Optimizing Compiler for the Cell Processor

Alexandre Eichenberger, Kathryn O’Brien, Kevin O’Brien, Peng Wu, Tong Chen, Peter Oden, Daniel Prener, Janice Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, and Michael Gschwind

www.research.ibm.com/cellcompiler/compiler.htm
Cell Broadband Engine

- **Multiprocessor on a chip**
  - 241M transistors, 235mm²
  - 200 GFlops (SP) @3.2GHz
  - 200 GB/s bus (internal) @ 3.2GHz

- **Power Proc. Element (PPE)**
  - general purpose
  - running full-fledged OSs

- **Synergistic Proc. Element (SPE)**
  - optimized for compute density
Cell Broadband Engine Overview

- **Heterogeneous, multi-core engine**
  - 1 multi-threaded power processor
  - up to 8 compute-intensive-ISA engines

- **Local Memories**
  - fast access to 256KB local memories
  - globally coherent DMA to transfer data

- **Pervasive SIMD**
  - PPE has VMX
  - SPEs are SIMD-only engines

- **High bandwidth**
  - fast internal bus (200GB/s)
  - dual XDR™ controller (25.6GB/s)
  - two configurable interfaces (76.8GB/s)
  - numbers based on 3.2GHz clock rate
Supporting a Broad Range of Expertise to Program Cell

Highest performance with help from programmers

- Multiple-ISA hand-tuned programs
- Explicit SIMD coding
- Explicit parallelization with local memories
  - SIMD
  - SIMD/alignment directives
  - Shared memory, single program abstraction

Highest Productivity with fully automatic compiler technology
Outline

Part 1: Automatic SPE tuning
Multiple-ISA hand-tuned programs
Automatic tuning for each ISA

Part 2: Automatic simdization
Explicit SIMD coding
SIMD/alignment directives

Part 3: Shared memory & single program abstr.
Explicit parallelization with
local memories
Shared memory, single program abstraction
Automatic parallelization
Architecture: Relevant SPE Features

Synergistic Processing Element (SPE)

- **SIMD-only functional units**
  - 16-bytes register/memory accesses

- **Simplified branch architecture**
  - no hardware branch predictor
  - compiler managed hint/predication

- **Dual-issue for instructions**
  - full dependence check in hardware
  - must be parallel & properly aligned

- **Single-ported local memory**
  - aligned accesses only
  - contentions alleviated by compiler

---

Even Pipe Floating/ Fixed Point
Odd Pipe Branch Memory Permute

- Dual-Issue Instruction Logic

- Instr.Buffer (3.5 x 32 instr)

- Register File (128 x 16Byte register)

- Local Store (256 KByte, Single Ported)

- DMA (Globally-Coherent)

- branch: 1,2
- branch hint: 1,2
- instr. fetch: 2
- dma request: 3

---

8 bytes (per dir)  16 bytes (one dir)  128 bytes (one dir)
Feature 1: Single-Ported Local Memory

- 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

Diagram:
- SPE
- Even Pipe Floating/Fixed Point
- Odd Pipe Branch Memory Permute
- Dual-Issue Instruction Logic
- Instr. Buffer (3.5 x 32 instr)
- Register File (128 x 16Byte register)
- Local Store (256 KByte, Single Ported)
- DMA (Globally-Coherent)
- Ifetch
- branch: 1,2 branch hint: 1,2 instr. fetch: 2 dma request: 3

- 8 Bytes (per dir)
- 16Bytes (one dir)
- 128Bytes (one dir)
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
Feature #2: 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**
  - predicing small if-then-else
  - hinting predictably taken branches
Hinting Branches & Instruction Starvation Prevention

- **SPE provides a HINT operation**
  - fetches the branch target into HINT buffer
  - no penalty for correctly predicted branches

-Compiler inserts hints when beneficial

**Impact on instruction starvation**
- after a correctly hinted branch, IFETCH window is smaller

**HINT** br, target
 fetches ops from target; needs a min of 15 cycles and 8 intervening ops
SPE Optimization Results

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

Part 1: Automatic SPE tuning
- Multiple-ISA hand-tuned programs
- Automatic tuning for each ISA

Part 2: Automatic SIMDization
- Explicit SIMD coding
- SIMD/alignment directives

Part 3: Shared memory & single program abstr.
- Explicit parallelization with local memories
- Shared memory, single program abstraction
- Automatic parallelization
Single Instruction Multiple Data (SIMD) Computation

Process multiple “b[i]+c[i]” data per operations

16-byte boundaries

R3

16-byte boundaries

R2

R1

Cell Broadband Engine
Successful Simdization

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

GENERIC

VMX

SPE

Optimizing Compiler for a Cell Processor
Alexandre Eichenberger
Example of SIMD-Parallelism Extraction

- **Loop level**
  - SIMD for a single statement across consecutive iterations
  - successful at:
    - efficiently handling misaligned data
    - pattern recognition (reduction, linear recursion)
    - leverage loop transformations in most compilers

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

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

References:
- Bik et al, IJPP 2002
- VAST compiler, 2004
- Eichenberger et al, PLDI 2004
- Wu et al, CGO 2005
- Naishlos, GCC Developer’s Summit 2004
Example of SIMD Constraints

- **Alignment in SIMD units matters:**
  - consider “b[i+1] + c[i+0]”

alignment constraints

16-byte boundaries

vload b[1]

this is not b[1] + c[0]

multiple targets

data size conversion

16-byte boundaries
Example of SIMD Constraints (cont.)

- **Alignment in SIMD units matters**
  - when alignments within inputs do not match
  - must realign the data

```
example
```

```
load b[i]
load a[i]
unpack
add
store
load a[i+4]
unpack
add
store
SHORT INT 2 INT 1
data size conversion
b0
b1
b2
b3
b4
b5
b6
b7
16-byte boundaries
vload b[1]

vpermute

b1
b2
b3
b4
R1

R2
c0
c1
c2
c3
c0
c1
c2
c3
c4
c5
c6
c7
16-byte boundaries

alignment constraints

16-byte boundaries

data size conversion

alignment constraints

alignment constraints

multiple targets

GENERIC VMX SPE
Automatic Simdization for Cell

- **Integrated Approach**
  - extract at multiple levels
  - satisfy all SIMD constraints
  - use “virtual SIMD vector” as glue

- **Minimize alignment overhead**
  - lazily insert data reorganization
  - handle compile time & runtime alignment
  - simdize prologue/epilogue for SPEs
    - memory accesses are always safe on SPE

- **Full throughput computations**
  - even in presence of data conversions
  - manually unrolled loops...
SPE Simdization Results

single SPE, optimized, automatic simdization vs. scalar code
Outline

Part 1: Automatic SPE tuning
Multiple-ISA hand-tuned programs

Part 2: Automatic simdization
Explicit SIMD coding
SIMD/alignment directives

Part 3: Shared memory & single program abstr.
Explicit parallelization with local memories
Shared memory, single program abstraction

Part 3: Shared memory & single program abstr.

Automatic tuning for each ISA
Automatic simdization
Automatic parallelization
Cell Memory & DMA Architecture

- **Local stores are mapped in global address space**

- **PPE**
  - can access/DMA memory
  - set access rights

- **SPE can initiate DMAs**
  - to any global addresses,
  - including local stores of others.
  - translation done by MMU

- **Note**
  - all elements may be masters, there are no designated slaves
Competing for the SPE Local Store

Local store is fast, need support when full.

Provided compiler support:

- **SPE code too large**
  - compiler partitions code
  - partition manager pulls in code as needed

- **Data with regular accesses is too large**
  - compiler stages data in & out
  - using static buffering
  - can hide latencies by using double buffering

- **Data with irregular accesses is present**
  - e.g. indirection, runtime pointers...
  - use a software cache approach to pull the data in & out (last resort solution)
Software Cache for Irregular Accesses

- **Data with irregular accesses**
  - cannot reside permanently in the SPE’s local memory (typically)
  - thus reside in global memory
  - when accessed,
    - must translate the global address into a local store address
    - must pull the data in/out when it's not already present

- **Use a software cache**
  - managed by the SPEs in the local store
  - generate DMA requests to transfer data to/from global memory
  - use 4-way set associative cache to naturally use the SIMD units of the SPE
Software Cache Architecture

- **Cache directory**
  - 128-set, 4-way set associative
  - pointers to data lines
  - use 16KByte of data

- **Data in a separate structure**
  - 512 x 128B lines
  - use 64KByte of data
Software Cache Access

extract & load set

set tags
0 a2c2e3h4
1 b2c3f4i3
... c4f6a1j5

data pointers

dirty bits...

compute addr

translate this global address

splat addr

addr offset

addr a1

subset of addr used by tags

compare '='

compare a1a1a1a1

SIMD comparator

hit

locate data ptr.

data array

d1...
d2 42...
dn...

when successful

hit latency: ~ 20 extra cycles
"Single Source" Compiler, using OpenMP

```c
#include <omp.h>

void foo()
{
#pragma omp parallel for
for(i = 0; i < 10000; i++)
    A[i] = x * B[i];
}

void foo_PPE()
{
    init_omp_rte();
#pragma omp parallel for
for(i = LB; i < UB; i++)
    A[i] = x * B[i];
}

void foo_SPE()
{
    for(k = LB; k < UB; k++)
        DMA 100 B elements into B´
    for(j = 0; j < 100; j++)
        A´[j] = cache_lookup(x) * B´[j];
    DMA 100 A elements out of A´
}
```

Running time:
- initialize OpenMP runtime
- compute its own work

Runtime:
- DMA in/out array, lookup software cache
- compute its own work
Results for Swim, Mgrid, & some of their kernels

- Softcache
- Optimization

Swim, Calc1, Calc2, Calc3, Mgrid, Resid, Psinv, Rprj3

Baseline: execution on one single PPE
Conclusions

- **Cell Broadband Engine architecture**
  - heterogeneous parallelism
  - dense compute architecture

- **Present the application writer with a wide range of tool**
  - from support to extract maximum performance
  - to support to achieve maximum productivity with automatic tools

- **Shown respectable speedups**
  - using automatic tuning, simdization, and support for shared-memory abstraction
Questions

For additional info:

www.research.ibm.com/cellcompiler/compiler.htm
Extra
Compiler Support For Single Source Program