE sends a number to SPEs indicating which function to execute

- Will investigate thread cost and code overlay utility in CELL SDK 2.x
  - Designate one SPE as work group leader
Reducing Thread Creation Cost.
Conclusion

- A dependence-based automatic code generation scheme
  - No parallelism directives/pragmas
  - Parallelism hierarchy and memory hierarchy
  - Parallelization, vectorization, multi-buffering data transfer, loop peeling on PPE, workload on PPE, data alignment on PEs, temporary allocation reduction, reduced lazy loop alignment, synchronization

- A source-to-source compiler
  - Procedure-outlines Fortran loop nest and rewrites in C to utilize parallelization libraries and vector data types and intrinsics
  - Adds support to vector instruction set on SPE

- Compilers developed for short vector processors can be extended for multi-core processors
Ongoing and Future Work

- More optimizations for memory hierarchy on CELL
  - On-chip data reuse to improve computation to data transfer ratio
  - Inside each SPE's LS
  - Across SPEs' LS

- More parallelization schemes
  - Across one loop's iteration space
  - Across multiple loop nests

- Load balancing
  - Performance modeling
  - Runtime scheduling

- Loop fusion and synchronization

- Longer term
  - Multiple CELL chips: programming model, memory hierarchy
  - CELL chips with other chips: heterogeneous computing
Thank you!