Outline

- Discussion of two papers
- Acknowledgments
  • PPoPP 2007 presentation from Filip Blagojevic
  • PACT 2005 and other presentations from Alex Eichenberger
“Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture”.


- Discussion will focus on two compiler optimizations areas that are important for Cell SPEs and related accelerators
  1. Instruction optimizations: branch hints, bundler, instruction fetch
  2. SIMDization

Motivation for Instruction Optimizations: Software-Assisted Branch Architecture

- Branch architecture
  - no hardware branch-predictor, but
  - compare/select ops for predication
  - software-managed branch-hint
  - one hint active at a time

- Lowering overhead by
  - predicating small if-then-else
  - hinting predictably taken branches
Branch Hints

- SPE provides a HINT operation
  - fetches the branch target into HINT buffer
  - no penalty for correctly predicted branches
  - compiler inserts hints when beneficial

Motivation for Instruction Fetch Optimizations

- Local store is single ported
  - less expensive hardware
  - asymmetric port
    - 16 bytes for load/store ops
    - 128 bytes for IFETCH/DMA
  - static priority
    - DMA > MEM > IFETCH

- If we are not careful, we may starve for instructions
Instruction Starvation Situation

- There are 2 instruction buffers
  — up to 64 ops along the fall-through path

- First buffer is half-empty
  — can initiate refill

- When MEM port is continuously used
  — starvation occurs (no ops left in buffers)

Instruction Starvation Prevention

- SPE has an explicit IFETCH op
  — which initiates a instruction fetch

- Scheduler monitors starvation situation
  — when MEM port is continuously used
  — insert IFETCH op within the (red) window

- Compiler design
  — scheduler must keep track of code layout
SPE Instruction Optimization Results

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Huffman</th>
<th>FFT</th>
<th>IDEA</th>
<th>LU</th>
<th>MD</th>
<th>Unlock</th>
<th>Convolution</th>
<th>OnerayXY</th>
<th>Mat Mult</th>
<th>Saxpy</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relative reductions in execution time</td>
<td>0.40</td>
<td>0.50</td>
<td>0.60</td>
<td>0.70</td>
<td>0.80</td>
<td>0.90</td>
<td>1.00</td>
<td>0.78</td>
<td>0.78</td>
<td>0.78</td>
<td>0.78</td>
</tr>
</tbody>
</table>

single SPE performance, optimized, simdized code (avg 1.00 → 0.78)

Integrated SIMDization Framework

Extract Parallelism

- Loop level
  - for (i=0; i<256; i++)
    - a[i] =

- Basic-block level
  - a[i+0] =
  - a[i+1] =
  - a[i+2] =
  - a[i+3] =

- Entire short loop
  - for (i=0; i<8; i++)
    - a[i] =

Satisfy Constraints

- Alignment constraints
- Data size conversion
- Multiple targets
- Non stride-one

COMP 635, Fall 2007 (V. Sarkar)
Key Abstraction: Virtual SIMD Vector

- **Virtual SIMD Vector**
  - has arbitrary length
  - has no alignment constraints

- **Extraction of SIMD Parallelism**
  - use virtual vector as representation
  - abstract away all the hardware complexity

- **Progressive “de-virtualization” of the virtual vector**
  - until each vector in the loop satisfies all hardware constraints
  - or revert vectors back to scalars (if too much overhead)

---

Basic-Block & Loop Level Aggregation

Original loop

```
for (i=0; i<256; i++) {
    a[i].x = a[i].y = a[i].z = b[i+1] =
}
```

Value streams

```
a0.x a0.y a0.z a1.x a1.y a1.z a2.x a2.y a2.z a3.x a3.y ...
```

4 iterations shown here for the purpose of illustration

- 1st iter
- 2nd iter
- 3rd iter
- 4th iter
Phase 1: Basic-Block Level Aggregation

Original loop

for (i=0; i<256; i++) {
    \(a[i].x = a[i].y = a[i].z = b[i+1] = \)
}

Value streams

\[
\begin{align*}
\text{a0.x} & \text{ a0.y} & \text{ a0.z} & \text{ a1.x} & \text{ a1.y} & \text{ a1.z} & \text{ a2.x} & \text{ a2.y} & \text{ a2.z} & \text{ a3.x} & \text{ a3.y} & \text{ a3.z} & \text{ a4.x} & \text{ a4.y} & \text{ a4.z} & \cdots
\end{align*}
\]

Basic-Block level aggregation

- pack \(a[i].x, a[i].y, a[i].z\) into a vector of 3 elements
- pack regardless of alignment

Phase 2: Loop-Level Aggregation

BB-aggregated loop

\[
\begin{align*}
\text{for (i=0; i<256; i++)} \{ \\
\text{\(a[i].x, a[i].y, a[i].z\) =} \\
\text{\(b[i+1] = \)} \\
\text{\}
\end{align*}
\]

Value streams

\[
\begin{align*}
\text{a0.x} & \text{ a0.y} & \text{ a0.z} & \text{ a1.x} & \text{ a1.y} & \text{ a1.z} & \text{ a2.x} & \text{ a2.y} & \text{ a2.z} & \text{ a3.x} & \text{ a3.y} & \text{ a3.z} & \text{ a4.x} & \text{ a4.y} & \text{ a4.z} & \cdots
\end{align*}
\]

Loop-level aggregation

- pack each statement with itself across consecutive iterations
- final vector lengths must be multiple of 16 bytes
- scalar “\(b[i]\)” or vector “\((a[i].x,y,z)\)” are treated alike
- pack regardless of alignment
Phase 3: Alignment Devirtualization

**Loop-aggregated**

```c
for (i=0; i<256; i+=4) {
    (a[i].x,...,a[i+3].z) = (b[i+1],...,b[i+4]) =
}
```

**Value streams**

- `a0.x, a0.y, a0.z, a1.x, a1.y, a1.z, ...`
- `b1, b2, b3, b4, ...`

**Alignment** *

- shift misaligned streams
- skew the computations so that loop computes \((b[i+4]...b[i+7])\)

* Arrays (e.g. &a[0], &b[0],...) are assumed here 16-byte aligned.

Phase 4: Length Devirtualization

**Aligned loop**

```c
(b[1]...b[3]) =
for (i=0; i<252; i+=4) {
    (a[i].x,...,a[i+3].z) = (b[i+4],...,b[i+7]) =
} (a[252].x,...,a[255].z) = b[256] =
```

**Value streams**

- `a0.x, a0.y, a0.z, a1.x, a1.y, a1.z, ...`
- `b1, b2, b3, b4, ...`

**Length**

- break into 16-byte chunks
Compiler Designers’ Notes

- Managing complexity includes
  - interactions among different phases within simdization
  - Interactions between simdization and other compiler optimizations

- Internal representation (IR)
  - virtual-length vectors capture effects of different aggregation phases
  - using generic operations in IR helps supporting multiple platforms

- Auxiliary analysis & transformations also important
  - Alignment analysis, pointer analysis, dependence analysis
  - Redundant conversion elimination
  - Data layout optimization for alignment and stride-one access

- Simdization profitability
  - not a big concern for Cell since scalar codes are much slower
  - very important for other arch with fast scalar units (e.g., VMX)
  - simdization may generate codes less easy to be optimized

Simdization Results

<table>
<thead>
<tr>
<th>single SPE, optimized, automatic simdization vs. scalar code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linpack</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>2.4</td>
</tr>
</tbody>
</table>

Speedup factors