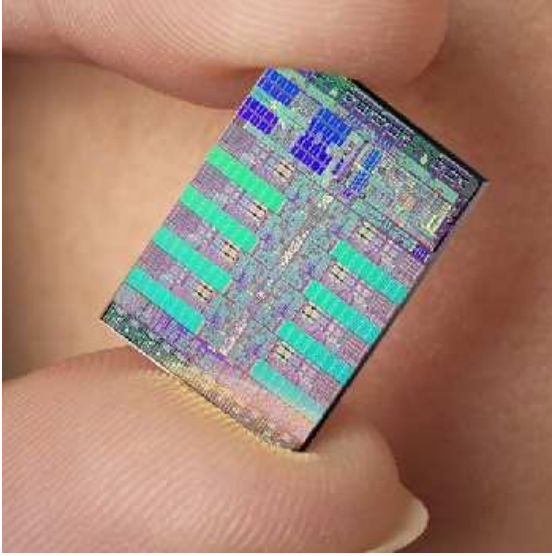



**The Cell
Architecture:
Supercomputer
on a Chip**

Feb 2010

Alex Ramirez
Heterogeneous
Architectures
Research Manager

Always nice to start with a picture





2

A bit of history



- Cell is short for Cell Broadband Engine Architecture
 - Aka CBEA, or CellBE
- Joint venture of Sony, Toshiba, and IBM
 - First design meetings in 2001
 - Over 400 engineers
 - First chips out in 2005
- Priorities set at design stage:
 - Optimize performance per watt
 - Optimize bandwidth over latency
 - Optimize total throughput over ease of programming

Cell based products



- Sony Playstation 3
- IBM QS20, QS21, QS22 blades
- Several others on plan
 - Toshiba HDTV set
 - Leadtek PCIe graphics accelerator board
 - ...

Roadrunner: breaking the Petaflop barrier

- First computer to break the Petaflop barrier
- Installed at Los Alamos National Laboratory (LANL)
- Built by IBM for the U.S. Department of Energy National Nuclear Security Administration
 - Aging of nuclear materials (nuclear weapons stock)
- Hybrid architecture
 - 12.960 IBM QS22 PowerXCell 8i
 - 6.480 AMD Opteron dual-core
 - Specially designed tri-blade computing node



5

Protein Folding @ Home

- Distributed computing problem managed by Stanford University
 - Protein folding
 - Molecular dynamics
- August 2006: PS3 version launched
- October 2006: GPU version launched
- Large contribution to TFLOPS from only a few clients

Platform	TFLOPS	NCPU
NVIDIA	2.182 (43%)	18.333 (4.6%)
PS3	1.387 (28%)	49.180 (12.4%)
TOTAL	5.033	394.853

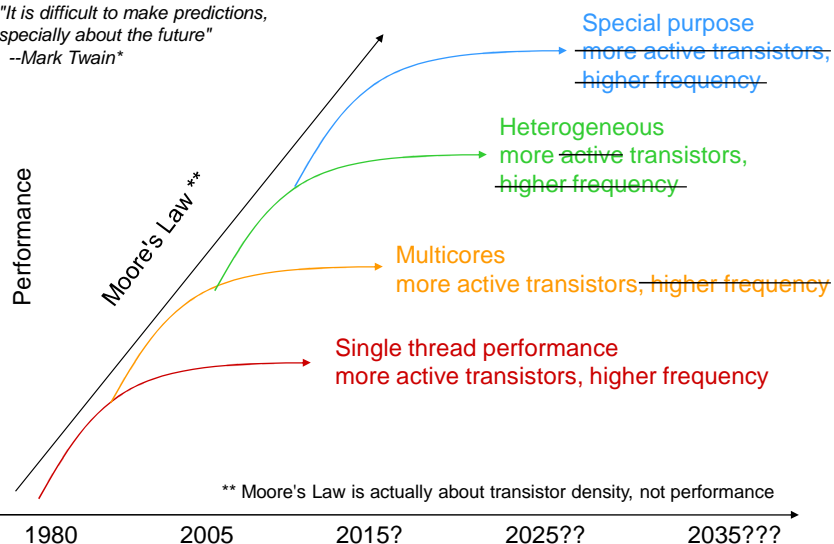
6

Outline

- Introduction
 - Multicore architectures
- Cell architecture
 - Overview
 - Memory architecture
 - PPE architecture
 - SPE architecture
 - Element Interconnection Bus
 - Cell-based systems
- Programming models
 - Streaming
 - Fork-join
 - Task queue

The compulsory Moore's Law slide

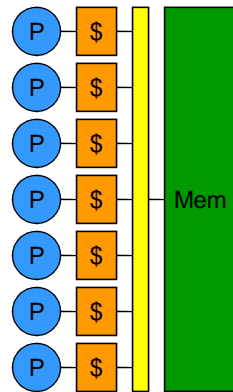
"It is difficult to make predictions,
specially about the future"
--Mark Twain*



Credits to Peter Hofstee, IBM Austin, for the original slide

* Also attributed to many others, including Yogui Berra, Albert Einstein, Confucius, Groucho Marx, ...

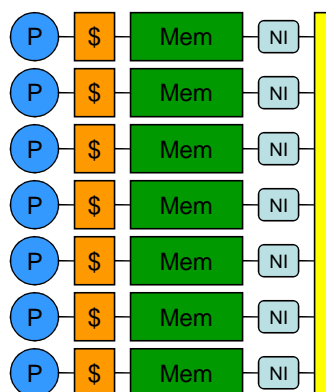
Shared Memory Multiprocessors



- All processors share the same memory
- Common memory access network
 - Usually a bus
- Caches maintain coherent copies of data
 - Snooping bus protocol
- **Scalability** issues
 - Adding processors does not increase memory size
 - Shared memory bandwidth
 - Bus traffic due to coherency messages

9

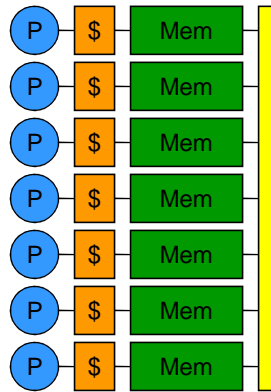
Distributed Memory



- Scalable solution
- Each processor has its own cache + memory
- Non-coherent memories
 - Processors explicitly send copies of data through interconnection network
 - Copies are non-coherent
- **Programmability** issues
 - Data and processing must be explicitly distributed
 - Communication is also explicit

10

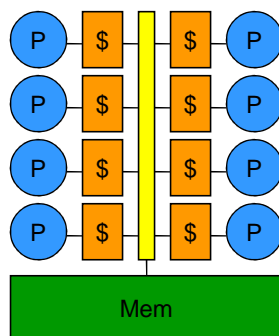
Distributed Shared Memory



- Best of both worlds:
 - Each processor has its own memory
 - Single address space
 - Non-uniform memory access time (NUMA)
 - Caches maintain coherent copies of data
- **Performance scalability** issues
 - Data must still be distributed
 - Excessive data sharing pushes NUMA too far
 - Large amount of memory spent on directories for coherency protocol

11

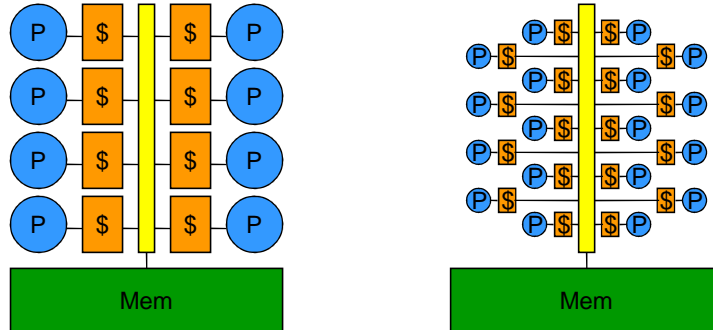
Homogeneous CMP



- All processors are equal
 - Same architecture
 - Same ISA
 - Same caches
- Ease of programming
 - No need to target a specific processor
- Ease of runtime management
 - Threads allocated to any processor
- Alternative designs:
 - Few complex cores
 - Many simple cores

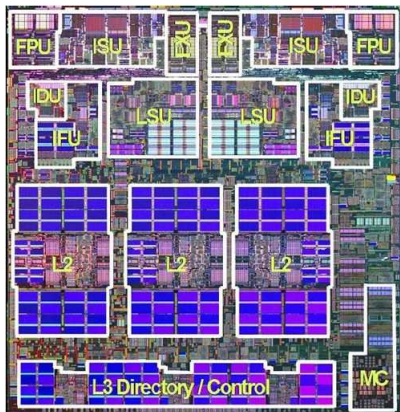
12

Few complex cores vs. Many simple cores

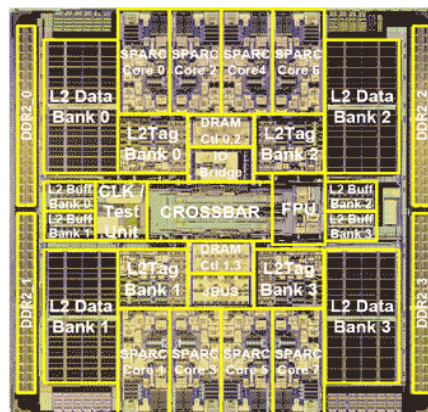


- Complex processors: high single thread performance
 - Good for serial (non-parallel) code
 - Poor parallel performance: simply can't exploit parallelism
- Simple processors: high parallelism
 - Good and efficient for parallel segment
 - Poor serial performance

Complex cores vs. Simple cores

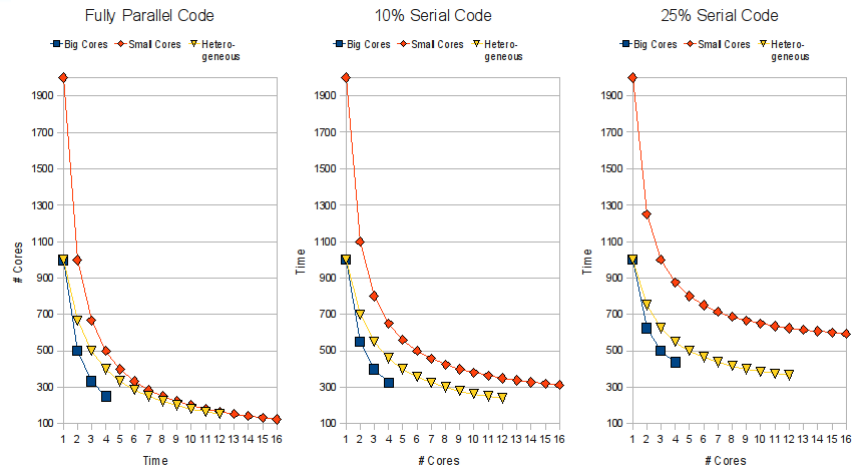


IBM Power5
2-cores
4 threads



Sun Niagara 2
8-core
32 threads

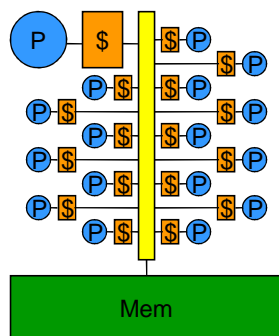
Big vs. Small vs. Heterogeneous



- Assumption: Big cores are 4x Size, 2x Performance of a Small core
- Small cores are better for fully parallel code
- Heterogeneous offers the best compromise

15

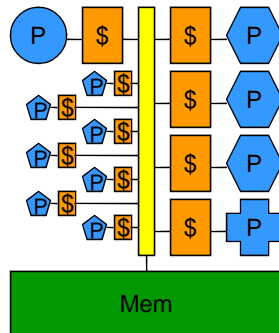
Heterogeneous CMP – Single ISA



- Amdahl's law
 - Performance ends up limited by serial phase
- Exploit both ILP + TLP
 - Sequential phase runs on large core
 - Parallel phase runs on small cores
- Runtime management complexity
 - Where to allocate each thread?

16

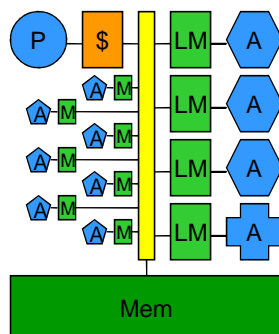
Heterogeneous CMP – Multiple ISA



- Two paths for increased performance
 - Parallelism
 - Specialization
- One step beyond running serial part on complex processor
 - Run **every task** on a custom processor
 - Vector
 - SIMD
 - Multithreaded
 - Special opcodes
 - ...

17

Heterogeneous CMP – Multiple ISA, Distributed Memory



- Caches are unpredictable
 - Huge variations in memory latency
- Cache coherency is not fully scalable
 - Snoopy protocols rely on bus
 - Directories take lots of space
- General purpose processor
 - "Classic" approach based on caches
- Accelerators
 - Work on local (private) memories
 - Move data in/out through DMA

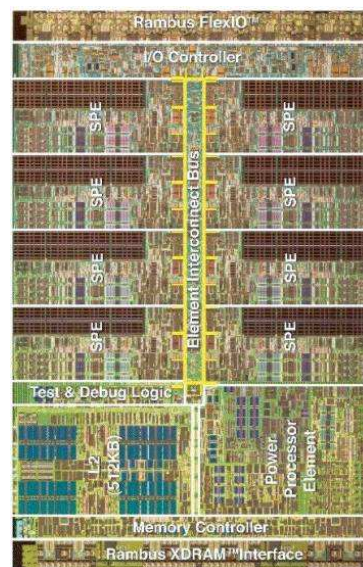
18

Outline

- Introduction
 - Multicore architectures
- Cell architecture
 - Overview
 - Memory architecture
 - PPE architecture
 - SPE architecture
 - Element Interconnection Bus
 - Cell-based systems
- Programming models
 - Streaming
 - Fork-join
 - Task queue

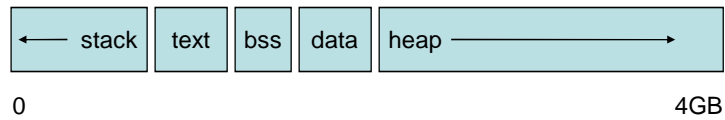
Cell Processor: 9 heterogeneous cores

- 1 Master Processor (PPE)
 - 2-way SMT
 - 2-way in-order issue
 - Minimal branch prediction
 - VMX support
- 8 Accelerators (SPE)
 - 2-way in-order issue
 - no branch prediction ("hints")
 - SIMD ISA
 - 256KB local memory
 - DMA controller
- 512KB L2 cache
- 4-ring interconnection network (EIB)
- Integrated memory and I/O controllers
- Simple design due to power/area restrictions



Address space

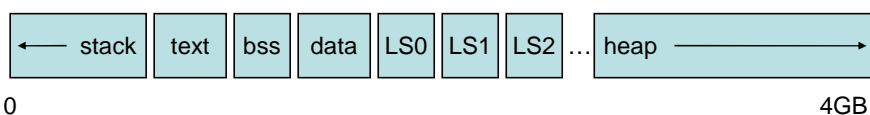
Logical @ space



- Applications access memory using logical addresses
- All segments mapped to the same logical @ space
 - Static mapping: Text / Code, Data, BSS
 - Dynamically allocated: Stack (grows to lower addresses), Heap
- All shared-memory threads share the same logical @space

Cell has multiple address spaces

PPE @ space



SPE @ space

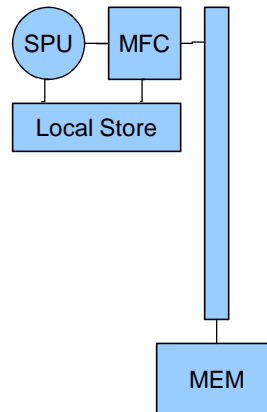


0 256KB

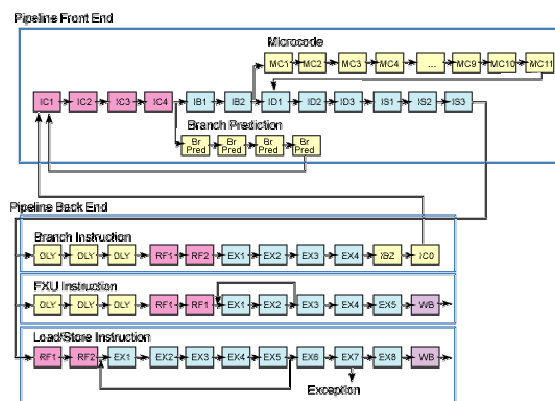
- The PPE sees the regular logical address space
 - The SPE Local Stores can be accessed through a virtual file system
 - Can also be memory mapped using mmap
- The SPE can only see its 256KB Local Store
 - Load / Store addresses truncated at 14 bits
 - Memory is always accessed at 128-bit granularity (16 bytes)
- The MFC's DMA controller accesses both address spaces

Moving data in and out of the Local Store

- SPU can only access Local Store
 - Load / Store
- MFC can access both Memory and Local Store
 - GET / PUT
 - LS address
 - MEM address
 - Size
 - Tag
- Implementation-specific restrictions
 - LS and MEM addresses must have the same alignment
 - Alignment restricted to 128-bytes

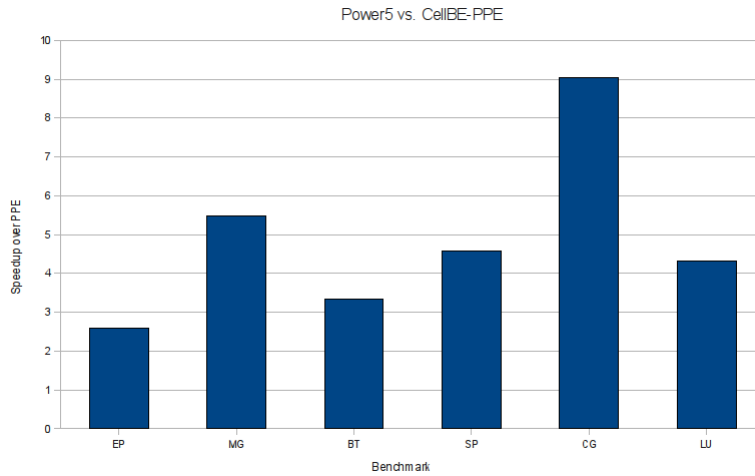


PPE architecture



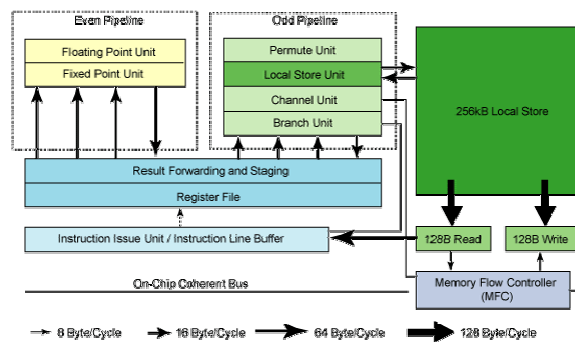
- Full PowerPC ISA compliant
- Low area, low performance
- Multithreaded for throughput, in-order execution

Power5 vs. CellBE PPE: 3x to 10x faster



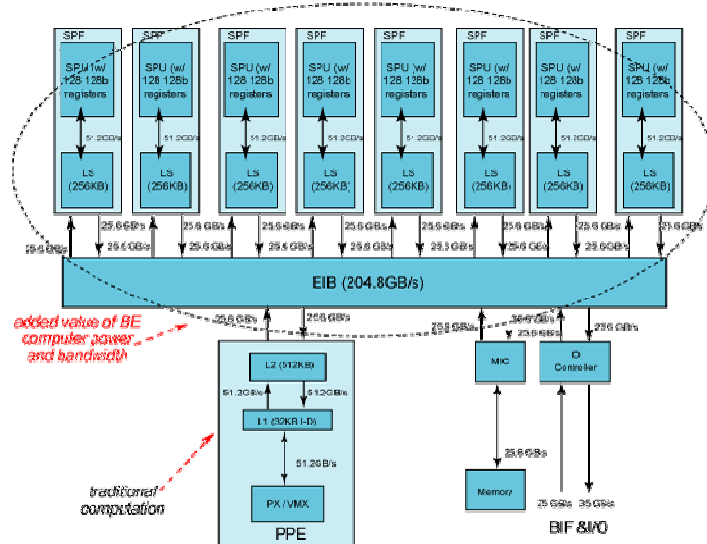
- Power5 outperforms PPE by 3x to 10x

SPE architecture



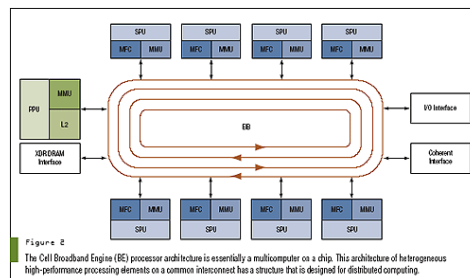
- SIMD only architecture with unified register file
- Superscalar vs VLIW
 - Even and odd pipeline
- No branch prediction
 - Branch hints
- Instruction fetch through instruction line buffer
- Single shared Local Store access port

High bandwidth Network on Chip



27

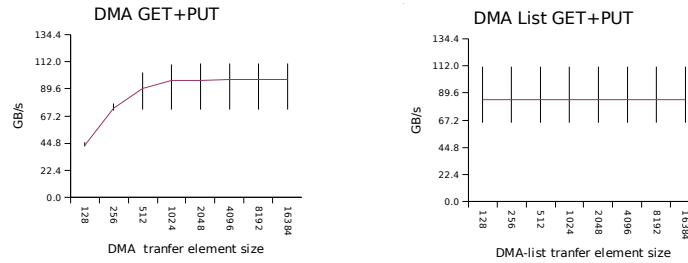
EIB Architecture



- 4 Global communication rings
 - 2 rings in each direction (left, right)
 - Segmented in 3 stages
- Up to 12 simultaneous communications
 - Maximum of 8 due to implementation restrictions

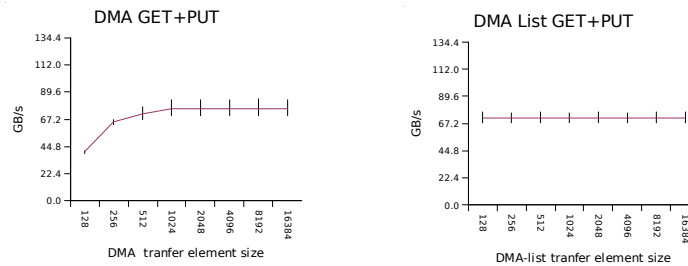
28

EIB performance greatly depends on physical layout



- Experiments run on a prototype QS20 blade @ 2.1 GHz
- 4 SPE transfer data to their “right” or “left” neighbor
 - 4 couples transfer data back + forth
- Vertical lines show variation across 10 different executions

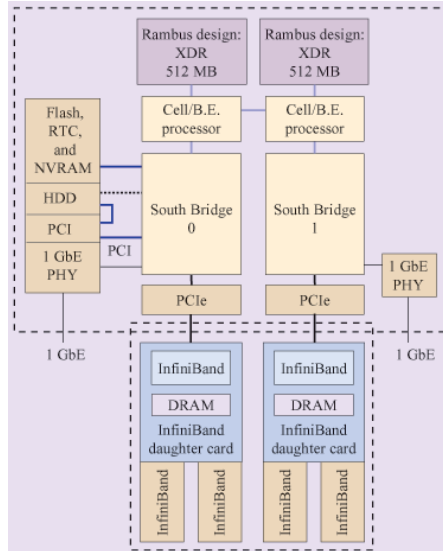
EIB Performance greatly suffers from conflicts



- Experiments run on a prototype QS20 blade @ 2.1 GHz
- All 8 SPE read data to both their right + left neighbors
 - 2 cycles of 8 SPE transfer data in each direction
- EIB performance drops by ~30% vs couples setup
- Much less variation depending on placement
 - All setups have too many conflicts

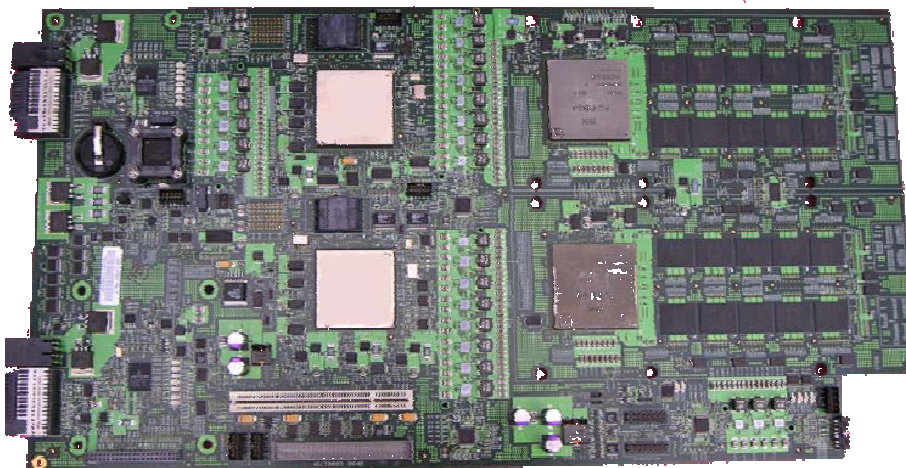
IBM QS20 / QS21 / QS22 Cell Blade

- IBM QS20 Cell Blade
 - Double width format (two slots)
 - 2x IBM Cell Processors @ 3.2 GHz
 - 2x 512MB XDRAM modules
 - 40GB IDE100 2.5" drive
 - Two Gigabit Ethernet ports
 - Optional InfiniBand 4X connectivity
- QS21
 - Single width format
 - QS21 has double the XDRAM (1GB per chip)
 - No on-board hard drive
- QS22
 - Uses the new IBM PowerXCell 8i
 - DDR2 memory controller
 - Up to 32GB of DDR2 memory (16GB per chip)
 - Adds double-precision pipelined SIMD functional unit



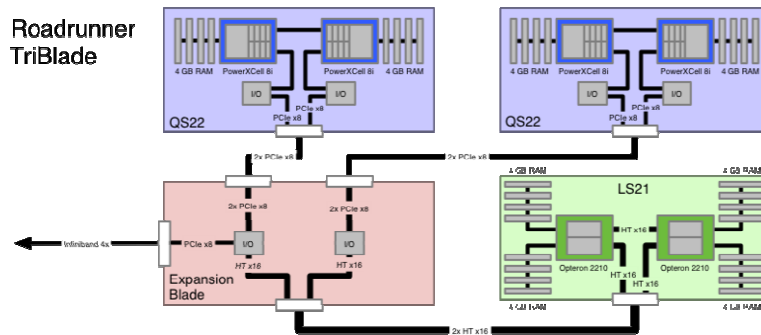
31

IBM QS20 Cell Blade (v1)



32

RoadRunner Tri-blade



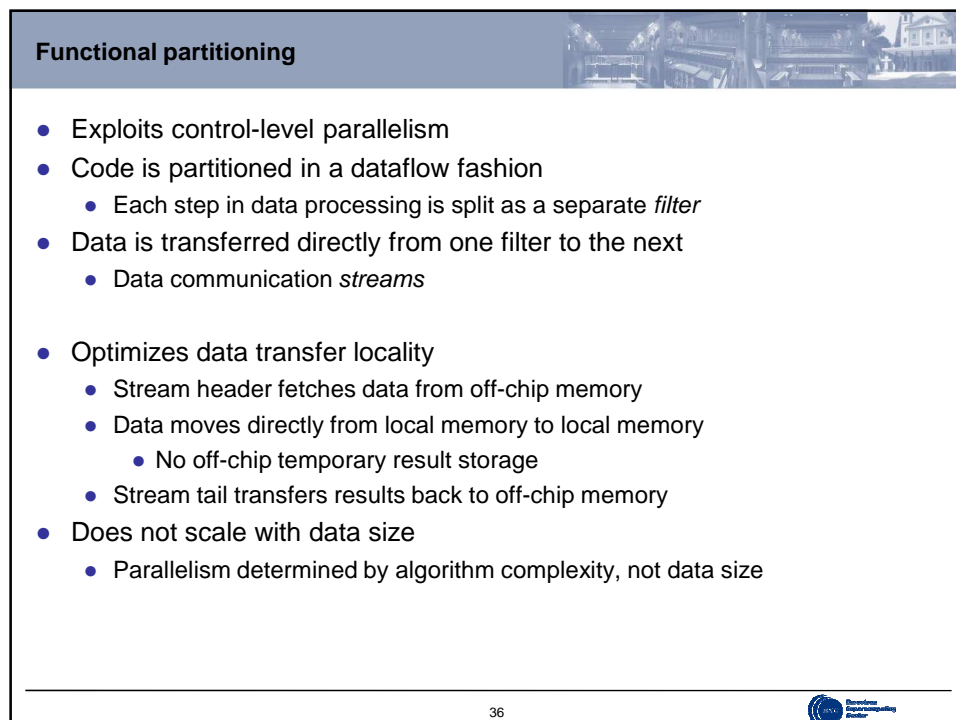
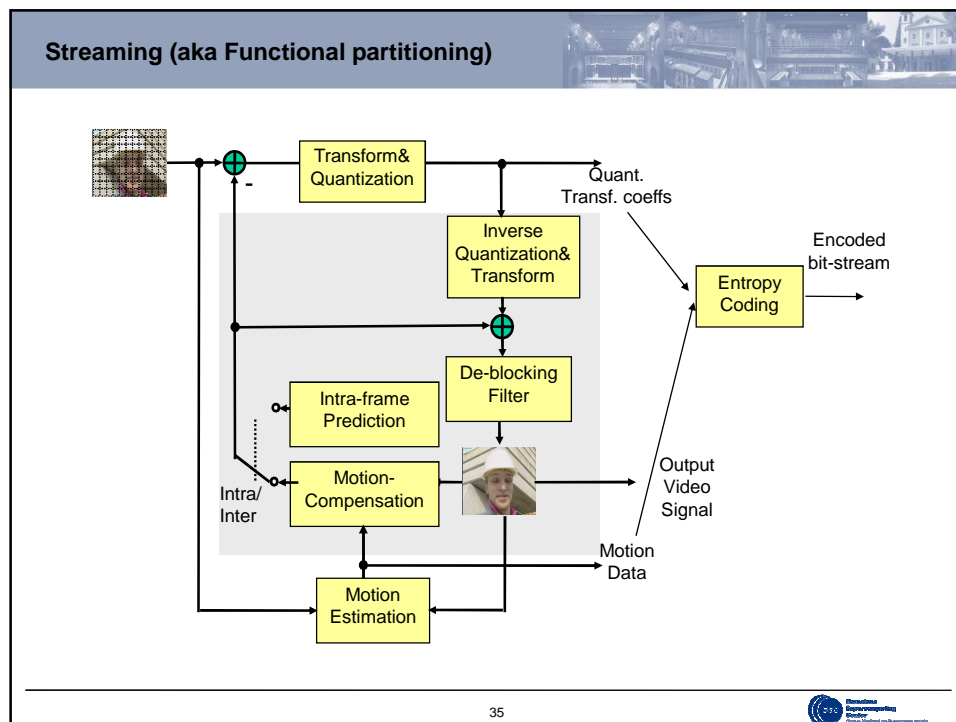
- AMD Opterons used to drive work to / from the QS22 blade
- Hypertransport + PCIe links for board-to-board memory transfers
 - Dedicated expansion blade for IO to cross from PCIe to HT
 - Application state is replicated on the LS21 and QS22

33

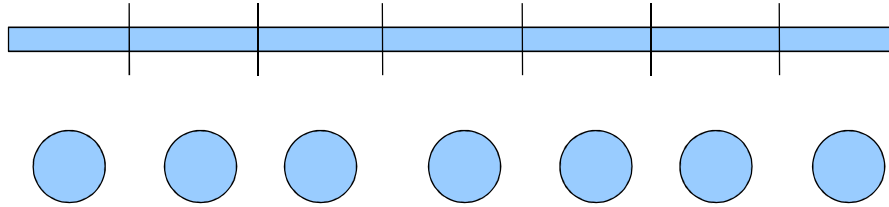
Outline

- Introduction
- Cell architecture
 - Overview
 - Memory architecture
 - PPE architecture
 - SPE architecture
 - Element Interconnection Bus
- Programming models
 - Streaming
 - Fork-join
 - Task queue

34



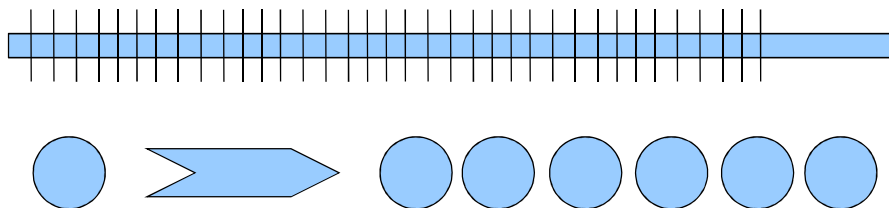
Fork-join (aka Loop partitioning)



- Exploits data-level parallelism: Single Program, Multiple Data (SPMD)
- Partition data space among the different processors
- Each SPE is responsible for fetching + processing its own data
 - The allocated data set is larger than the local memory
 - Local Store managed as a Software Cache

37

Task queue

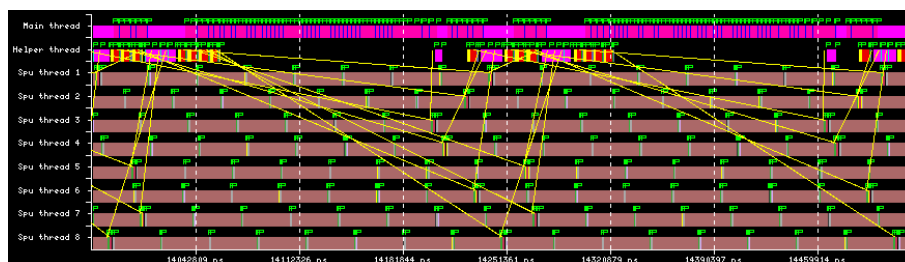


- Exploits data-level parallelism in a dataflow fashion
 - Multiple Program, Multiple Data (MPMD)
- Data is partitioned in blocks, each block processing is a task
- Master thread adds tasks to task queue
 - Each job requires fetch + processing of a single data block
- Workers get jobs from the queue
 - Use multiple queues to avoid critical sections
 - If own queue is empty, *steal* jobs from another queue

38

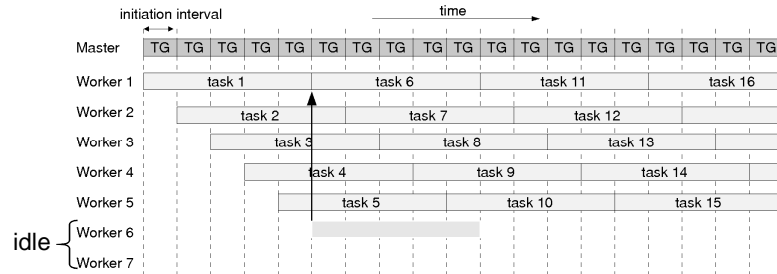
- ```
#pragma css task in(A), out(B)
void transpose(float A[64][64],
 float B[64][64])
{
 ... //executes on the SPEs
}

int main() //executes on the PPE
{
 ...
 #pragma css start
 ...
 transpose(matA, matB);
 ...
 #pragma css finish
 ...
}
```



- **Master thread (PPE 1<sup>st</sup> SMT slot):**
  - Starts *main* execution.
  - On an annotated function call, creates the task and adds it to the dependence graph.
- **Helper thread (PPE 2<sup>nd</sup> SMT slot):**
  - Schedules tasks from the dependence graph.
  - Groups tasks in bundles (default: 8 tasks/bundle).
  - Dispatches task bundles to SPEs.
- **Worker threads (SPEs):**
  - Execute tasks and notify task finalization to the Helper thread.

## Task Queue Scalability Analysis

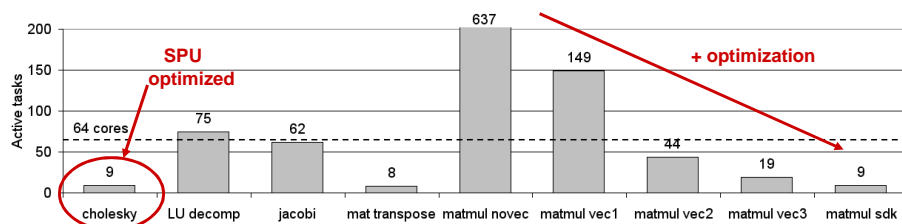


- Max Active tasks =  $\frac{\text{task execution time}}{\text{task generation time}}$
- Task generation and task execution were measured and averaged over 10 executions of each application.
  - IBM QS21 blades, 2xCell @ 3.2GHz, 512MB XDR

41



## Limited Scalability



- Only less-optimized versions of matmul and LU decomposition could fully utilize a 64-core multiprocessor (expected by Moore's Law by 2013).
- The more optimized, the less parallelism:
  - LU decomposition and jacobi are not optimized for Cell.
  - E.g. matmul SDK version tasks execute in ~25us while task generation takes ~3us.
- A next-generation Cell using the same PPE and an increased number of SPEs would not scale for most applications.

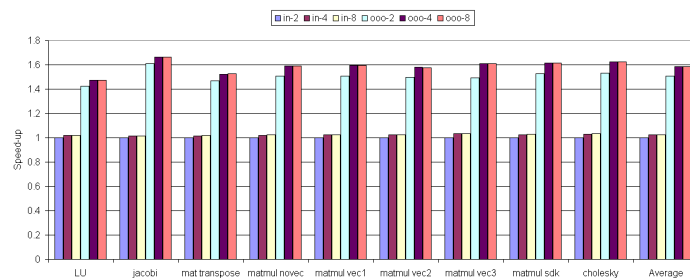
42



## How to increase the parallelism?

- Max Active tasks =  $\frac{\text{task execution time}}{\text{task generation time}}$
- Increase the task size
  - This would not work for application with
    - Fixed total problem size
      - Larger tasks → Less tasks (ie. less parallelism)
    - Fixed task size
      - E.g. multimedia applications work on fixed-size block (H.264, MPEG-4)
  - Task size is also limited by the data that fits in the Local Store
  - NOT A GLOBAL SOLUTION
- Faster task generation
  - Faster PPE: faster processor, better caches, ...
  - Quantitative evaluation is very important: power and area restrictions

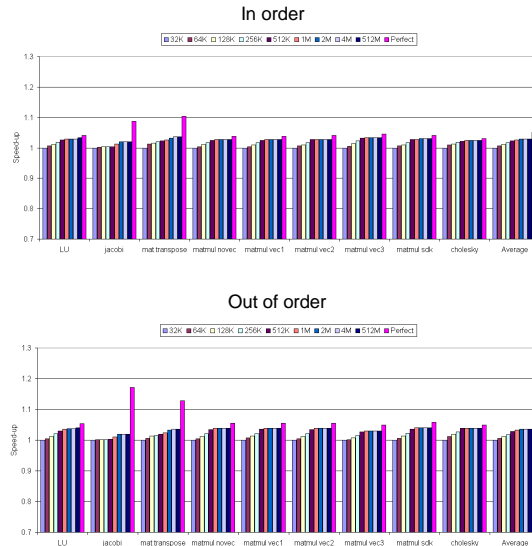
## Out of order execution gets 50% speedup



- All in-order and out-of-order configurations are evaluated.
- Out of order execution improves performance by a 50% on average.
  - More functional units provides up to 10% extra improvement
- This results shows that the task generation phase on CellSs is very sensitive to memory latency on cache misses.

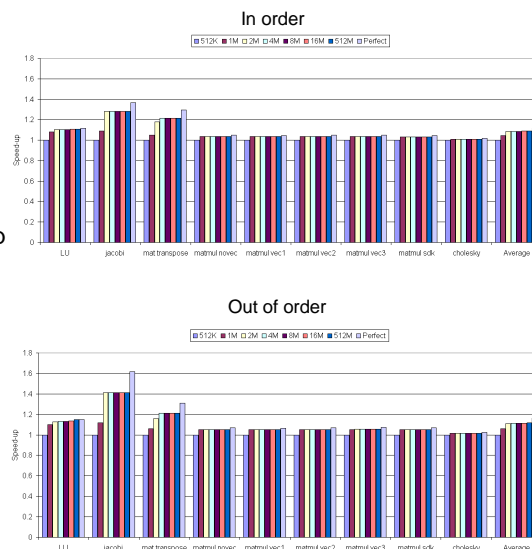
## L1 cache size has small impact up to 2-4MB

- L1 cache sizes evaluated:
  - 32KB to 4MB
  - a pseudo-infinite 512MB
  - Perfect (always hit).
- L2 is 512MB.
- Performance improves up to 2-4MB.
- Small impact:
  - In-order: 3%
  - OOO: 3.5%
- Perfect:
  - In-order: 5.2%
  - OOO: 7.5%

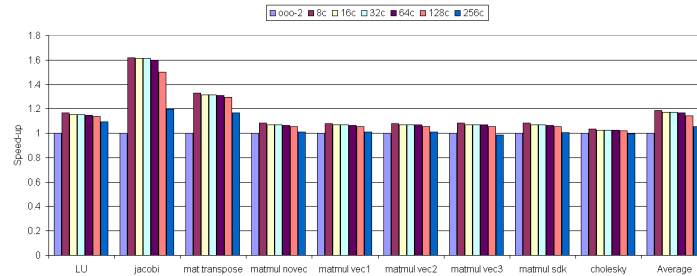


## L2 cache size provides ~10% improvement for 2-4MB

- L2 cache sizes evaluated:
  - 512KB to 16MB
  - a pseudo-infinite 512MB
  - Perfect (always hit).
- L1 is 32KB.
- Performance also improves up to 2-4MB.
- Small impact:
  - In-order: 9%
  - OOO: 11.5%
- Perfect:
  - In-order: 11.5%
  - OOO: 16%



## Local memory for task generation improves out-of-order performance by 20%



- As seen in the previous experiments, the data used by task generation could fit in a 2-4MB local memory.
- Such a large on-chip memory would be slower, so latencies from 8 to 256 cycles are evaluated and compared to the ooo-2 configuration.
- Having a local memory would improve out-of-order performance by close to 20% even with a 64-cycle latency on average.

## Conclusions (I)

- Cell is a 9-processor heterogeneous CMP
  - Multiple ISA
  - Distributed memory
- Programming Cell is difficult
  - Distributed memory programming model
    - Must partition + transfer data to processors
    - Performance heavily depends on good use of the DMA engine
  - DMA programming harder than it should be
    - Lots of implementation-specific restrictions
  - SIMD-only SPU instruction set
    - Automatic vectorization not up to the challenge
    - Poor performance on scalar / control code
      - Even worse since there is no branch prediction
  - VLIW pipeline on the SPU
    - Instruction scheduler not always good enough
  - Poor PPE performance



## Conclusions (II)



- But, there is nothing wrong with the concept!
  - Heterogeneous CMP, multiple ISA
    - Assign each task to the most adequate processor
  - Distributed memory
    - Higher scalability + efficiency
- Lots of room for improvement on the implementation
  - Easier ISA on the SPU side
    - Automatic code generation by the compiler
  - Avoid architecture-driven restrictions
    - Even if they have a cost in performance?
  - Provide at least one high-performance processor
    - Task generation, sequential parts of the code