Multicores, Multiprocessors, and Clusters

Introduction

- Goal: connecting multiple computers to get higher performance
  - Multiprocessors
  - Scalability, availability, power efficiency
- Job-level (process-level) parallelism
  - High throughput for independent jobs
- Parallel processing program
  - Single program run on multiple processors
- Multicore microprocessors
  - Chips with multiple processors (cores)

THE UNIVERSITY OF ARIZONA

Hardware and Software

- Hardware
  - Serial: e.g., Pentium 4
  - Parallel: e.g., quad-core Xeon e5345
- Software
  - Sequential: e.g., matrix multiplication
  - Concurrent: e.g., operating system
- Sequential/concurrent software can run on serial/parallel hardware
  - Challenge: making effective use of parallel hardware

THE UNIVERSITY OF ARIZONA

Parallel Programming

- Parallel software is the problem
- Need to get significant performance improvement
  - Otherwise, just use a faster uniprocessor, since it's easier!
- Difficulties
  - Partitioning
  - Coordination
  - Communications overhead

ece

Amdahl's Law

- Sequential part can limit speedup
- Example: 100 processors, 90× speedup?
  - $T_{new} = T_{parallelizable}/100 + T_{sequential}$

$$\text{Speedup} = \frac{1}{(1 - F_{parallelizable}) + F_{parallelizable}/100} = 90$$

  -
  - Solving: $F_{parallelizable} = 0.999$
- Need sequential part to be 0.1% of original time

Scaling Example

- Workload: sum of 10 scalars, and 10 × 10 matrix sum
  - Speed up from 10 to 100 processors
- Single processor: Time = $(10 + 100) \times t_{add}$
- 10 processors
  - Time = $10 \times t_{add} + 100/10 \times t_{add} = 20 \times t_{add}$
  - Speedup = 110/20 = 5.5 (55% of potential)
- 100 processors
  - Time = $10 \times t_{add} + 100/100 \times t_{add} = 11 \times t_{add}$
  - Speedup = 110/11 = 10 (10% of potential)
- Assumes load can be balanced across processors

Scaling Example (cont)

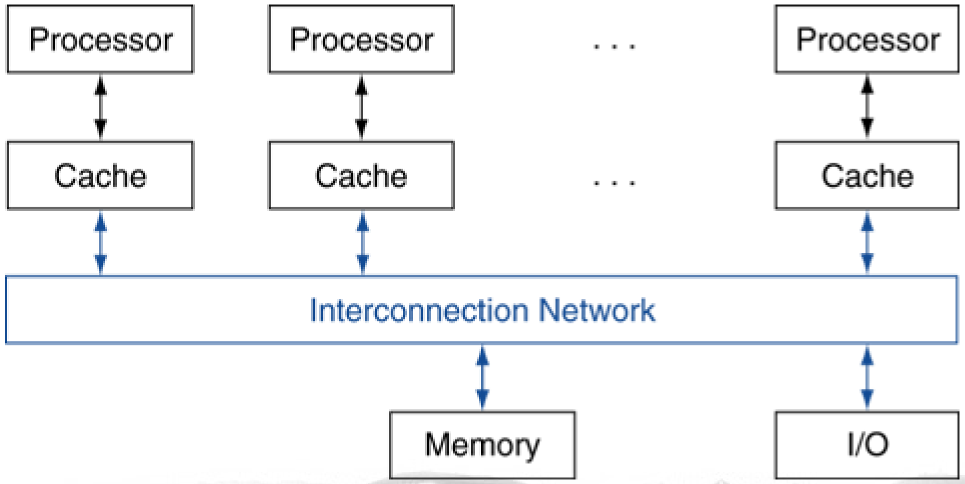
- What if matrix size is 100 × 100?
- Single processor: Time = $(10 + 10000) \times t_{add}$
- 10 processors
  - Time = $10 \times t_{add} + 10000/10 \times t_{add} = 1010 \times t_{add}$
  - Speedup = 10010/1010 = 9.9 (99% of potential)
- 100 processors
  - Time = $10 \times t_{add} + 10000/100 \times t_{add} = 110 \times t_{add}$
  - Speedup = 10010/110 = 91 (91% of potential)
- Assuming load balanced

Strong vs Weak Scaling

- Strong scaling: problem size fixed
  - As in example
- Weak scaling: problem size proportional to number of processors
  - 10 processors, 10 × 10 matrix
    - Time = $20 \times t_{add}$
  - 100 processors, 32 × 32 matrix
    - Time = $10 \times t_{add} + 1000/100 \times t_{add} = 20 \times t_{add}$
  - Constant performance in this example

Shared Memory

- SMP: shared memory multiprocessor
  - Hardware provides single physical address space for all processors
  - Synchronize shared variables using locks
  - Memory access time
    - UMA (uniform) vs. NUMA (nonuniform)



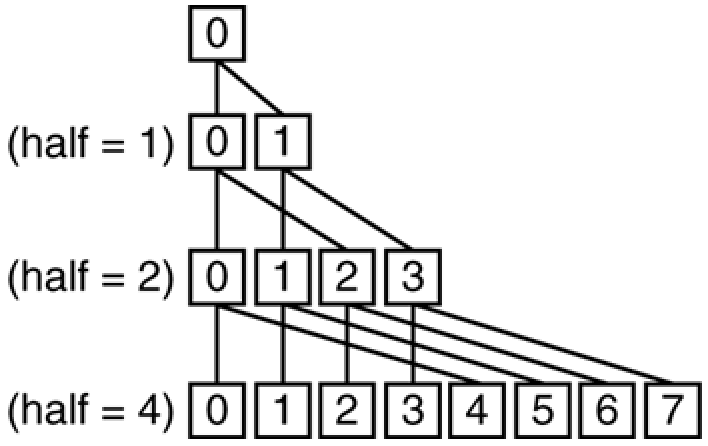
Example: Sum Reduction

- Sum 100,000 numbers on 100 processor UMA
  - Each processor has ID: $0 \le Pn \le 99$
  - Partition 1000 numbers per processor
  - Initial summation on each processor

```
sum[Pn] = 0;
  for (i = 1000*Pn;
       i < 1000*(Pn+1); i = i + 1)
     sum[Pn] = sum[Pn] + A[i];
```

- Now need to add these partial sums
  - Reduction: divide and conquer
  - Half the processors add pairs, then quarter, …
  - Need to synchronize between reduction steps

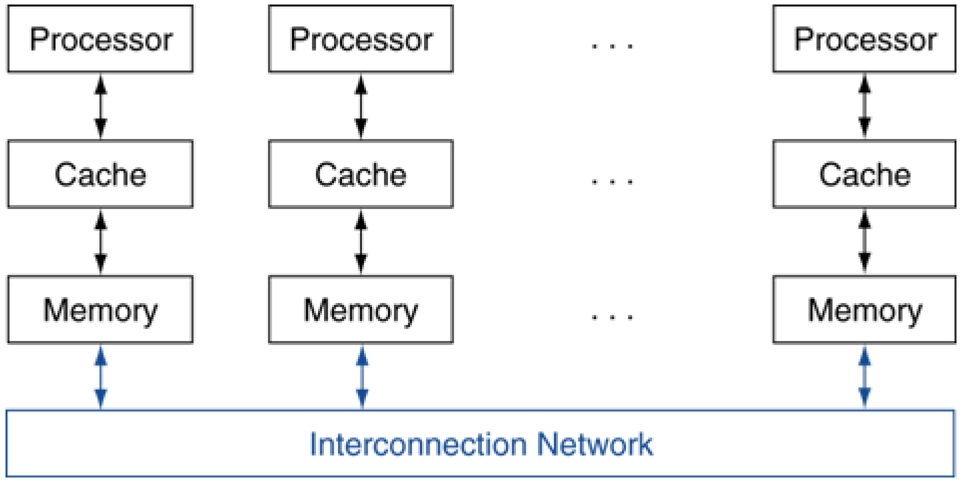
Example: Sum Reduction



```
half = 100;
repeat
  synch();
  if (half%2 != 0 && Pn == 0)
    sum[0] = sum[0] + sum[half-1];
    /* Conditional sum needed when half is odd;
       Processor0 gets missing element */
  half = half/2; /* dividing line on who sums */
  if (Pn < half) sum[Pn] = sum[Pn] +
  sum[Pn+half];
until (half == 1);
```

Message Passing

- Each processor has private physical address space
- Hardware sends/receives messages between processors



Loosely Coupled Clusters

- Network of independent computers
  - Each has private memory and OS
  - Connected using I/O system
    - E.g., Ethernet/switch, Internet
- Suitable for applications with independent tasks
  - Web servers, databases, simulations, …
- High availability, scalable, affordable
- Problems
  - Administration cost (prefer virtual machines)
  - Low interconnect bandwidth
    - c.f. processor/memory bandwidth on an SMP

Sum Reduction (Again)

- Sum 100,000 on 100 processors
- First distribute 100 numbers to each
  - The do partial sums

```
sum = 0;
for (i = 0; i<1000; i = i + 1)
   sum = sum + AN[i];
```

- Reduction
  - Half the processors send, other half receive and add
  - The quarter send, quarter receive and add,…

Sum Reduction (Again)

- Given send() and receive() operations

```
limit = 100; half = 100;/* 100 processors */
repeat
  half = (half+1)/2; /* send vs. receive
                        dividing line */
  if (Pn >= half && Pn < limit)
    send(Pn - half, sum);
  if (Pn < (limit/2))
    sum = sum + receive();
  limit = half; /* upper limit of senders */
until (half == 1); /* exit with final sum */
```

  – Send/receive also provide synchronization
  – Assumes send/receive take similar time to addition

Grid Computing

- Separate computers interconnected by long-haul networks
  - E.g., Internet connections
  - Work units farmed out, results sent back
- Can make use of idle time on PCs

Multithreading

- Performing multiple threads of execution in parallel
  - Replicate registers, PC, etc.
  - Fast switching between threads
- In multiple-issue dynamically scheduled processor
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when function units are available
  - Within threads, dependencies handled by scheduling and register renaming
- Example: Intel Pentium-4 HT
  - Two threads: duplicated registers, shared function units and caches

Future of Multithreading

- Will it survive? In what form?
- Power considerations $\Rightarrow$ simplified microarchitectures
  - Simpler forms of multithreading
- Tolerating cache-miss latency
- Multiple simple cores might share resources more effectively

Instruction and Data Streams

- An alternate classification

		Data Streams	
		Single	Multiple
Instruction Streams	Single	**SISD**: Intel Pentium 4	**SIMD**: SSE instructions of x86
	Multiple	**MISD**: No examples today	**MIMD**: Intel Xeon e5345

- SPMD: Single Program Multiple Data
  – A parallel program on a MIMD computer

SIMD

- Operate elementwise on vectors of data
  - E.g., MMX and SSE instructions in x86
    - Multiple data elements in 128-bit wide registers
- All processors execute the same instruction at the same time
  - Each with different data address, etc.
- Simplifies synchronization
- Reduced instruction control hardware
- Works best for highly data-parallel applications

Vector Processors

- Highly pipelined function units
- Stream data from/to vector registers to units
  - Data collected from memory into registers
  - Results stored from registers to memory
- Example: Vector extension
  - 32 × 64-element registers (64-bit elements)
  - Vector instructions
    - `lv`, `sv`: load/store vector
    - `addv.d`: add vectors of double
    - `addvs.d`: add scalar to each element of vector of double
- Significantly reduces instruction-fetch bandwidth

Example: (Y = a × X + Y)

- Conventional MIPS code

```
        l.d   $f0,a($sp)      ;load scalar a
        addiu r4,$s0,#512     ;upper bound of what to
load
loop:   l.d   $f2,0($s0)      ;load x(i)
        mul.d $f2,$f2,$f0     ;a × x(i)
        l.d   $f4,0($s1)      ;load y(i)
        add.d $f4,$f4,$f2     ;a × x(i) + y(i)
        s.d   $f4,0($s1)      ;store into y(i)
        addiu $s0,$s0,#8      ;increment index to x
        addiu $s1,$s1,#8      ;increment index to y
        subu  $t0,r4,$s0      ;compute bound
        bne   $t0,$zero,loop  ;check if done
```

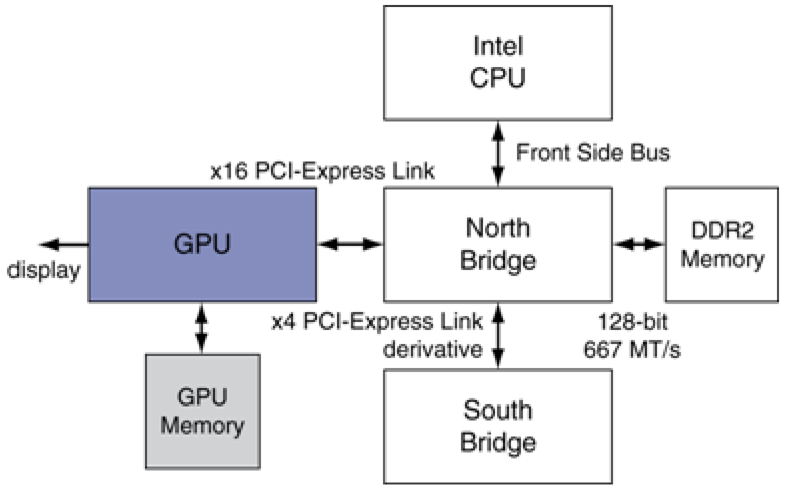
- Vector MIPS code

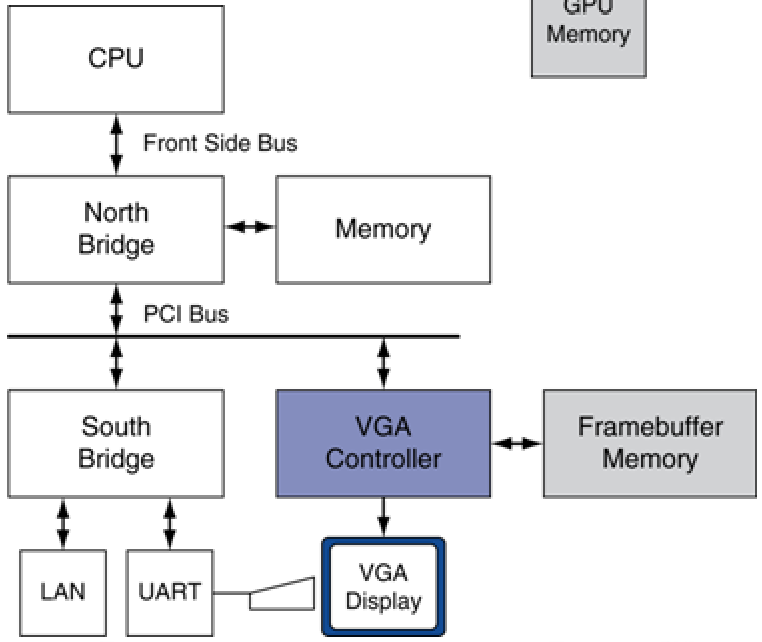
```
        l.d     $f0,a($sp)    ;load scalar a
        lv      $v1,0($s0)    ;load vector x
        mulvs.d $v2,$v1,$f0   ;vector-scalar multiply
        lv      $v3,0($s1)    ;load vector y
        addv.d  $v4,$v2,$v3   ;add y to product
        sv      $v4,0($s1)    ;store the result
```

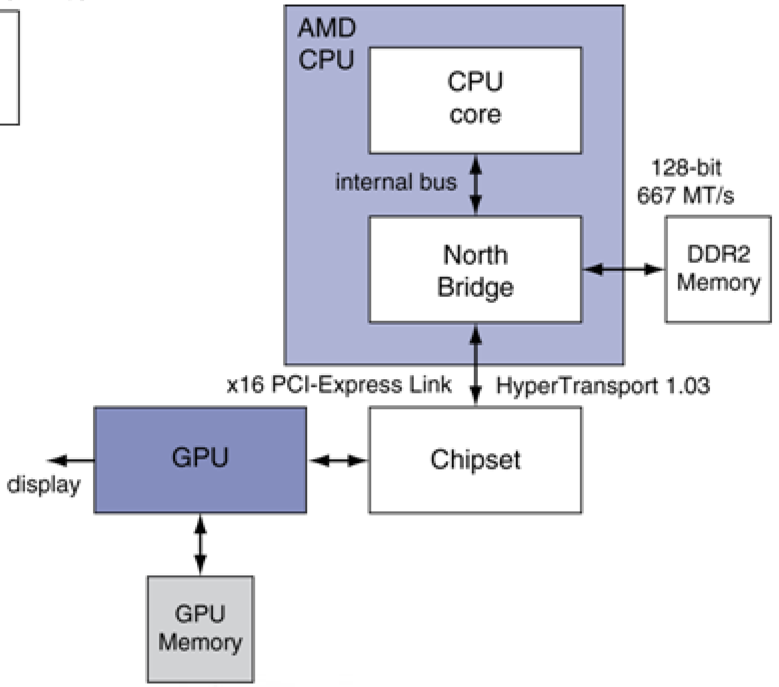
History of GPUs

- Early video cards
  - Frame buffer memory with address generation for video output
- 3D graphics processing
  - Originally high-end computers (e.g., SGI)
  - 3D graphics cards for PCs and game consoles
- Graphics Processing Units
  - Processors oriented to 3D graphics tasks
  - Vertex/pixel processing, shading, texture mapping, ray tracing

Graphics in the System







THE UNIVERSITY OF ARIZONA

GPU Architectures

- Processing is highly data-parallel
  - GPUs are highly multithreaded
  - Use thread switching to hide memory latency
    - Less reliance on multi-level caches
  - Graphics memory is wide and high-bandwidth
- Trend toward general purpose GPUs
  - Heterogeneous CPU/GPU systems
  - CPU for sequential code, GPU for parallel code
- Programming languages/APIs
  - DirectX, OpenGL
  - C for Graphics (Cg), High Level Shader Language (HLSL)
  - Compute Unified Device Architecture (CUDA)

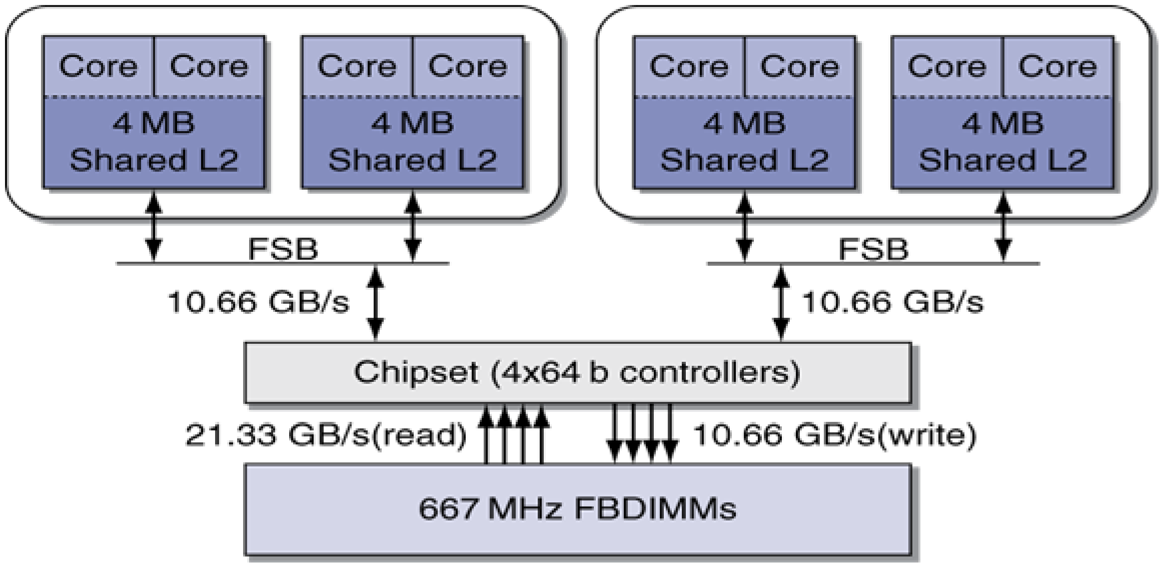
THE UNIVERSITY OF ARIZONA

Classifying GPUs

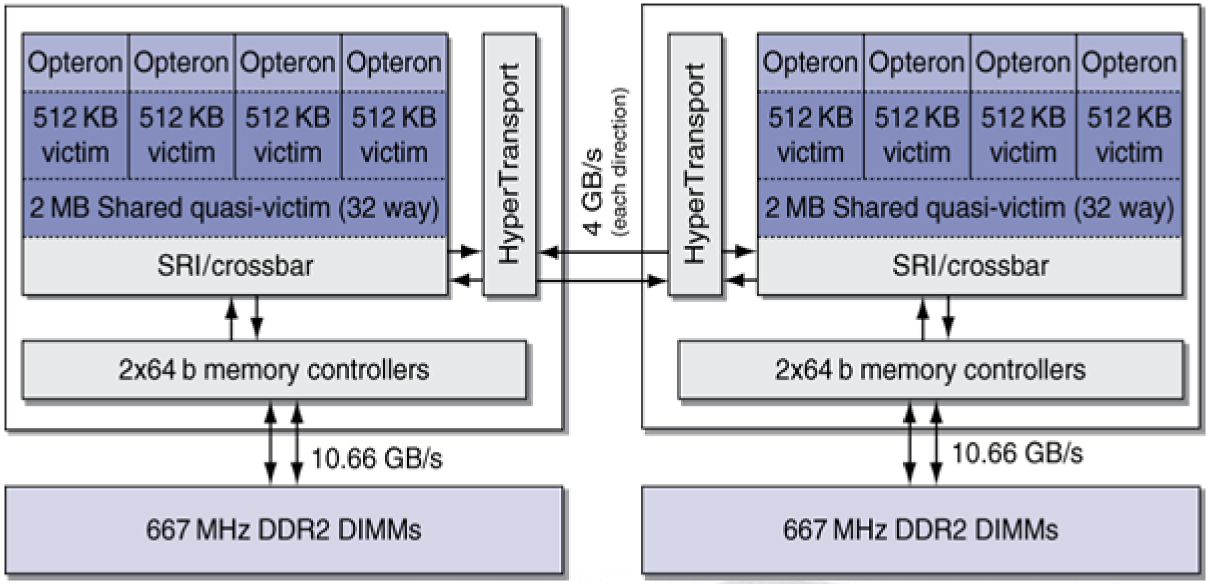
- Don’t fit nicely into SIMD/MIMD model
  - Conditional execution in a thread allows an illusion of MIMD
    - But with performance degredation
    - Need to write general purpose code with care

	Static: Discovered at Compile Time	Dynamic: Discovered at Runtime
Instruction-Level Parallelism	VLIW	Superscalar
Data-Level Parallelism	SIMD or Vector	**Tesla Multiprocessor**

Four Example Systems

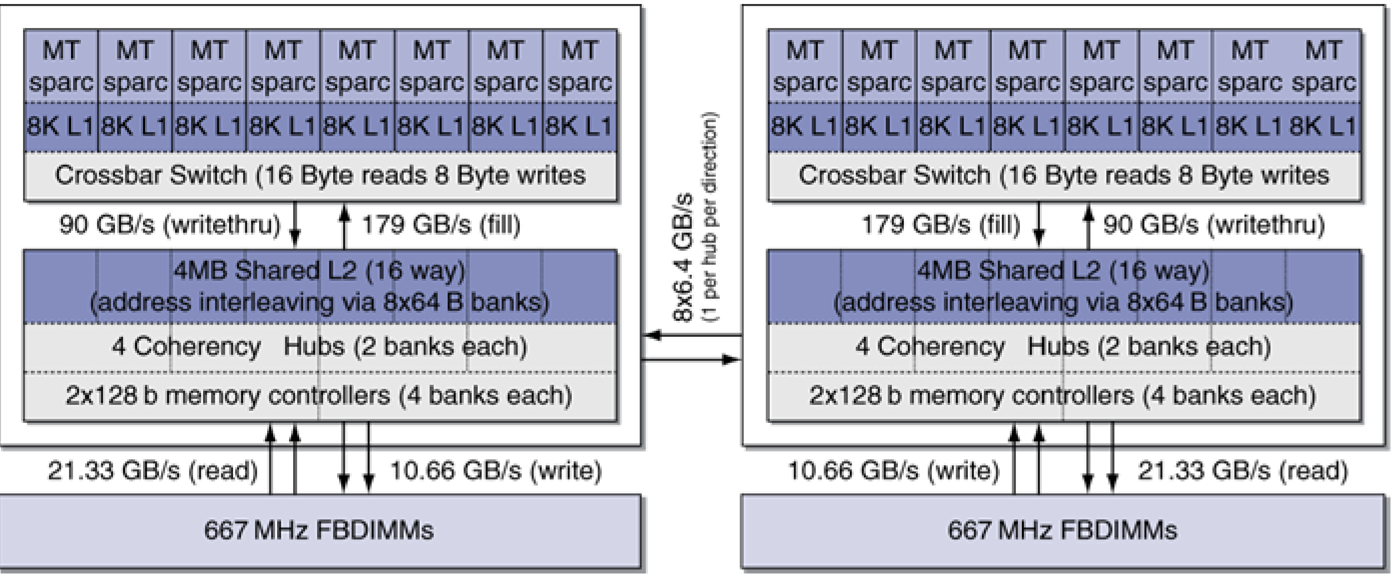


2 × quad-core
Intel Xeon e5345
(Clovertown)

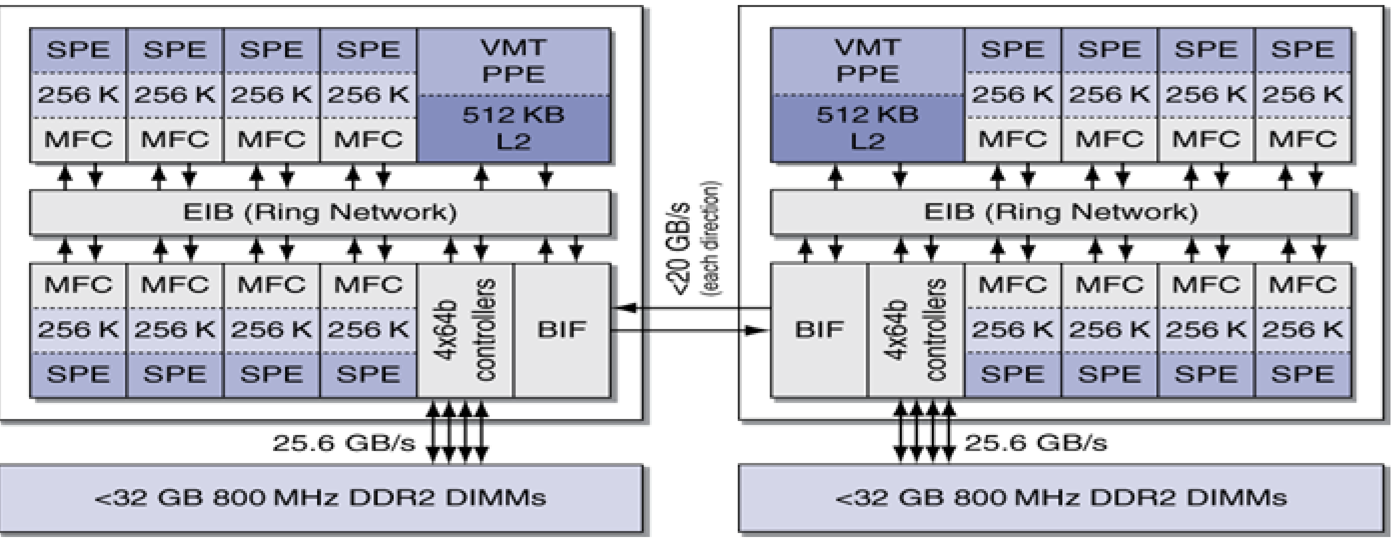


2 × quad-core
AMD Opteron X4 2356
(Barcelona)

Four Example Systems

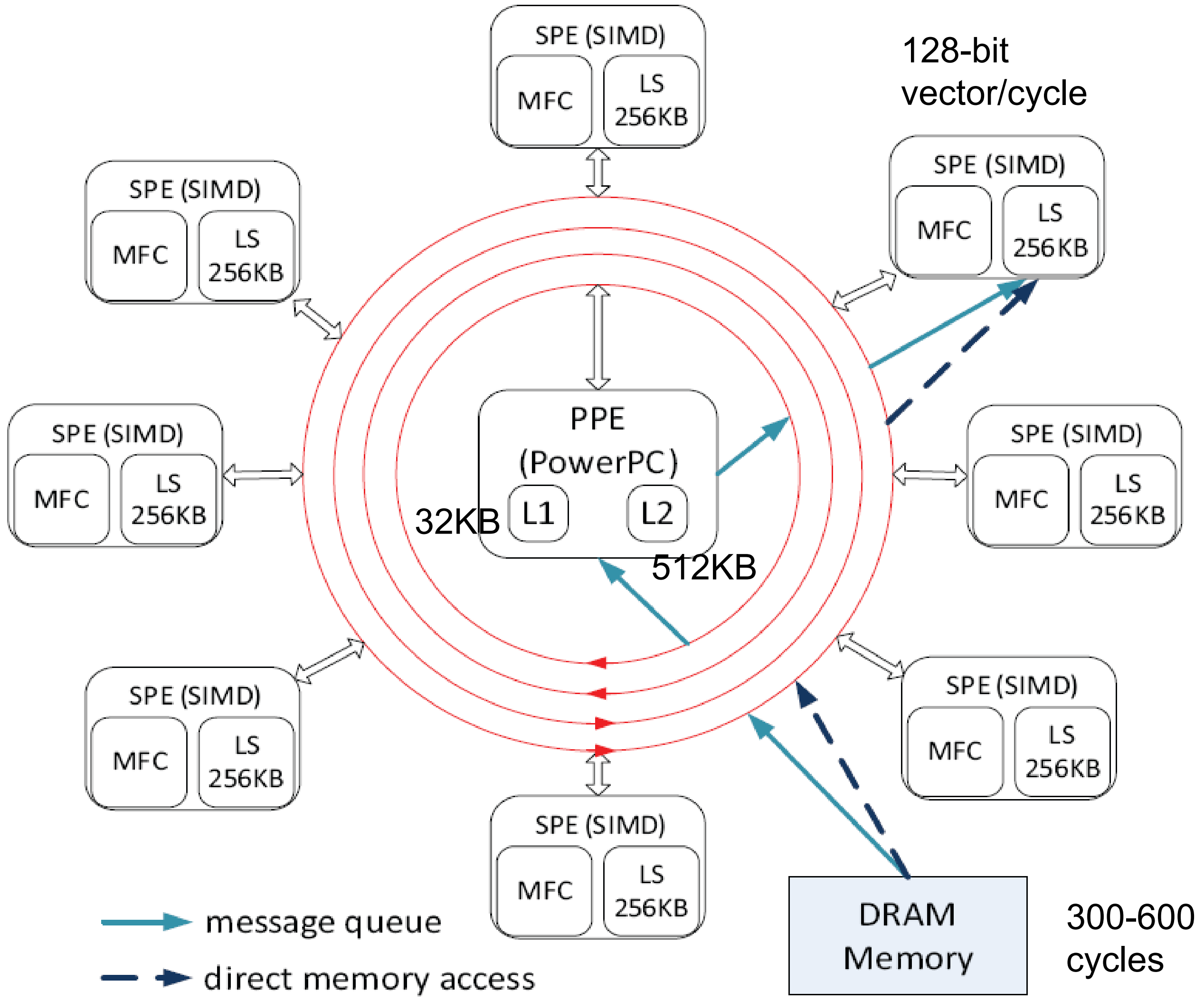

MT sparc
8K L1
Crossbar Switch (16 Byte reads 8 Byte writes
90 GB/s (writethru)
179 GB/s (fill)
4MB Shared L2 (16 way) (address interleaving via 8x64 B banks)
4 Coherency Hubs (2 banks each)
2x128 b memory controllers (4 banks each)
8x6.4 GB/s (1 per hub per direction)
21.33 GB/s (read)
10.66 GB/s (write)
667 MHz FBDIMMs

2 × oct-core
Sun UltraSPARC
T2 5140 (Niagara 2)



2 × oct-core
IBM Cell QS20

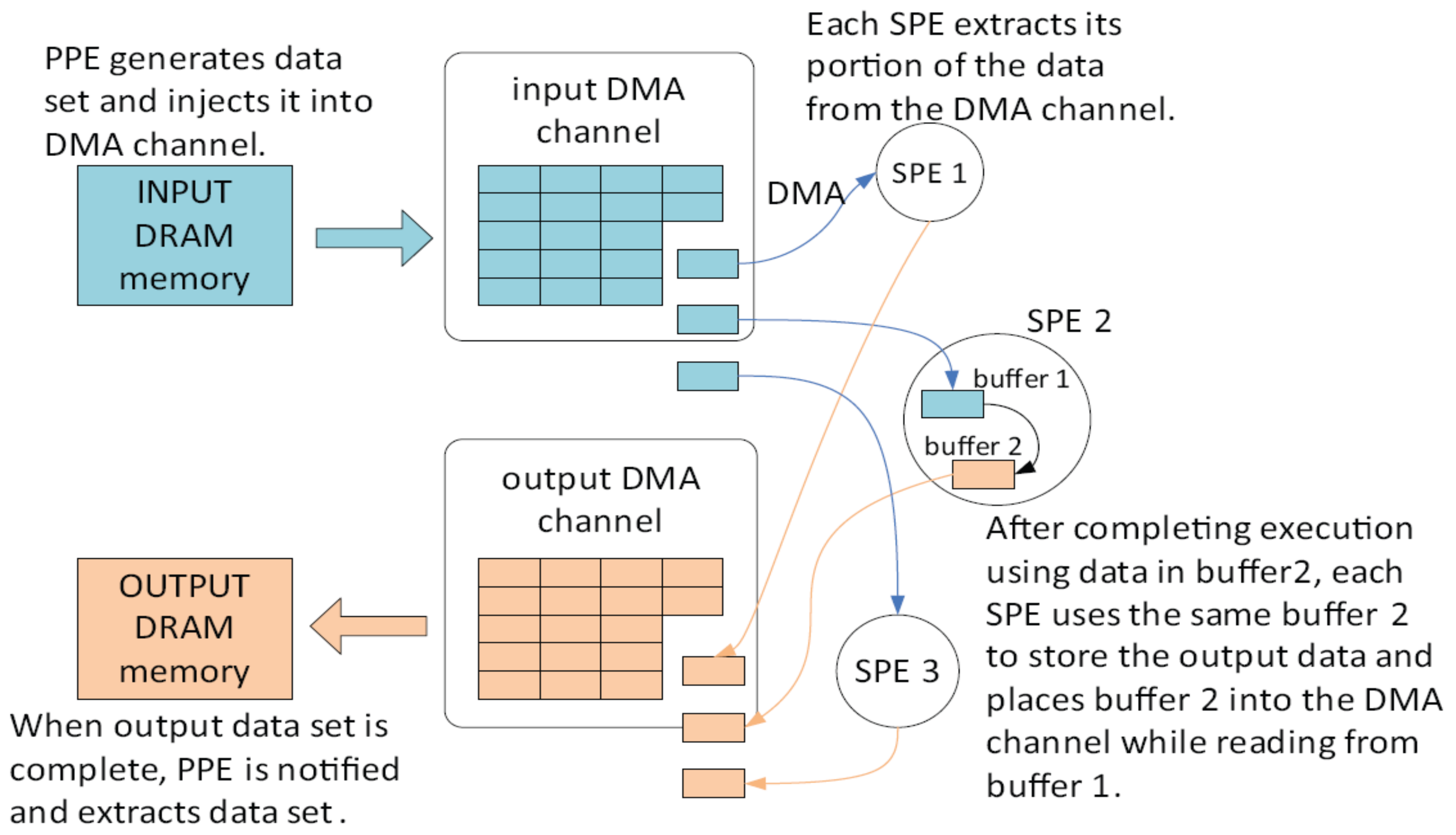
IBM Cell Broadband Engine



Abbreviations

PPE:	*PowerPC Engine*
SPE:	*Synergistic Processing Element*
MFC:	*Memory Flow Controller*
LS:	*Local Store*
SIMD:	*Single Instruction Multiple Data*

CELL BE Programming Model



PPE generates data set and injects it into DMA channel.
INPUT DRAM memory
input DMA channel
Each SPE extracts its portion of the data from the DMA channel.
DMA
SPE 1
SPE 2
buffer 1
buffer 2
SPE 3
output DMA channel
OUTPUT DRAM memory
After completing execution using data in buffer2, each SPE uses the same buffer 2 to store the output data and places buffer 2 into the DMA channel while reading from buffer 1.
When output data set is complete, PPE is notified and extracts data set.

No direct access to DRAM from LS of SPE,
Buffer size: 16KB

Reading #1

PARALLEL COMPUTING EXPERIENCES WITH CUDA

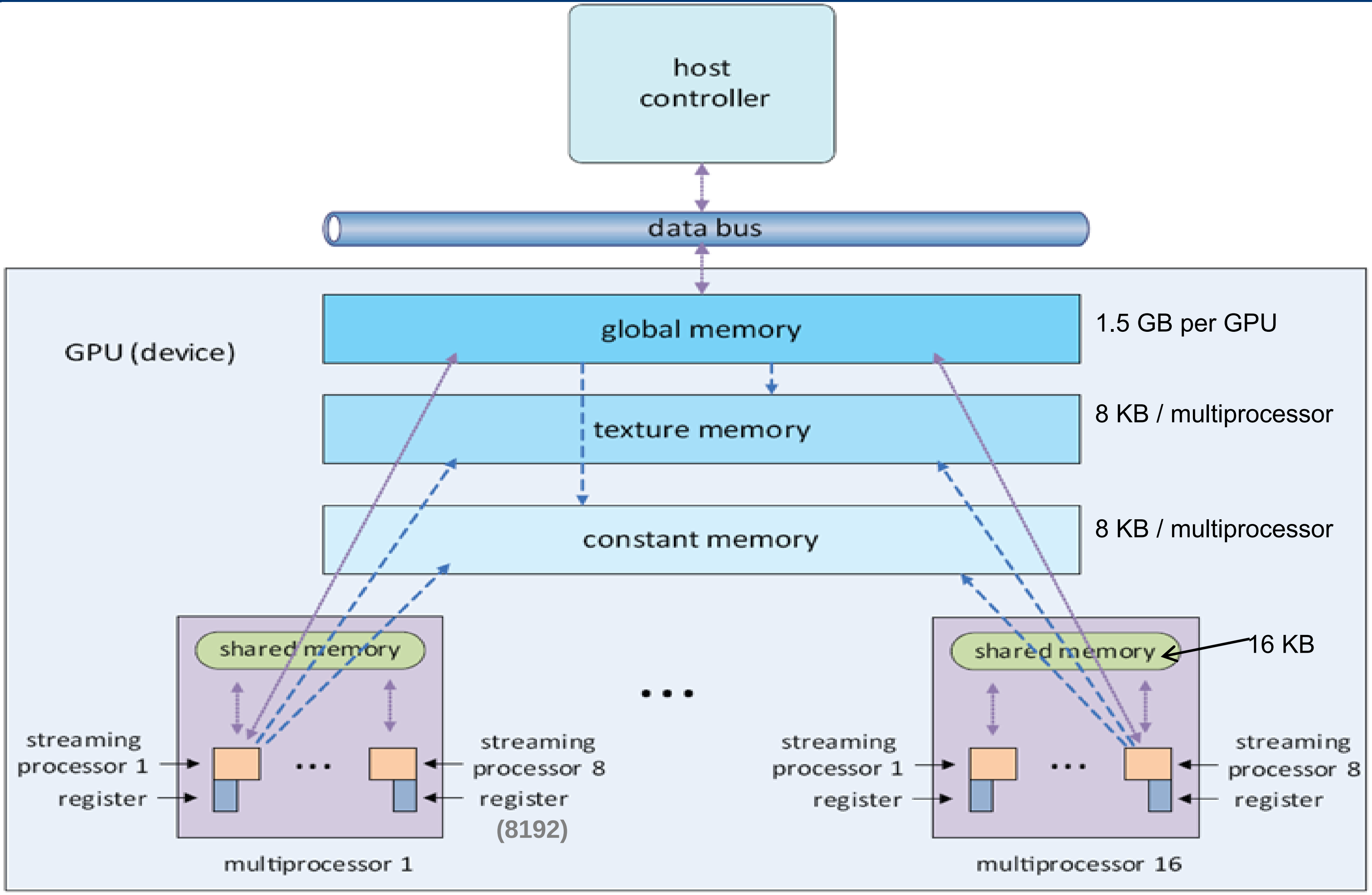
Reading #1

- Increasing parallelism vs. Clock rate
- Amdahl’s Law doesn’t apply to parallel computers
  - Since we can achieve linear speedup
  - But only on applications with weak scaling

Programming Model

- CUDA's design goals
  - extend a standard sequential programming language, specifically C/C++,
    - focus on the important issues of parallelism—how to craft efficient parallel algorithms—rather than grappling with the mechanics of an unfamiliar and complicated language.
  - minimalist set of abstractions for expressing parallelism
    - highly scalable parallel code that can run across tens of thousands of concurrent threads and hundreds of processor cores.

Tesla C870 GPU

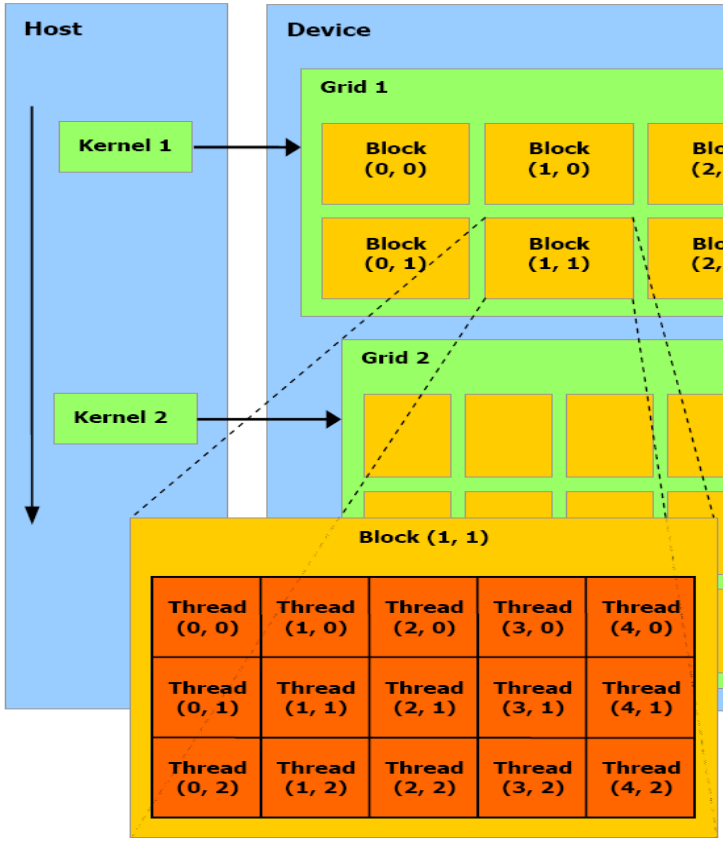


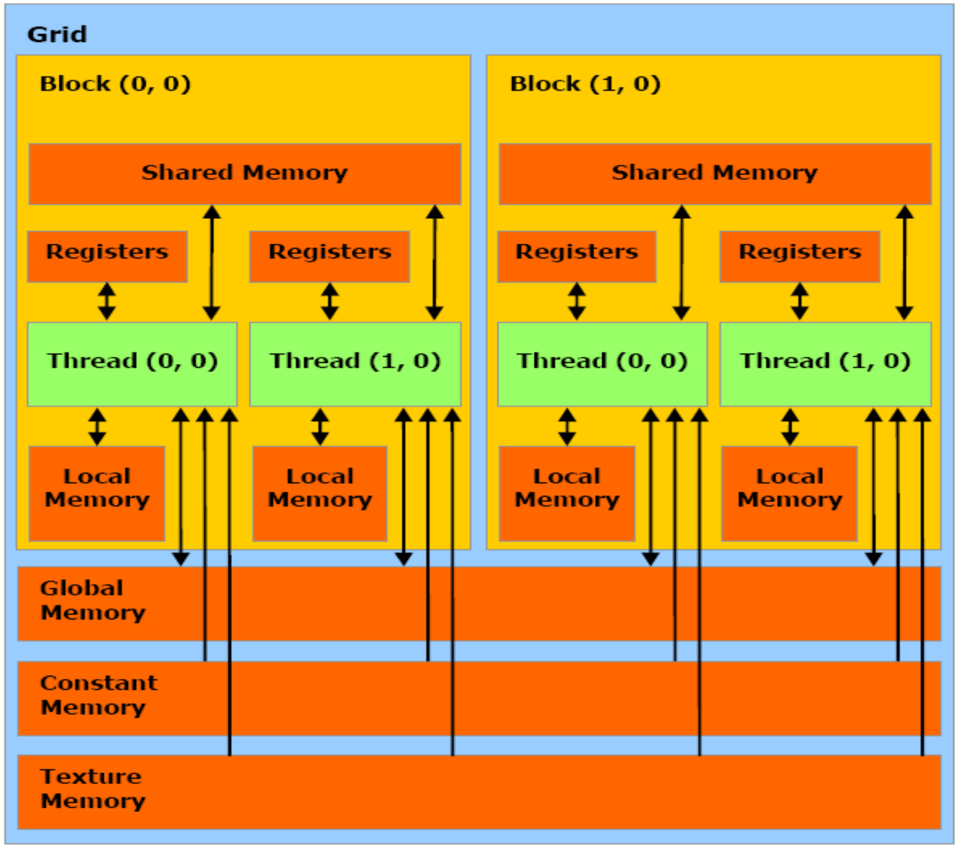
up to 768 threads (21 bytes of shared memory and 10 registers/thread)

CUDA Program Organization

- Host program and GPU
- One or more parallel kernels
  - Kernel
    - scalar sequential program on a set of parallel threads.
  - Threads
    - The programmer organizes grid of thread blocks.
  - Threads of a Block
    - allowed to synchronize via barriers
    - have access to a high-speed, per-block shared on-chip memory for interthread communication.
  - Threads from different blocks in the same grid
    - coordinate only via shared global memory space visible to all threads.
- CUDA requires that thread blocks be independent, meaning:
  - kernel must execute correctly no matter the order in which blocks are run,
  - provides scalability
- Main consideration in decomposing workload into kernels
  - the need for global communication
  - synchronization amongst threads

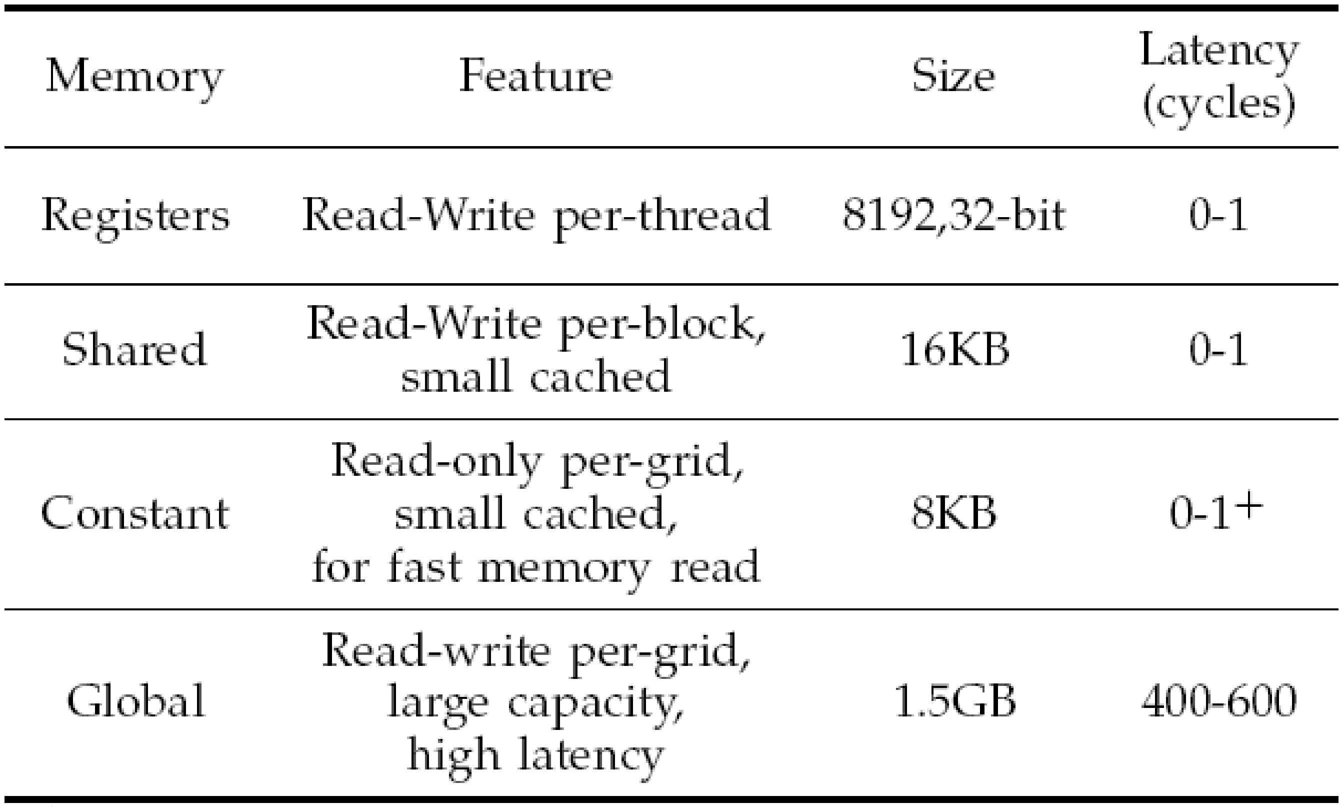
Tesla C870 Threads





THE UNIVERSITY OF ARIZONA

Tesla C870 Memory Hierarchy



Memory	Feature	Size	Latency (cycles)
Registers	Read-Write per-thread	8192,32-bit	0-1
Shared	Read-Write per-block, small cached	16KB	0-1
Constant	Read-only per-grid, small cached, for fast memory read	8KB	0-1+
Global	Read-write per-grid, large capacity, high latency	1.5GB	400-600

texture and constant caches: utilized by all the threads within the GPU

shared memory: individual and protected region available to threads of a multiprocessor

Example

Which element am I processing?

```
void saxpy(uint n, float a,
           float *x, float *y)
{
   for (uint i = 0; i<n; ++i)
      y[i] = a*x[i] + y[i];
}


void serial_sample ()
{
   // Call serial SAXPY function
   saxpy (n, 2.0, x,y);
}
```

(a)

```
__global__ void saxpy(uint n, float a,
                      float *x, float *y)
{
   uint i = blockIdx.x*blockDim.x
          + threadIdx.x;

   if( i<n ) y[i] = a*x[i] + y[i];
}

void parallel_sample()
{
   // Launch parallel SAXPY kernel
   // using ⌈n/256⌉ blocks of 256
   // threads each
   saxpy<<<ceil(n/256),256>>>(n, 2, x, y);
}
```

B blocks of T threads

Data parallel vs. Task parallel kernels !
Expose more fine-grained parallelism

SIMT Architecture

- Threads of a block grouped into warps
  - containing 32 threads each.
- A warp’s threads
  - free to follow arbitrary and independent execution paths (DIVERGE)
  - collectively execute only a single instruction at any particular instant.
- Divergence and reconvergence
  - wait in turn while other threads of the warp execute the instructions
  - managed in hardware.
- Ideal match for typical data-parallel programs,
- SIMT execution of threads transparent to the programmer
- a scalar computational model.

Molecular dynamics

- particles are updated independently
- atom to a single thread.
- Millions of time steps
- Multiple kernels
- Fine grained parallelism
- Simple memory access

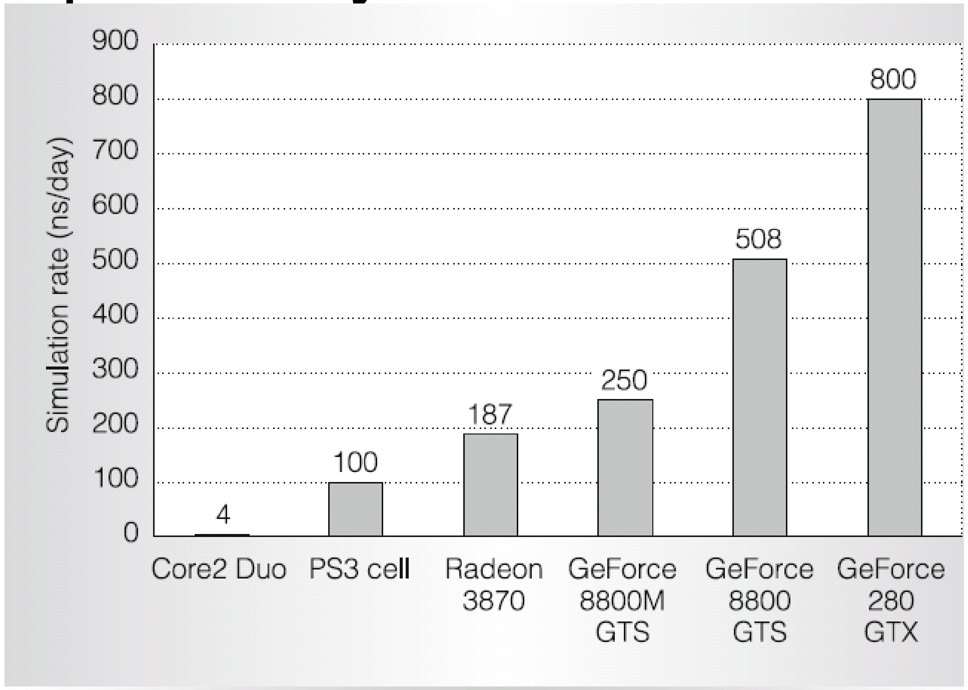


Figure 5. Performance of Folding@Home energy kernel on various platforms.

-Synchronous threads in a warp
-Each thread can interact with one atom's data
in shared memory at a time over p iterations
no need for additional synchronization

Potential Tricks

- Large register file
  - primary scratch space for computation.
- Small blocks of elements to each thread, rather than a single element to each thread
- The nonblocking nature of loads
  - software prefetching for hiding memory latency.

Fluid Dynamics

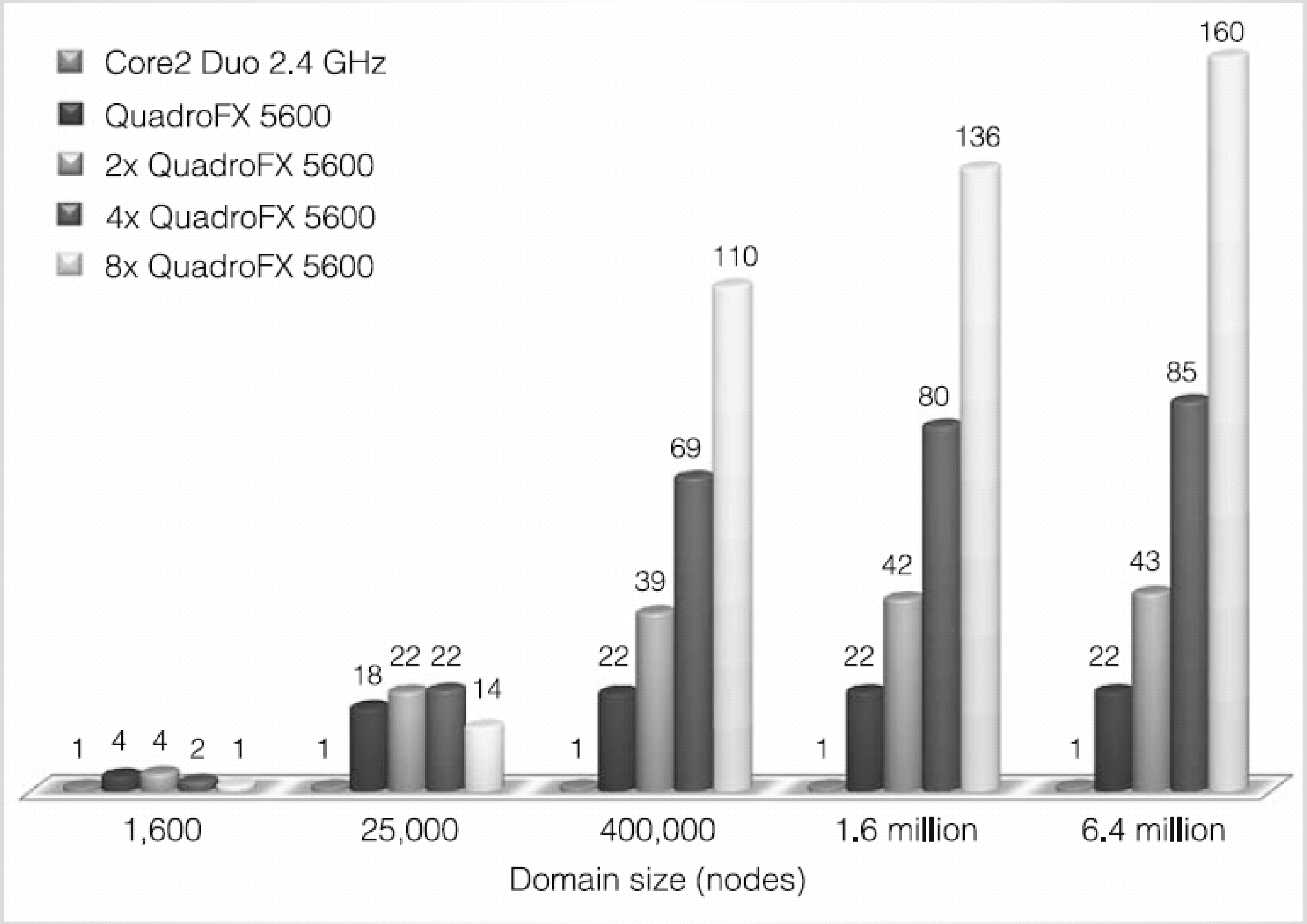
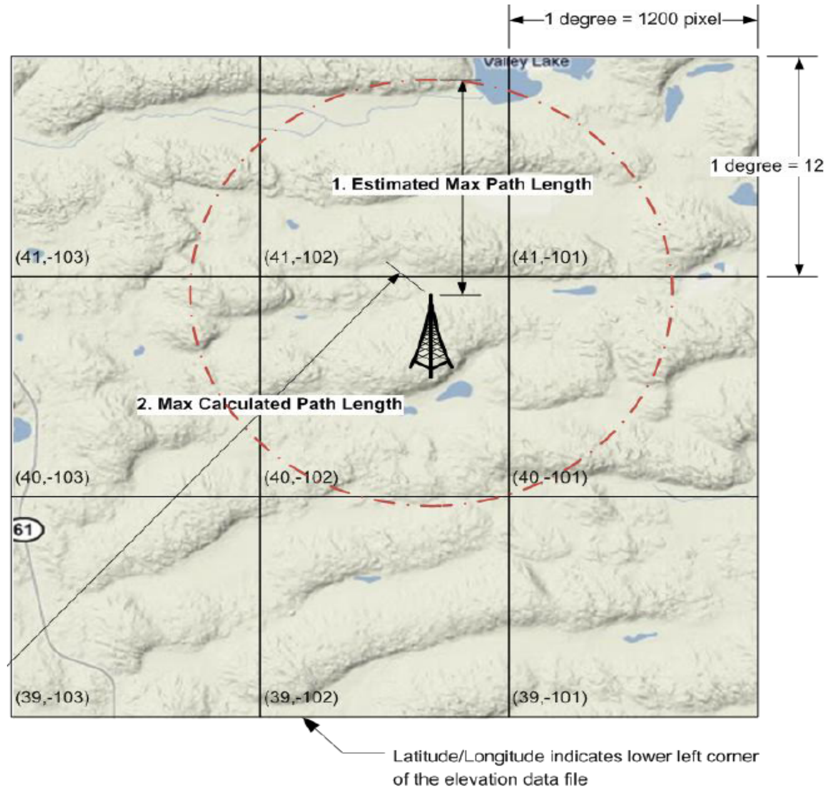
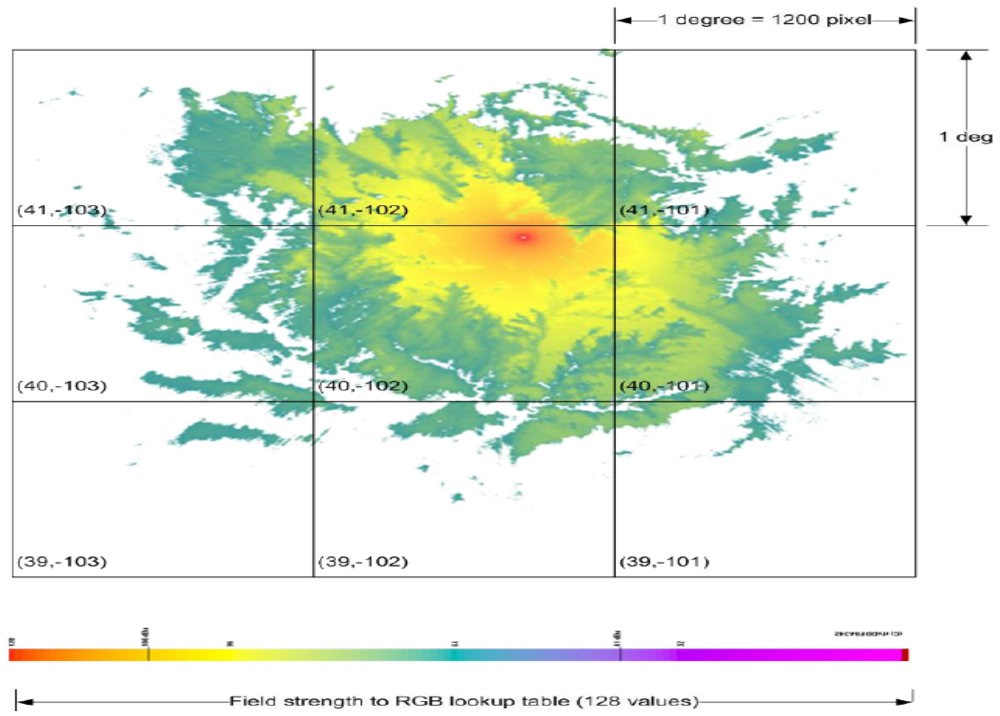


Figure 12. Performance speedup for GPU clusters of varying size over a single CPU for solving 2D Euler equations.

Case Study: Irregular Terrain Model



1 degree = 1200 pixel
Valley Lake
1 degree = 12
1. Estimated Max Path Length
2. Max Calculated Path Length
(41,-103)
(41,-102)
(41,-101)
(40,-103)
(40,-102)
(40,-101)
(39,-103)
(39,-102)
(39,-101)
61
Latitude/Longitude indicates lower left corner of the elevation data file



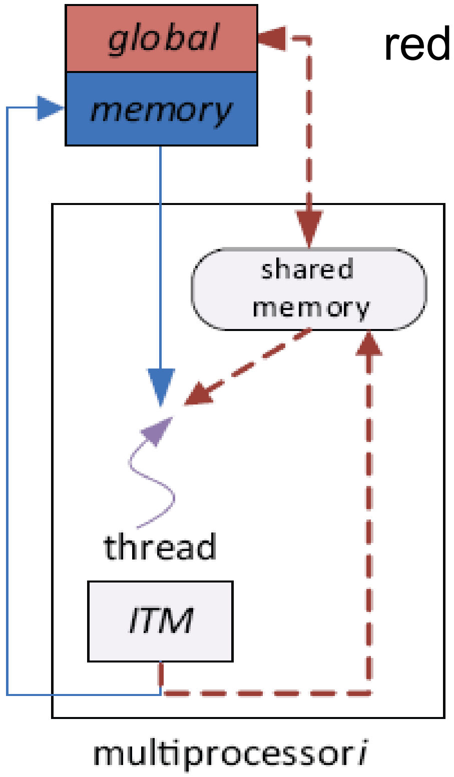
White space (Vacant frequency bands) exceeded the occupied spectrum after transition to digital TV

Utilization is possible with propagation loss models to detect occupied bands

Constraint: Near real-time!

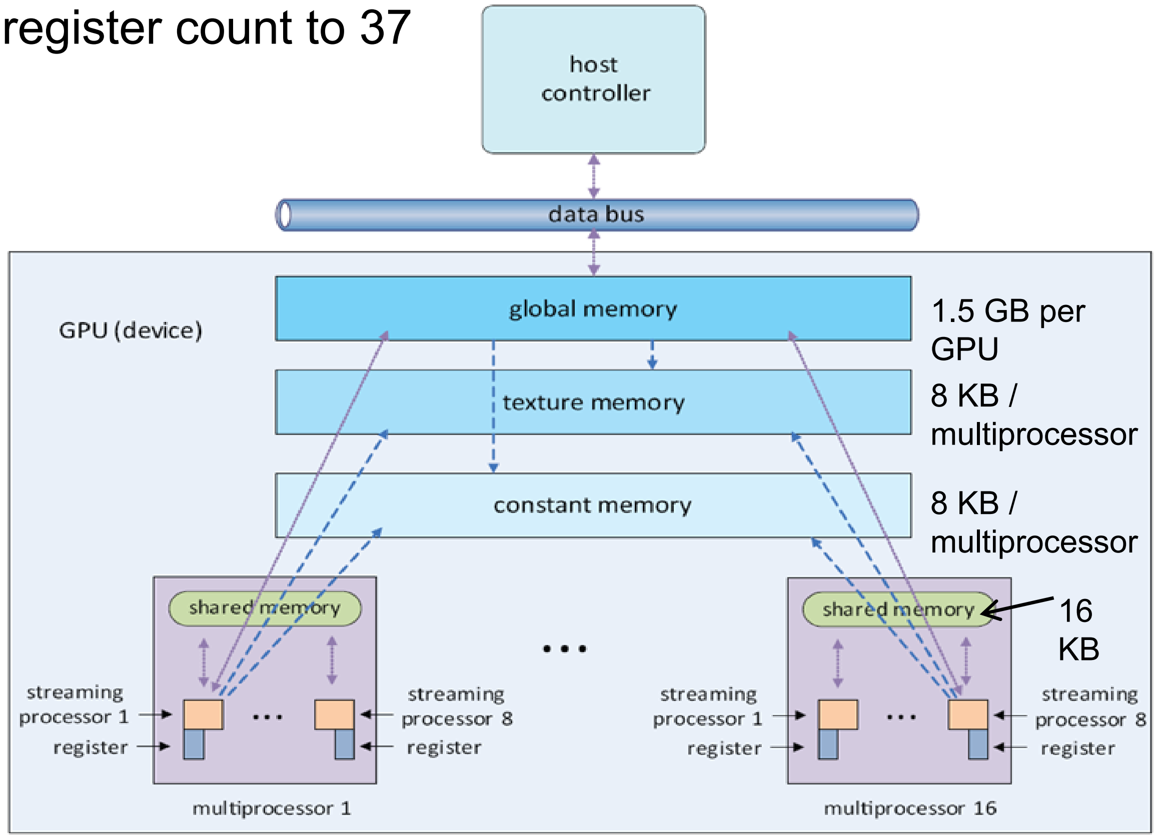
GPU Strategies for ITM

- ITM requires 45 registers
- Each profile is 1KB (radio frequency, path length, antenna heights, surface transfer impedance, plus 157 elevation points)



(c) strategy 3

reduces register count to 37



1.5 GB per GPU

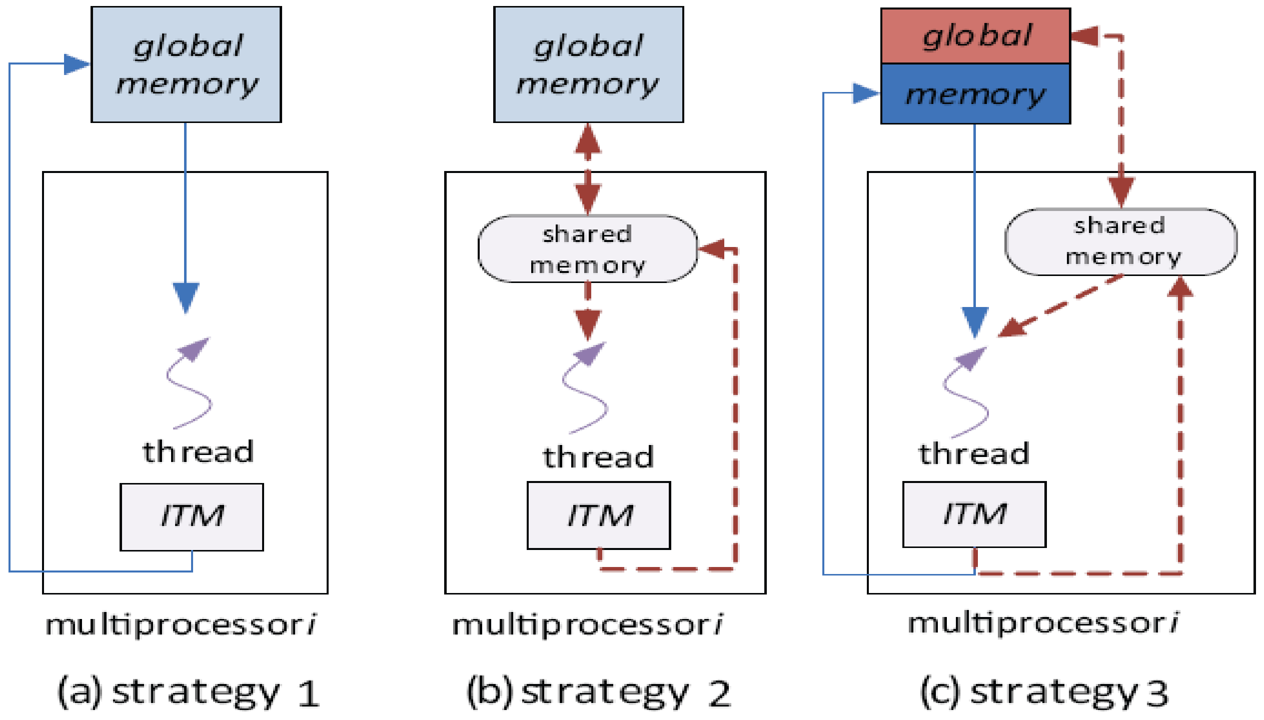
8 KB / multiprocessor

8 KB / multiprocessor

16 KB

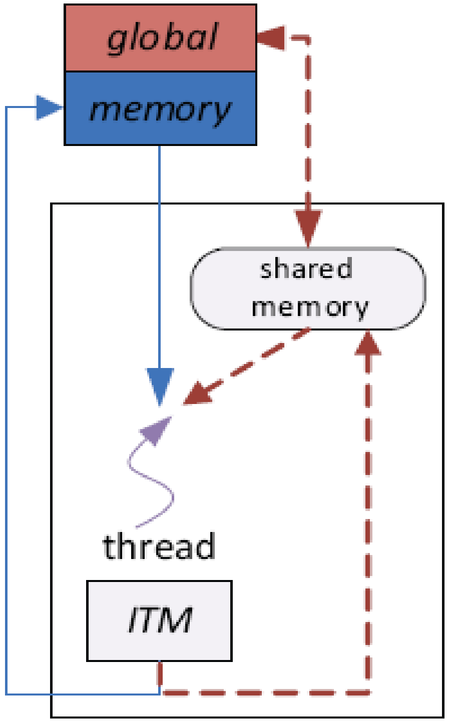
How many threads / MP?

GPU Strategies for ITM



128*16 threads   16*16 threads   192*16 threads

GPU Strategies for ITM



(c) strategy 3

t

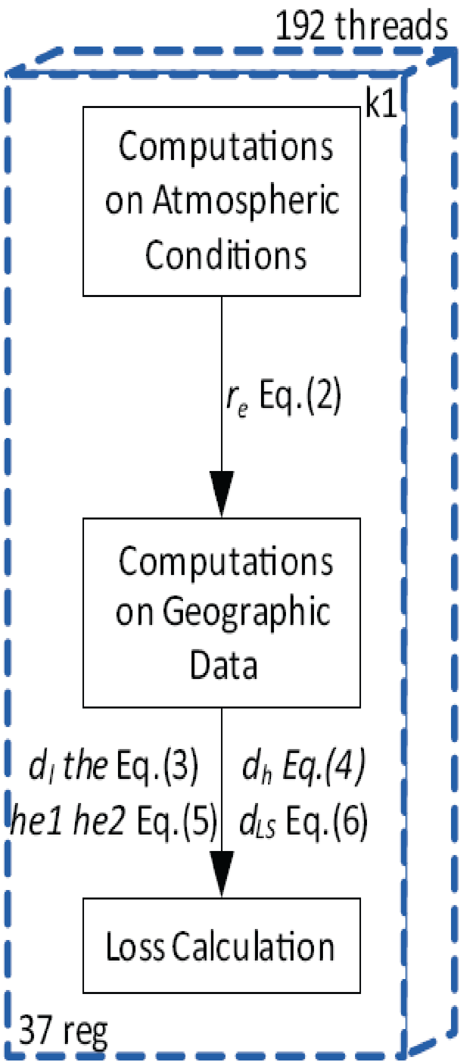
Execution time of strategy 3 (global and shared memory) with the available 192 threads for four different "threads per block" choices on Tesla C870.

Job Size	time(ms)			
	1 block 192 threads	2 blocks 96 threads	3 blocks 64 threads	6 blocks 32 threads
128	1.115	1.069	**1.033**	1.034
256	1.220	1.115	**1.035**	1.035
512	1.238	1.119	**1.038**	1.035
1024	1.366	1.302	**1.116**	1.110
2k	2.312	2.249	**1.404**	2.227
3k	2.602	2.511	**2.020**	2.479
4k	3.675	3.589	**3.003**	3.522
8k	7.210	7.109	**5.423**	7.015
16k	13.849	13.201	**11.036**	13.099
32k	27.121	26.854	**20.957**	26.177
64k	54.148	53.199	**42.085**	51.431
128k	109.672	105.571	**83.090**	102.881
256k	217.214	209.147	**166.434**	204.671

GPU Strategies for ITM

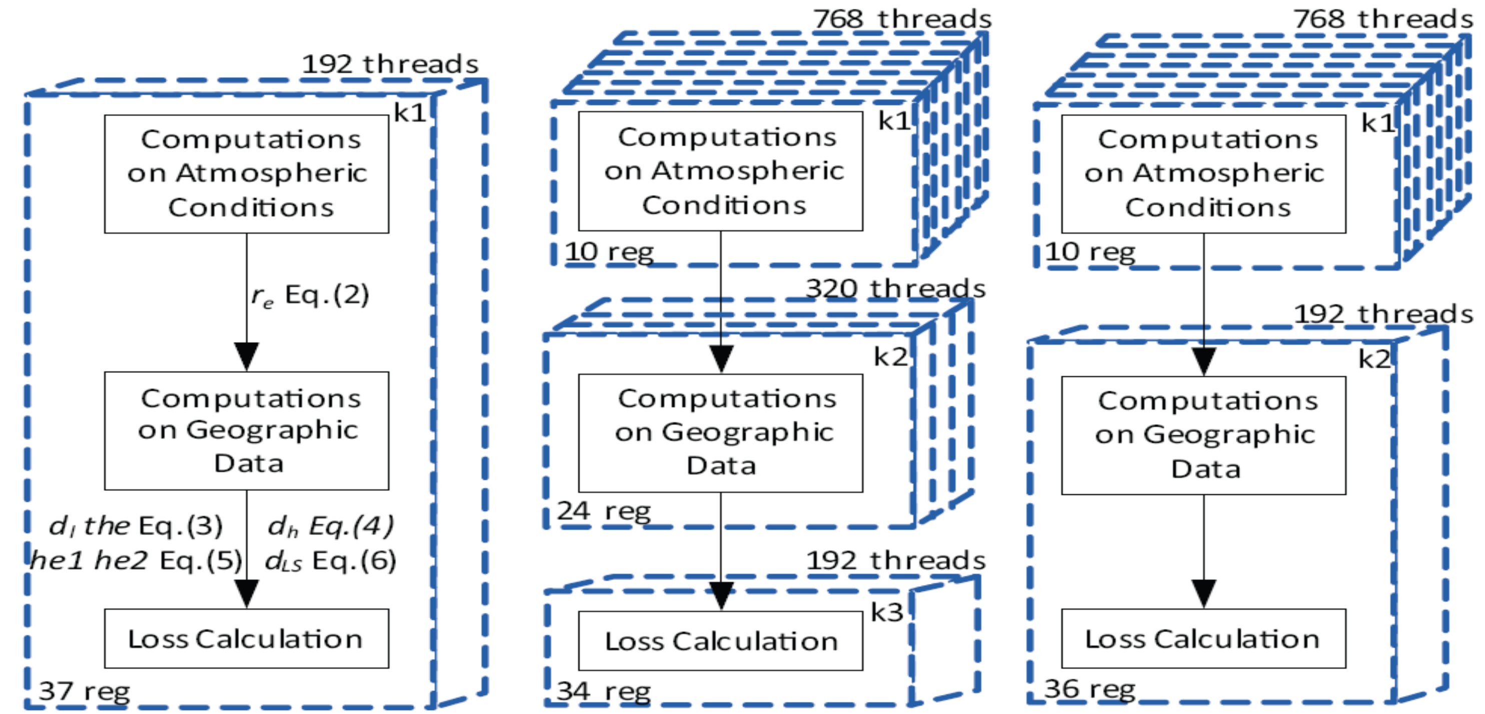
Execution time of the three optimization strategies on Tesla C870. (Strategy 1: global memory only, Strategy 2: shared memory only, Strategy 3: global and shared memory)

Job Size	time(ms)		
	strategy 1	strategy 2	strategy 3
128	1.554	**0.867**	1.033
256	1.563	**0.873**	1.035
512	1.568	1.714	**1.038**
1024	1.708	3.388	**1.116**
2k	2.309	6.732	**1.404**
4k	4.058	13.428	**3.003**
8k	7.518	36.812	**5.423**
16k	14.378	53.636	**11.036**
32k	28.073	108.137	**20.957**
64k	55.172	215.175	**42.085**
128k	109.603	430.041	**83.090**
256k	219.380	859.007	**166.434**



THE UNIVERSITY OF ARIZONA

GPU Strategies for ITM

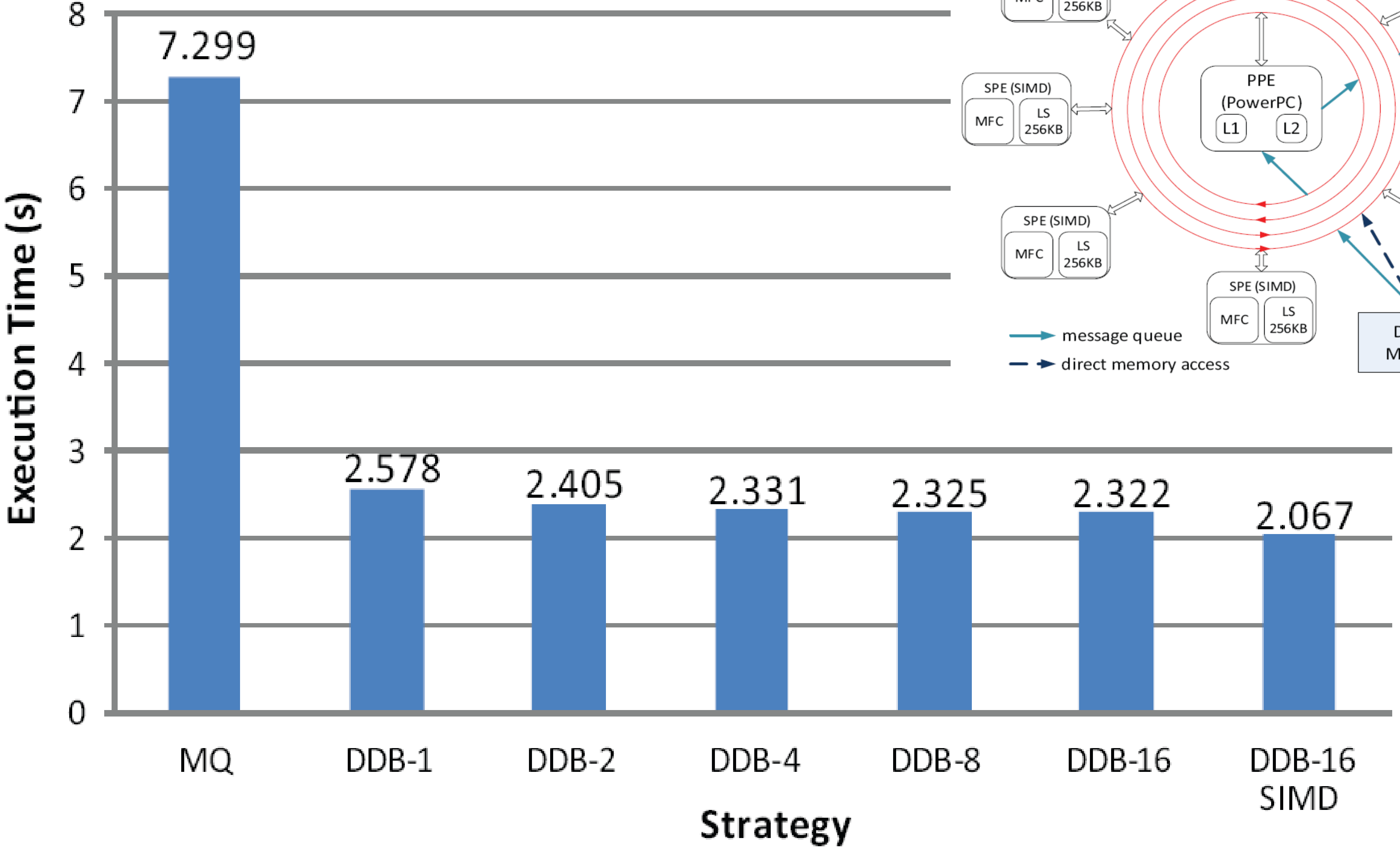


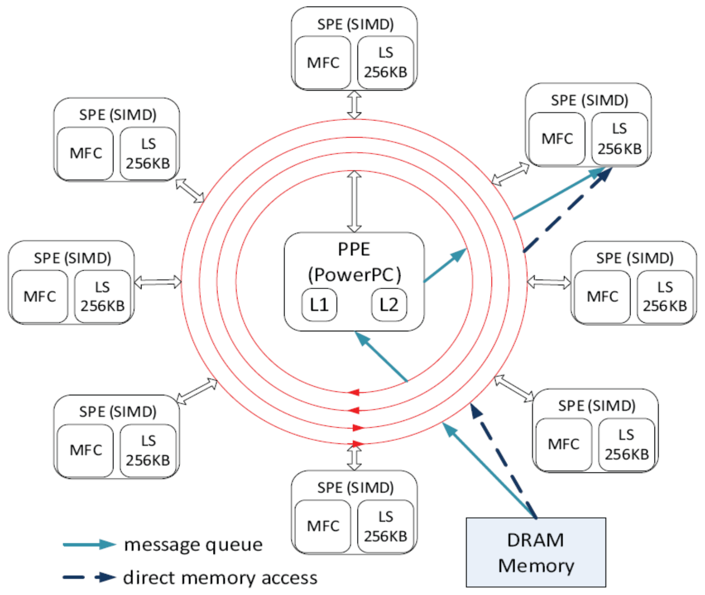
(a) strategy 1 with 1 kernel (b) strategy 2 with 3 kernels (c) strategy 3 with 2 kernels

Strategy	1	2	3
Number of kernels	1	3	2
Time (ms)	166.434	148.268	138.637

THE UNIVERSITY OF ARIZONA

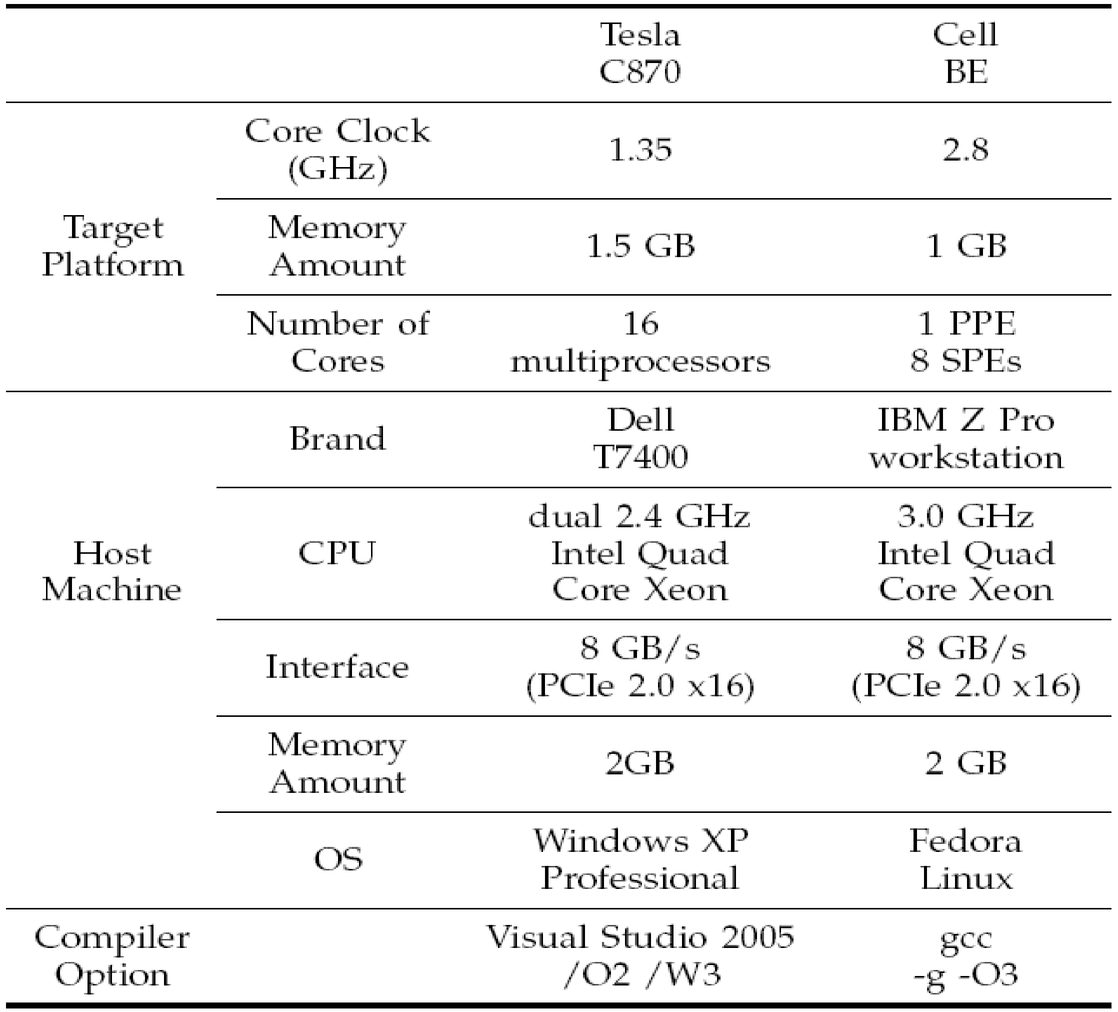
CELL BE Strategies for ITM





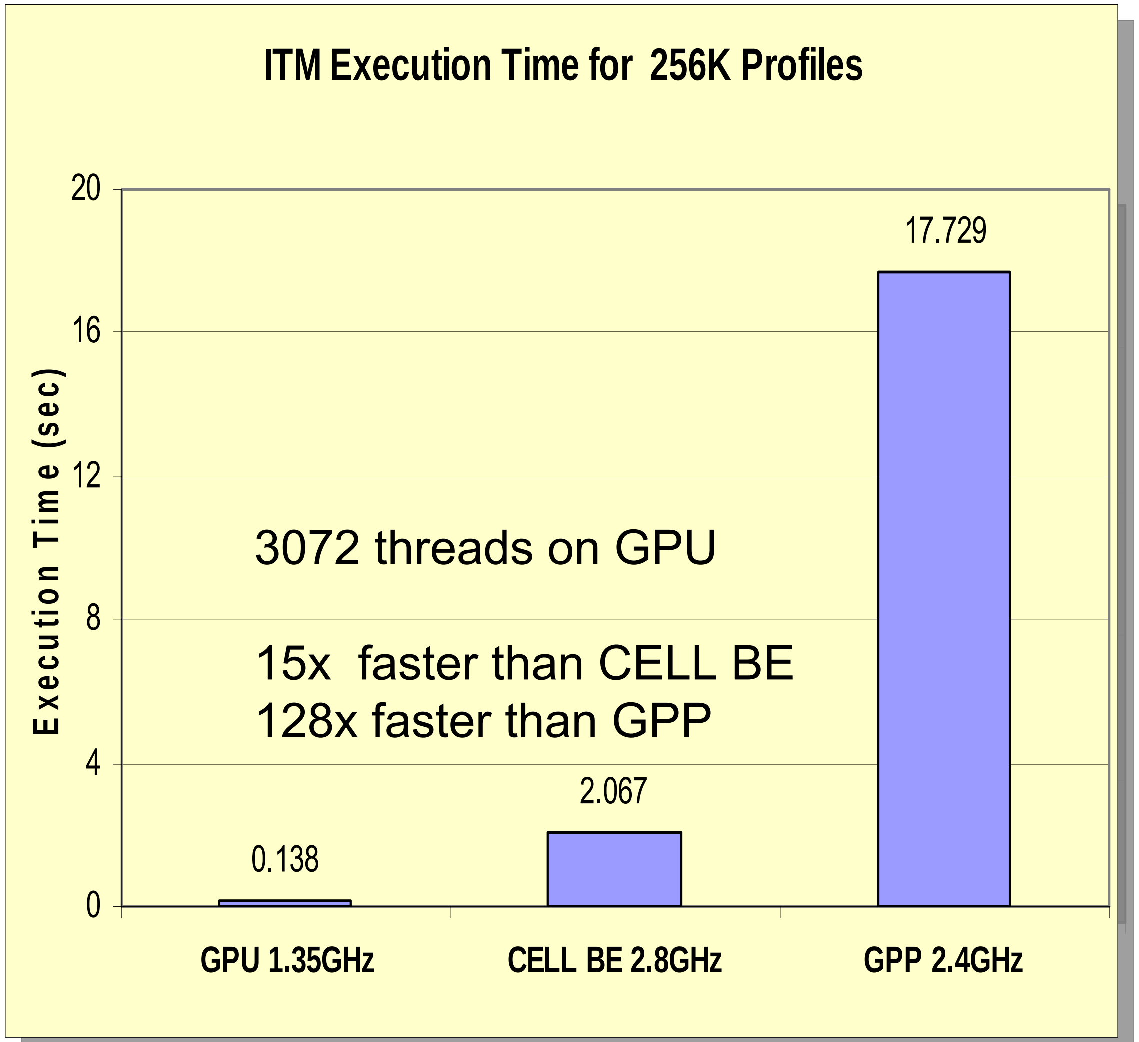
Configurations

Specifications of the Tesla C870 GPU and Cell BE, along with their host machines and the compiler options.



		Tesla C870	Cell BE
Target Platform	Core Clock (GHz)	1.35	2.8
	Memory Amount	1.5 GB	1 GB
	Number of Cores	16 multiprocessors	1 PPE 8 SPEs
Host Machine	Brand	Dell T7400	IBM Z Pro workstation
	CPU	dual 2.4 GHz Intel Quad Core Xeon	3.0 GHz Intel Quad Core Xeon
	Interface	8 GB/s (PCIe 2.0 x16)	8 GB/s (PCIe 2.0 x16)
	Memory Amount	2GB	2 GB
	OS	Windows XP Professional	Fedora Linux
Compiler Option		Visual Studio 2005 /O2 /W3	gcc -g -O3

Tesla C870 GPU vs. CELL BE



Job size: 256k	GPU	Cell BE
Speedup (execution time)	14.9	1
Speedup (cycle count)	30.7	1
Power (watt)	190	130
Energy	12.8	130
Power efficiency (Performance per watt)	10.2	1
Estimated cost in $	600	1200
Cost efficiency (Performance per $)	29.8	1
Development Time 1 Ph.D. student (Days)	32	134
Lines of Code	2365	2706

Tesla C1060: 30 cores, double amount of registers ($1,500) with 8X GFLOPS over Intel Xeon W5590 Quad Core ($1600)

Fermi: Next Generation GPU

- 32 cores, 1536 threads per core
- No complex mechanisms
  - speculation, out of order execution, superscalar, instruction level parallelism, branch predictors
- Column access to memory
  - faster search, read
- First GPU with error correcting code
  - registers, shared memory, caches, DRAM
- Languages supported
  - C, C++, FORTRAN, Java, Matlab, and Python
- IEEE 754’08 Double-precision floating point
  - fused multiply-add operations,
- Streaming and task switching (25 microseconds)
  - Launching multiple kernels (up to 16) simultaneously
    - visualization and computing

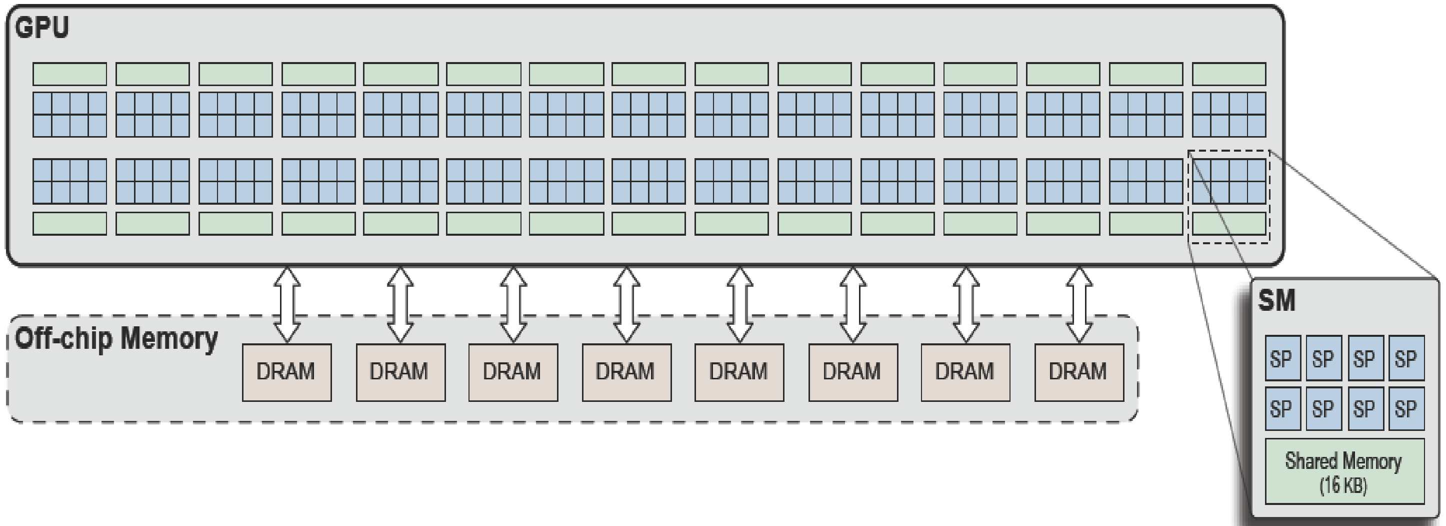
Reading 2

Designing Efficient Sorting Algorithms for Manycore GPUs

Radix and Merge Sort

From Hardware Perspective

- SP has its register space and shared memory
- Threads executed in groups of 32 (warps)
- Threads of a warp on separate SPs and share a single multithreaded instruction unit.
- SM transparently manages any divergence in the execution of threads in a warp.



From Software Perspective

- A kernel is a SPMD-style
- Programmer organizes threads into thread blocks;
- Kernel consists of a grid of one or more blocks
- A thread block is a group of concurrent threads that can cooperate amongst themselves through
  - barrier synchronization
  - a per-block private shared memory space
- Programmer specifies
  - number of blocks
  - number of threads per block
- Each thread block = virtual multiprocessor
  - each thread has a fixed register
  - each block has a fixed allocation of per-block shared memory.

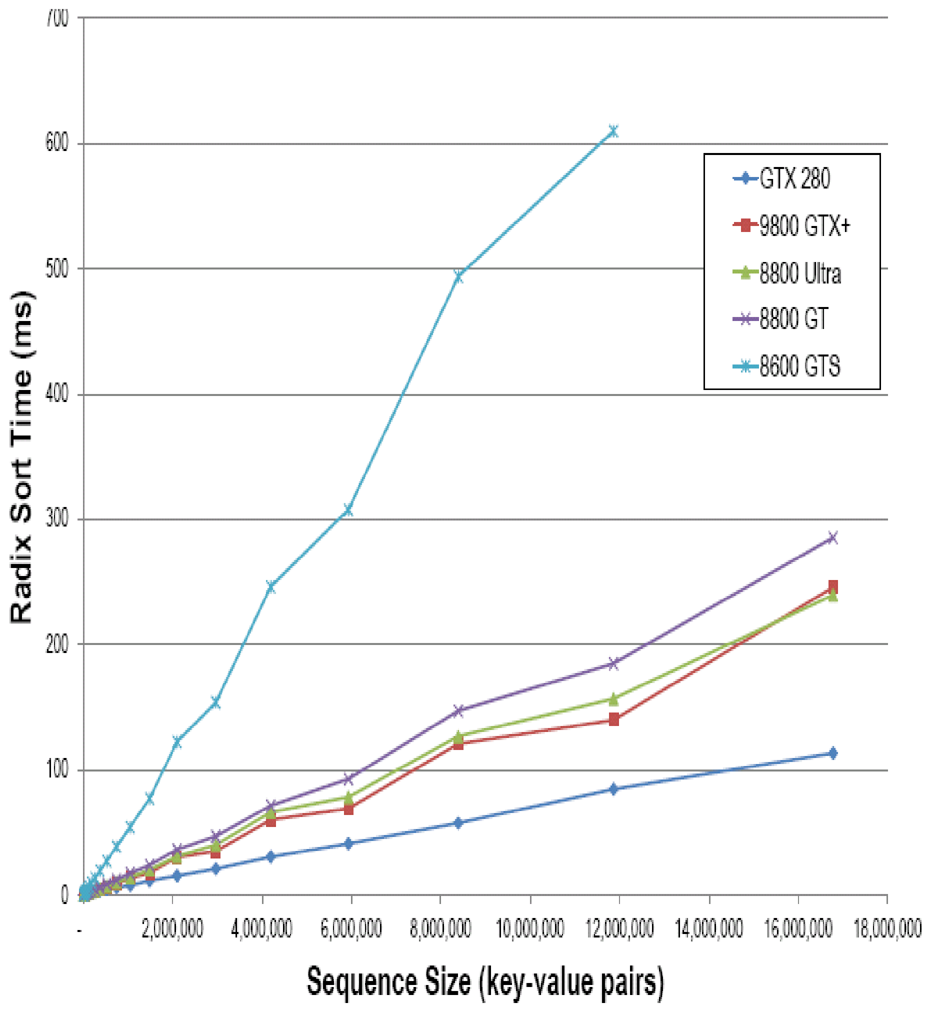
Efficiency Considerations

- Avoid execution divergence
  - threads within a warp follow different execution paths.
  - Divergence between warps is ok
- Allow loading a block of data into SM
  - process it there, and then write the final result back out to external memory.
- Coalesce memory accesses
  - Access executive words instead of gather-scatter
- Create enough parallel work
  - 5K to 10K threads

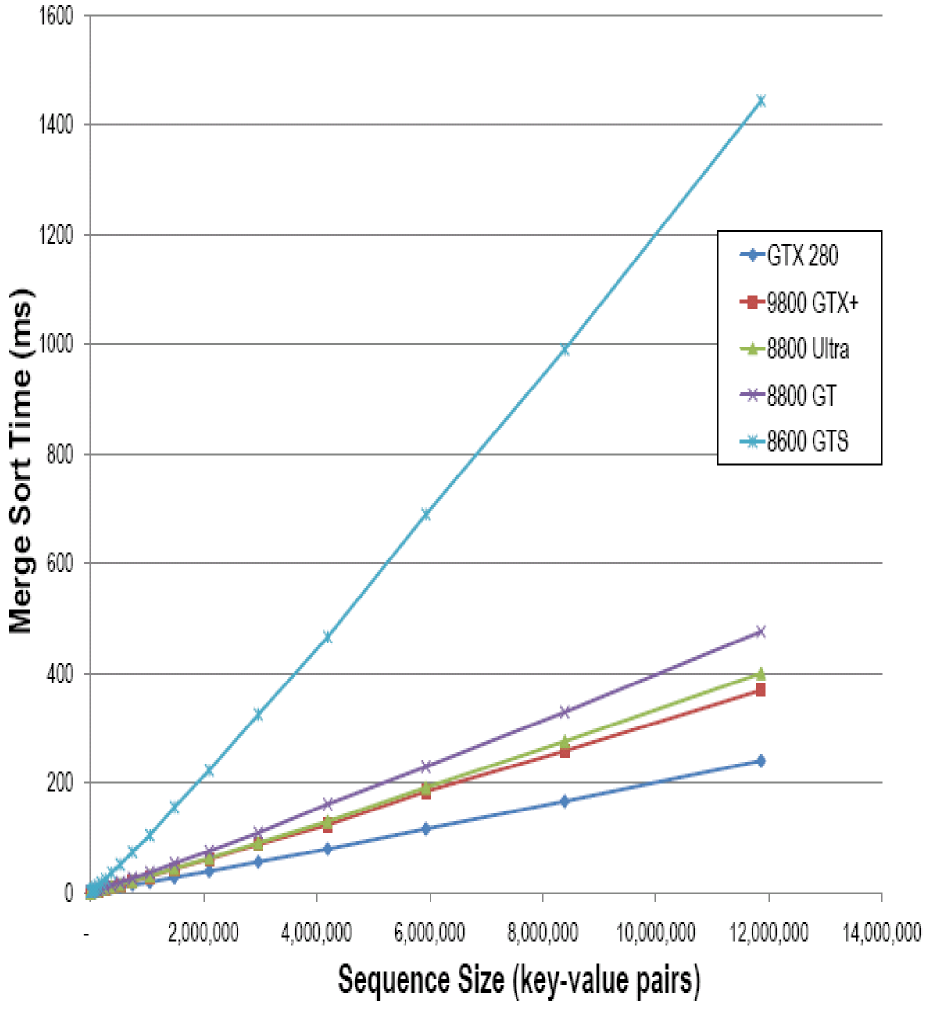
Most important factor

- number of thread blocks =“processor count”
- At thread block (or SM) level
  - Internal communication is cheap
- Focusing on decomposing the work between the “p” thread blocks

Performance



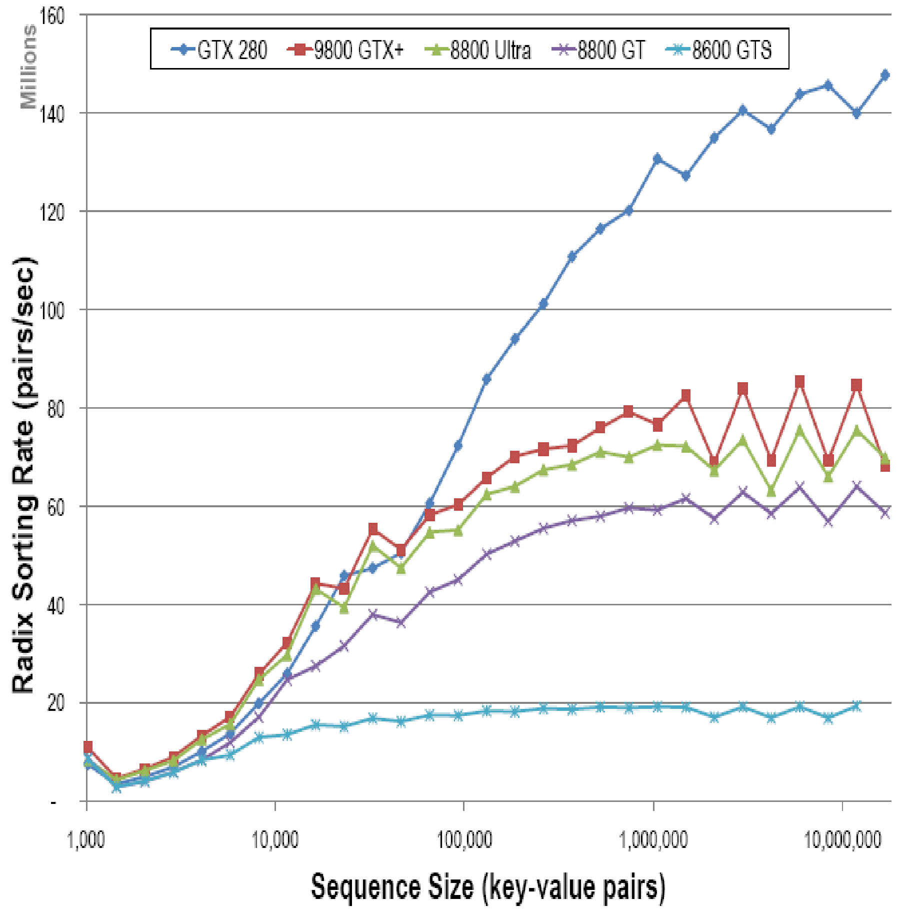
(a) Radix sort



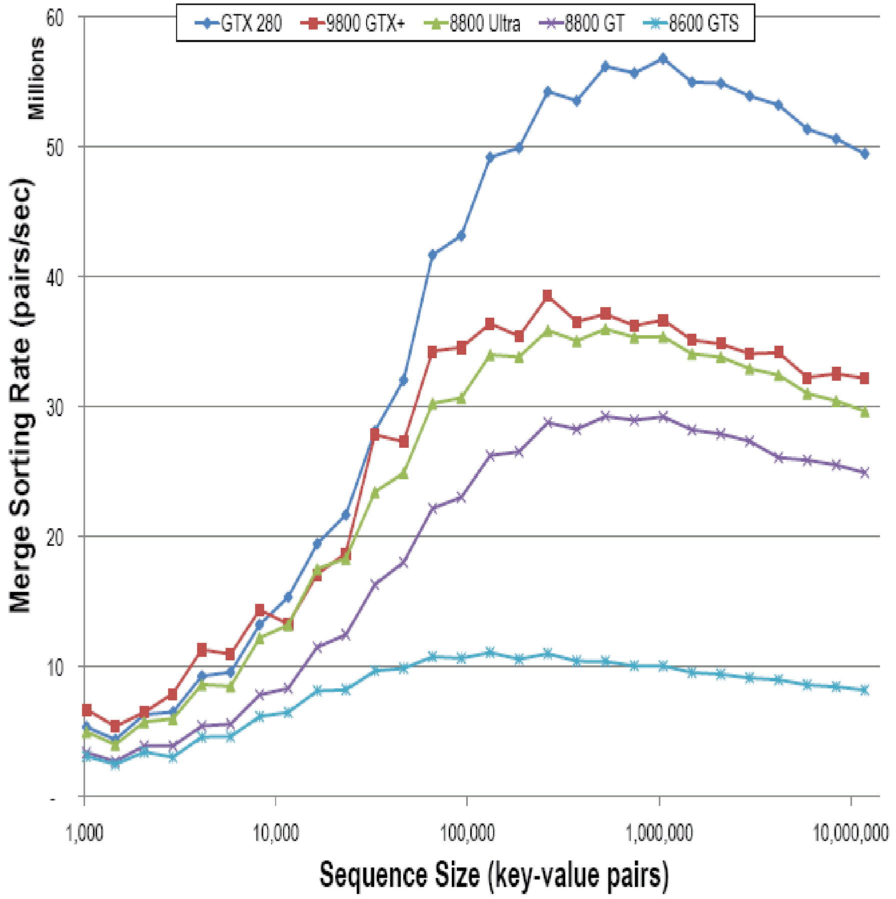
(b) Merge sort

THE UNIVERSITY OF ARIZONA

Performance

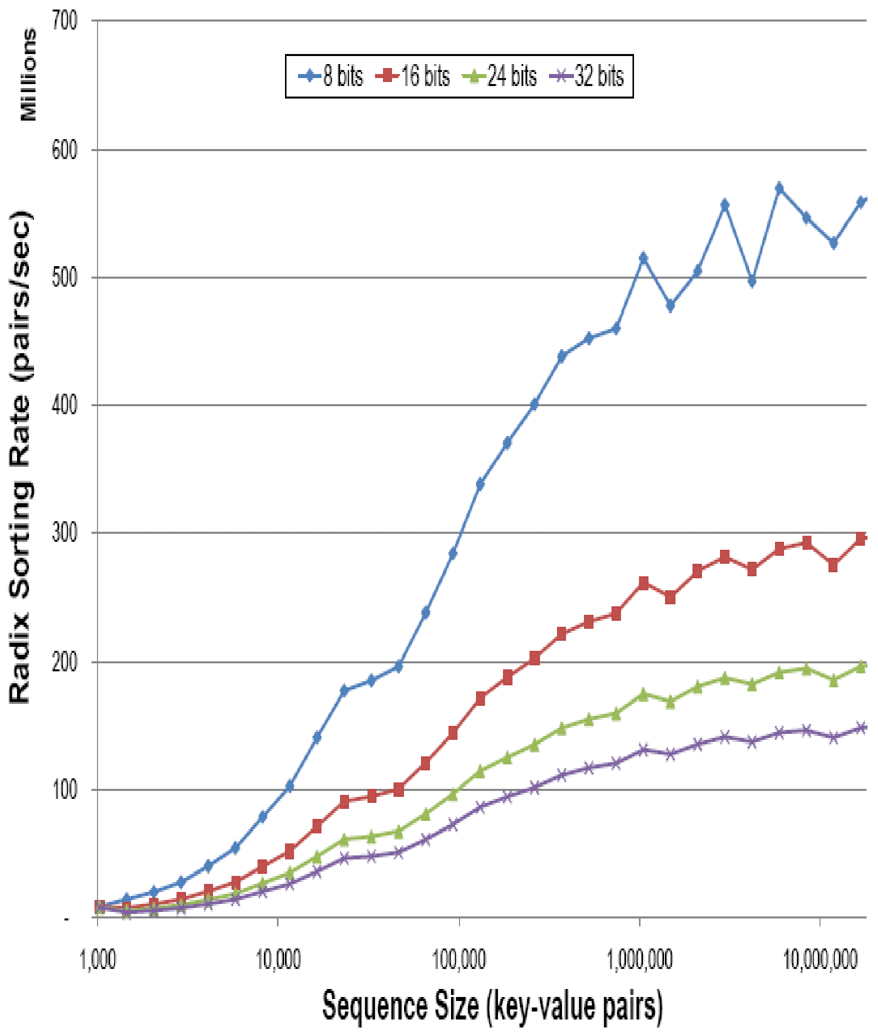


(a) Radix sort

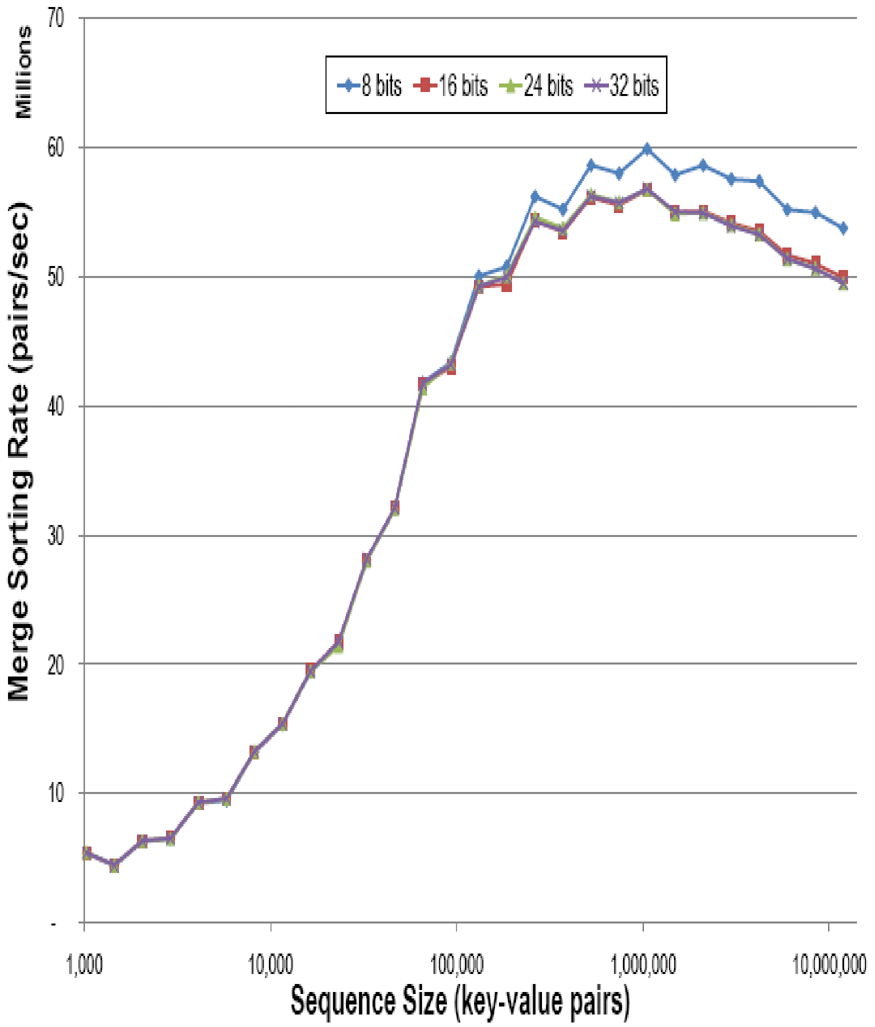


(b) Merge sort

Performance

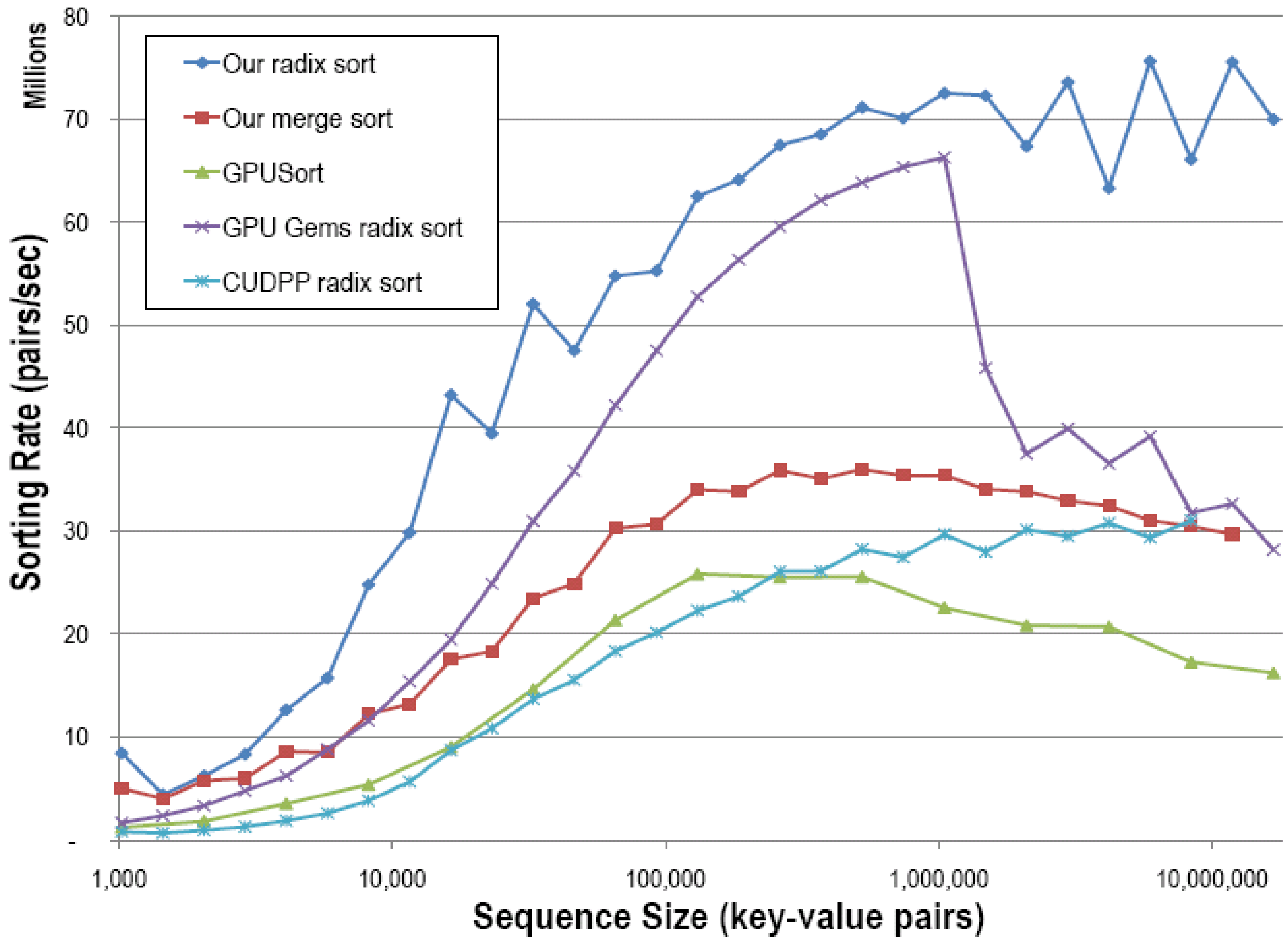


(a) Radix sort



(b) Merge sort

Performance



THE UNIVERSITY OF ARIZONA

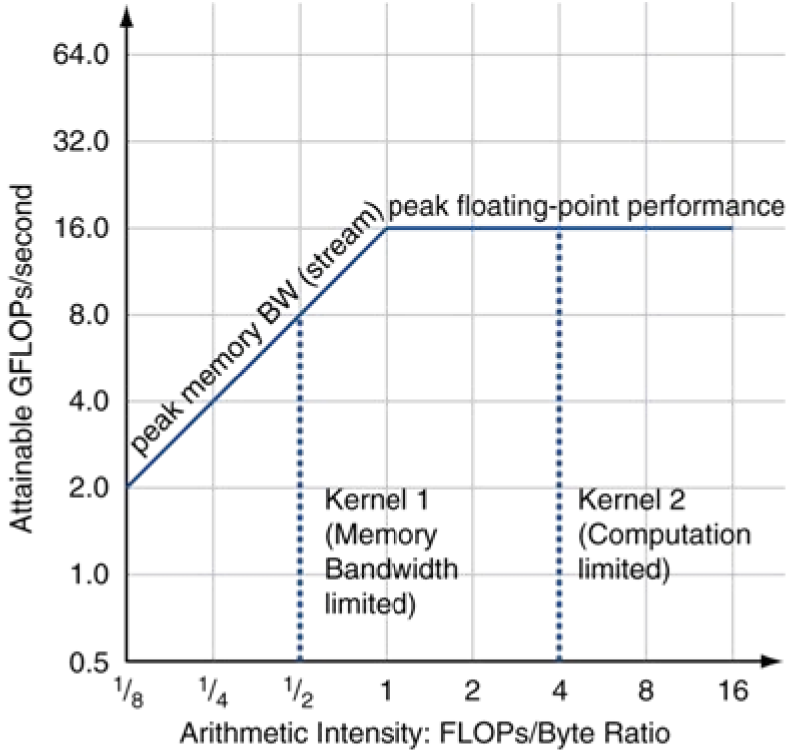
Fallacies

- Amdahl's Law doesn't apply to parallel computers
  - Since we can achieve linear speedup
  - But only on applications with weak scaling
- Peak performance tracks observed performance
  - Marketers like this approach!
  - But compare Xeon with others in example
  - Need to be aware of bottlenecks

Concluding Remarks

- Goal: higher performance by using multiple processors
- Difficulties
  - Developing parallel software
  - Devising appropriate architectures
- Many reasons for optimism
  - Changing software and application environment
  - Chip-level multiprocessors with lower latency, higher bandwidth interconnect
- An ongoing challenge for computer architects!

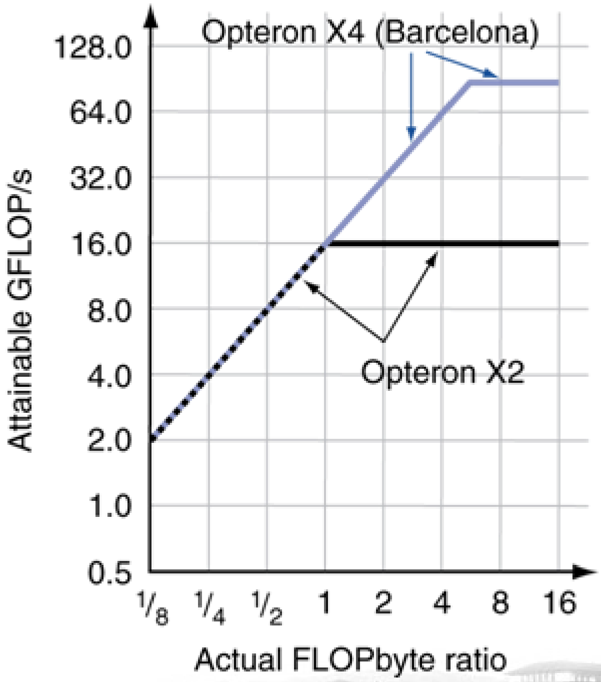
Roofline Diagram



Attainable GFLOPs/sec
= Max (Peak Memory BW × Arithmetic Intensity, Peak FP Performance)

Comparing Systems

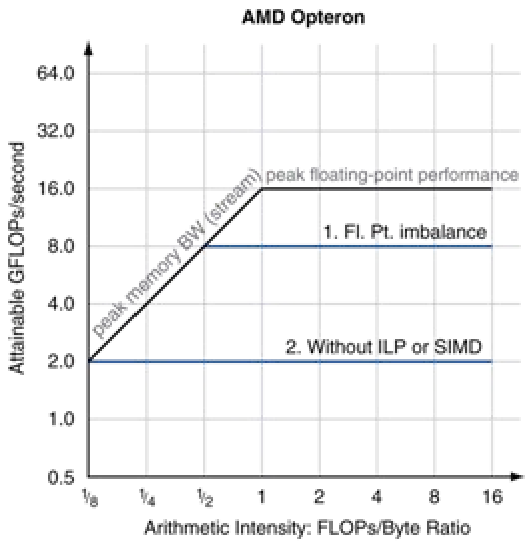
- Example: Opteron X2 vs. Opteron X4
  - 2-core vs. 4-core, 2× FP performance/core, 2.2GHz vs. 2.3GHz
  - Same memory system

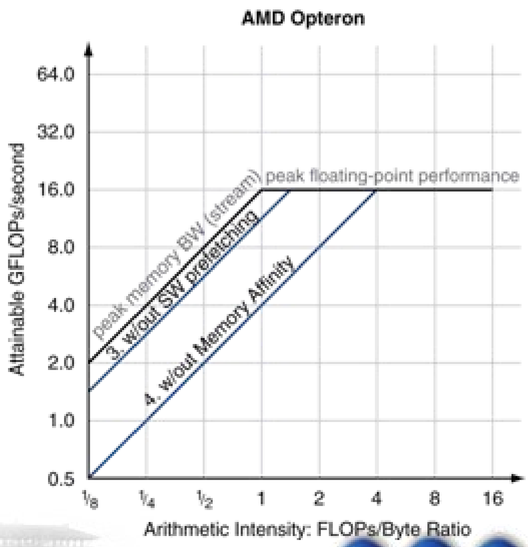


- To get higher performance on X4 than X2
  - Need high arithmetic intensity
  - Or working set must fit in X4's 2MB L-3 cache

Optimizing Performance

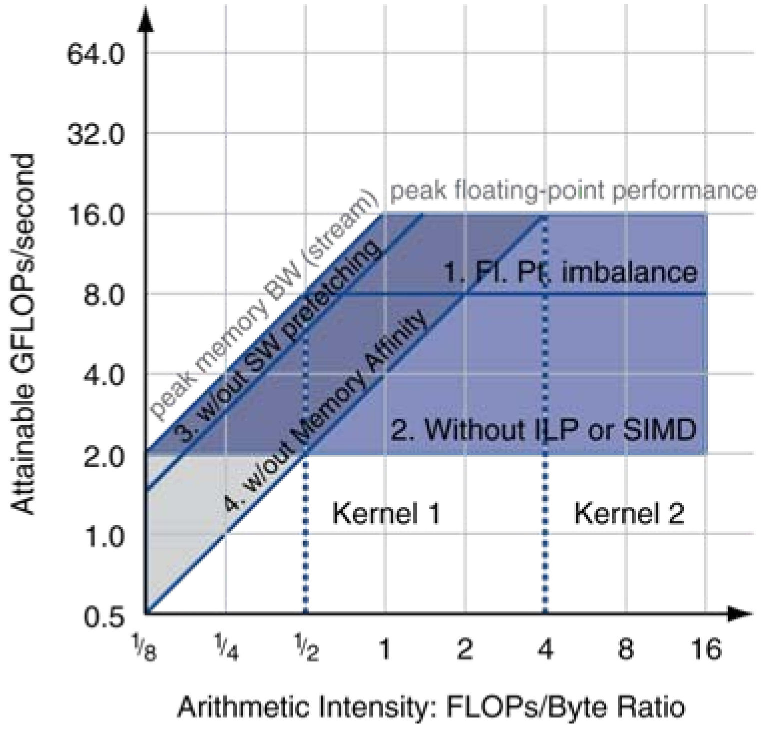
- Optimize FP performance
  - Balance adds & multiplies
  - Improve superscalar ILP and use of SIMD instructions
- Optimize memory usage
  - Software prefetch
    - Avoid load stalls
  - Memory affinity
    - Avoid non-local data accesses





Optimizing Performance

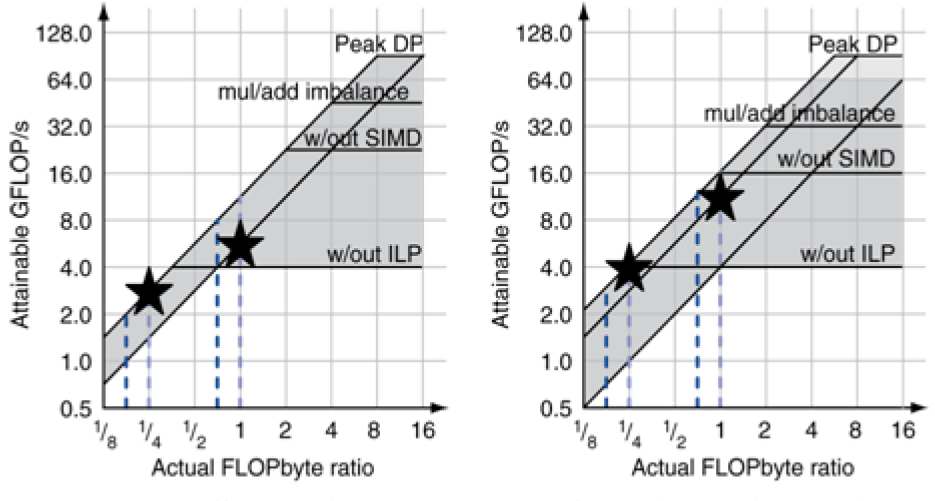
- Choice of optimization depends on arithmetic intensity of code



- Arithmetic intensity is not always fixed
  - May scale with problem size
  - Caching reduces memory accesses
    - Increases arithmetic intensity

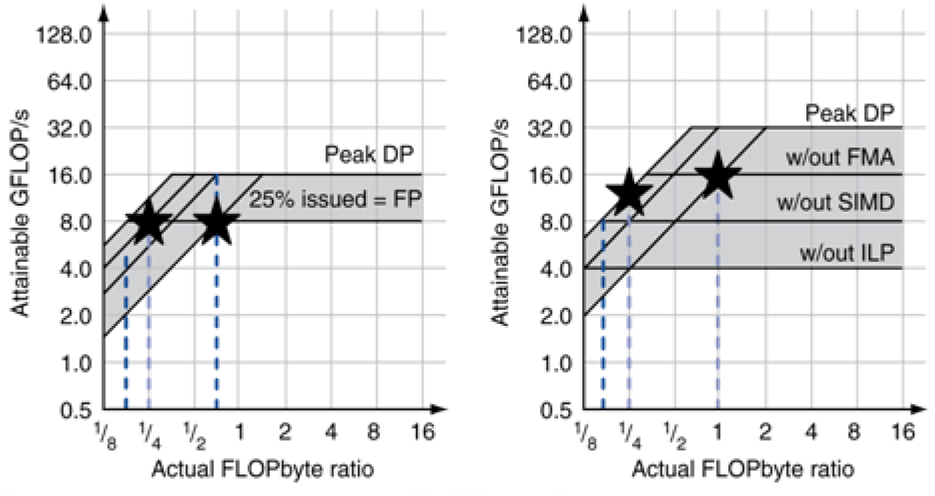
And Their Rooflines

- Kernels
  - SpMV (left)
  - LBHMD (right)
- Some optimizations change arithmetic intensity
- x86 systems have higher peak GFLOPs
  - But harder to achieve, given memory bandwidth



a. Intel Xeon e5345 (Clovertown)

b. AMD Opteron X4 2356 (Barcelona)

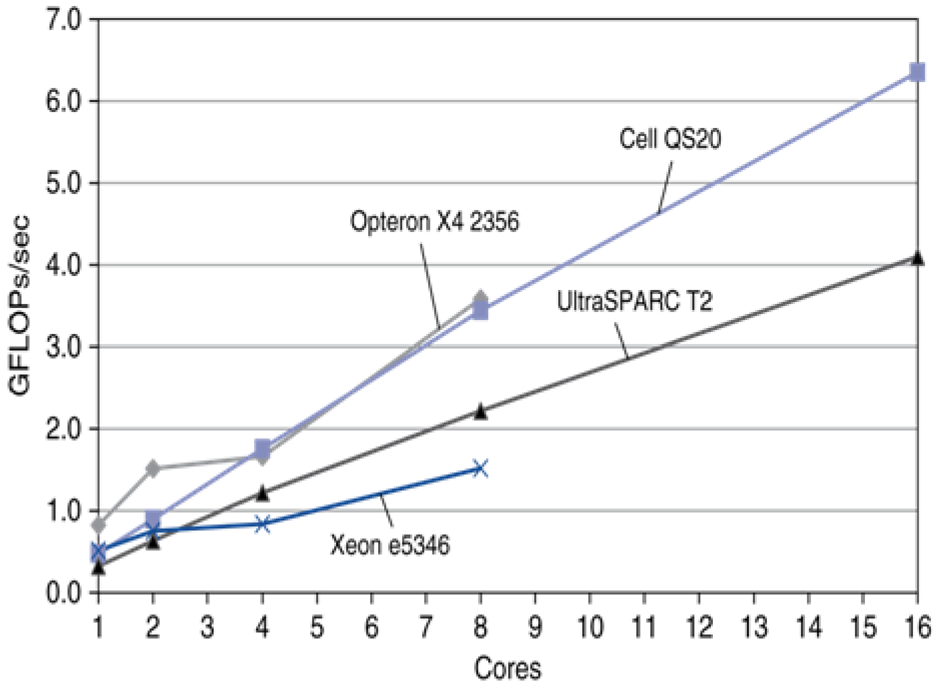


c. Sun UltraSPARC T2 5140 (Niagara 2)

d. IBM Cell QS20

Performance on SpMV

- Sparse matrix/vector multiply
  - Irregular memory accesses, memory bound
- Arithmetic intensity
  - 0.166 before memory optimization, 0.25 after

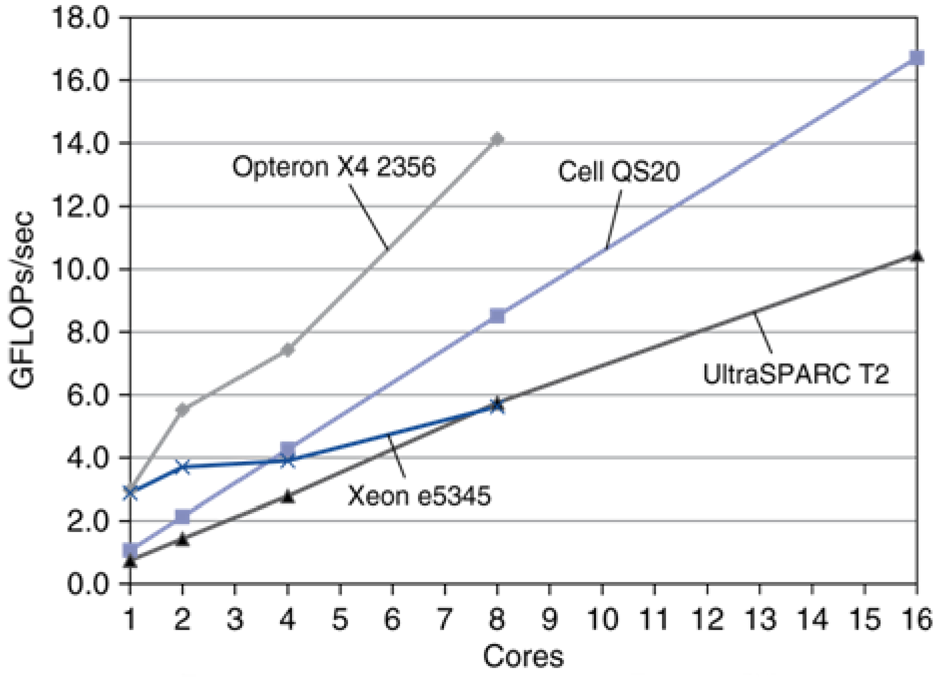


- Xeon vs. Opteron
  - Similar peak FLOPS
  - Xeon limited by shared FSBs and chipset
- UltraSPARC/Cell vs. x86
  - 20 – 30 vs. 75 peak GFLOPs
  - More cores and memory bandwidth

THE UNIVERSITY OF ARIZONA

Performance on LBMHD

- Fluid dynamics: structured grid over time steps
  - Each point: 75 FP read/write, 1300 FP ops
- Arithmetic intensity
  - 0.70 before optimization, 1.07 after



- Opteron vs. UltraSPARC
  - More powerful cores, not limited by memory bandwidth
- Xeon vs. others
  - Still suffers from memory bottlenecks

Achieving Performance

- Compare naïve vs. optimized code
  - If naïve code performs well, it's easier to write high performance code for the system

System	Kernel	Naïve GFLOPs/sec	Optimized GFLOPs/sec	Naïve as % of optimized
Intel Xeon	SpMV	1.0	1.5	64%
	LBMHD	4.6	5.6	82%
AMD Opteron X4	SpMV	1.4	3.6	38%
	LBMHD	7.1	14.1	50%
Sun UltraSPARC T2	SpMV	3.5	4.1	86%
	LBMHD	9.7	10.5	93%
IBM Cell QS20	SpMV	Naïve code not feasible	6.4	0%
	LBMHD		16.7	0%