Multicore Programming Case Studies: Cell BE and NVIDIA Tesla

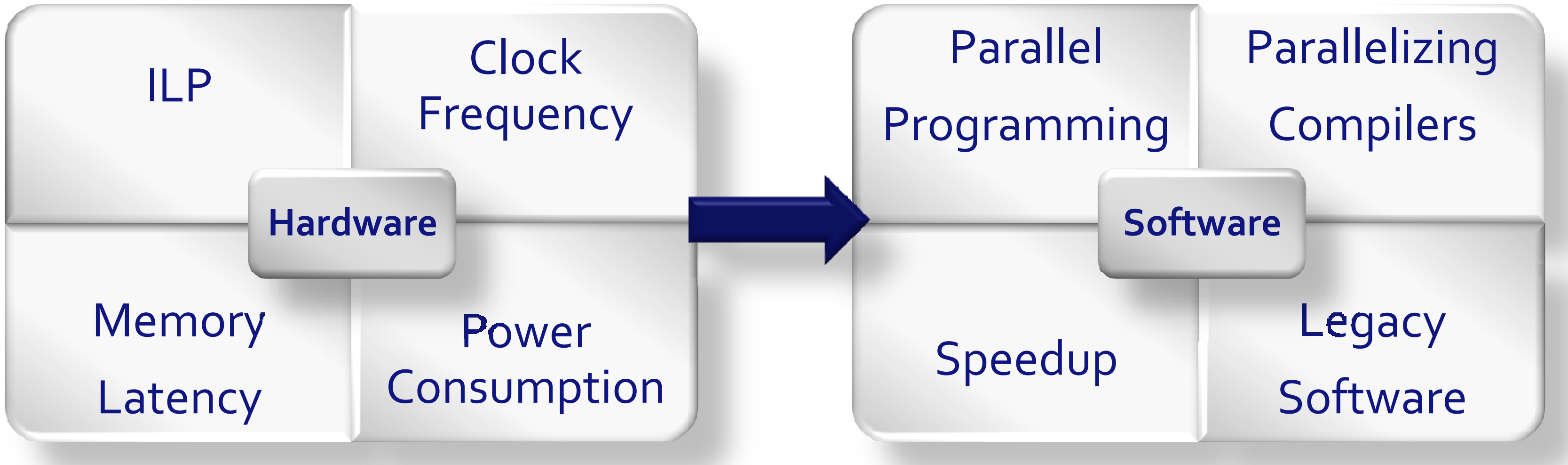
Meeting on Parallel Routine Optimization and Applications

Juan Fernández (juanf@ditec.um.es)
Gregorio Bernabé
Manuel E. Acacio
José L. Abellán
Joaquín Franco

May 26-27, 2008

Introduction
Motivation

- Monolithic Processors
  - Single-core processors
  - Ever-increasing hardware design complexity
- Modular Processors
  - Multi-core processors
  - Software development and optimization complexity



Introduction

Challenges

- Technological constraints
  - Simpler processing cores
    - Less transistors devoted to control logic and storage
      - Lack of branch prediction and other aggressive speculative execution techniques
      - Limited amount of on-chip memory per core (memory size scales at a slower rate)
  - Limited amount of off-chip bandwidth
- Software constraints
  - Complexity of developing and optimizing new parallel applications
  - Difficulty to build new compilers to automatically extract parallelism
  - *Actual* fraction of ideal performance that can be achieved with real applications
  - Migration path for legacy software built atop MPI, OpenMP, Cray Shmem, etc.

Introduction
Goal

- Two multicore platforms are concentrating an enormous attention due to their tremendous potential in terms of sustained performance: Cell BE and NVIDIA GPGPUs
- Common denominator is a non-traditional programming model
- In this talk we try to provide some insight into their programming models along with an overview of their most outstanding architectural features

Introduction

Target architectures

- Cell Broadband Engine
  - Jointly developed by Sony (PS3), Toshiba and IBM
  - Heterogeneous multicore processor specifically designed to exploit not only thread-level parallelism but also SIMD parallelism
  - Peak performance of 204.8 Gflops (SP) and 14.64 Gflops (DP)
- Compute Unified Device Architecture (CUDA)
  - Hardware and software architecture for managing computations on GPUs
    - CUDA-enabled devices can be seen as massively-threaded processors with on-board memory
  - No need to map algorithms and data to a graphics API
  - Common to NVIDIA Geforce 8 series, Quadro FX 5600/4600, Tesla C870/D870/S870
  - Peak performance of 518 Gflops (SP) for the NVIDIA Tesla C870

Agenda

Introduction

Cell Broaband Engine Architecture and Programming

CUDA Architecture and Programming

Comparison

Concluding remarks

Agenda

Introduction

Cell Broaband Engine Architecture and Programming

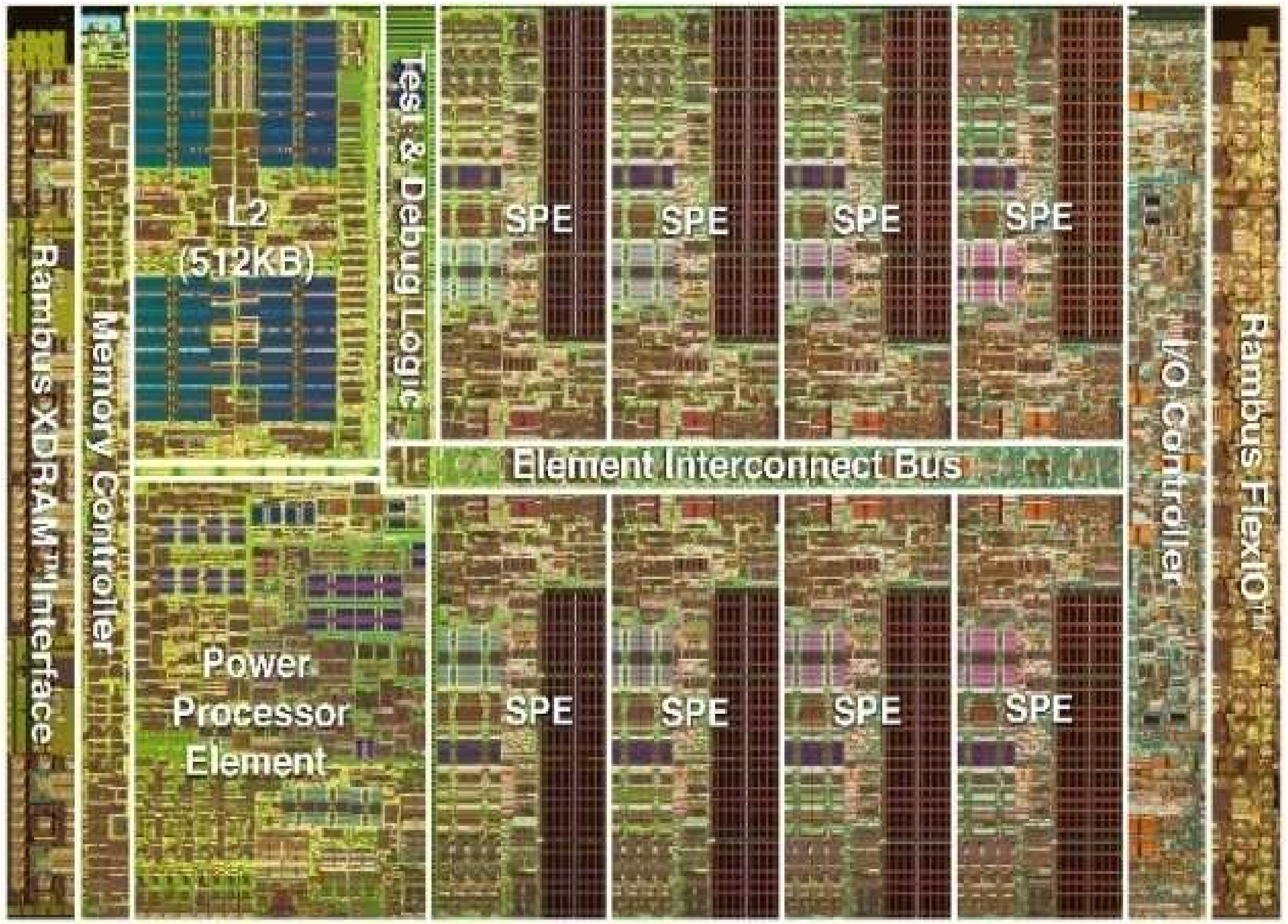
CUDA Architecture and Programming

Comparison

Concluding remarks

Cell Broadband Engine

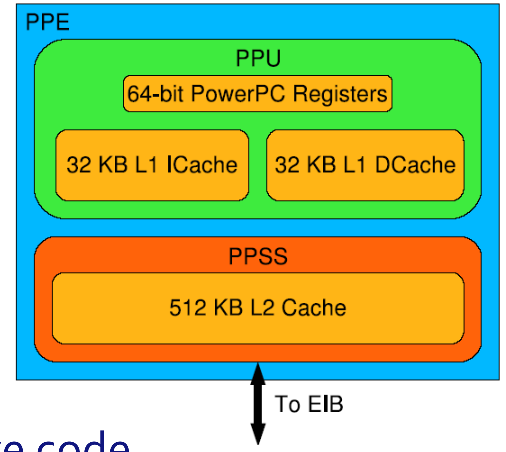
Architecture



Cell Broadband Engine

Architecture

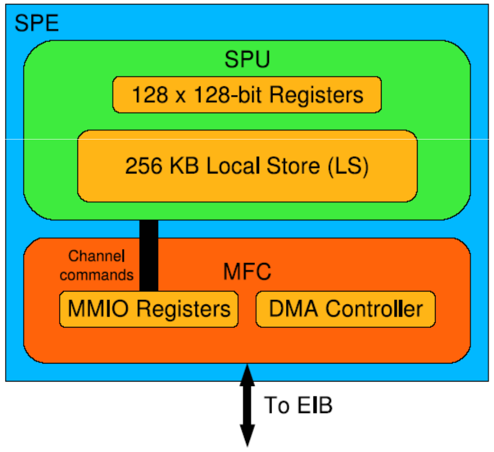
- Heterogeneous multicore processor
  - 1 x Power Processor Element (PPE)
    - 64-bit Power-architecture-compliant processor
    - Dual-issue, in-order execution, 2-way SMT processor
    - PowerPC Processor Unit (PPU)
      - 32 KB L1 IC, 32 KB L1 DC, VMX unit
    - PowerPC Processor Storage Subsystem (PPSS)
      - 512 KB L2 Cache
    - General-purpose processor to run OS and control-intensive code
    - Coordinates the tasks performed by the remaining cores



Cell Broadband Engine

Architecture

- Heterogeneous multicore processor
  - 8 x Synergistic Processing Element (SPE)
    - Dual-issue, in-order execution, 128-bit SIMD processors
    - Synergistic Processor Unit (SPU)
      - SIMD ISA (four different granularities)
      - 128 x 128-bit SIMD register file
      - **256 KB Local Storage (LS) for code/data**
    - Memory Flow Controller (MFC)
      - Memory-mapped I/O registers (MMIO Registers)
      - DMA Controller: commands to transfer data in and out
    - Custom processors specifically designed for data-intensive code
    - Provide the main computing power of the Cell BE

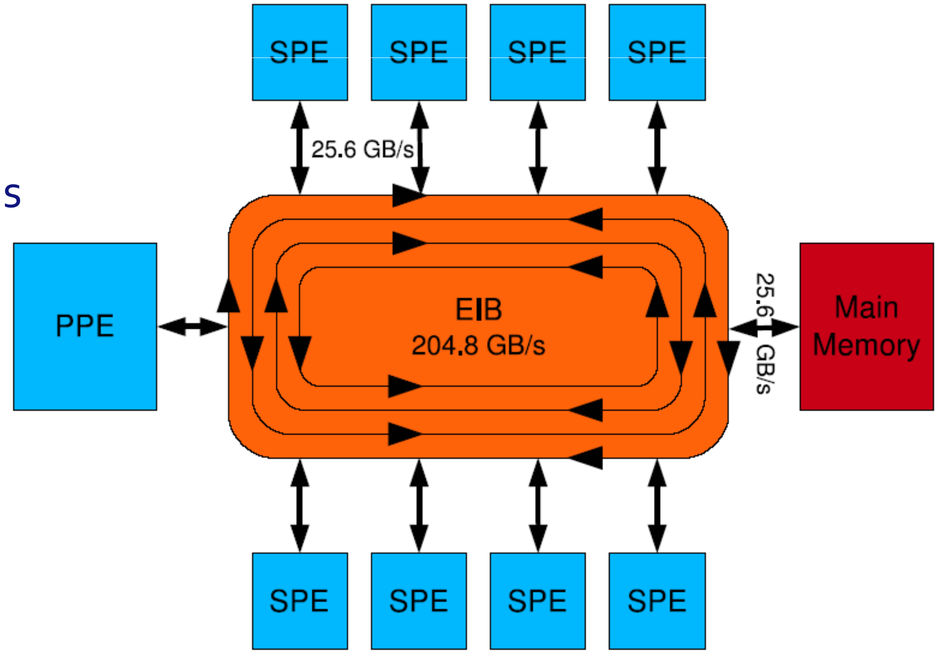


Cell Broadband Engine

Architecture

- Element Interconnect Bus (EIB)
  - Interconnects PPE, SPEs, and the memory and I/O interface controllers
    - 4 x 16 Byte-wide rings (2 clockwise and 2 counterclockwise)
  - Up to three simultaneous data transfers per ring
  - Shortest path algorithm for transfers
- Memory Interface Controller (MIC)
  - 2 x Rambus XDR I/O memory channels

    (accesses on each channel of 1-8, 16, 32, 64 or 128 Bytes)
- Cell BE Interface (BEI)
  - 2 x Rambus FlexIO I/O channels



Cell Broadband Engine

Programming

- SPEs are intended to run threads spawned by the PPE
- PPE and SPEs communicate and synchronize using a bunch of hardware-supported mechanisms:
  - PPE- or SPE-initiated, blocking or non-blocking DMA transfers between main memory and an SPE' s LS (GET) or vice versa (PUT) up to a maximum of 16KB
  - Mailboxes support the exchange of 32-bit messages among SPEs and PPE (4-entry *SPU Read Inbound Mailbox* / *SPU Write Outbound Mailbox*)
  - Signals allow SPEs to collect 32-bit incoming notifications (*SPU Signal Notification 1* / *SPU Signal Notification 2*)
  - Read-modify-write atomic operations enable simple transactions on single words residing in main memory (e.g. `atomic_add_return`)

Cell Broadband Engine
Programming

- Usually SPEs run the same code but on different data (SPMD)
- Other collaboration schemes are feasible
  - Function offload
  - Device extension
  - Pipeline
  - Shared-memory multiprocessor
- Critical issues
  - Data movement
  - Load balancing

Cell Broadband Engine

Programming

- Separate programs written in C/C++ for PPE and SPEs (ADA and Fortran also supported as of SDK v3.0)
  - PPE program
    - Vector data types and intrinsics to use the VMX unit (e.g. `vector float` or `vec_add`)
    - Library function calls to manage threads and perform communication and synchronization operations (e.g. `spe_context_run`, `spe_mfcio_put`, `spe_in_mbox_write`)
  - SPE program
    - Vector data types and intrinsics (e.g. `vector float` or `spu_add`)
      - SP FP fully pipelined in 4-way SIMD fashion while DP FP only partially pipelined
    - Library function calls to perform communication and synchronization operations (e.g. `mfc_get`, `spu_read_in_mbox`)

Cell Broadband Engine
Programming

- SPE Program (cont.)
  - BLAS, LAPACK and SIMD Math libraries
  - Compiler directives to tackle with a number of memory alignment issues and branch hinting (e.g. `__attribute__(aligned(16))` or `__builtin_expect`)
  - Manual optimizations
    - Double-buffering DMA data transfers to overlap communication and computation
    - SPE code SIMDization to fully exploit SPE architecture
    - Reordering instruction scheduling to maximize dual-issue cycles

Cell Broadband Engine

Programming example

```
// LS buffers for DMA transfers must be aligned
control_block cb __attribute__(aligned (128));
int vecx[MAX_VECTOR] __attribute__(aligned (128));
int vecy[MAX_VECTOR] __attribute__(aligned (128));

// SPE code for saxpy_parallel
int main(uint64_t speid, uint64_t id, EA cb_ptr)
{
  . . .
  /* copy control block from main memory to LS buffer */
  mfc_get(&cb, cb_ptr, sizeof(control_block), DMA_TAG, 0, 0);
  mfc_write_tag_mask(1<<DMA_TAG);
  mfc_read_tag_status_all();

  /* wait for PPE approval to proceed */
  unsigned int in_mbox_data = spu_read_in_mbox();
  assert(in_mbox_data == DMA_START);
  . . .
```

Cell Broadband Engine

Programming example

```
      /* copy vecx and vecy from main memory to LS buffers */
      mfc_get(&vecx, cb.vecx_ptr, MAX_VECTOR * sizeof(float), DMA_TAG, 0, 0);
      mfc_get(&vecy, cb.vecxy_ptr, MAX_VECTOR * sizeof(float), DMA_TAG, 0, 0);
      mfc_write_tag_mask(1<<DMA_TAG);
      mfc_read_tag_status_all();

      /* compute vecy using non-SIMDized code (saxpy_spu BLAS function available) */
      for(int i=0; i < cb.n; i++) { vecy = cb.alpha * vecx[i] + vecy[i];}

     /* copy vecy back to main memory */
      mfc_put(&vecy, cb.vecy_ptr, MAX_VECTOR * sizeof(float), DMA_TAG, 0, 0);
      mfc_write_tag_mask(1<<DMA_TAG);
      mfc_read_tag_status_all();

      /* tell PPE we are done */
      spu_write_out_mbox(DMA_END);
      . . .
  }
```

Cell Broadband Engine

Programming

- Other libraries/environments that support Cell BE programming
  - Data Communications and Synchronization Library (DACS) and Accelerated Library Framework (ALF) are included in the latest releases of the IBM SDK
  - Cell Superscalar (CellS) developed at Barcelona Supercomputing Center
  - Multicore Plus SDK Software developed by Mercury Computing Systems Inc.
  - RapidMind Multicore Development Platform for AMD and Intel multicore x86 CPUs, ATI/AMD and NVIDIA GPUs and Cell BE

Cell Broadband Engine

Commercial systems

- Cell BE comes in different flavors
  - Play Station 3
    - Cheapest alternative but…
    - …6 out of 8 SPEs and < 200 MB left to applications
  - IBM Blade Center QS20/21/22
  - Mercury dual Cell-based blade
  - Mercury Cell-based PCI Express board

Agenda

Introduction

Cell Broaband Engine Architecture and Programming

CUDA Architecture and Programming

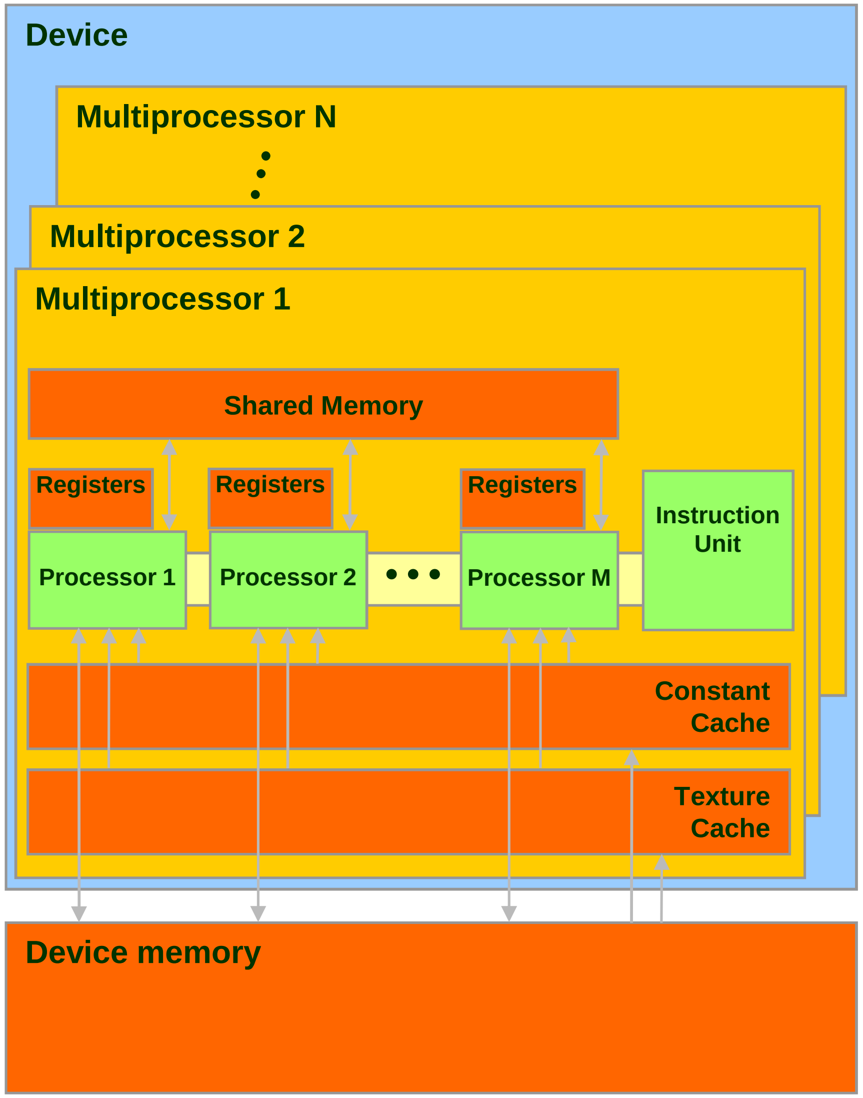
Comparison

Concluding remarks

Compute Unified Device Architecture

Hardware Model

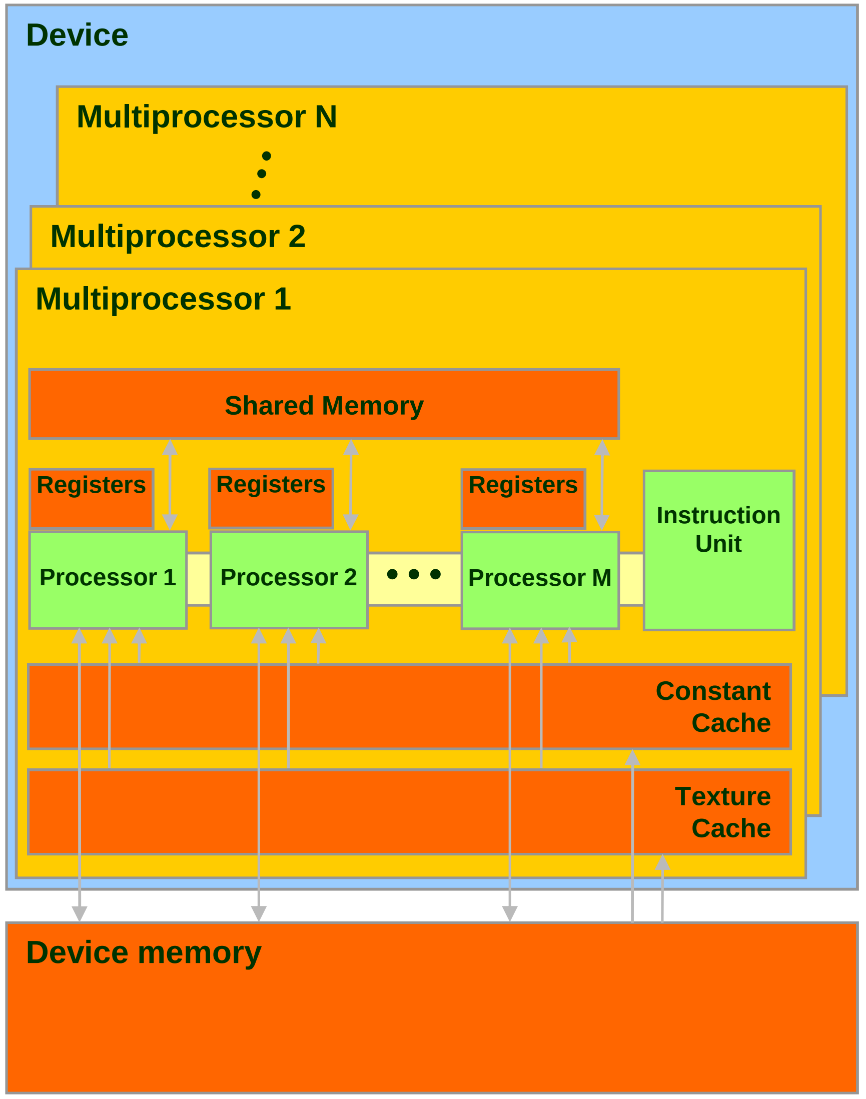
- GPU as a coprocessor to CPU
  - N x Multiprocessors
    - M x processors execute the same instruction on different data based on `threadId` at any given clock cycle
    - 32-bit read-write registers per processor
    - Parallel data cache (SHM) shared by all processors
    - Read-only constant and texture caches shared by all processors
    - NVIDIA Tesla C870: N=16 and M=8, 8192 registers and 16 KB of SHM per multiprocessor (16 memory banks)



Compute Unified Device Architecture

Hardware Model

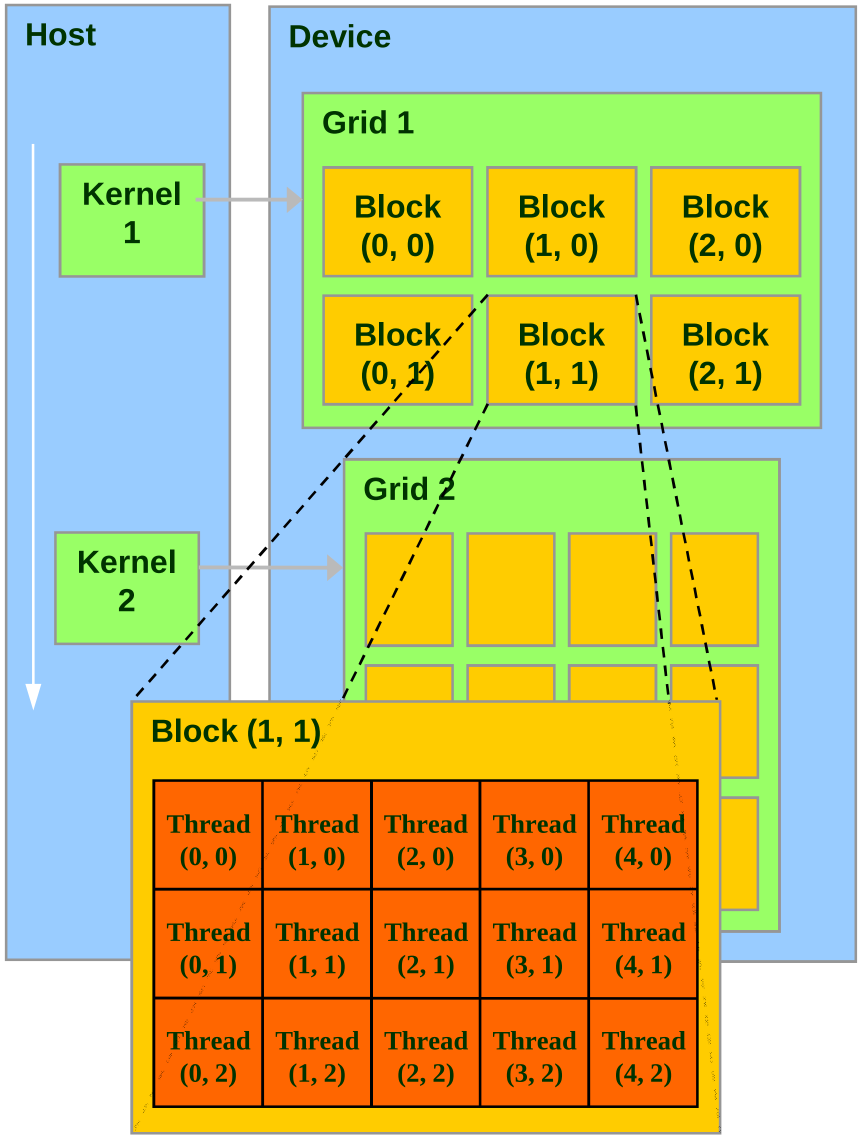
- GPU as a coprocessor to CPU
  - Device memory (on-board)
    - Global, constant and texture memory optimized for different usages
    - Written to or read from by CPU
    - Persistent through the life of the application
    - NVIDIA Tesla C870: 1.5 GB of device memory



Compute Unified Device Architecture

Execution Model

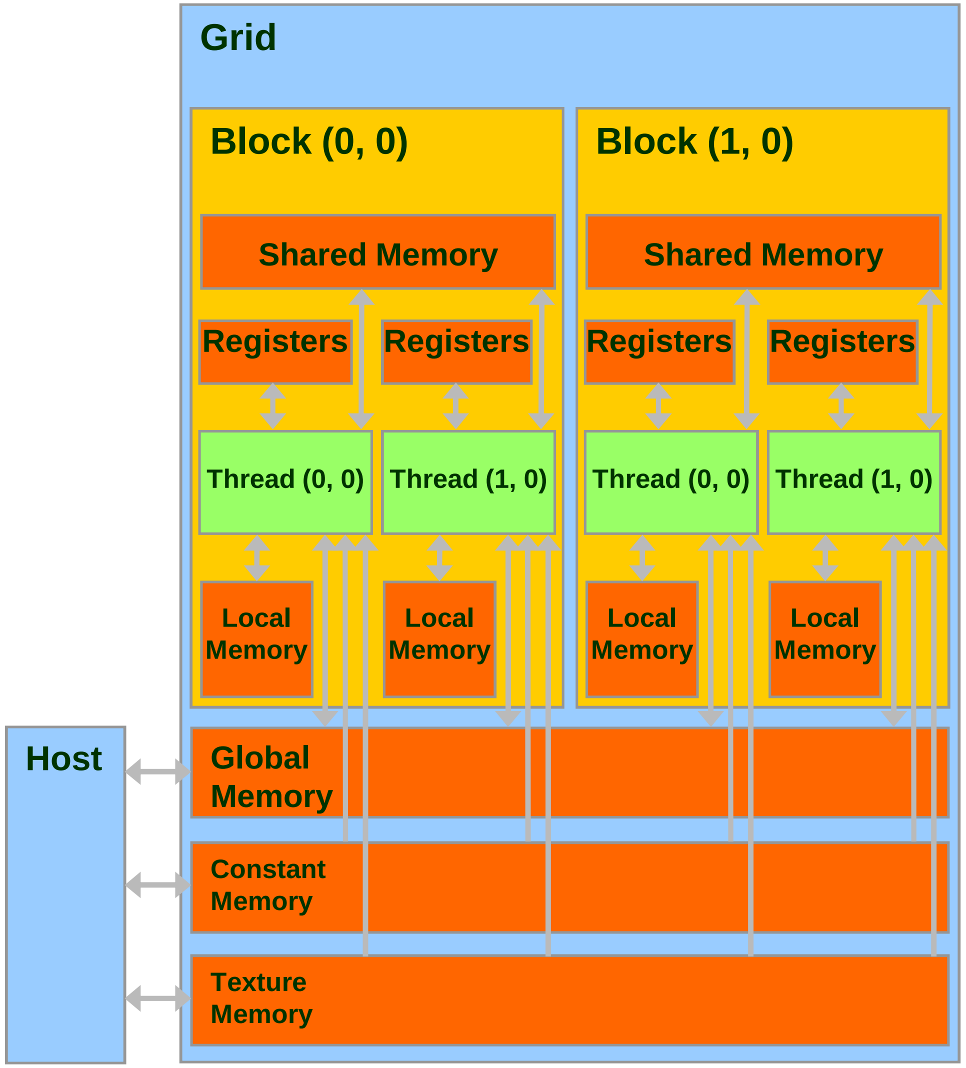
- Kernel: portion of application run by many threads in parallel
  - Thread block: 1D-, 2D- or 3D-array of threads (`threadId`)
    - Assigned to a single multiprocessor
    - Cooperation (SHM)
    - Synchronization (`__syncthreads()`)
  - Grid: 1D- or 2D-array of thread blocks
    - Thread blocks can be assigned to different multiprocessors
    - Threads from different blocks cannot synchronize their execution but at kernel launches



Compute Unified Device Architecture

Memory Model

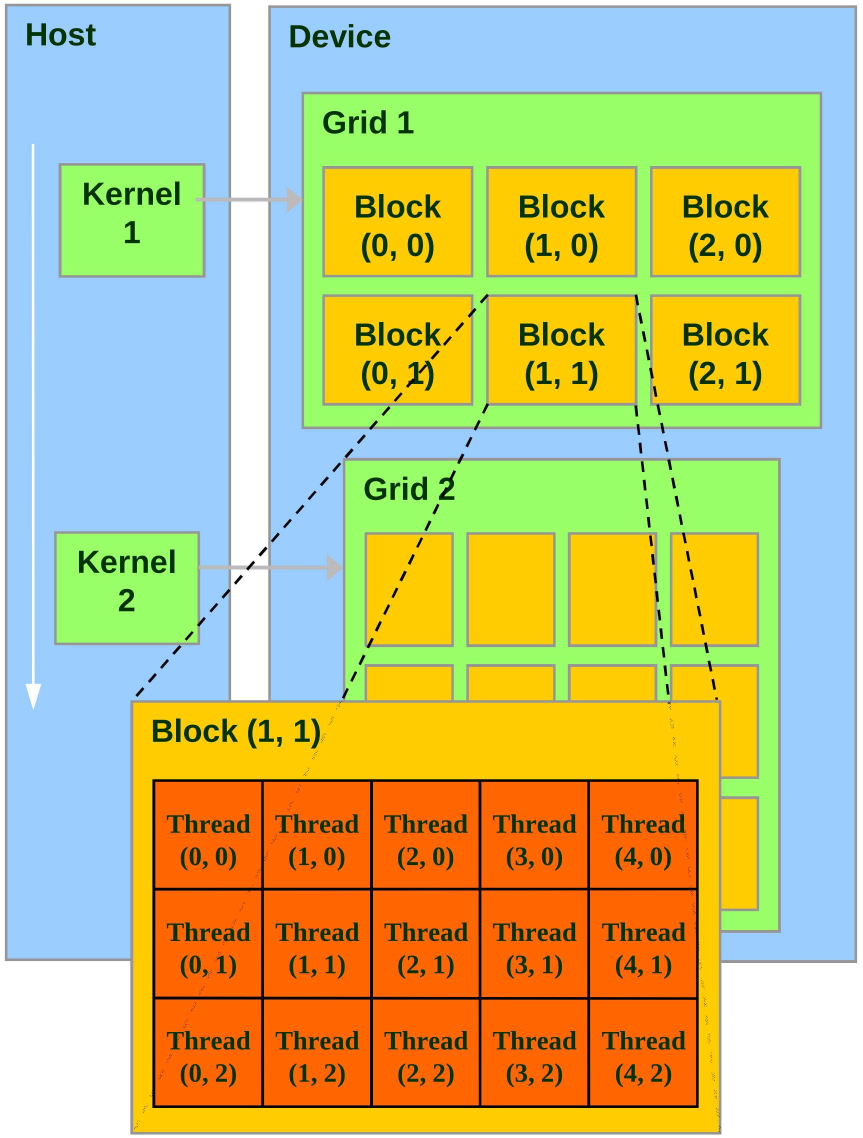
- Each thread can:
  - Read/write per-thread registers
  - Read/write per-block shared memory
  - Read/write per-grid global memory
- Each thread can also:
  - Read/write per-thread local memory (register spilling)
  - Read only per-grid constant memory
  - Read only per-grid texture memory



Compute Unified Device Architecture

Execution Model

- Multiprocessors can process one or more blocks concurrently
- Each active block is split into warps (32 threads) and threads within a warp are executed physically in parallel (4 cycles)
- Registers and SHM are split among active threads from active blocks
- Maximum number of active blocks depends on how many registers and SHM the kernel requires
- High parallelism is crucial to hide memory latency by overlapping memory accesses with computation



Compute Unified Device Architecture
Programming

- Memory is allocated on device memory (global memory)
- Data set is copied from main memory to device memory
- Kernel is invoked with as many threads as *desired*
- Next `cudaThreadSynchronize()` function call blocks CPU
- Thereafter threads proceed in lockstep in a SIMD fashion acting as a data-parallel computing device
  - A thread execution manager handles threading automatically
  - Threads are extremely lightweight: creation overhead is negligible and context switching is essentially free
- Results are copied back from device memory to main memory

Compute Unified Device Architecture

Programming

- Programmers don't have to write explicitly threaded code
- Data layout is the key issue and requires some explicit code
- Single program written in C/C++ with extensions for CPU and CUDA device
  - Function type qualifiers (`__global__` and `__device__`)
  - Variable type qualifiers (`__device__`, `__shared__` and `__constant__`)
  - Four built-in variables (`gridDim`, `blockDim`, `blockIdx` and `ThreadIdx`)
  - Execution configuration construct
    - `function<<<gridDim, blockDim, shm_size>>>(parameter_list);`
  - Runtime library
    - Built-in vector data types (`dim3`), texture types, mathematical functions, type conversion and casting functions, thread synchronization functions, device and memory management functions (`cudaMalloc()`, `cudaFree()` and `cudaMemcpy()`)
  - CUBLAS and CUFFT libraries

Compute Unified Device Architecture

Programming example

```
/* compute y (cublas_saxpy function available) */
__global__ void saxpy_parallel(int n, float alpha, float *y, float *x)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i<n) y[i] = alpha * x[i] + y[i];
}
/* copy data from host memory to global memory */
cudaMemcpy(d_y, h_y, n * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_x, h_x, n * sizeof(float), cudaMemcpyHostToDevice);
/* Invoke SAXPY kernel (256 threads per block) */
dim3 bdim.x = 256; dim3 gdim.x = (n + gdim.x-1) / gdim.x;
saxpy_parallel<<<gdim, bdim, 0>>>(n, 2.0, x, y);
cudaThreadSynchronize();
/* copy data back from global memory to host memory */
cudaMemcpy(d_y, h_y, n * sizeof(float), cudaMemcpyDeviceToHost);
```

Compute Unified Device Architecture

Programming

- Single program written in C/C++ with extensions for CPU and CUDA device
  - Manual optimizations
    - Expose as much parallelism as possible
      - Take advantage of asynchronous kernel launches and data transfers (CUDA streams)
      - Maximize occupancy running as many threads per multiprocessor as possible (CUDA occupancy calc)
    - Optimize memory usage
      - Use page-locked host memory
      - Minimize and group data transfers across the PCIe link
      - Avoid non-coalesced global memory accesses
      - Maximize use of shared memory (favor access patterns with no of few bank conflicts)
    - Optimize instruction usage
      - Minimize use of low throughtput instructions (`func()` vs. `__func()`)
      - Minimize divergent warps

NVIDIA Tesla C870

Commercial systems

- NVIDIA Tesla series:
  - C870: PCIe x16 board
    - 16 x Multiprocessors: 8 x processor
    - 518 Gflops (single precision)
    - 1.5 GB 76.8 GB/s device memory
    - 4 GB/s main memory
      - **1.5 GB/s regular memory**
      - **3.1 GB/s page-locked memory**
  - D870: desktop (2 x C870)
  - S870: 1U rack-mount chassis (4 x C870)

Agenda

Introduction

Cell Broaband Engine Architecture and Programming

CUDA Architecture and Programming

Comparison

Concluding remarks

Comparison

Cell BE and CUDA architecture and programming

Comparison	Cell BE (IBM Blade Center QSXX)	CUDA (NVIDIA Tesla C870)
Cost	Expensive	Affordable
Memory bandwidth	25.6 GB/s	3.1 GB/s (page-locked memory)
Peak performance	204.8 Gflops (SP) 14.64 Gflops (DP)	518 Gflops (SP) -
Learning curve	Steep	Smooth
Code Optimization	Very Complex	Complex
Debuggability	Very Hard	Hard
Portability	None	CUDA-enabled devices
Integration	Easy	Easy

Agenda

Introduction

Cell Broaband Engine Architecture and Programming

CUDA Architecture and Programming

Comparison

Concluding remarks

Concluding remarks

Cell BE and CUDA architecture and programming

- Multicore platforms are here to stay
- Cell BE and NVIDIA GPUs are two popular representatives
  - PROS
    - Tremendous potencial in terms of sustained performance
    - Libraries and tools to facilitate programming and debugging coming up
    - Easy integration into higher-level hierarchical parallel systems
  - CONS
    - Non-tradicional programming models
    - Hard to program and debug
    - Code optimization is complex

Thank you!
Any questions?

Multicore Programming Case Studies: Cell BE and NVIDIA Tesla

Meeting on Parallel Routine Optimization and Applications

Juan Fernández (juanf@ditec.um.es)
Gregorio Bernabé
Manuel E. Acacio
José L. Abellán
Joaquín Franco

May 26-27, 2008