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
Cell Processor Components

Power Processor Element (PPE)

- Industry-standard 64-bit IBM Power Architecture™ processor
  - PowerPC AS 2.0.2
- 2-Way Hardware Multithreaded
- L1: 32KB I; 32KB D
- L2: 512KB
- Coherent load/store
- VMX
- 3.2+ GHz
- Realtime Control
  - Locking L2 Cache & TLB
  - Bandwidth Reservation

In the Beginning
- the Power Architecture™ Processor

Custom Designed
- for high frequency, area and power efficiency
Cell Processor Components

Element Interconnect Bus
- data ring for internal communication
  - Four 16 byte data rings, supporting multiple transfers
  - 96B/cycle peak bandwidth
  - Over 100 outstanding requests
  - 200+ GByte/s @ 3.2+ GHz

96 Byte/Cycle  200+GB/sec @ 3.2+GHz
NCU
Power Core (PPE)
L2 Cache
Element Interconnect Bus
Cell Processor Components

**SPE provides computational performance**
- Dual issue, up to 8-way 32-bit SIMD
- Dedicated resources
  - 128-entry 128-bit VRF
  - 256KB Local Store
- Each SMF can be dynamically configured to protect resources
- Dedicated DMA engine
  - Up to 16 outstanding requests
Cell Processor Components

**SMF provides memory management & mapping**
- SPE Local Store aliased into system memory map
- SMF controls SPE DMA accesses
  - Compatible with Power Architecture™ Virtual Memory architecture
  - S/W controllable from PPE MMIO
- DMA 1,2,4,8,16,128 B ⇒ 16Kbyte transfers for I/O access
- SPE DMA access protected by SMF
  - Based on Power Architecture™ system memory map
Cell Processor Components

I/O provides wide bandwidth

- Dual XDR™ controller
  - 25.6GB/s @ 3.2Gbps
- Two configurable interfaces
  - 76.8GB/s @ 6.4Gbps
  - Configurable number of Bytes
  - Coherent or I/O Mode Interconnect
- Supports multiple system configurations
Cell Processor Components

IIC – Internal Interrupt Controller
- Handles SPE Interrupts
- Handles External Interrupts
  - From Coherent Interconnect
  - From IOIF0 or IOIF1
- Interrupt Priority Level Control
- Interrupt Generation ports for IPI
- Duplicated for each PPE hardware thread
Cell Processor Components

**IOT implements I/O Bus Master Translation**
- Translates bus address to system address
- Two Level translation
  - I/O Segments: 256 MB
  - I/O Pages: 4KB, 64KB, 1MB, 16MB
- I/O Device Identifier / page for LPAR
- IOST and IOPT Cache
  - hardware/software managed

---

![Diagram of Cell Broadband Engine](image)
Cell Processor Components

Token Manager provides Bandwidth Reservation for shared resources
- Optionally used for RT tasks or LPAR
- Multiple Resource Allocation Groups
- Generates access tokens at configurable rate for each allocation group
  - 1 per each memory bank (16 total)
  - 2 for each IOIF (4 total)
- Requestors assigned RAG ID by OS/hypervisor
  - Each SPE
  - PPE L2 / NCU
  - IOIF 0 Bus Master
  - IOIF 1 Bus Master
- Priority order for using another RAGs unused tokens
- Resource overcommit warning interrupt
Supporting a Broad Range of Expertise to Program Cell

Highest performance with help from programmers

<table>
<thead>
<tr>
<th>Programs</th>
<th>SIMD</th>
<th>Parallelization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple-ISA hand-tuned</td>
<td>Explicit SIMD coding</td>
<td>Explicit parallelization with local memories</td>
</tr>
<tr>
<td>programs</td>
<td>SIMD/alignment directives</td>
<td>Shared memory, Single program abstraction</td>
</tr>
<tr>
<td>Automatic tuning for each ISA</td>
<td>Automatic simdization</td>
<td>Automatic parallelization</td>
</tr>
</tbody>
</table>

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

Outline
SPE’s Functional Units are SIMD Only

- **Functional units are SIMD only**
  - all transfers are 16 Bytes wide,
  - including register file and memory

- **How to handle scalar code?**
Single Instruction Multiple Data (SIMD)

Meant to process multiple “b[i]+c[i]” data per operations

16-byte boundaries

memory stream

registers

16-byte boundaries

memory stream

VADD
Scalar code on Scalar Functional Units


```
LOAD b[1]
LOAD c[3]
ADD
STORE a[2]
```
Scalar Code on SIMD Functional Units


- Problem #1: Memory alignment defines data location in register
- Problem #2: Adding aligned values yield wrong result
- Problem #3: Vector store clobbers neighboring values
Scalar Load Handling

- **Use read-rotate sequence**
  
  ![Diagram showing scalar load handling with read-rotate sequence]

- **Overhead (1 op, in blue)**
  - one quad-word byte rotate

- **Outcome**
  - desired scalar value always in the first slot of the register
  - this addresses Problems 1 & 2
Scalar Store Handling

- Use read-modify-write sequence

  ![Diagram](image)

  - Overhead (1 to 3 ops, in blue)
    - one shuffle byte, one mask formation (may reuse), one load (may reuse)

  - Outcome
    - SIMD store does not clobber memory (this addresses Problem 3)
Optimizations for Scalar on SIMD

- Significant overhead for scalar load/store can be lowered

- For vectorizable code
  - generate SIMD code directly to fully utilize SIMD units
  - done by expert programmers or compilers (see SIMD presentation)

- For scalar variable
  - allocate scalar variables in first slot, by themselves
  - eliminate need for rotate when loading
    - data is guaranteed to be in first slot (Problems 1 & 2)
  - eliminate need for read-modify-write when storing
    - other data in 16-byte line is garbage (Problem 3)
SPE Features Optimized for by the Compiler

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
SPE Optimization Results

The diagram shows the relative reductions in execution time for various algorithms on the Cell Broadband Engine. The x-axis represents different algorithms: Huffman, FFT, IDEA, LU, VLD, Linpack, Convolution, OnerayXY, Mat Mult, Saxpy, and Average. The y-axis represents the relative reductions in execution time.

The bars indicate the improvements achieved by different optimization techniques:
- Original
- +Bundle
- +Branch Hint
- +Ifetch

The bar colors correspond to these techniques. For example, Huffman shows a significant reduction in execution time across all optimization techniques compared to the original version.

The graph indicates that optimizing compiler techniques for the Cell Broadband Engine can lead to substantial improvements in performance, with an average reduction of 1.00 to 0.78.

The text box at the bottom of the image states: "single SPE performance, optimized, simdized code (avg 1.00 → 0.78)"
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
Successful Simdization

Extract Parallelism

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

16-byte boundaries

data size conversion

multiple targets

GENERIC

VMX

SPE
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

- 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] =

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 in SIMD units matters:

1. **Consider “b[i+1] + c[i+0]”**

   - **vload b[1]**

   - **R1**
     - b0 b1 b2 b3

   - **R2**
     - c0 c1 c2 c3

   - **b0 + b1**

   - **+**

   - **+**

   - This is not b[1] + c[0]

   - **16-byte boundaries**

   - **data size conversion**

   - **vmx**

   - **spear**

   - **Generic VMX**

   - **SPE**

   - **Multiple targets**

   - **alignment constraints**

   - 16-byte boundaries

   - **load b[1]**

   - **unpacked**

   - **add**

   - **store**

   - INT 1

   - **load b[i]**

   - **short**

   - **unpacked**

   - **add**

   - **store**

   - INT 2

   - **load ajl**

   - **unpack**

   - **add**

   - **store**

   - **load ajl+4**

   - **add**

   - **store**

   - **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 of SIMD Constraints (cont.)

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

```
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...
A Unified Simdization Framework

**Global information gathering**
- Pointer Analysis
- Alignment Analysis
- Constant Propagation

**General Transformation for SIMD**
- Dependence Elimination
- Data Layout Optimization
- Idiom Recognition

**Simdization**
- Straightline-code Simdization
- Loop-level Simdization

**SIMD Intrinsic Generator**
- Architecture independent
- Architecture specific

**Diagnostic output**
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

PROGRAMS
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

- Local store 1

- ALIAS TO

- Local store 8

- TLBs
- MFC Regs

- QoS* / L3*

- IO Devices

- Memory requests
- DMA
- Mapped

* external

© 2006 IBM Corporation
Anatomy of a Cell Program

A. Invoke PPE program
   A1. Invoke thread lib to start threads
   A2. Load SPE code “loop1” and initiate
   A3. Wait for SPE to finish

B. SPE code “loop(lb, ub)
   B1. dma_get B,C[lb : ub];
   B2. for ( i=lb; i<ub; i++)
       A[i] = B[i] + C[i];
   B3. dma_put A[lb : ub];

for (i=0; i<10K; i++)
   A[i] = B[i] + C[i]
Manual Compilation of a Cell Program

PPE/SPE Progs. Manual Compiling & Binding Executable

Compiling Multiple ISA Programs into a single unit

A1. Invoke thread to start threads
A2. Load SPE code "loop1" and initiate
A3. Wait for SPE to finish

B1. dma_get B[C[i]; ub];
B2. for (i=lb; i<ub; i++)
B3. dma_put A[ib; ub];

PPE Source → SPE Compiler → SPE Object
SPE Source → SPE Compiler → SPE Object
SPE Libraries → SPE Linker → SPE Exec → SPE Embedder
PPE Source → PPE Compiler → PPE Object
PPE Source → PPE Compiler → PPE Object
PPE Libraries → PPE Linker → PPE Executable

PPE Linker
PPE code
SPE Linker
SPE code
PPE program
Data
Memory Image

Highest performance: use platform specific intrinsics
Higher productivity using automatic simdization compiler
"Single-Source" Compilation of a Cell Program

Single Source Prog.  Automatic Compiling & Binding  Executable

Highest productivity using single-source compiler
Parallelization and Partitioning Approach

- Parallel systems present a complex programming task

- Heterogeneous parallel systems increase the complexity because …

- Must consider processor characteristics in addition to parallelism
  - different processor performance
  - different system services (no O/S on SPEs)
  - different memory latencies
  - small local memories
  - to what extent is the PPE involved?
  - concept that PPE is only a service provider, most computation takes place on SPE
Our Parallel Implementation

- Avoid pitfalls of fully automatic approach by ...

- Allowing user to guide partitioning with directives, initially OpenMP
  - reasonable acceptance of OpenMP in the parallel programming community
  - directives and clauses easily mapped to SPE parallel execution
  - parallel_for, parallel_sections (possible pipelining functionality), parallel regions etc
  - relaxed-consistency memory-model incorporated into software cache approach

- Provide solution for C and Fortran

- Sidesteps need to partition for code size, at least initially
  - since most loop bodies typically fit within SPE storage

- Roadmap to fully automatic approaches
  - can piggyback on compilers that automatically generate OpenMP directives
  - XL compiler has an initial version of automatic OpenMP directive generation
“Single-Source” Compiler

- User prepares an application as a collection of one or more source files containing user directives/pragmas

- Compiler uses directives to partition code between PPE and SPE

- Compiler handles data transfers.
  - identify accesses in SPE functions that refer to data in system memory locations
  - use static buffers or software cache to transfer his data to/from SPE local stores

- Compiler handles code size
  - explore extending Code partitioning to Single Source, i.e. automatic partitioning based on functionality rather than size
Single-Source Compiler using OpenMP pragma

master thread for PPE

...  
  omp_rte_init();  
  omp_rte_do_par(ol$1);  
...

master thread

...  
  void ol$1_PPE(LB, UB)  
  for( i=LB;  i<UB;  i++)  
    A[i] = B[i] + D[C[i]] ;
...

PPE worker thread (optional)

void ol$1_PPE(LB, UB)  
  for( i=LB;  i<UB;  i++)  
    A[i] = B[i] + D[C[i]] ;

SPE worker threads

...  

clone for PPE

void ol$1()  
  for( i =0; i<10000; i++)  
    A[i] = B[i] + D[C[i]] ;

clone for SPE

...  

 outlined work

...  

#pragma Omp parallel for
  for( i =0; i<10000; i++)
    A[i] = B[i] + D[C[i]] ;

...  

single source

outline parallel region

void ol$1()  
  for( i =0; i<10000; i++)  
    A[i] = B[i] + D[C[i]] ;

void ol$1_SPE(LB, UB)  
  for( k=LB;  k<UB;  k+=100)  
    DMA 100 B,D elements into B',C'  
    for ( i=0; i<100; i++)  
      A'[i] = B'[i] + cache_lookup(D[C'[i]]);

  DMA 100 A elements out of A'
Outlining a Parallel Region

- **Creates a new architecture-independent function**
  - step proceeds in Pass 1 of TPO
  - add an extra node in the call graph
  - cannot have jumps outside region

- procedures are nested to access shared variables
- memory alias sets must accurately represent use/defs
- private variables are transformed to automatic in the new procedure; must transfer alias relationships
Cloning and Specializing a Parallel Region

- **Create new architecture-dependent functions:**
  - step proceeds in Pass 2 of TPO (uses whole program call graph)
  - outlined function is cloned and then specialized to create a PPE and an SPE version

- **All called functions must also be cloned**
  - SPE call sites modified to call SPE versions of cloned subroutines

- **Compiler creates SPE and PPE partitions**
  - invokes lower-level optimizer for each partition to perform machine specific optimization
PPE Runtime

- First OMP construct initializes the runtime system
  - create SPE threads and loads the SPE runtime
  - create work queue and get DMA queue addresses
  - send address of work queue to each SPE
  - set global options

- Sends a “setup_done” to SPEs after partitioning/scheduling the work items

- Calls PPE outlined procedures for its own work share.

```c
void ol$1_PPE(LB, UB)
for( i=LB;  i<UB;  i++)
A[i] = B[i] + D[ C[i] ];
```

... `omp_rte_init();`  `omp_rte_do_par(ol$1);` ...

*master thread*
**SPE Runtime**

- **Infinite loop while waiting for signals from PPE runtime**
- **DMA-fetches work items from work queue in system memory**

- **Depending on the work type:**
  - translates the address of SPE outlined procedure from PPE outlined procedure
  - invokes SPE outlined procedures.

![Loop continuously looking for work](loopDiagram.png)
Runtime Interaction

PPE RUNTIME
- Partitioning
- Scheduling
- Synchronization
- Communication

SPE RUNTIME
- Perform work items
- Communication

void ol$1_PPE(LB, UB)
for( i=LB; i<UB; i++)
A[i] = B[i] + D[C[i]];

PPE worker thread

void ol$1_SPE(LB, UB)
for( k=LB; k<UB; k+=100)
DMA 100 B,D elements into B',C'

for( i=0; i<100; i++)
A'[i] = B'[i] + cache_lookup(D[C'[i]]);

DMA 100 A elements out of A'

master thread
Limitations

- No nested parallelism
- Parallel constructs with system calls serialized (execute on PPE)
- Aggregated SPE binary not partitioned yet
- Some less-frequently used features under development
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)
Accessing Global Variables in System Memory

float x,y,z;
void main()
{
x=y+z;
}

start spe_thread
wait

call dma_get t.x
load r100=.y
load r101=t.y
load r102=.z
call dma_get t.z
add r104=r101
store t.x=r104

load r105=.x
load r106=r104
load r107=.z
load r108=r102
load r109=.y
load r1010=r103
add r111=r106
add r112=r109
add r113=r107
add r114=r1010
add r115=r111
add r116=r112
add r117=r113
add r118=r114
store t.x=r118
store t.y=r117
store t.z=r116

start spe_thread
wait
Hiding Communication using Double Buffering

**Original Code**

```c
for (i=0; i<100000; i++)
    A[i] = B[i] + C[i];
```

**Single Buffering**

```c
for (i=0; i<100000; i+=100)
    dma_get(B', B[i], 400);
    dma_get(C', C[i], 400);
    for (ii=0; ii<100; ii++)
        A'[ii] = B'[ii] + C'[ii];
    dma_put(A[i], A', 400);
```

**Double Buffering**

```c
dma_get(B', B[0], 400);
dma_get(C', C[0], 400);
for (i=0; i<99800; i+=200)
    dma_get(B", B[i+100], 400);
    dma_get(C", C[i+100], 400);
    for (ii=0; ii<100; ii++)
        A'[ii] = B'[ii] + C'[ii];
    dma_put(A[i], A', 400);
    dma_get(B', B[i+200], 400);
    dma_get(C', C[i+200], 400);
    for (ii=100; ii<200; ii++)
        A"[ii] = B"[ii] + C"[ii];
    dma_put(A[i+100], A", 400);
    for (ii=0; ii<100; ii++)
        A'[ii] = B'[ii] + C'[ii];
    dma_put(A[i+99900], A', 400);
```

Communication is blocked (100 elements at a time)

Computation and communication overlap as their phases are software pipelined
Handling Irregular Accesses using Software Cache

Original Code

\[
\text{for}(i=0; i<100000; i++)
\]
\[
= \ldots D[ C[i] ];
\]

Code with explicit Cache Lookup

\[
\text{for}(i=0; i<100000; i++)
\]
\[
t=\text{cache\_lookup}( D[ C[i] ]);\]
\[
= \ldots t;
\]

Code Lookup Sequence

\[
\text{inline vector cache\_lookup (addr)}
\]
\[
\text{if (cache\_directory[addr&key\_mask] != (addr&tag\_mask))}
\]
\[
\text{miss\_handler(addr);}\]
\[
\text{return cache\_data[addr&key\_mask][addr+offset\_mask];}
\]

miss handler DMA the required data, and some suitable quantity of surrounding data
higher degrees of associativity can be supported, for little extra cost on a SIMD processor
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

Diagram:
- Set tags
- Data pointers
- Dirty bits
- Data array
- Data points to data lines
- Use 16KByte of data
- Use 64KByte of data
Software Cache Access

set tags
data ptrs.
dirty bits...
0 a2 c2 e3 h4
1 b2 c3 f4 i3
...
x c4 f6 a1 j5 d2

compute addr
data array
d1 d2 dn

extract & load set

translate this global address
addr a1
subset of addr used by tags
splat addr
addr offset
a1 a1 a1 a1
compare ‘=‘

when successful
hit latency: ~ 20 extra cycles
locate data ptr.
hit

SIMD comparison

addr offset
Single Source Compiler Results

- Results for whole Swim, Mgrid, and some of their kernels

![Graph showing speedup with 8 SPEs for different kernels.](image)

- 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