Chapter 7C

Multicores, Multiprocessors, and Clusters

C & CUDA Codes for SAXPY

Computing y = ax + y with a serial loop:

```
void saxpy_serial(int n, float alpha, float *x, float *y)
{
    for(int i = 0; i<n; ++i)
        y[i] = alpha*x[i] + y[i];
}

// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

Computing y = ax + y in parallel using CUDA:

```
__global__
void saxpy_parallel(int n, float alpha, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if( i<n )  y[i] = alpha*x[i] + y[i];
}

// Invoke parallel SAXPY kernel (256 threads per block)
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

kernel launching

FIGURE A.3.4 Sequential code (top) in C versus parallel code (bottom) in CUDA for SAXPY

CUDA programming

- Parallel execution and thread management are automatic.
  - All thread creation, scheduling, and termination is handled for the programmer by the underlying system.
  - A Tesla architecture GPU performs all thread management directly in hardware.
  - The threads of a block execute concurrently and may synchronize at a synchronization barrier by calling _syncthreads() intrinsic.
  - Threads in a block, which should reside on the same multiprocessor, may communicate with each other by writing and reading per-block shared memory at a synchronization barrier.
  - # of threads ? # of thread blocks ? # of processors

CUDA programming

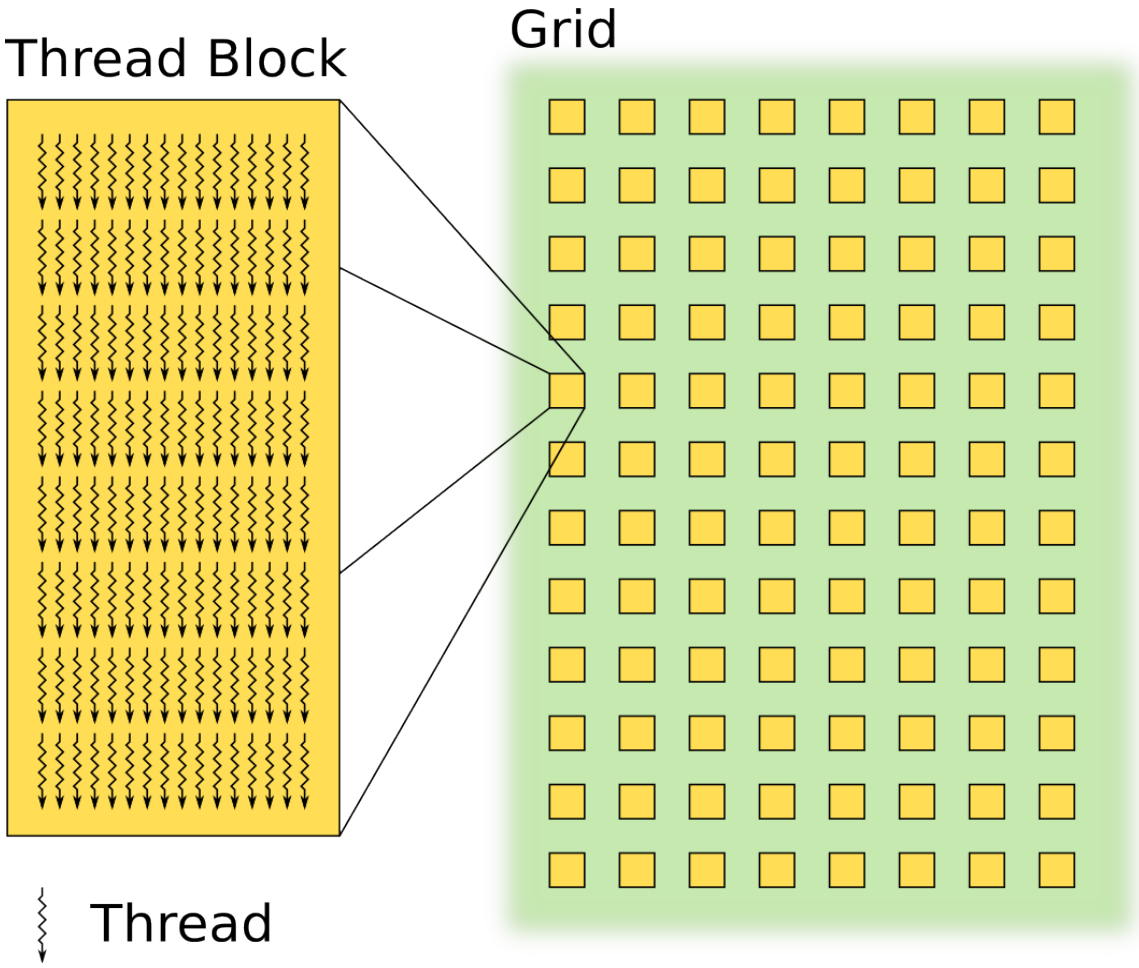
- A CUDA program is a unified C/C++ program for a heterogeneous CPU and GPU system.
- It execute on the CPU and dispatches parallel work to the GPU.
  - This work consists of a data transfer from main memory and a thread dispatch.
  - A thread is a piece of the program for the GPU.
- Programmers specify the number of threads in a thread block and the number of thread blocks they wish to start executing on the GPU.
  - All the threads in the thread block are scheduled to run on the same multiprocessor so they all share the same local memory.

CUDA programming

- Thus, they can communicate via loads and stores instead of messages.
- The CUDA compiler allocates registers to each thread, under the constraints that the registers per thread time thread per thread block does not exceed the 8192 registers per multiprocessors.
- A thread block can be up to 512 threads
  - Each group of 32 threads in a thread block is packed into warps.

CUDA Programming

Massive number (>10000) of light-weight threads.



CUDA Program

- CUDA program expresses data level parallelism (DLP) in terms of thread level parallelism (TLP).
- Hardware converts TLP into DLP at run time.

Scalar program

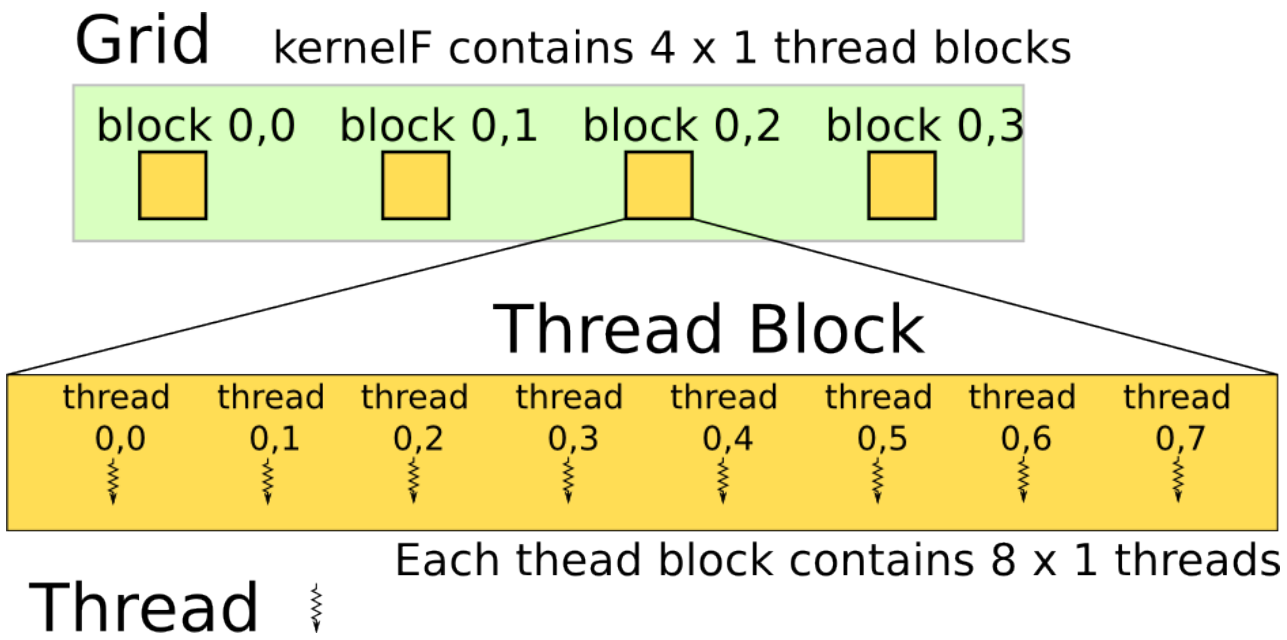
```
float A[4][8];
do-all(i=0;i<4;i++){
    do-all(j=0;j<8;j++){
        A[i][j]++;
    }
}
```

CUDA program

```
float A[4][8];

kernelF<<<(4,1),(8,1)>>>(A);

__global__    kernelF(A){
    i = blockIdx.x;
    j = threadIdx.x;
    A[i][j]++;
}
```

Two Levels of Thread Hierarchy

```
kernelF<<<(4,1),(8,1)>>>(A);

__global__    kernelF(A){
    i = blockIdx.x;
    j = threadIdx.x;
    A[i][j]++;
}
```

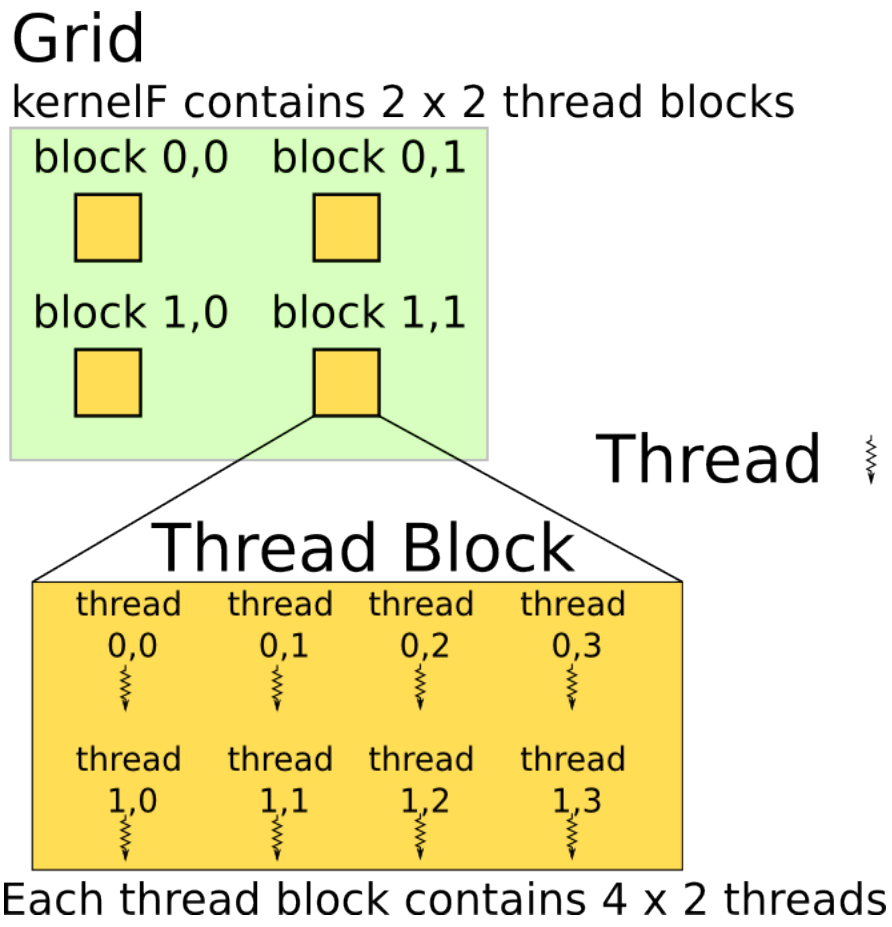


y, x ? wrong
x,y ! correct

Multi-dimension Thread and Block ID

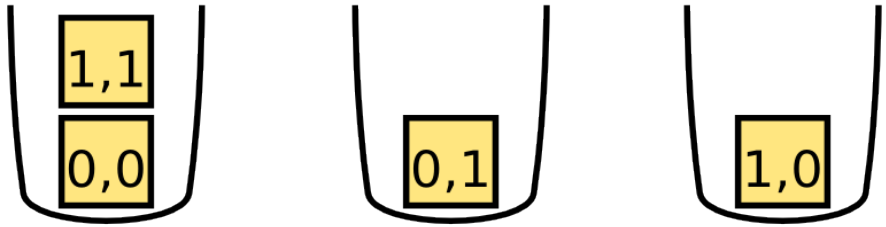
Both grid and thread block can have two dimensional index.

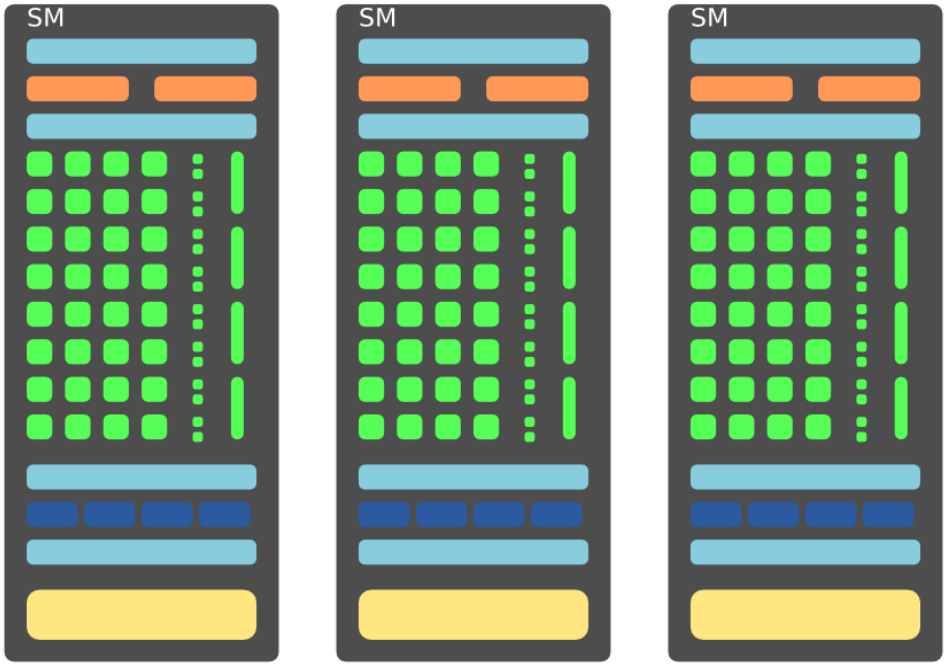
```
kernelF<<<(2,2),(4,2)>>>(A);

__global__    kernelF(A){
    i = blockDim.x * blockIdx.y
        + blockIdx.x;
    j = threadDim.x * threadIdx.y
        + threadIdx.x;
    A[i][j]++;
}
```



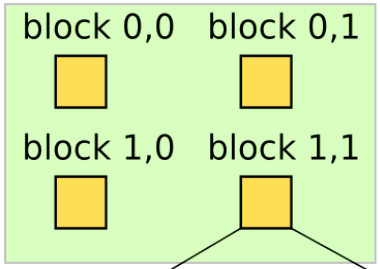
Scheduling Thread Blocks on SM

Example:
Scheduling 4 thread blocks on 3 SMs.



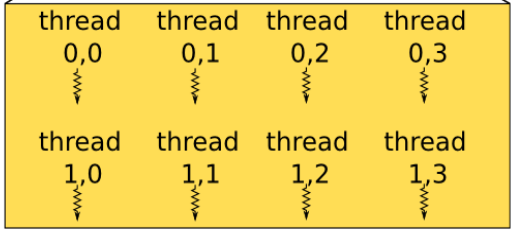


Grid
kernelF contains 2 x 2 thread blocks



Thread

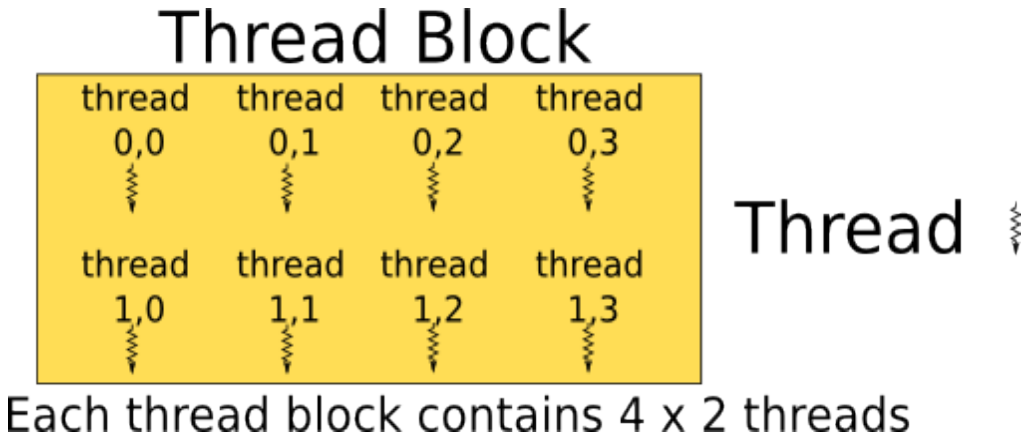
Thread Block



Each thread block contains 4 x 2 threads

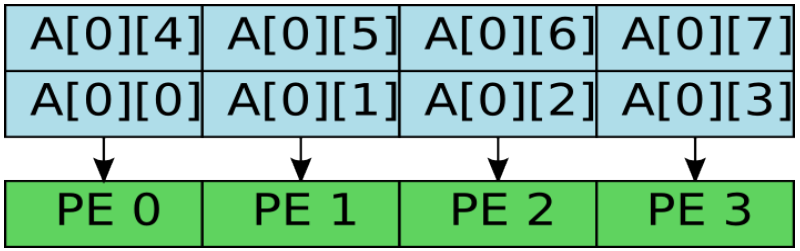
Executing Thread Block on SM

```
kernelF<<<(2,2),(4,2)>>>(A);

__device__    kernelF(A){
    i = blockDim.x * blockIdx.y
        + blockIdx.x;
    j = threadDim.x * threadIdx.y
        + threadIdx.x;
    A[i][j]++;
}
```

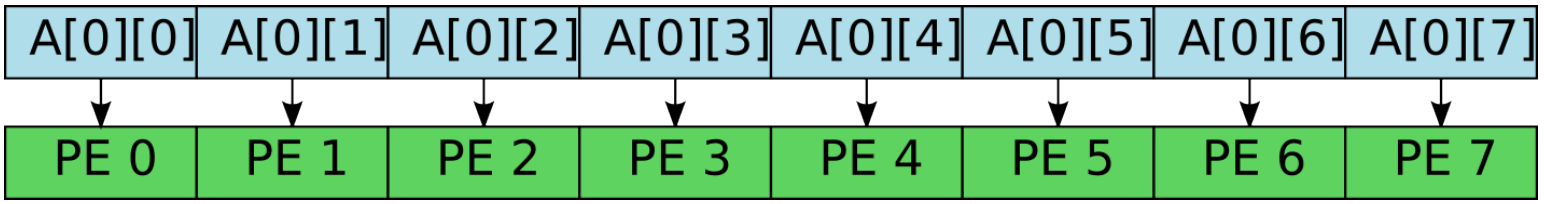


Notes: the number of Processing Elements (PEs) is transparent to programmer.

Executed on machine with width of 4:

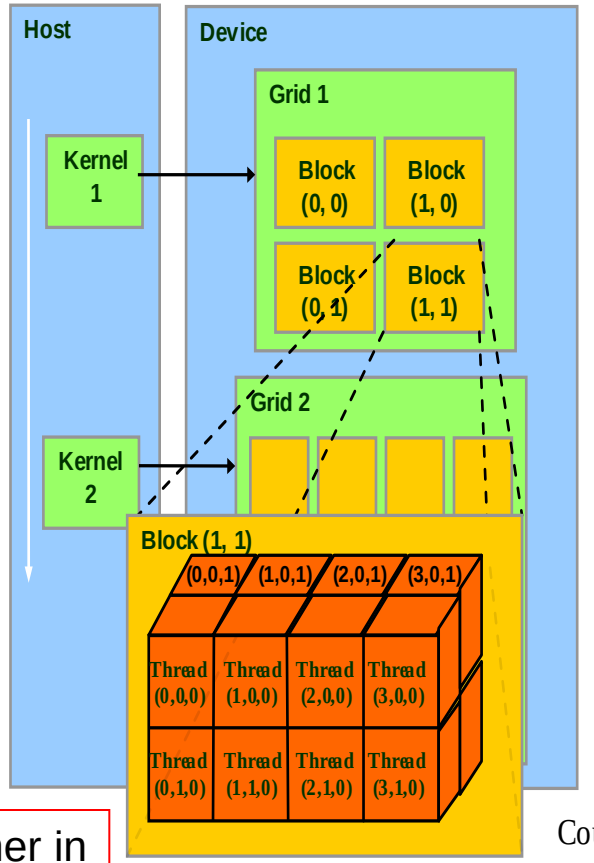


Executed on machine with width of 8:



Locate Thread by Built-in Variables

- Built-in Variables
  - gridDim
    - x, y
  - blockIdx
    - x, y
  - blockDim
    - x, y, z
  - threadIdx
    - x, y, z

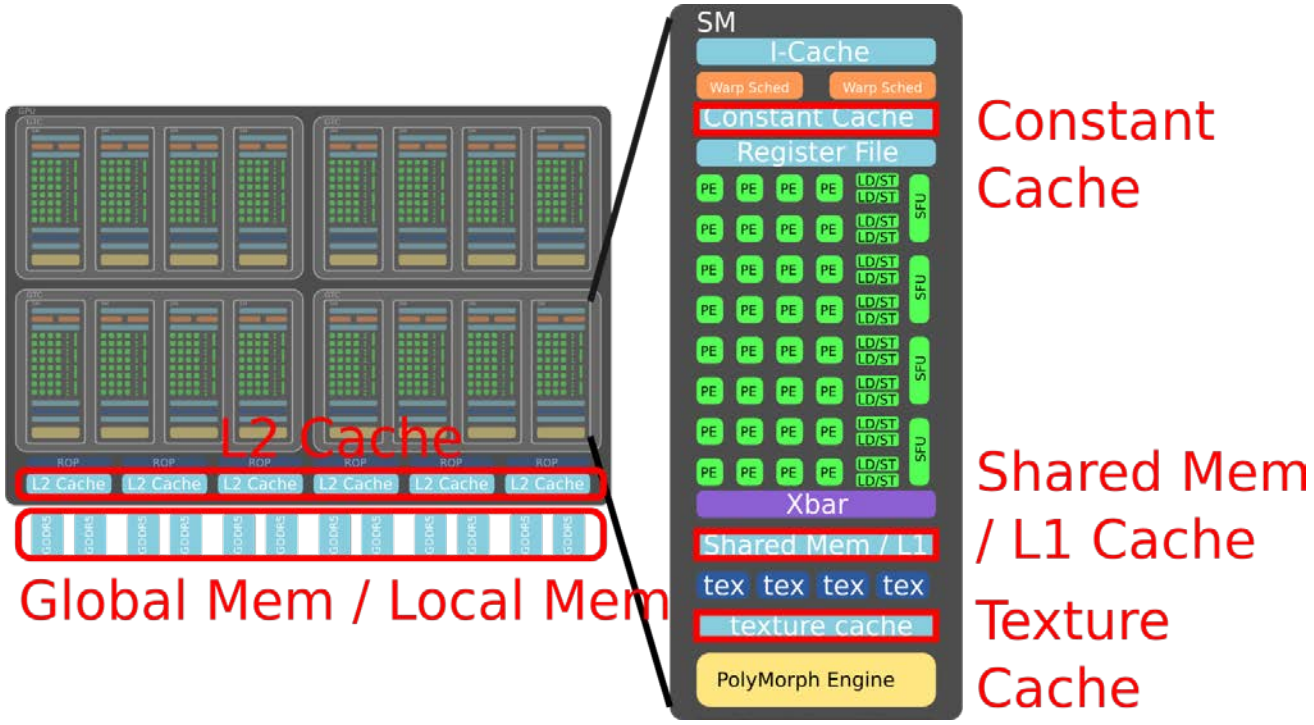


Courtesy: NDVIA

The index of a thread and its thread ID relate to each other in a straightforward way: For a one-dimensional block, they are the same; for a two-dimensional block of size *(Dx, Dy)*, the thread ID of a thread of index *(x, y)* is *(x + y Dx)*; for a three-dimensional block of size *(Dx, Dy, Dz)*, the thread ID of a thread of index *(x, y, z)* is *(x + y Dx + z Dx Dy)*.

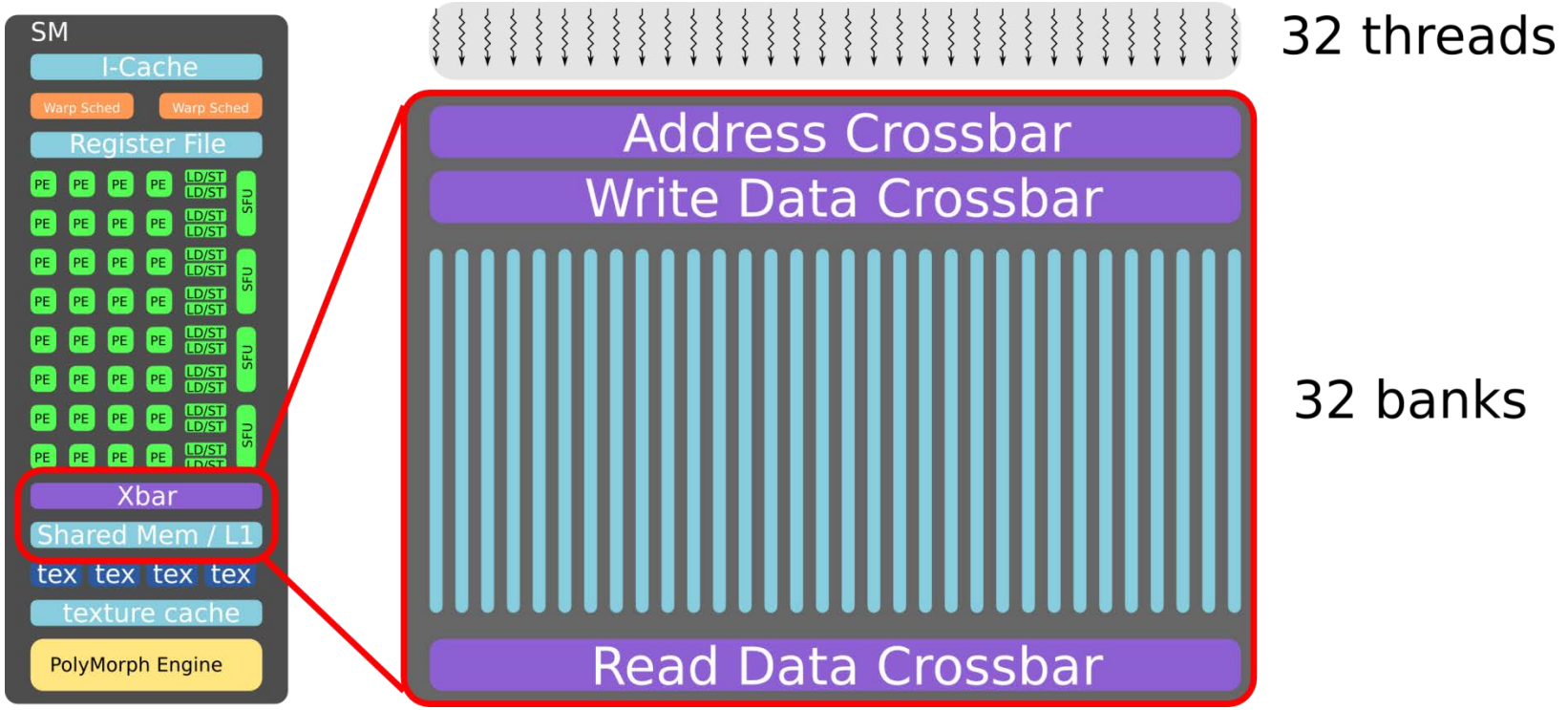
Multiple Levels of Memory Hierarchy

Name	Cache?	cycle	read-only?
Global	L1/L2	200~400 (cache miss)	R/W
Shared	No	1~3	R/W
Constant	Yes	1~3	Read-only
Texture	Yes	~100	Read-only
Local	L1/L2	200~400 (cache miss)	R/W



Explicit Management of Shared Mem

Shared memory is frequently used to exploit locality.



Shared Memory and Synchronization

Example: average filter with 3x3 window

```
kernelF<<<(1,1),(16,16)>>>(A);

__global__    kernelF(A){
    __shared__ smem[16][16]; //allocate smem
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j];
    __sync();
    A[i][j] = ( smem[i-1][j-1]
                + smem[i-1][j]
                ...
                + smem[i+1][i+1] ) / 9;
}
```

3x3 window on image

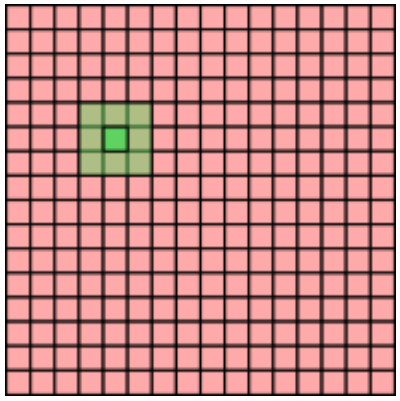
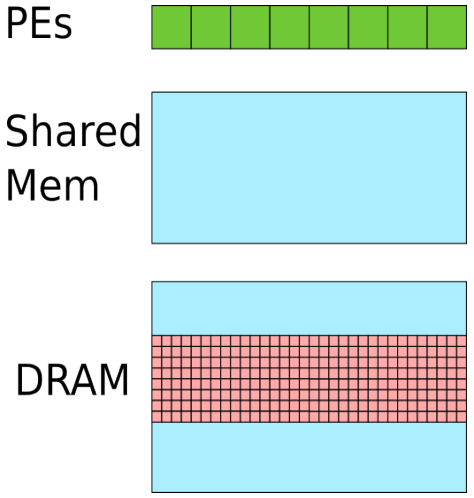


Image data in DRAM

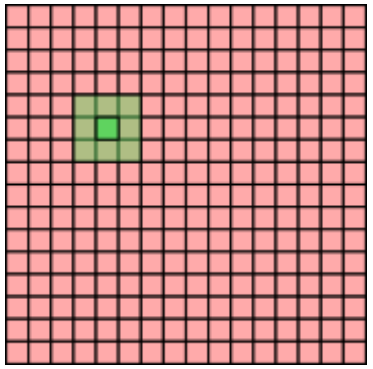


Shared Memory and Synchronization

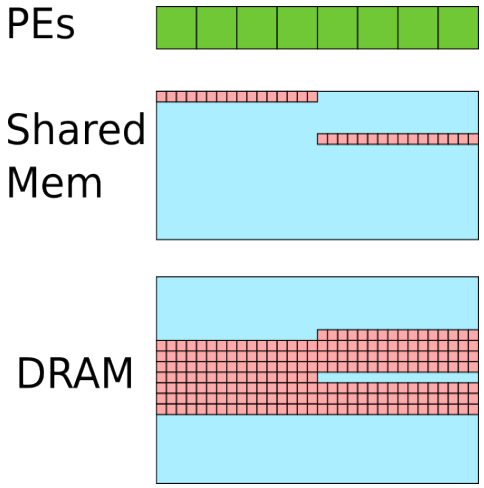
Example: average filter over 3x3 window

```
kernelF<<<(1,1),(16,16)>>>(A);

__global__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    __sync(); // thread wait at barrier
    A[i][j] = ( smem[i-1][j-1]
                + smem[i-1][j]
                ...
                + smem[i+1][i+1] ) / 9;
}
```

3x3 window on image



Stage data in shared mem

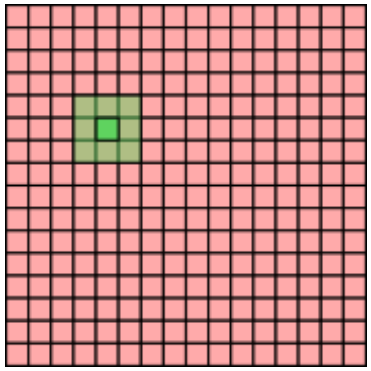


Shared Memory and Synchronization

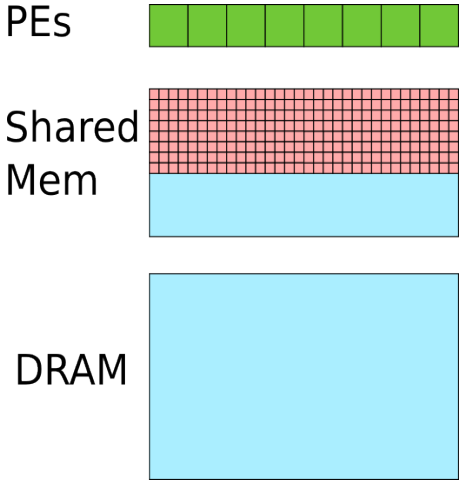
Example: average filter over 3x3 window

```
kernelF<<<(1,1),(16,16)>>>(A);

__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j];
    __sync(); // every thread is ready
    A[i][j] = ( smem[i-1][j-1]
                + smem[i-1][j]
                ...
                + smem[i+1][i+1] ) / 9;
}
```

3x3 window on image



all threads finish the load

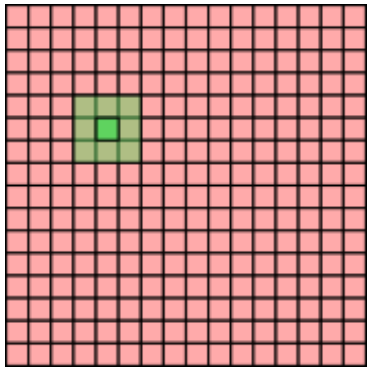


Shared Memory and Synchronization

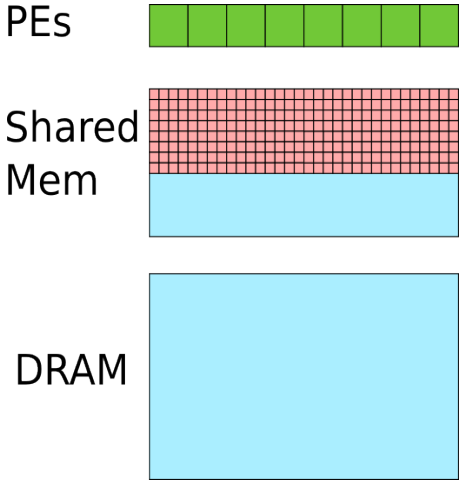
Example: average filter over 3x3 window

```
kernelF<<<(1,1),(16,16)>>>(A);

__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j];
    __sync();
    A[i][j] = ( smem[i-1][j-1]
                + smem[i-1][j]
                ...
                + smem[i+1][i+1] ) / 9;
}
```

3x3 window on image



all threads finish the load



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**

Interconnection Networks

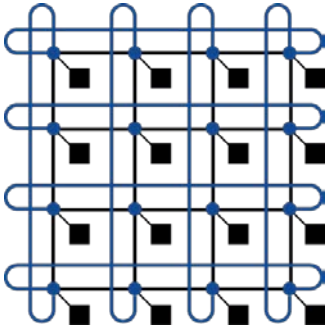
- Network costs include
  - number of switches
  - number of links
  - width per link
  - length of the links
- Networks are normally drawn as graphs, with each arc representing a link
  - Processor-memory node as a black square
  - Switch as a blue circle
- All links are bidirectional in this book

Networks Topology

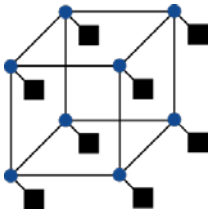
- Network topologies
  - Arrangements of processors, switches, and links

Bus

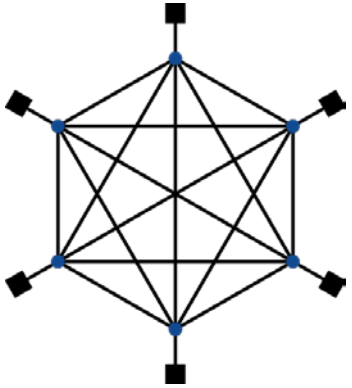
Ring



2D Torus (Grid)



N-cube (N = 3)



Fully connected

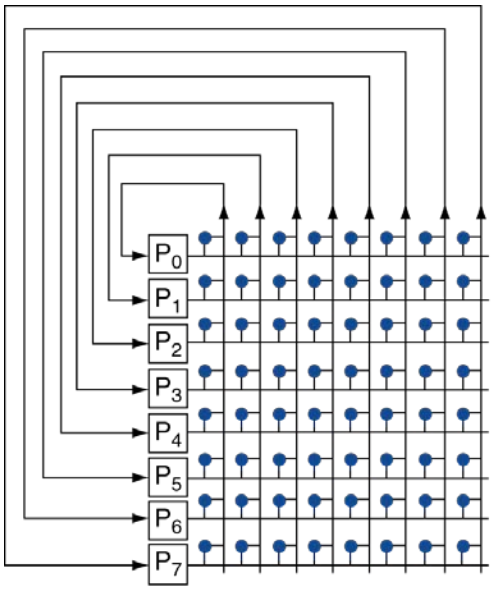
Network Characteristics

- Network bandwidth: informally the peak transfer rate of a network.
  - Can refer to the speed of a single link or the collective transfer rate of all links in the network.
- Bisection bandwidth: the bandwidth between two equal parts of a multiprocessors
  - This measure is for a worst case split of the multiprocessor.
- A fully connected network
  - total network BW: $O(P(P-1)/2)$
  - bisection BW: $O(P/2)^2$

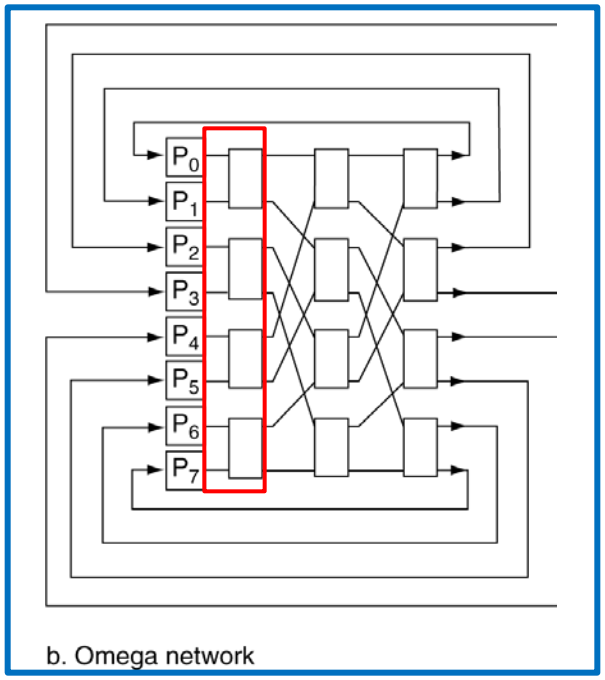
Network Characteristics

- Single-stage vs. multi-stage network:
  - One switch or more between the input and the output
- Omega network: identical perfect shuffle between each stage
  - used less hardware than the crossbar networks
  - $2n\log_2 n$ versus n^2 switches
  - Contention can occurs between messages
- No contention in the crossbar or fully connected networks

Omega Networks



a. Crossbar



b. Omega network

? The connection of 1st stage is wrong

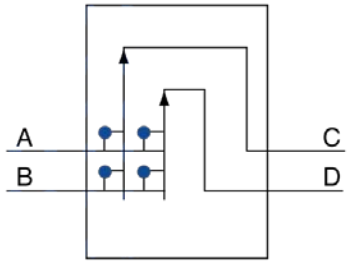
64 switches

Support arbitrary connection

48 switches

Some connection are not supported

Is the crossbar network a multi-stage network?



c. Omega network switch box

pass through or cross over

Single stage

Multi-stage

Crossbar

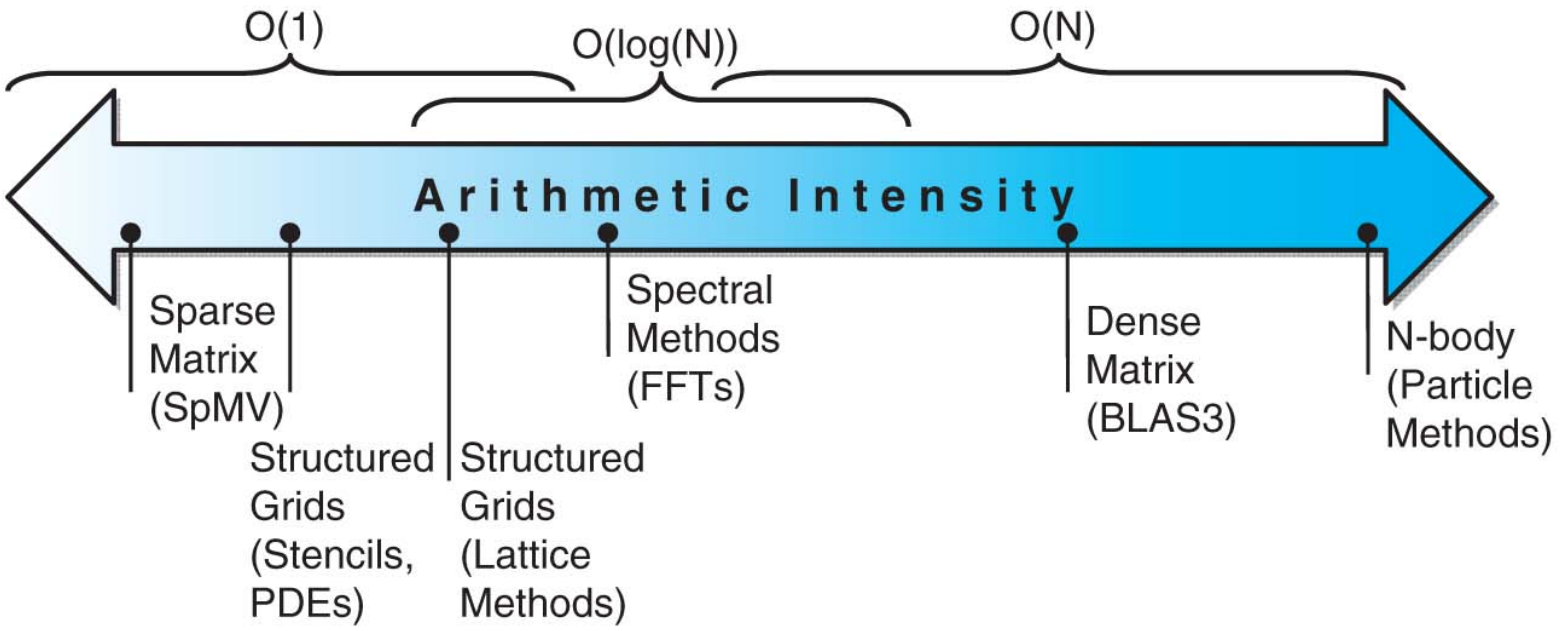
Modeling Performance

- Assume performance metric of interest is achievable GFLOPs/sec
  - Measured using computational kernels from Berkeley Design Patterns
- For a given computer, determine
  - Peak GFLOPS : from data sheet
  - Peak memory bytes/sec: by using Stream benchmark
- Demands on the memory system [bytes/sec]
  - Floating-point operation per sec divided by the arithmetic intensity, the average number of floating-point operation per byte.

Stream Benchmark

- The STREAM benchmark is a simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/s), which intended to measure the bandwidth from main memory.
- The STREAM benchmark is specifically designed to work with data sets much larger than the available cache on any given system and no temporal locality.
- Computer CPUs are getting faster much more quickly than computer memory systems.
- As this progresses, more and more programs will be limited in performance by the memory bandwidth of the system, rather than by the computational performance of the CPU.

Arithmetic Intensity



Arithmetic intensity: specified as the number of float-point operations to run the program divided by the number of bytes accessed in main memory [Williams, Patterson, 2008].

FLOPs per byte of memory accessed

Roofline Model for the computer

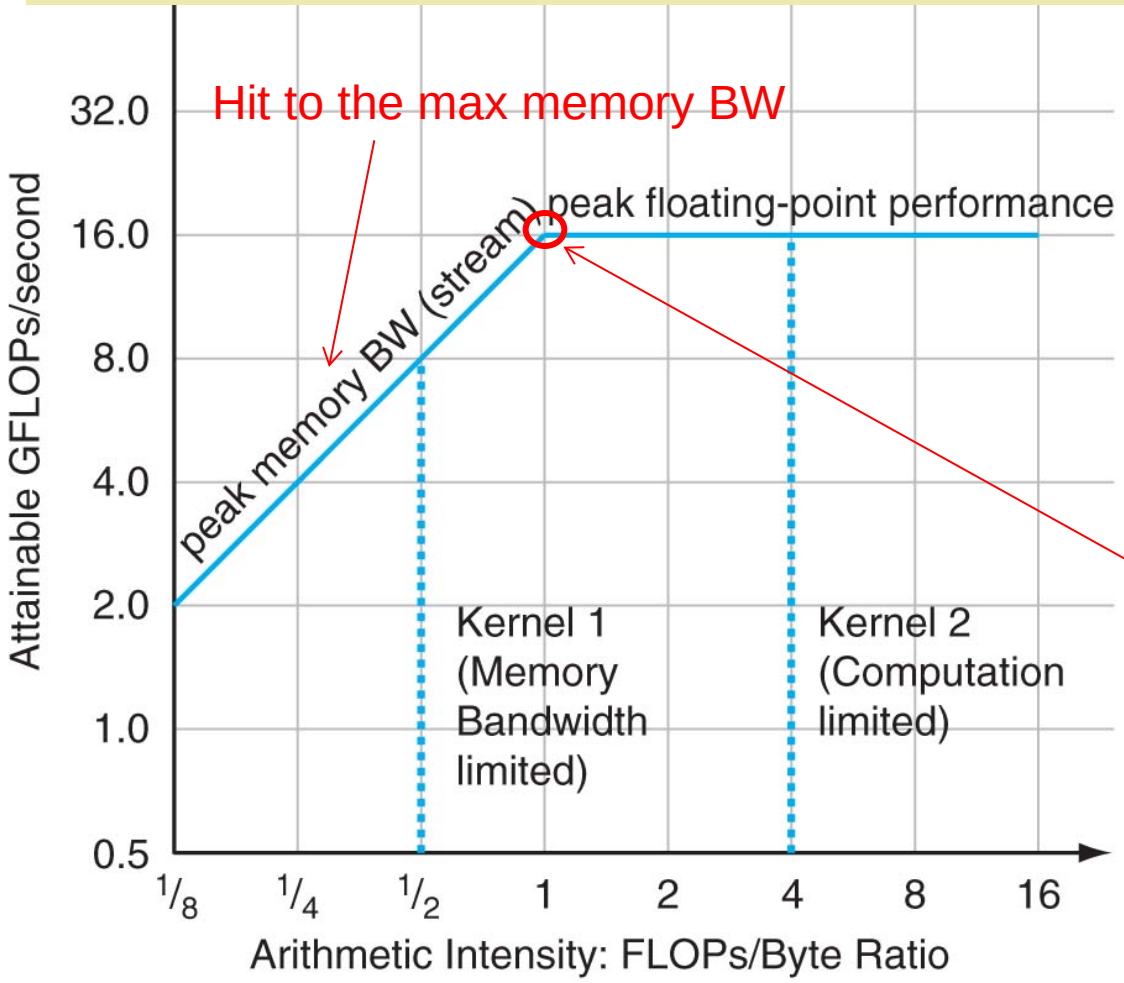
- This model ties floating-point performance, arithmetic intensity, memory performance together in a 2D graph.
- The roofline diagram in Fig 7.13 is for a computer
  - Y-axis: available floating-point performance [GFLOPS/second]
  - X-axis: arithmetic intensity

    [FLOPS/DRAM bytes accessed]
- For a given kernel, we can find a point on the X-axis.
  - The performance of the kernel on that computer must lie where along the line which can be obtained by drawing a vertical line through the point

Roofline Diagram

a log-log scale

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



The ridge point offers an interesting insight into the computer.

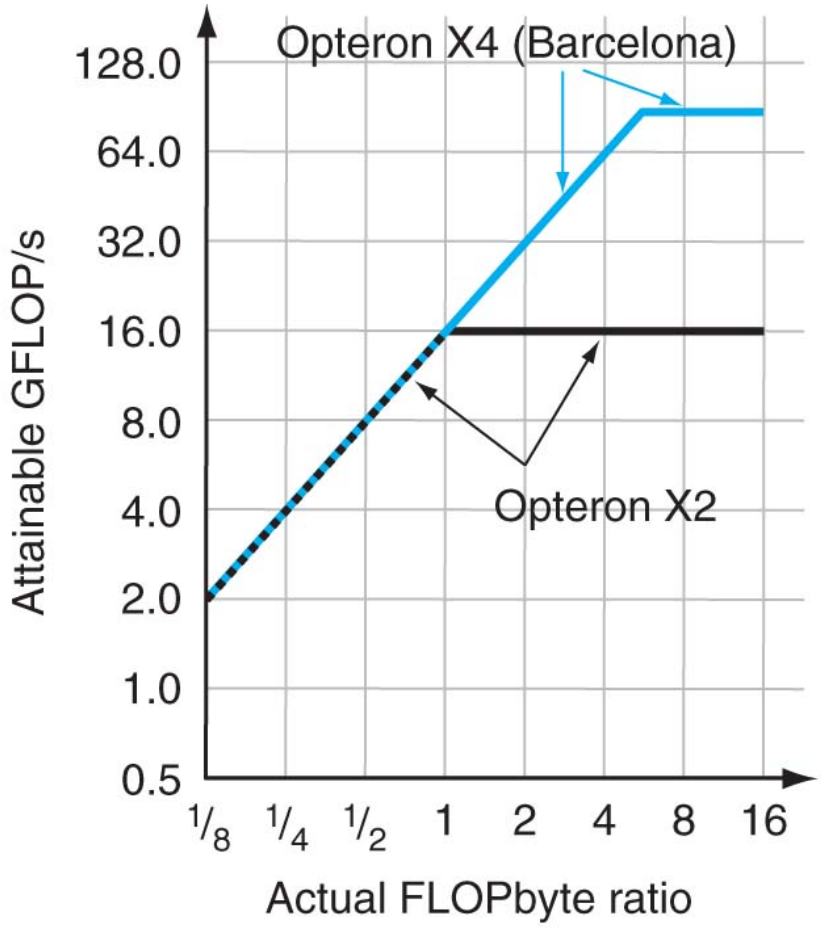
If the ridge point is far to the right, then only kernels with very high arithmetic intensity can achieve the max performance of the computer.

If it is far to the left, then almost any kernel can potentially hit the max performance.

Roofline Model

- This example on the Opteron X2 has
  - a peak floating-point performance of 16 GFLOPS/sec
  - a peak memory bandwidth of 16 GB/sec from the Stream benchmark.
  - Since stream is actually four measurements, this line is the average of the four.
  - Kernel 1, which has an arithmetic intensity of 0.5 FLOPs/byte, is limited by memory bandwidth to no more than 8 GFLOPS/sec.
  - Kernel 2, which has an arithmetic intensity of 4 FLOPs/byte is limited only computationally to 16 GFLOPS/s.
  - This data is based on the AMD Opteron X2 (Revision F) using dual cores running at 2 GHz in a dual socket system.

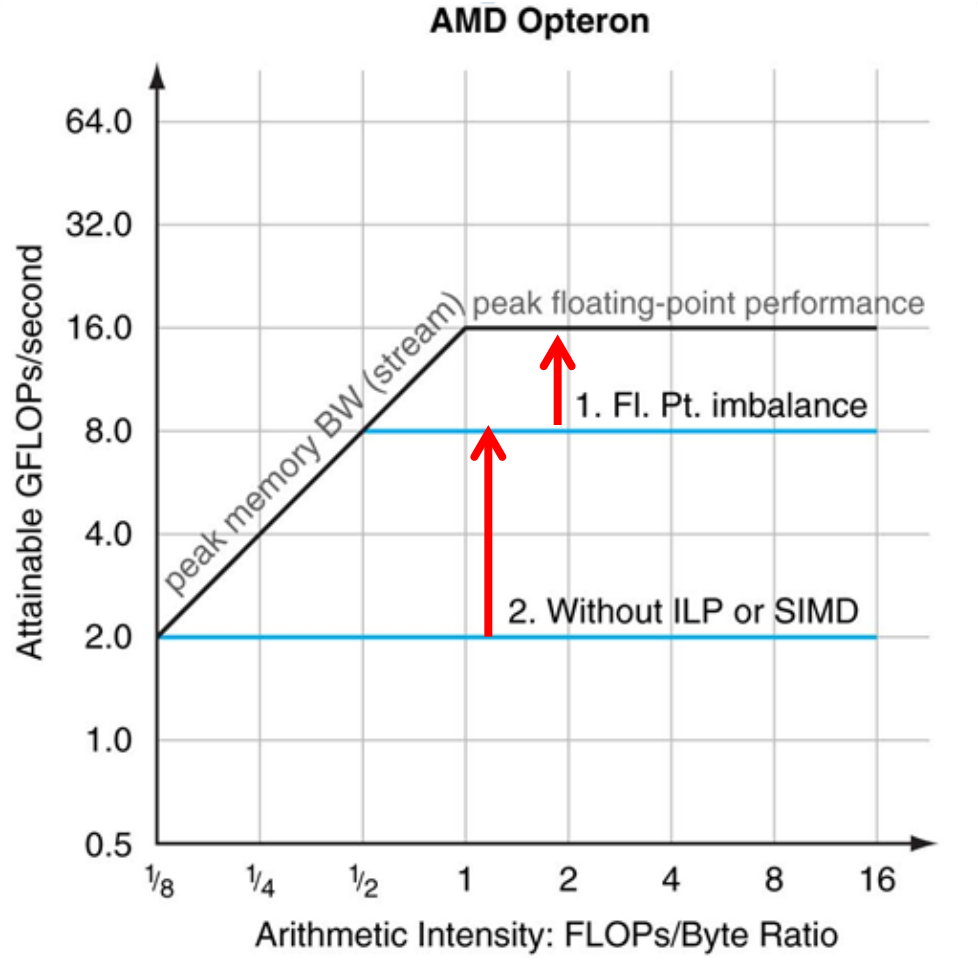
Comparing two systems



- Example: Opteron X2 vs. Opteron X4
  - 2-core vs. 4-core,
  - 2x peak FP performance/core
  - Similar clock 2.2GHz vs. 2.3GHz
  - Used the same sockets → 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

The higher edge point in X4 means memory-performance bound for lower arithmetic intensity on X4

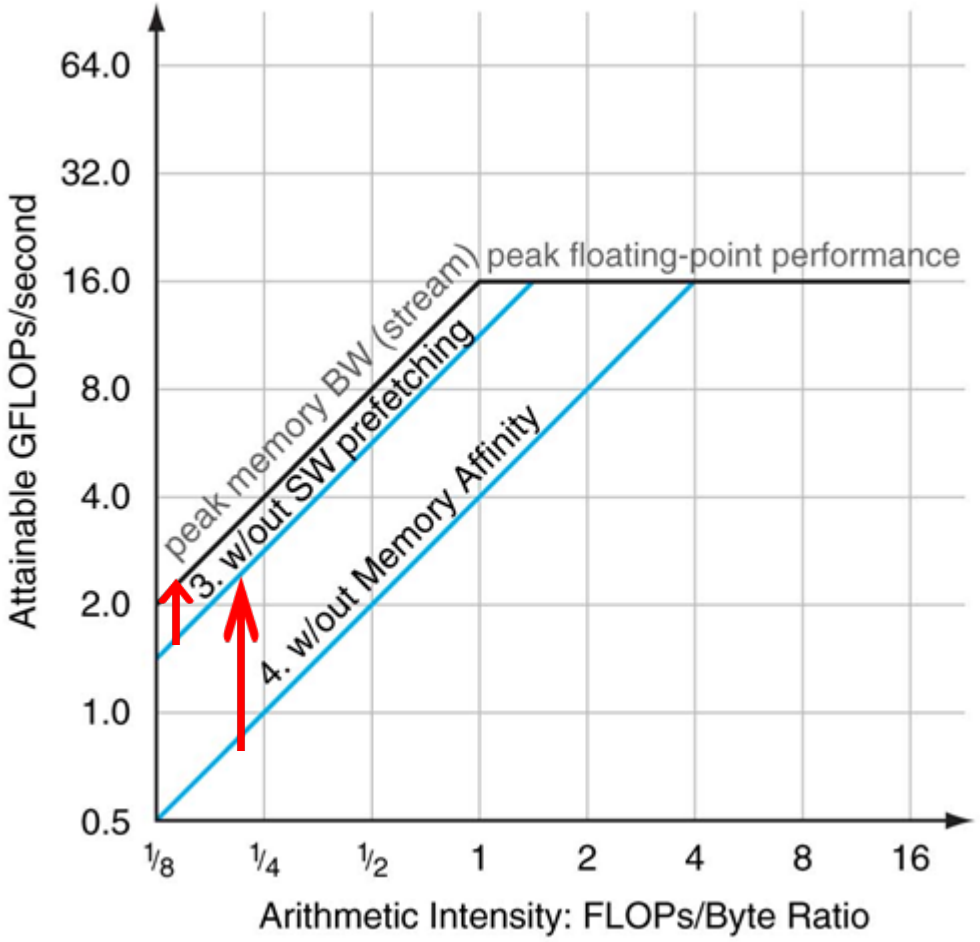
Optimizing Performance



To reduce computational bottleneck, the following two optimizations can help almost any kernel

- 1. Floating-point operation mix by balancing adds & multiplies
  - For a fused multiply-add instructions or
  - Because # of fp adders = # of fp multipliers
- 2. Improve ILP and apply SIMD

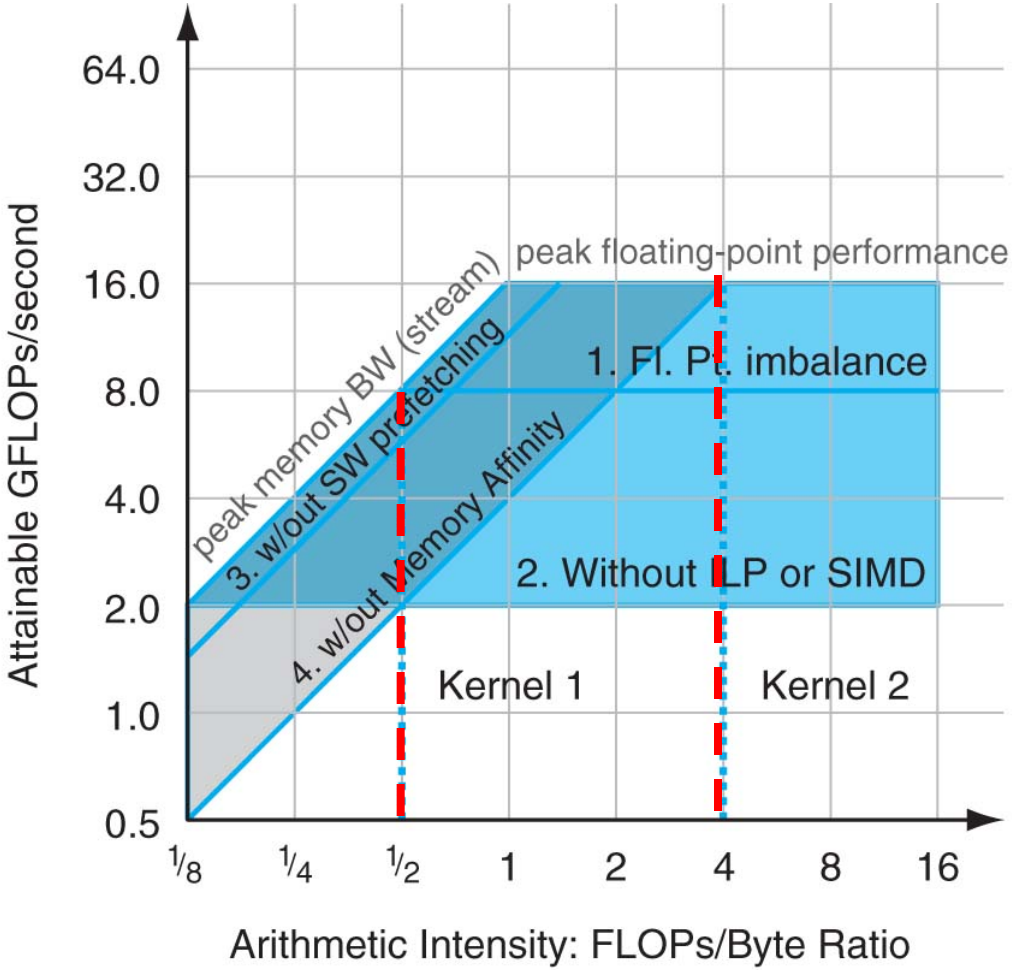
Optimizing Performance



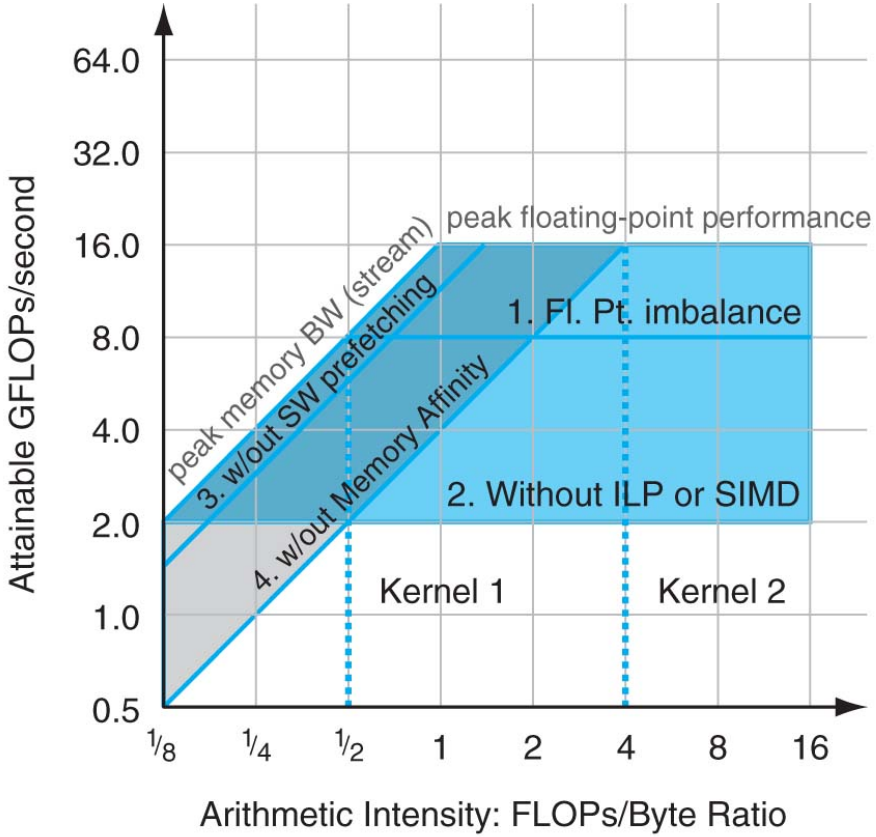
- To reduce memory bottleneck, the following two optimizations can help
- 3. Software prefetch
  - Avoid load stalls
- 4. Memory affinity
  - Memory controller on a chip
  - Multiple memory-processor pairs on a system.
  - Avoid non-local data accesses

Optimizing Performance

- Choice of optimization depends on arithmetic intensity of code



Optimizing Performance



- Arithmetic intensity is not always fixed
- Arithmetic intensity may scale with problem size (for Dense Matrix, and N-boy problems): so weak scaling is better than strong scaling
- Caching reduces memory accesses → Increases arithmetic intensity
  - loop unrolling

Benchmarking 4 multicores

- Examine 4 multicore systems for two kernels of the design patterns: sparse matrix and structured grid. All dual socket systems.
  - (a) Intel Xeon e5345 (Clovertown) : 8 cores
  - (b) AMD Opteron X4 2356 (Barcleona): 8 cores
  - (c) Sun UltraSPARC T2 5140 (Niagara 2): 16 cores
  - (d) IBM Cell QS20: 16 cores

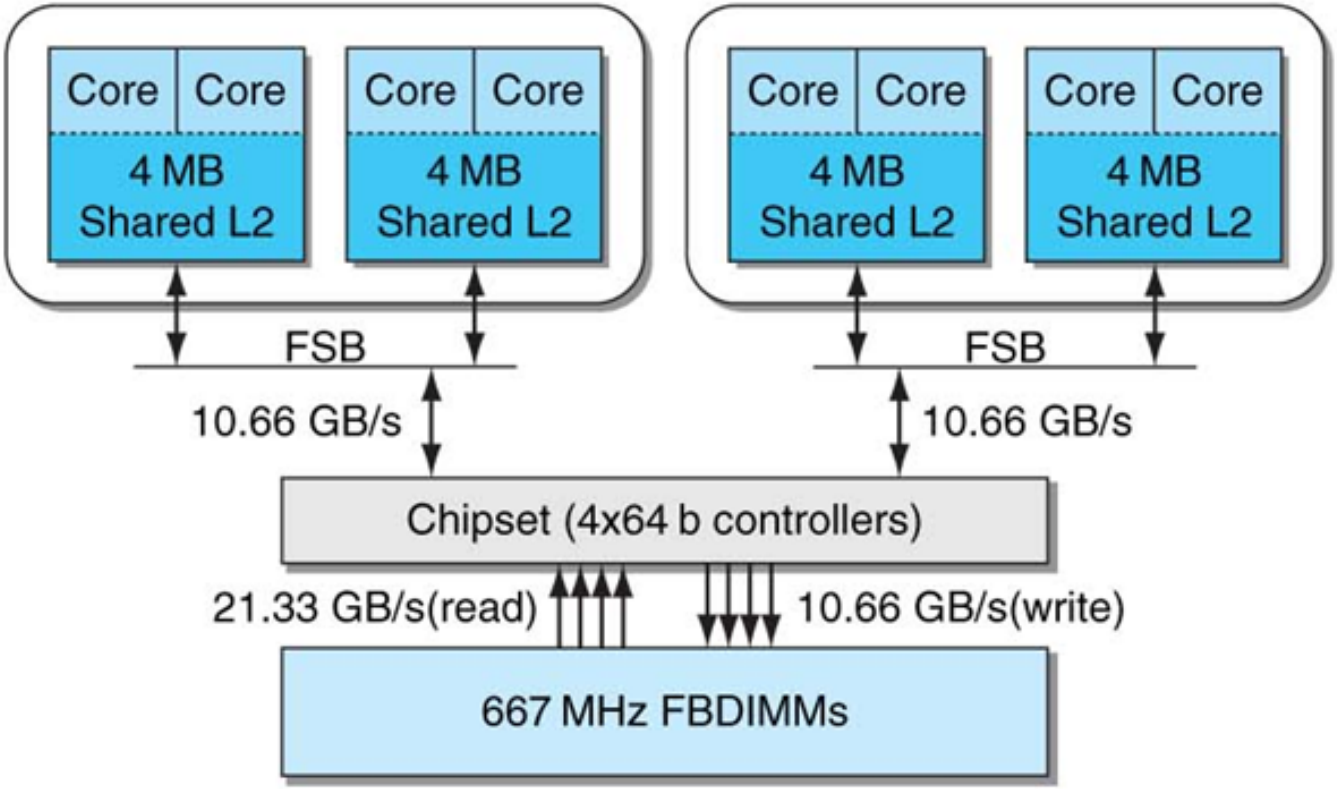
MPU Type	ISA	Number Threads	Number Cores	Number Sockets	Clock GHz	Peak GFLOP/s	DRAM: Peak GB/s, Clock Rate, Type	
Intel Xeon e5345 (Clovertown)	x86/64	8	8	2	2.33	75	FSB: 2 x 10.6	667 MHz FBDIMM
AMD Opteron X4 2356 (Barcelona)	x86/64	8	8	2	2.30	74	2 x 10.6	667 MHz DDR2
Sun UltraSPARC T2 5140 (Niagara 2)	Sparc	128	16	2	1.17	22	2 x 21.3 (read) 2 x 10.6 (write)	667 MHz FBDIMM
IBM Cell QS20	Cell	16	16	2	3.20	29	2 x 25.6	XDR

Xeon e5345 (Clovertown)

Core: 2.33 GHz

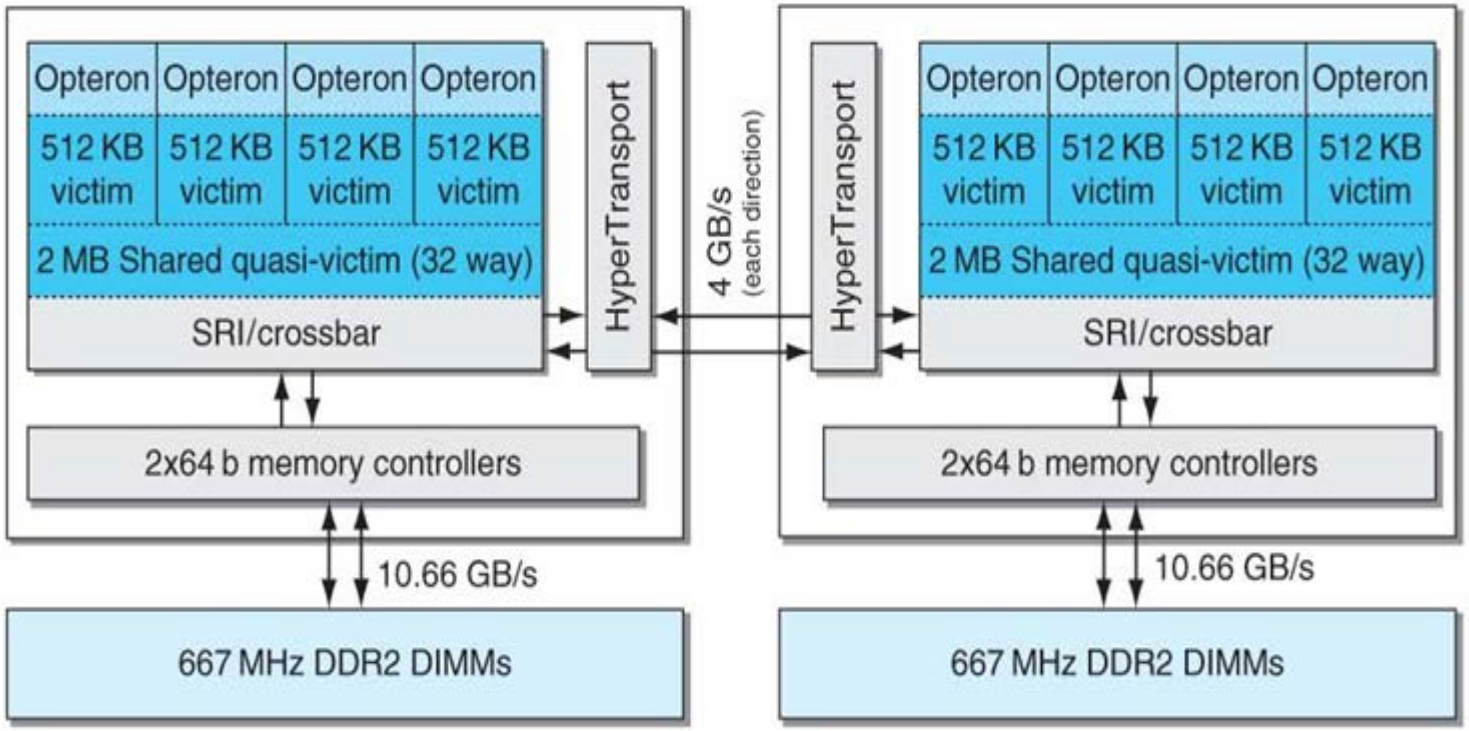
Two dual-core chips per socket

FSB: a weak link



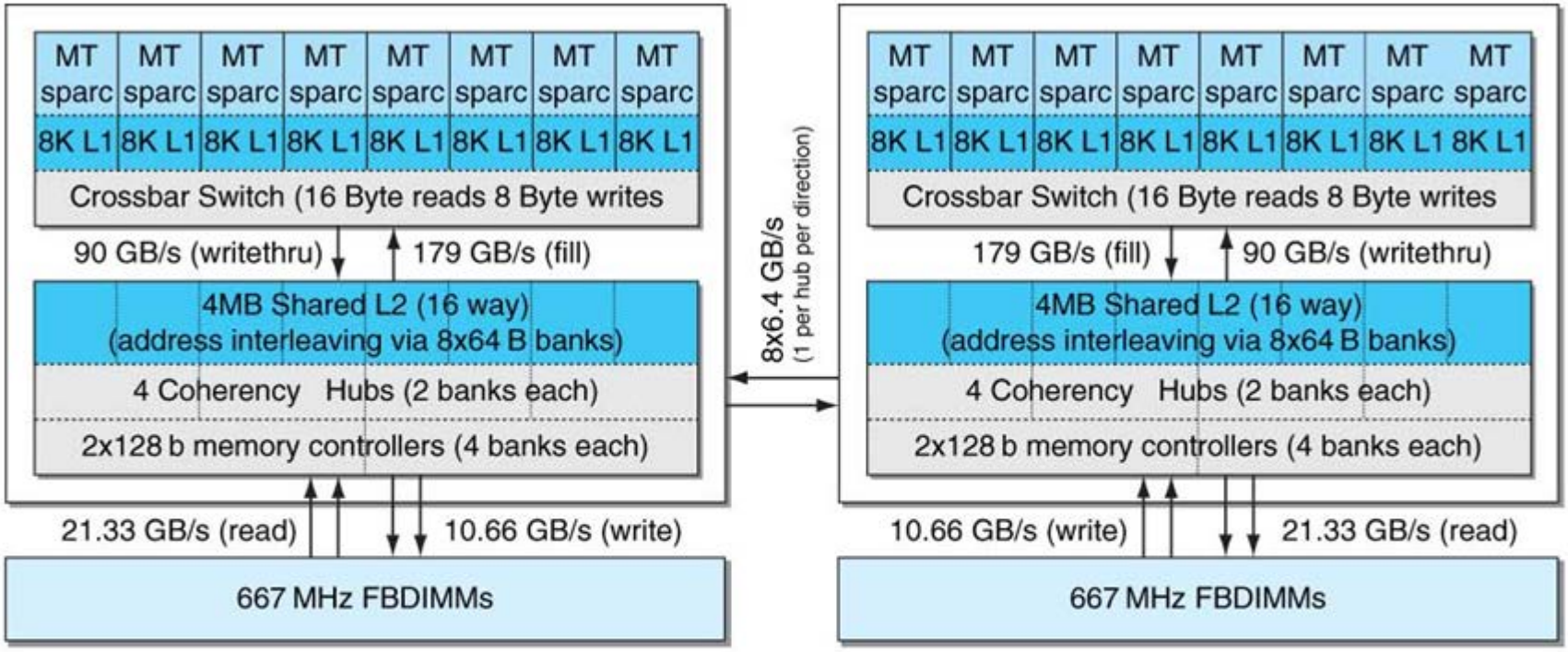
Peak performance: 75 GFLOPS, which cacn be obtained
at arithmetic intensity of 8 or more
Dual FSBs interfere each other → yield relatively low memory BW to programs

Opteron X4 2356 (Barcelona)



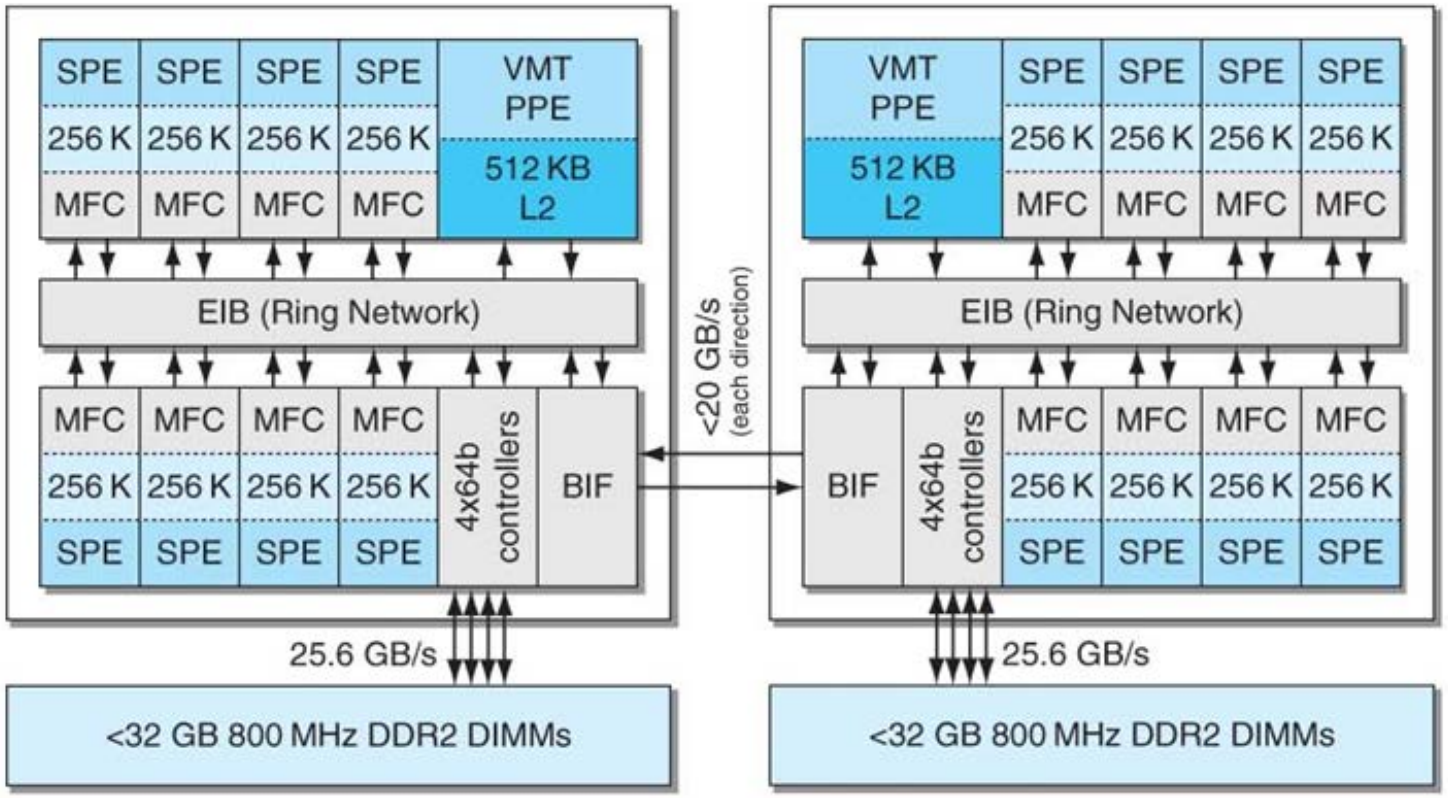
A quad-core chip per socket. Each core: 2.30 GHz
Peak performance: 74 GFLOPS,
Ridge point is at an arithmetic intensity of 5 FLOPS/byte
Two sockets communicates through Hypertransport links
→ glueless multichip systems

UltraSPARC T2 5140 (Niagara 2)



8 relatively simple cores per chip. Each core: 1.17 GHz
Fine-grained multithreading with 8 thread per core
A chip has 4 memory controllers, but two of them are connected to join two T2 chips
Peak performance: 22 GFLOPS,
The ridge point is an very low arithmetic intensity of 1/3 FLOPS/byte

IBM Cell QS20



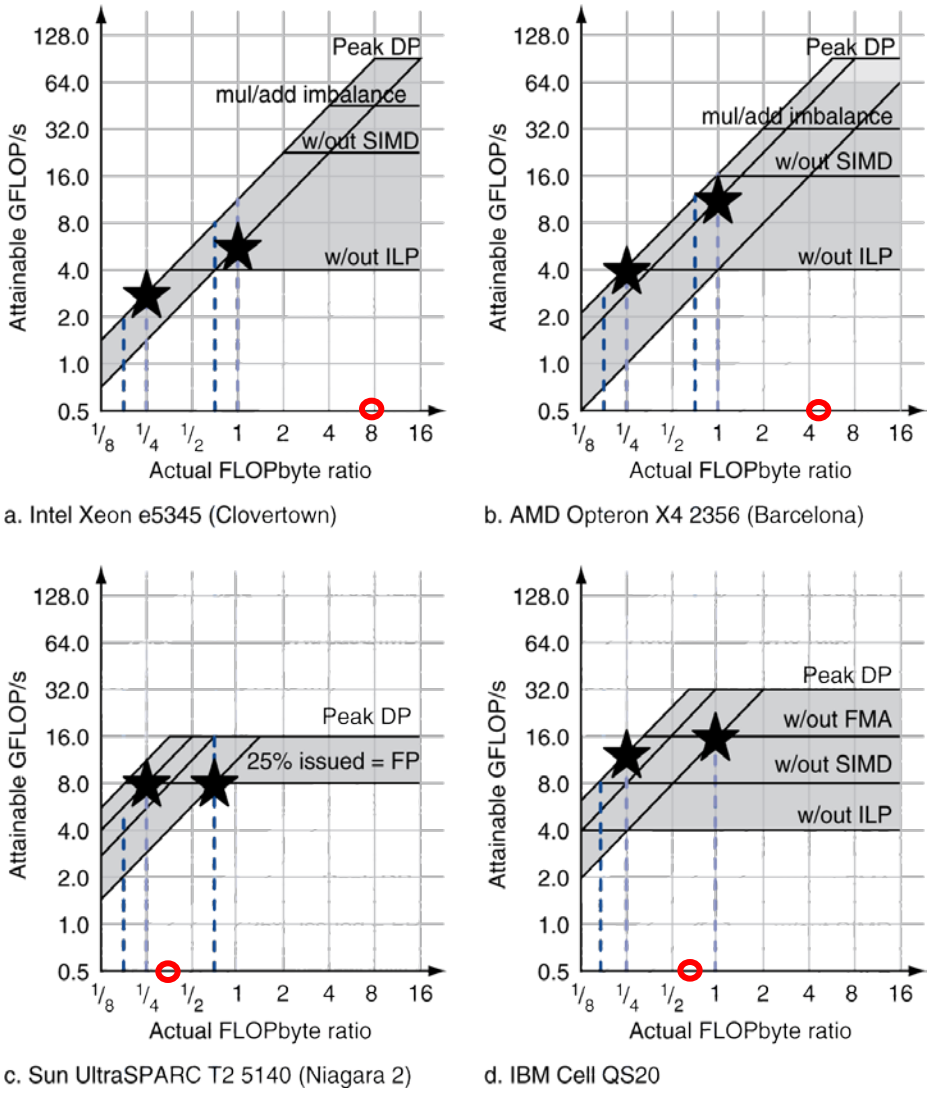
It is a heterogeneous design: one PowerPC core and 8 SPEs with SIMD-like ISA
An SPE with 256K local memory loads/stores data from/to main memory with DMA
tow chips are connected via dedicated links. A core operates at 3.2 GHz and uses XDR DRAMs chips with high BW and low capacity. Peak double precision performance of SPEs is 29 GFLOPS. The ridge point is an low arithmetic intensity of 3/4 FLOPS/byte.

Memory systems

- They have very different approaches to the memory system.
  - (a) Intel Xeon e5345 (Clovertown) : each core has an L1 cache. Each pair of cores shares an L2 cache. They are connected through a shared memory controller.
  - (b) AMD Opteron X4 2356 (Barcleona): has a separate memory controller and memory per chip. Each core has private L1 and L2 caches.
  - (c) Sun UltraSPARC T2 5140 (Niagara 2): has a on-chip memory controller and 4 separate DRAM channels per chip. 8 Cores shares the L2 cache with 4 banks
  - (d) IBM Cell QS20: local private memory to SPE and DMA between the local memory and DRAM.
  - In (c) and (d): many memory accesses in flight are sustained with many cores and multithreading/DMA

And Their Rooflines

- Kernels
  - SpMV: sparse vector multiply (left)
  - LBHMD: Lattice-Boltzmann Magneto-Hydrodynamics (right)
- Some optimizations change arithmetic intensity
- x86 systems have higher peak GFLOPs
  - But harder to achieve, given memory bandwidth



And Their Rooflines

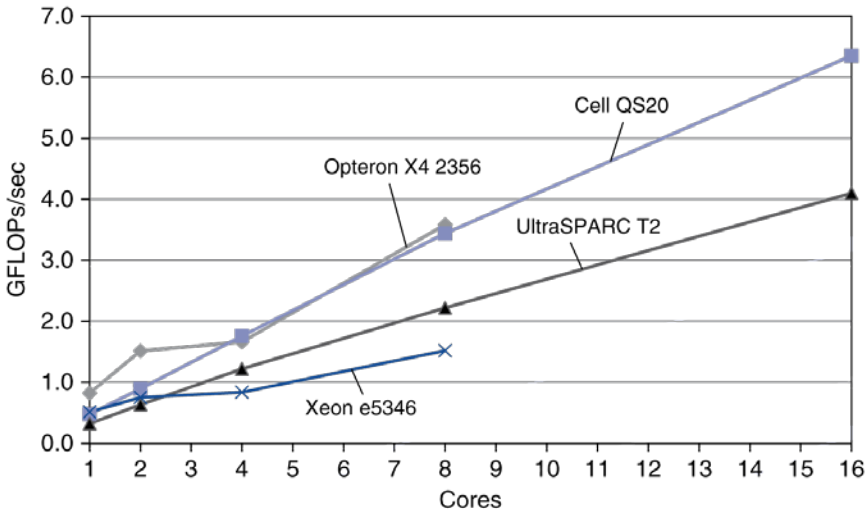
- Note the ridge points for the four microprocessors intersect the X-axis at the arithmetic intensities of 8, 5, 1/3, and 3/4, respectively.
- The dashed vertical lines are for the two kernels of this section and the stars mark the performance achieved for these kernels after all the optimizations.
- SpMV is the pair of dashed vertical lines on the left. It has two lines because its arithmetic intensity improved from 0.166 to 0.255 based on register blocking optimizations.

And Their Rooflines

- LBMHD is the dashed vertical lines on the right. It has a pair of lines in (a) and (b) because a cache optimization skips filling the cache block on a miss when the processor would write new data into the entire block.
- That optimization increases the arithmetic intensity from 0.70 to 1.07.
- It's a single line in (c) at 0.70 because UltraSPARC T2 does not offer the cache optimization.
- It is a single line at 1.07 in (d) because Cell has local store loaded by DMA, so the program doesn't fetch unnecessary data as do caches.

Performance on SpMV

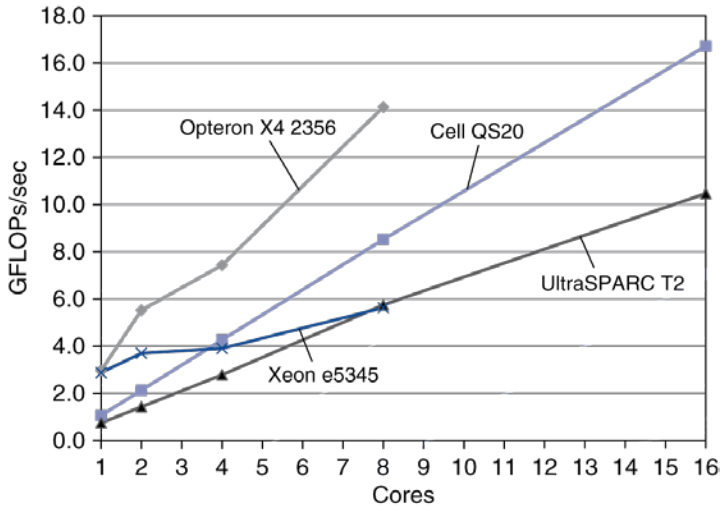
- Sparse matrix/vector multiply
  - Irregular memory accesses, memory bound
- Arithmetic intensity
  - 0.166 before memory optimization, 0.25 after



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

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%