Lecture 6: Introduction to CELL B.E. and GPU Programming

Department of Electrical & Computer Engineering
Rutgers University

Agenda

- Background
- CELL B.E. Architecture Overview
- CELL B.E. Programming Environment
- GPU Architecture Overview
- CUDA Programming Model
- A Comparison: CELL B.E. vs. GPU
- Resources

Sources:
- IBM Cell Programming Workshop, 03/02/2008, GaTech
- UIUC course “Programming Massively Parallel Processors”, Fall 2007
• Background
  • CELL B.E. Architecture Overview
  • CELL B.E. Programming Environment
  • GPU Architecture Overview
  • CUDA Programming Model
  • A Comparison: CELL B.E. vs. GPU
• Resources

GPU / CPU Performance

Cell B.E. 200Gflops
8 SPEs
3.0 GHz
Quad Core
80 GFLOPS

3.0G Xeon
Quad Core
~80 GFLOPS

GT200 = Tesla T10P
~1000 GFLOPS

Source: NVIDIA

Single Precision Floating-Point Operations per Second for the CPU and GPU

*NVIDIA*
Successful Projects


Major Limiters to Processor Performance

- Instruction Level Parallelism (ILP) Wall
  - Diminishing returns from deeper pipeline
- Memory Wall
  - DRAM latency vs. processor cores frequency
- Power Wall
  - Limits in CMOS technology
  - System power density

The amount of transistors doing direct computation is shrinking relative to the total number of transistors.
What’s Next? Multicore to Manycore

- Chip level multi-processors
- Vector Units/SIMD
- Rethink memory organization

The question is not whether this will happen but whether we are ready

*Jack Dongarra,

- Background
- CELL B.E. Architecture Overview
- CELL B.E. Programming Environment
- GPU Architecture Overview
- CUDA Programming Model
- A Comparison: CELL B.E. vs. GPU
- Resources
Cell B.E. Highlights (3.2GHz)

- 241M transistors
- 235mm²
- 9 cores, 10 threads
- >200 GFlops (SP)
- >20 GFlops (DP)
- Up to 25 GB/s memory BW
- Up to 75 GB/s I/O BW
- >300 GB/s EIB
- Top frequency >4GHz (observed in lab)

Cell B.E. Products

---

Consumer | Professional | Business | High Perf Computing
---|---|---|---
Common Operating Systems, Infrastructure, Tools, Libraries, Code...
Roadrunner

IBM to Build World’s First Cell Broadband Engine™ Based Supercomputer

Revolutionary Hybrid Supercomputer at Los Alamos National Laboratory Will Harness Cell Game Chips and AMD Opteron™ Technology

- x86 Linux Master Cluster
  - AMD Opteron™ System x3755

- Cell BE Accelerator Linux Cluster
  - 8000+ Blades

- Goal 1 PetaFlop Double Precision Floating Point Sustained
  - 1.6 PetaFlop Peak DP Floating point (3.2 GF)
  - 280 server racks that take up around 12,000 square feet—about three basketball courts.
  - Hybrid of Opteron x3754 AMD processors (system x3755 servers) and Cell BE Blade servers connected via high speed network
  - Modular Approach means the Master Cluster could be made up of Any Type System—Including Power, Intel

Cell B.E. Architecture Roadmap

Performance Enhancements/Scaling

<table>
<thead>
<tr>
<th>Year</th>
<th>Performance Enhancements/Scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>2006</td>
<td>Cell BE (1+8) 90nm SOI</td>
</tr>
<tr>
<td>2007</td>
<td>Cell BE (1+8) 65nm SOI</td>
</tr>
<tr>
<td>2008</td>
<td>Enhanced Cell BE (1+8) 65nm SOI</td>
</tr>
<tr>
<td>2009</td>
<td>Next Gen (2PPE+32SPE) 65nm SOI</td>
</tr>
<tr>
<td>2010</td>
<td>- 1 TFlop (est.)</td>
</tr>
</tbody>
</table>

*All future dates and specifications are estimations only. Subject to change without notice. Dashed outlines indicate concept designs.*
Cell B.E. Block Diagram

- PPE: Power Processor Element
- SPE: Synergistic Processing Elements
- EIB: Element Interconnect Bus

- SPU Core: Registers & Logic
- Channel Unit: Message passing interface for I/O
- Local Store: 256KB of SRAM private to the SPU Core
- DMA Unit: Transfers data between Local Store and Main Memory

PPE and SPE Architectural Difference

<table>
<thead>
<tr>
<th>Feature</th>
<th>PPE</th>
<th>SPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of SIMD registers</td>
<td>32 (128-bit)</td>
<td>128 (128-bit)</td>
</tr>
<tr>
<td>Organization of register files</td>
<td>separate fixed-point, floating-point, and vector multimedia registers</td>
<td>unified</td>
</tr>
<tr>
<td>Load latency</td>
<td>variable (cache)</td>
<td>fixed</td>
</tr>
<tr>
<td>Addressability</td>
<td>2^64 bytes</td>
<td>256KB local store</td>
</tr>
<tr>
<td>Instruction set</td>
<td>more orthogonal</td>
<td>optimized for single-precision float</td>
</tr>
<tr>
<td>Single-precision</td>
<td>IEEE 754-1985</td>
<td>extended range</td>
</tr>
<tr>
<td>Doubleword</td>
<td>no doubleword SIMD</td>
<td>double-precision floating-point SIMD</td>
</tr>
</tbody>
</table>
• Background
• CELL B.E. Architecture Overview
• CELL B.E. Programming Environment
• GPU Architecture Overview
• CUDA Programming Model
• A Comparison: CELL B.E. vs. GPU
• Resources

Cell Software Environment
Cell/BE Basic Programming Concepts

- The PPE is just a PowerPC running Linux.
  - No special programming techniques or compilers are needed.
- The PPE manages SPE processes as POSIX pthreads*.
- IBM-provided library (libspe2) handles SPE process management within the threads.
- Compiler tools embed SPE executables into PPE executables: one file provides instructions for all execution units.

Control & Data Flow of PPE & SPE

1) (PPE Program) Loads the SPE program to the LS.
2) (PPE Program) Instructs the SPEs to execute the SPE program.
3) (SPE Program) Transfers required data from the main memory to the LS.
4) (SPE Program) Processes the received data in accordance with the requirements.
5) (SPE Program) Transfers the processed result from the LS to the main memory.
6) (SPE Program) Notifies the PPE program of the termination of processing.
PPE Programming Environment

- PPE runs PowerPC applications and operating system
- PPE handles thread allocation and resource management among SPEs
- PPE’s Linux kernel controls the SPUs’ execution of programs
  - Schedule SPE execution independent from regular Linux threads
  - Responsible for runtime loading, passing parameters to SPE programs, notification of SPE events and errors, and debugger support
- PPE’s Linux kernel manages virtual memory, including mapping each SPE’s local store (LS) and problem state (PS) into the effective-address space
- The kernel also controls virtual-memory mapping of MFC resources, as well as MFC segment-fault and page-fault handling
- Large pages (16-MB pages, using the hugetlbfs Linux extension) are supported
- Compiler tools embed SPE executables into PPE executables

SPE Programming Environment

- Each SPE has a SIMD instruction set, 128 vector registers and two in-order execution units, and no operating system
- Data must be moved between main memory and the 256 KB of SPE local store with explicit DMA commands
- Standard compilers are provided
  - GNU and XL compilers, C, C++ and Fortran
  - Will compile scalar code into the SIMD-only SPE instruction set
  - Language extensions provide SIMD types and instructions.
- SDK provides math and programming libraries as well as documentation

The programmer must handle
- A set of processors with varied strengths and unequal access to data and communication
- Data layout and SIMD instructions to exploit SIMD utilization
- Local store management (data locality and overlapping communication and computational)
PPE C/C++ Language Extensions (Intrinsics)

- C-language extensions: vector data types and vector commands (Intrinsics)
  - Intrinsics - inline assembly-language instructions
- Vector data types – 128-bit vector types
  - Sixteen 8-bit values, signed or unsigned
  - Eight 16-bit values, signed or unsigned
  - Four 32-bit values, signed or unsigned
  - Four single-precision IEEE-754 floating-point values
  - Example: vector signed int: 128-bit operand containing four 32-bit signed ints
- Vector intrinsics
  - Specific Intrinsics—Intrinsics that have a one-to-one mapping with a single assembly-language instruction
  - Generic Intrinsics—Intrinsics that map to one or more assembly-language instructions as a function of the type of input parameters
  - Predicates Intrinsics—Intrinsics that compare values and return an integer that may be used directly as a value or as a condition for branching

SPE C/C++ Language Extensions (Intrinsics)

Vector Data Types

Three classes of intrinsics

- **Specific Intrinsics** - one-to-one mapping with a single assembly-language instruction
  - prefixed by the string, si_
  - e.g., si_to_char // Cast byte element 3 of qword to char
- **Generic Intrinsics and Built-Ins** - map to one or more assembly-language instructions as a function of the type of input parameters
  - prefixed by the string, spu_
  - e.g., d = spu_add(a, b) // Vector add
- **Composite Intrinsics** - constructed from a sequence of specific or generic intrinsics
  - prefixed by the string, spu_
  - e.g., spu_mfcdma32(ls, ea, size, tagid, cmd) //Initiate DMA to or from 32-bit effective address
Hello World – SPE code

Compiled to **hello_spu.o**

```c
#include <stdio.h>

int main(unsigned long long speid, unsigned long long argp, unsigned long long envp)
{
    printf("Hello World\n");
    return 0;
}
```

Hello World – PPE: Single Thread

```c
#include <errno.h>
#include <stdlib.h>
#include <libspe2.h>

extern spe_program_handle_t hello_spu;

int main(void)
{
    // Structure for an SPE context
    spe_context_ptr_t speid;
    unsigned int flags = 0;
    unsigned int entry = SPE_DEFAULT_ENTRY;
    void * argp = NULL;
    void * envp = NULL;
    spe_stop_info_t stop_info;
    int rc;

    // Create an SPE context
    speid = spe_context_create(0, NULL);
    if (speid == NULL) {
        perror("spe_context_create");
        return -2;
    }

    // Load an SPE executable object into the SPE context local store
    if (spe_program_load(speid, &hello_spu)) {
        perror("spe_program_load");
        return -3;
    }

    // Run the SPE context
    rc = spe_context_run(speid, &entry, 0, argp, envp, &stop_info);
    if (rc < 0)
        perror("spe_context_run");

    // Destroy the SPE context
    spe_context_destroy(speid);
    return 0;
}
Hello World – PPE: Multi-Thread

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/syscall.h>

typedef struct 
{
    uint32_t entry; 
    unsigned int flags;
} SPU_thread_t;

int main(void)
{
    SPU_thread_t ppe_thread;
    ppe_thread.entry = 0;
    ppe_thread.flags = 0;
    printf("Hello World!");
    return 0;
}
```

PPE SPE Communication

- PPE communicates with SPEs through MMIO registers supported by the MFC of each SPE
- Three primary communication mechanisms between the PPE and SPEs
  - Mailboxes
    - Queues for exchanging 32-bit messages
    - Two mailboxes (the SPU Write Outbound Mailbox and the SPU Write Outbound Interrupt Mailbox) are provided for sending messages from the SPE to the PPE
    - One mailbox (the SPU Read Inbound Mailbox) is provided for sending messages to the SPE
  - Signal notification registers
    - Each SPE has two 32-bit signal-notification registers, each has a corresponding memory-mapped I/O (MMIO) register into which the signal-notification data is written by the sending processor
    - Signal-notification channels, or signals, are inbound (to an SPE) registers
    - They can be used by other SPEs, the PPE, or other devices to send information, such as a buffer-completion synchronization flag, to an SPE
  - DMAs
    - To transfer data between main storage and the LS
• Background
• CELL B.E. Architecture Overview
• CELL B.E. Programming Environment
• GPU Architecture Overview
• CUDA Programming Model
• A Comparison: CELL B.E. vs. GPU
• Resources

NVIDIA’s Tesla T10P

• T10P chip
  – 240 cores; 1.3~1.5 GHz
  – Tpeak, 1 Tflop/s, 32bit, single precision
  – Tpeak, 100 Gflop/s, 64bit, double precision
  – IEEE 754r capabilities
• C1060 Card - PCIe 16x
  – 1 T10P; 1.33 Ghz
  – 4GB DRAM
  – ~160W
  – Tpeak ~936 Gflop
• S 1060 Computing Server
  – 4 T10P devices
  – ~700W
CPU vs. GPU: Memory Models

- **CPU**
  - One linear memory spaces
  - Cached

- **GPU**
  - Several memory spaces
  - R/W capabilities
  - Cached/non-cached

**Background**
- CELL B.E. Architecture Overview
- CELL B.E. Programming Environment
- GPU Architecture Overview
- CUDA Programming Model
- A Comparison: CELL B.E. vs. GPU
- Resources
CUDA Programming Model: A Highly Multithreaded Coprocessor

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

GPU Programming Model w/ CUDA

- Compute device
- CUDA kernels
- A grid of thread blocks

Source: NVIDIA
Block and Thread IDs

- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …

CUDA Device Memory Space Overview

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory
- The host can R/W global, constant, and texture memories
Global, Constant, and Texture Memories (Long Latency Accesses)

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads

- Texture and Constant Memories
  - Constants initialized by host
  - Contents visible to all threads

Access Times

- **Register** – dedicated HW - single cycle
- **Shared Memory** – dedicated HW - single cycle
- **Local Memory** – DRAM, no cache - *slow* *
- **Global Memory** – DRAM, no cache - *slow* *
- **Constant Memory** – DRAM, **cached**, 1…10s…100s of cycles, depending on cache locality
- **Texture Memory** – DRAM, **cached**, 1…10s…100s of cycles, depending on cache locality
- **Instruction Memory (invisible)** – DRAM, **cached**
CUDA Device Memory Allocation

- **cudaMalloc()**
  - Allocates object in the device **Global Memory**
  - Requires two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** of allocated object

- **cudaFree()**
  - Frees object from device **Global Memory**
  - Pointer to freed object

CUDA Host-Device Data Transfer

- **cudaMemcpy(…)**
  - Memory data transfer
  - Requires four parameters
    - Pointer to source
    - Pointer to destination
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device

- **cudaMemcpyAsync(…)**
CUDA Function Declarations

<table>
<thead>
<tr>
<th></th>
<th>Executed on the:</th>
<th>Only callable from the:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong> float DeviceFunc()</td>
<td>device</td>
<td>device</td>
</tr>
<tr>
<td><strong>global</strong> void KernelFunc()</td>
<td>device</td>
<td>host</td>
</tr>
<tr>
<td><strong>host</strong> float HostFunc()</td>
<td>host</td>
<td>host</td>
</tr>
</tbody>
</table>

- __global__ defines a kernel function
  - Must return void
- __device__ and __host__ can be used together

CUDA Function Declarations (cont.)

- __device__ functions cannot have their address taken
- For functions executed on the device:
  - No recursion
  - No static variable declarations inside the function
  - No variable number of arguments
Language Extensions: Variable Type Qualifiers

<table>
<thead>
<tr>
<th></th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong> <strong>local</strong> int LocalVar;</td>
<td>local thread</td>
<td>thread</td>
<td></td>
</tr>
<tr>
<td><strong>device</strong> <strong>shared</strong> int SharedVar;</td>
<td>shared block</td>
<td>block</td>
<td></td>
</tr>
<tr>
<td><strong>device</strong> int GlobalVar;</td>
<td>global grid</td>
<td>application</td>
<td></td>
</tr>
<tr>
<td><strong>device</strong> <strong>constant</strong> int ConstantVar;</td>
<td>constant grid</td>
<td>application</td>
<td></td>
</tr>
</tbody>
</table>

- __device__ is optional when used with __local__, __shared__, or __constant__
- Automatic variables without any qualifier reside in a register
  - Except arrays that reside in local memory

Calling a Kernel Function – Thread Creation

- A kernel function must be called with an execution configuration:

```c
__global__ void KernelFunc(...);
dim3 DimGrid(100, 50);  // 5000 thread blocks
dim3 DimBlock(4, 8, 8);  // 256 threads per block
size_t SharedMemBytes = 64;  // 64 bytes of shared memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```
- Any call to a kernel function is asynchronous, explicit synch needed for blocking
Dense Matrix Multiplication

Each thread block computes one sub-matrix $C_{ab}$ of $C$. Each thread within the block computes one element of $C_{ab}$.

Dense Matrix Multiplication - Host Side

```c
// Compute C = A * B
// hA is the height of A
// wA is the width of A
// wB is the width of B

// Compute the execution configuration assuming
// the matrix dimensions are multiples of BLOCK_SIZE
int dimBlock = BLOCK_SIZE,
int hA = BLOCK_SIZE,
int wA = BLOCK_SIZE;

int dimGrid = (wB / dimBlock.x,
               hA / dimBlock.y);

// Launch the device computation
Mull<<dimGrid, dimBlock>>(Ad, Bd, wA, wB, Cd);

// Read C from the device
cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);
```
Dense Matrix Multiplication - Device Side

```c
// Device multiplication function called by Host()
// Compute C = A * B
// A in the width of B
__global__ void mult(float* A, float* B, int w, int m, int n, float* C) {
    // Block index
    int tx = blockIdx.x % blockDim.x;
    int ty = blockIdx.y;
    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    // Index of the first sub-matrix of A processed by the block
    int albegin = tx * BLOCK_SIZE + ty;
    // Index of the last sub-matrix of A processed by the block
    int alend = albegin + w + ty - 1;
    // Step size used to iterate through the sub-matrices of A
    int lstep = BLOCK_SIZE + 1;
    // Index of the first sub-matrix of B processed by the block
    int blbegin = blockIdx.x * BLOCK_SIZE + tx;
    // Step size used to iterate through the sub-matrices of B
    int lstep = BLOCK_SIZE + w;
    // The element of the block sub-matrix that is computed
    // by the thread
    float Csub = 0;
    // Loop over all the sub-matrices of A and B required to
    // compute the block sub-matrix
    for (int l = albegin, b = blbegin; 
        l < alend, b < blend; 
        l += lstep, b += lstep)
    {
        // Shared memory for the sub-matrix of A
        __shared__ float A[BLOCK_SIZE][BLOCK_SIZE];
        // Shared memory for the sub-matrix of B
        __shared__ float B[BLOCK_SIZE][BLOCK_SIZE];
        // Load the matrices from global memory to shared mem
        // each thread loads one element of each matrix
        A[ty][tx] = A[l + w * ty + tx];
        B[ty][tx] = B[b + w * ty + tx];
        // Synchronize to make sure the matrices are loaded
        __syncthreads();
        // Multiply the two matrices together
        // each thread computes one element
        // of the block sub-matrix
        for (int k = 0; k < m; k++)
            Csub += A[ty][k] * B[k][tx];
        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }
    // Write the block sub-matrix to global memory
    // each thread writes one element
    int c = w * BLOCK_SIZE + ty * BLOCK_SIZE + tx;
    A[c + w * ty + tx] = Csub;
}
```

- Background
- CELL B.E. Architecture Overview
- CELL B.E. Programming Environment
- GPU Architecture Overview
- CUDA Programming Model
- A Comparison: CELL B.E. vs. GPU
- Resources
# Cell B.E. vs. Tesla T10P GPU

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Cell B.E.</th>
<th>Tesla T10P</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cores</strong></td>
<td>Heterogeneous 8 SPE/1 PPE (dual threads). Clocked @ 3.2GHz. AltVec ISA</td>
<td>Uniform simple Thread Processors; co-processor to CPU (30 Multiprocessors, 8 cores / MP, total 240 cores/threads), clocked @600MHz. NVIDIA private ISA</td>
</tr>
<tr>
<td><strong>Peak SP</strong></td>
<td>SPEs: 25.6GFLOPS, PEAK DP 14 GFLOPS, 102 GFLOPS (PowerXCell 8i)</td>
<td>Peak SP 1 TFLOPS*</td>
</tr>
<tr>
<td><strong>PPE</strong></td>
<td>SPEs: 25.6 GFLOPS</td>
<td>Peak DP 125 GFLOPS*</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>LS (256KB/spe), main memory (no direct access for SPU)</td>
<td>Device Memory, Shared Memory (16KB/MP)</td>
</tr>
<tr>
<td><strong>Memory Bandwidth</strong></td>
<td>128bits, 256GB/s to main memory.</td>
<td>512bits, 102GB/s to DRAM; PCIe 16x to CPU side (4GB/s one-way, 8GB/s bi-direction)</td>
</tr>
<tr>
<td><strong>Inter-core communication</strong></td>
<td>Very fast, 204.8GB/s on EIB. mailbox, signal, DMA</td>
<td>Local barrier. Shard memory, or return to CPU for global sync. DMA to CPU memory</td>
</tr>
<tr>
<td><strong>Programming</strong></td>
<td>C/C++ Extensions, support Fortran, Stacks on SPE; full debug support</td>
<td>C/C++ Extensions, no-stacks on GPU Cores; limited debug support</td>
</tr>
<tr>
<td></td>
<td>2-level SIMD; has to manually SIMD’ize code</td>
<td>1-level SIMD, scalar unit exposed to programmer directly. Kernel code length limitation</td>
</tr>
<tr>
<td><strong>Library</strong></td>
<td>FFT, BLAS, High level Acceleration Lib, …</td>
<td>FFT, BLAS</td>
</tr>
</tbody>
</table>

### Background
- CELL B.E. Architecture Overview
- CELL B.E. Programming Environment
- GPU Architecture Overview
- CUDA Programming Model
- A Comparison: CELL B.E. vs. GPU

### Resources
Cell Resources

- Cell resource center at developerWorks
- Cell developer's corner at power.org
  - http://www.power.org/resources/devcorner/cellcorner/
- The cell project at IBM Research
- The Cell BE at IBM alphaWorks
  - http://www.alphaworks.ibm.com/topics/cell
- Cell BE at IBM Engineering & Technical Services
  - http://www-03.ibm.com/technology/
- IBM Power Architecture
  - http://www-03.ibm.com/chips/power/
- Cell BE documentation at IBM Microelectronics
- Linux info at the Barcelona Supercomputing Center website
  - http://www.bsc.es/projects/deepcomputing/linuxoncell/

GPU Resources

- NVIDIA CUDA Center
  http://www.nvidia.com/cuda/
- UIUC Course: Programming Massively Parallel Processors
  http://courses.ece.uiuc.edu/ece498/al1/
- GP-GPU resources
  http://www.gpgpu.org/
- Books: GPU-GEMS 2/3