Chapter 6

Parallel Processors from Client to Cloud

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)

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

Hardware and Software

		Software	
		serial	Parallel
Hardware	Serial	Matrix Multiply written in MATLAB running on an Intel Pentium 4	Windows Vista Operating System running on an Intel Pentium 4
	parallel	Matrix Multiply written in MATLAB running on an Intel Corei7	Windows Vista Operating System running on an Intel Corei7

What We've Already Covered

- §2.11: Parallelism and Instructions
  - Synchronization
- §3.6: Parallelism and Computer Arithmetic
  - Associativity
- §4.10: Parallelism and Advanced Instruction-Level Parallelism
- §5.8: Parallelism and Memory Hierarchies
  - Cache Coherence

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 (load balancing)
  - Coordination (scheduling)
  - Communications overhead
  - Synchronization

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

$$\text{Speedup} = T_{old}/((T_{par}/100) + T_{seq})$$

Scaling Example

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

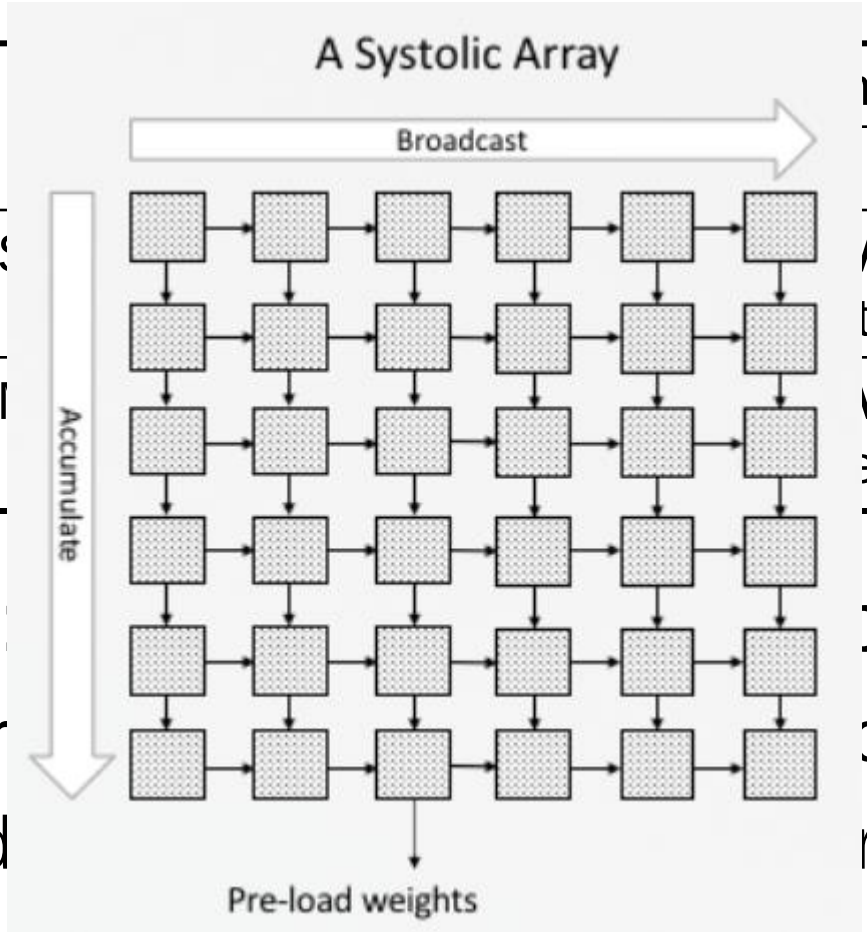
Strong vs Weak Scaling

- 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

Instruction and Data Streams

- An alternate classification

		ns
		Multiple
Instruction Streams	S	**MD**: SSE tructions of x86
	I	**MD**: el Xeon e5345



- SPMD ole Data
  - A par computer
  - Cond rocessors
- SIMD: Data-level parallelism

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

Example: DAXPY (Y = a × X + Y)

- Conventional MIPS code

600

```
        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
```

Many stalls

- Vector MIPS code

6

```
        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
```

One stall

Vector Processors

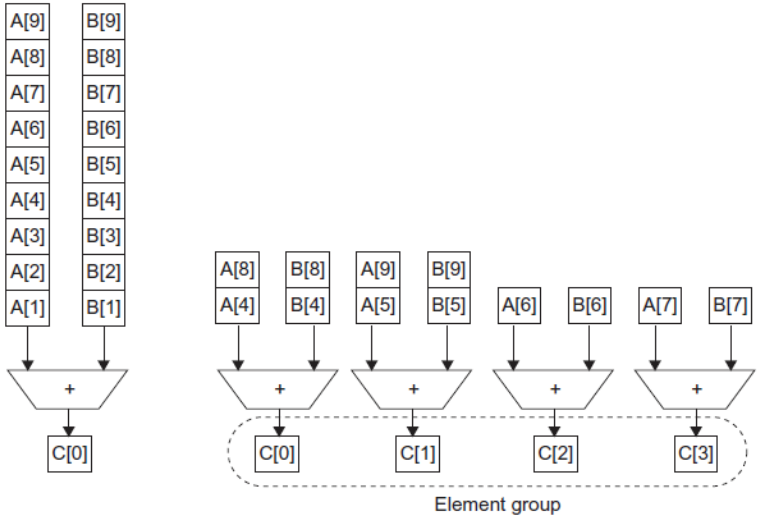
- Highly pipelined function units
- Stream data from/to vector registers to units
  - Data collected from memory into registers
  - Results stored from registers to memory
- Example: Vector extension to MIPS
  - 32 × 64-element registers (64-bit elements)
  - Vector instructions
    - `lv`, `sv`: load/store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

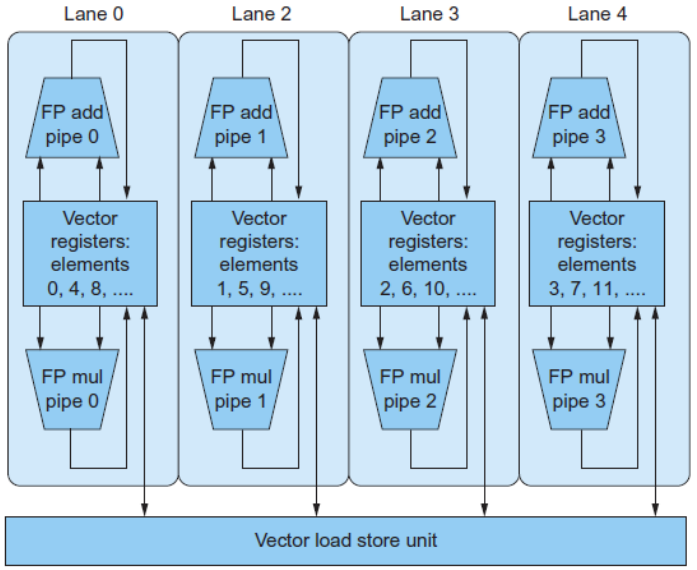
Vector vs. Scalar

- Vector architectures and compilers
  - Few instructions replace many iterations of a group of instructions
  - Explicit statement of absence of loop-carried dependences
    - Reduced checking in hardware
  - Checking data hazard once for two vector instructions rather than for every element within two vectors.
  - Regular access patterns benefit from interleaved and burst memory
  - Avoid control hazards by avoiding loops
- More general than ad-hoc media extensions (such as MMX, SSE)
  - Better match with compiler technology

Vector vs. Multimedia Extensions

- Vector instructions have a variable vector width, MMX has a fixed width
- Vector instructions support strode access, gathering and scattering, MMX does not
- Vector units can be combination of pipelined and arrayed functional units





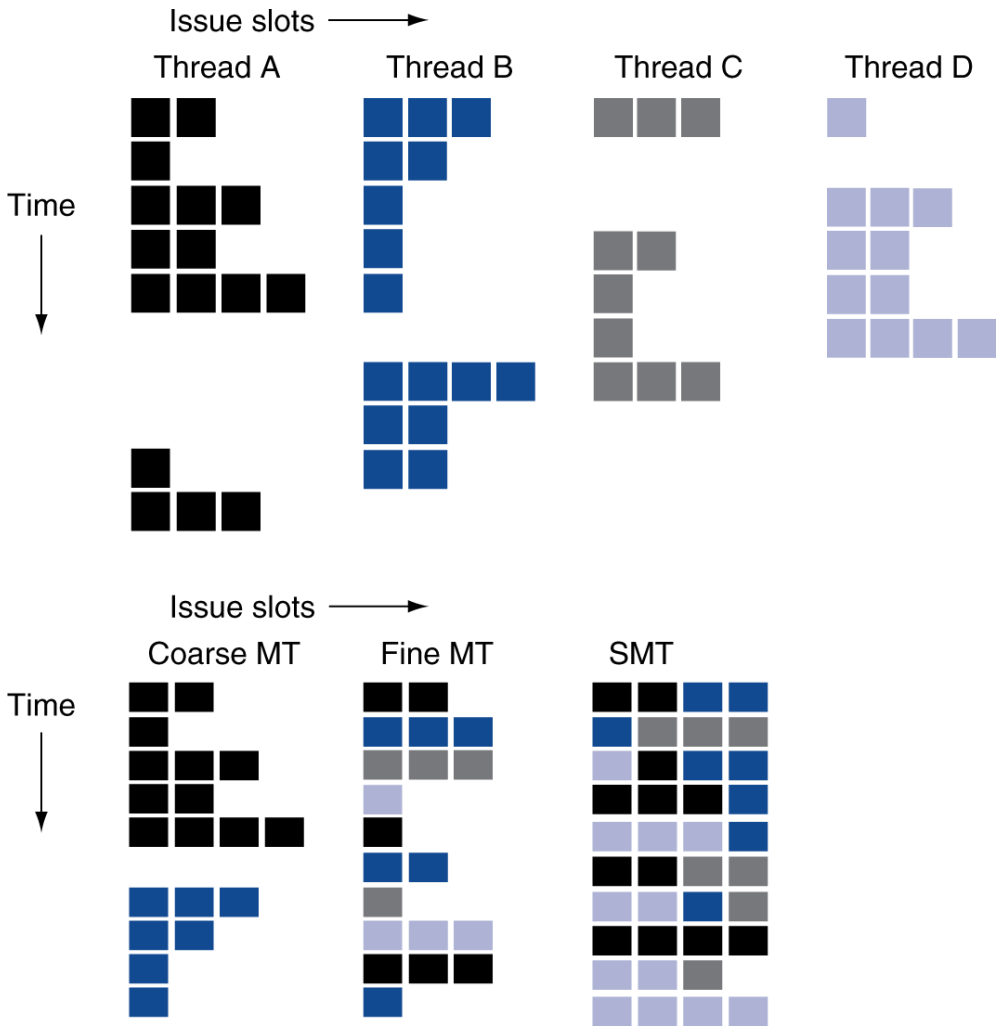
Multithreading

- Performing multiple threads of execution in parallel
  - Replicate registers, PC, etc.
  - Fast switching between threads
- Fine-grain multithreading
  - Switch threads after each cycle
  - Interleave instruction execution
  - If one thread stalls, others are executed
- Coarse-grain multithreading
  - Only switch on long stall (e.g., L2-cache miss)
  - Simplifies hardware, but doesn't hide short stalls (eg, data hazards)
- The shortcoming of FGM and CGM

Simultaneous Multithreading

- 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

Multithreading Example

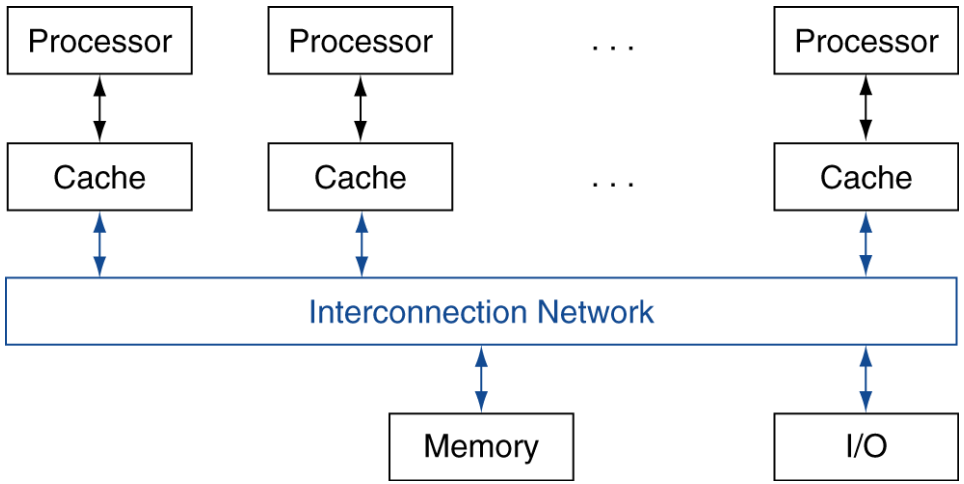


Future of Multithreading

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

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 but scalable)



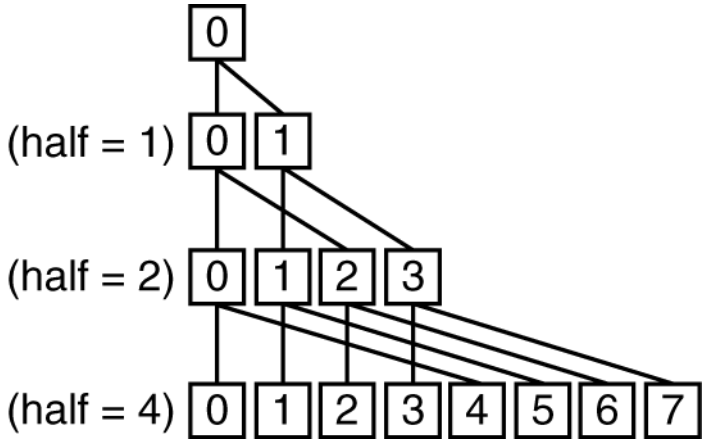
Example: Sum Reduction

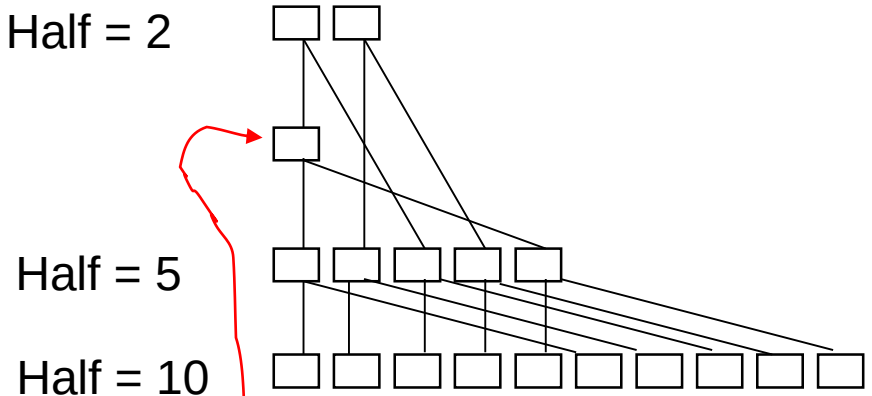
- Sum 100,000 numbers on 100 processor UMA
  - Each processor has ID: 0 ≤ Pn ≤ 99
  - Partition 1000 numbers per processor
  - Initial summation on each processor

```
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



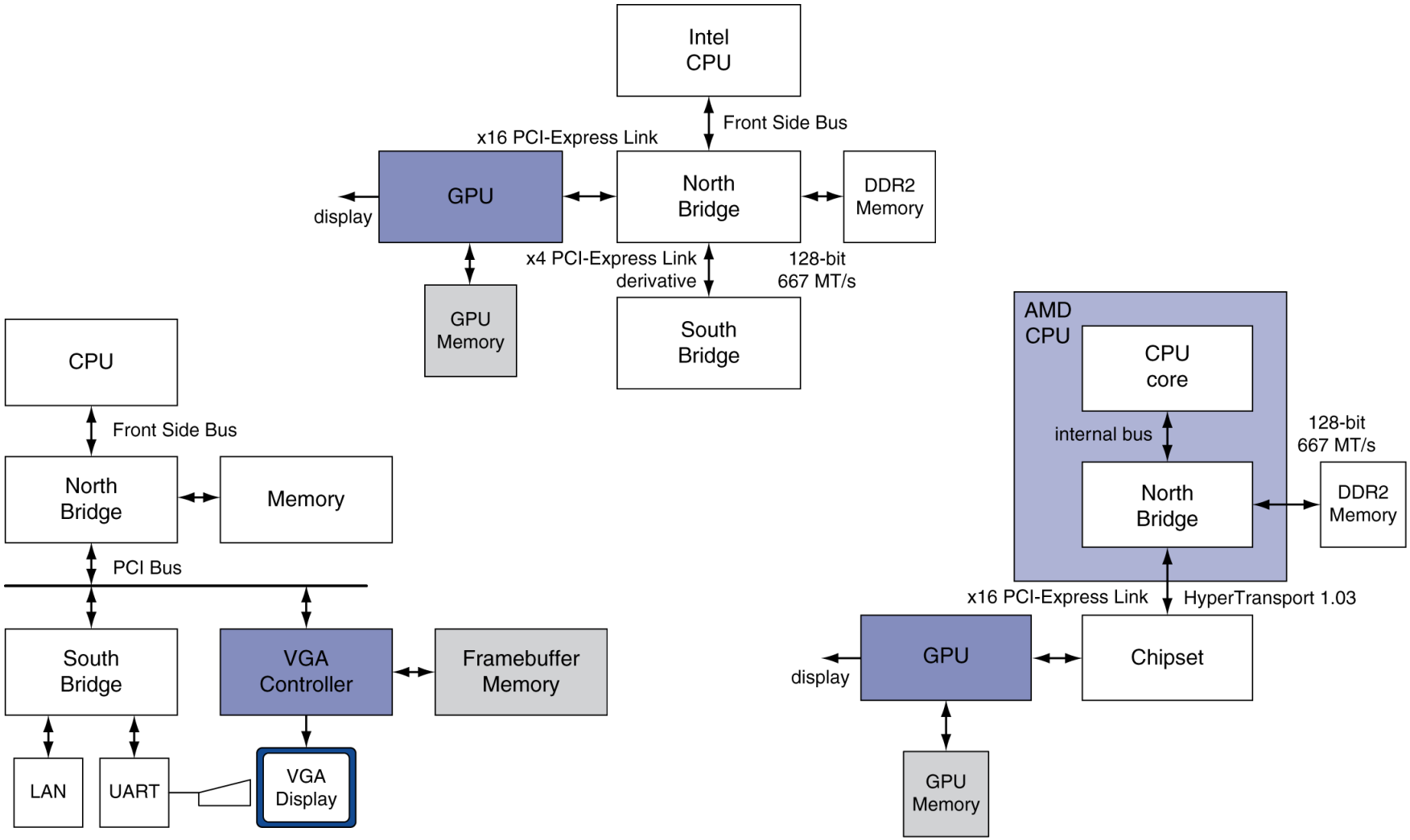



```
half = 100;
repeat
  synch();
  if (half%2 != 0 && Pn == 0)
    sum[0] = sum[0] + sum[half-1];
    /* Conditional sum needed when half is odd;
       Processor0 gets missing element */
  half = half/2; /* dividing line on who sums */
  if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half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

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)
  - Moore's Law $\Rightarrow$ lower cost, higher density
  - 3D graphics cards for PCs and game consoles
- Graphics Processing Units
  - Processors oriented to 3D graphics tasks
  - Vertex/pixel processing, shading, texture mapping, rasterization

Graphics in the System



Differences between GPU and CPU

- Characteristics
  - GPU does not need to perform everything
  - GPU problem sizes: hundreds of megabytes to gigabytes
- Architecture
  - More hardware multithreading rather than small memory latency by multi-level cache.
  - Bandwidth-oriented GPU memory
  - GPUs can accommodate many parallel processors (MIMD) as well as many threads

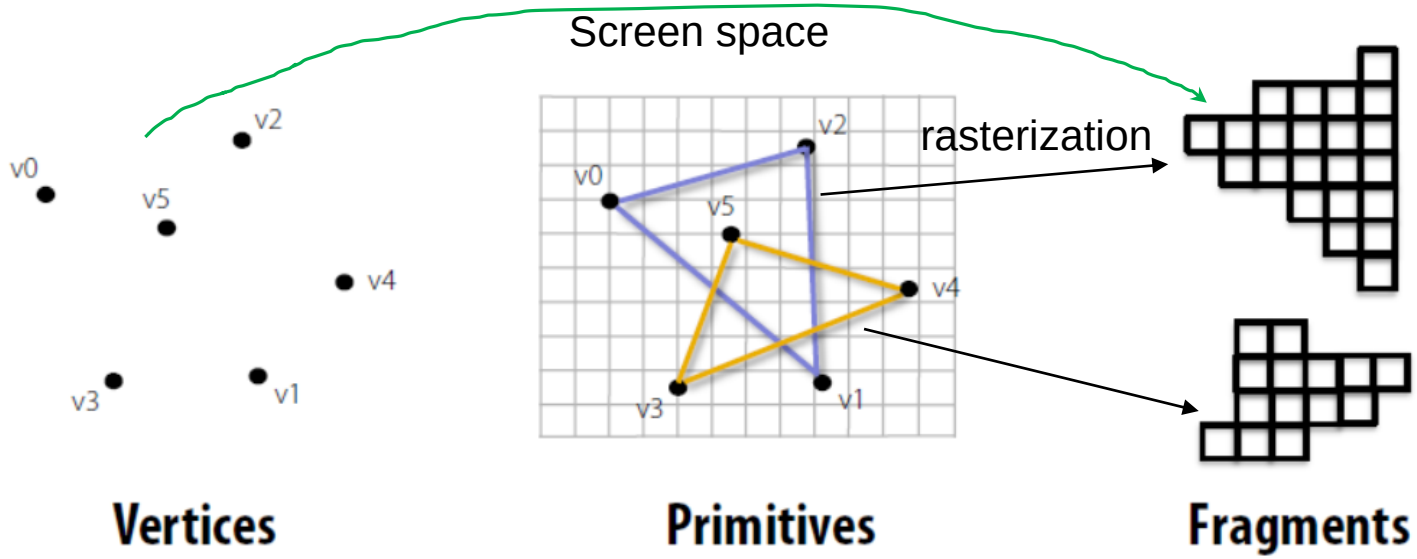
GPU Architectures

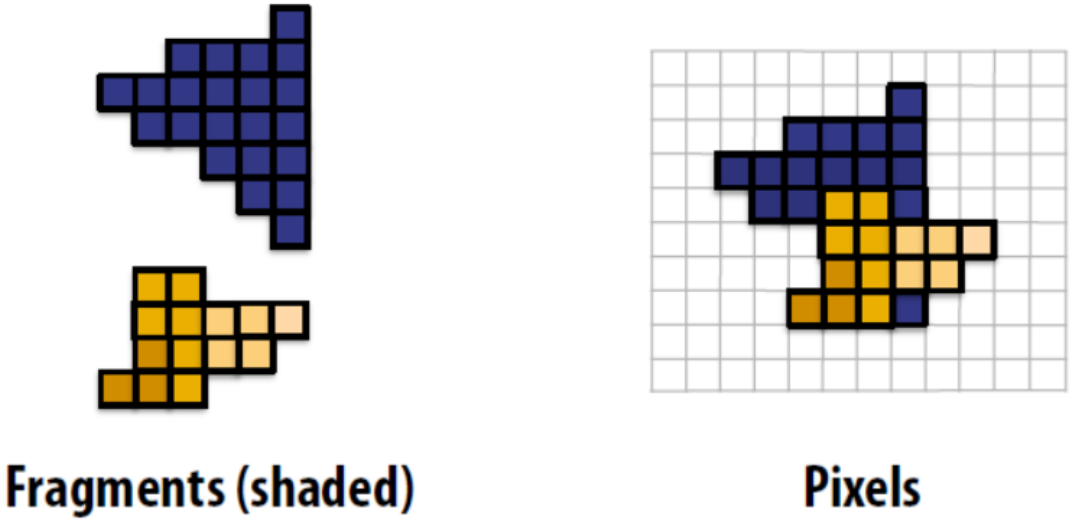
- Processing is highly data-parallel
  - GPUs are highly multithreaded
  - Use thread switching to hide memory latency
    - Less reliance on multi-level caches
  - Graphics memory is wide and high-bandwidth
- Trend toward general purpose GPUs
  - Heterogeneous CPU/GPU systems
  - CPU for sequential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

Cuda vs. OpenCL

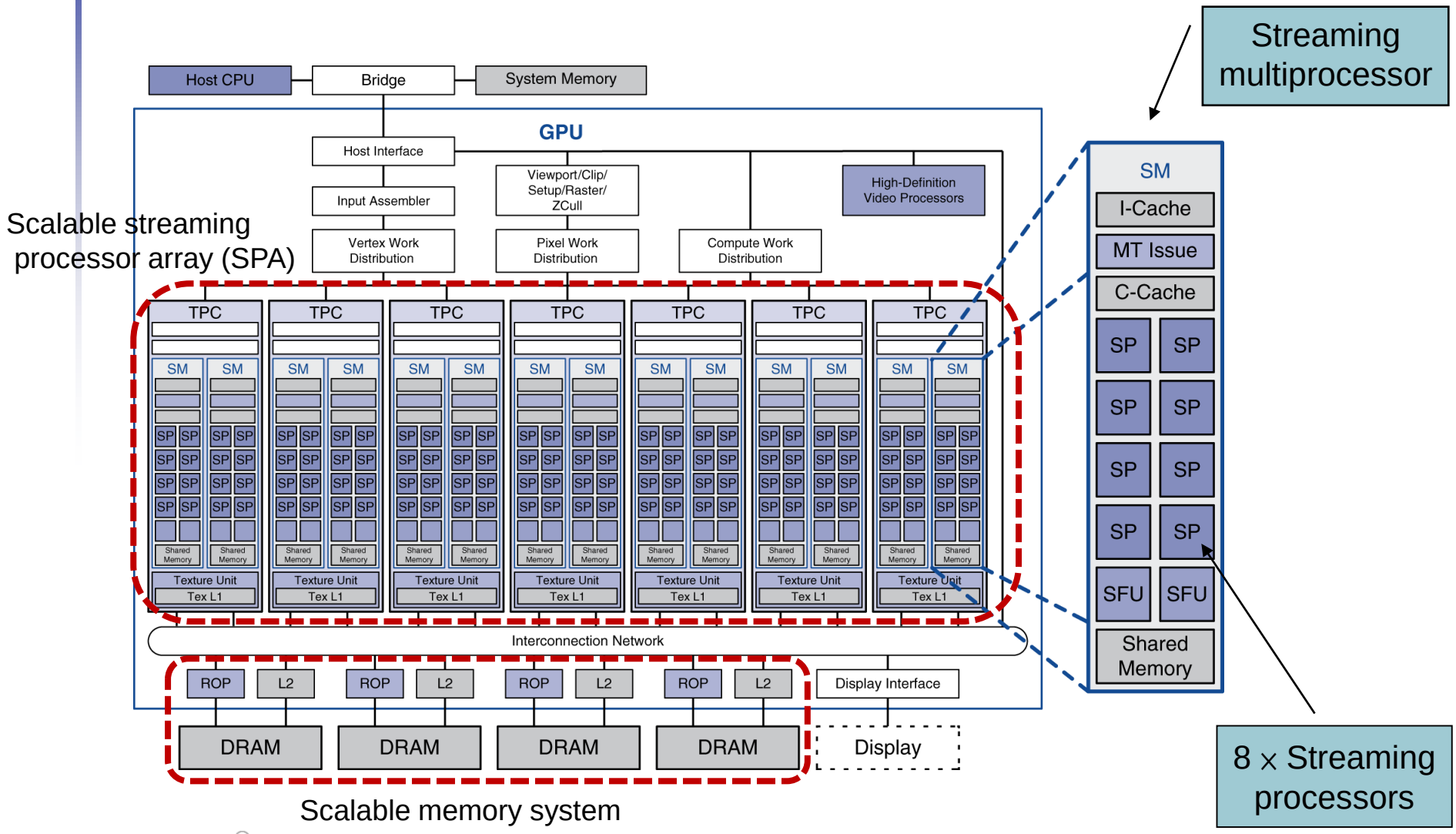
	OpenCL	CUDA
Programming Language	C	C/C++
Supported GPUs	AMD, NVIDIA	NVIDIA
Supported CPUs	AMD, Intel, ARM	None
Method of Creating GPU Work	Kernel	Kernel
Run-time compilation of kernels	Yes	No
Multiple Kernel Execution	Yes (in certain hardware)	Yes (in certain hardware)
Execution Across Multiple Components	Yes	Yes – only GPUs
Need to Optimize for Best Performance	High	High
Coding Complexity	High	Medium

How GPUs Process Images



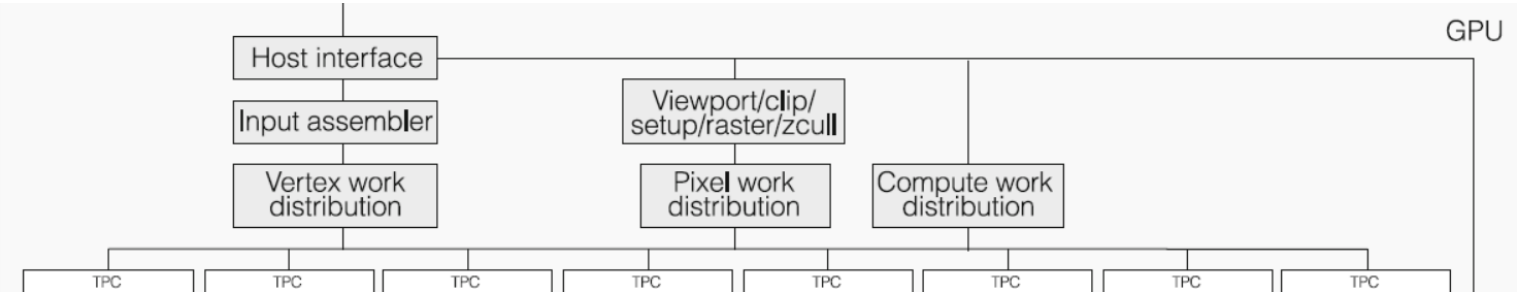


Example: NVIDIA Tesla



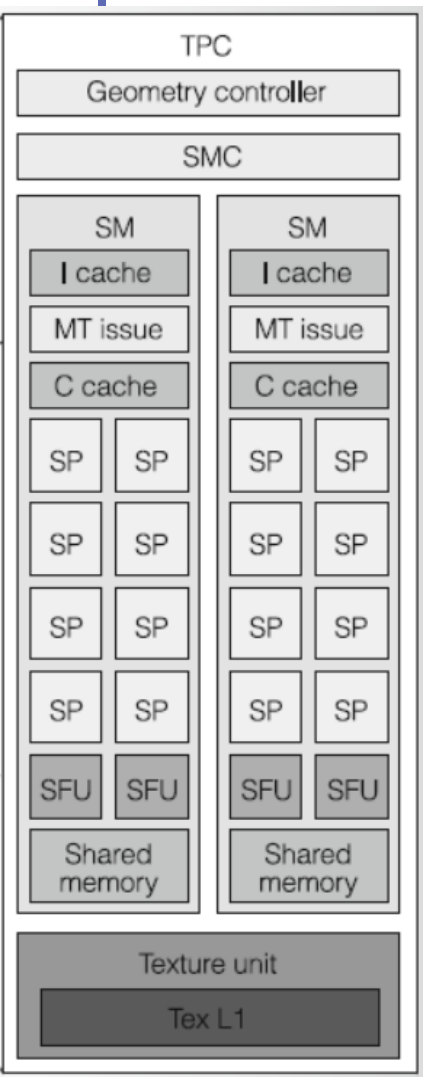
GeForce 8800 GPU Architecture

- 128 streaming processor (SP) cores organized as 16 streaming multiprocessors (SM) in 8 independent processing units, called texture/processor clusters (TPC)
- Streaming processor array is scalable
- Scalable memory system – external DRAM and fixed-function raster operation processors (ROPs) that perform color and depth frame buffer operations directly on memory.



- Input assembler has peak rates of one primitive per clock and eight scalar attributes per clock at the GPU core clock,

GeForce 8800 GPU Architecture

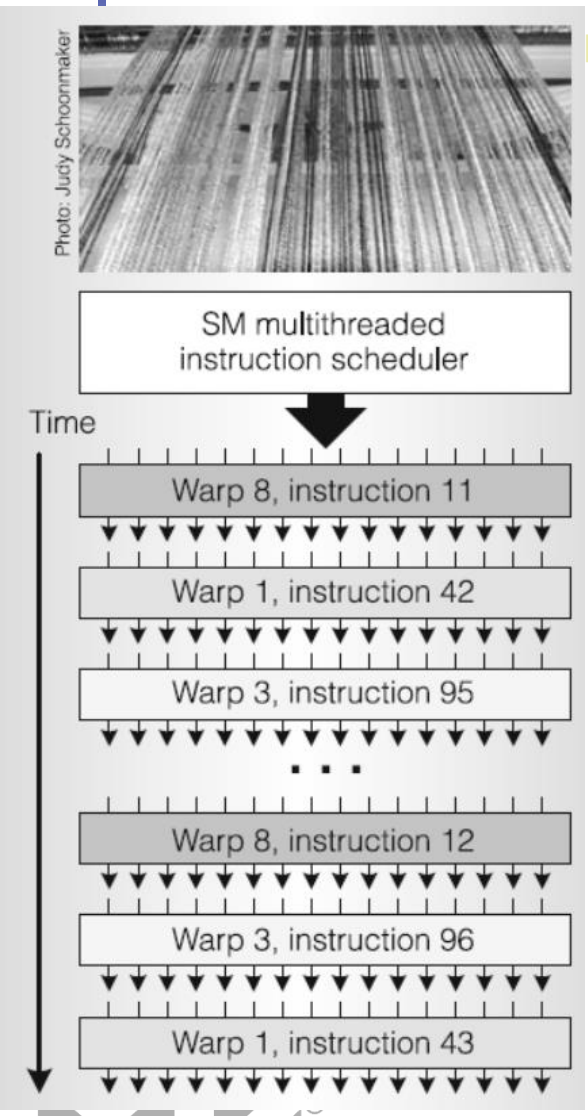


- Geometrical controller
- MT issue – multithreaded instruction fetch and issue unit
- Shared memory – vertex, geometry, and pixel threads have independent input and output buffers. Workloads can arrive and depart independently of thread execution.
- Streaming multiprocessor (SM) – executes vertex, geometry, and pixel-fragment shader programs and parallel computing programs
  - SFU – special function unit for transcendental functions and attribute interpolation

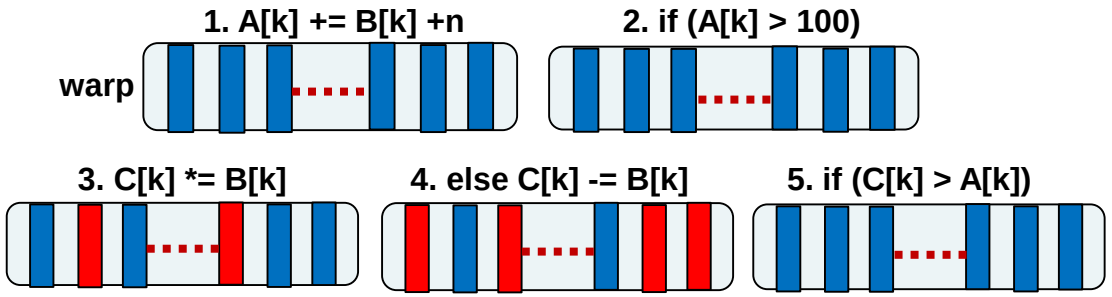
GeForce 8800 GPU Architecture

- SM … continue
  - SP – contains a scalar multiply-add (MAD)
  - Register-based instruction set including floating-point, integer, bit, conversion, transcendental, flow control, memory load/store, and texture operations.
- SM multithreading
  - Thread – a graphic vertex or a pixel shader that is a program to describe how to process a vertex or a pixel; or just a C program
  - Up to 768 concurrent threads with 0 thread scheduling overhead
  - Thread synchronization at a barrier is realized by a SM instruction
  - Lightweight thread creation, zero-overhead thread scheduling, and fast barrier synchronization support fine-grained parallelism

GeForce 8800 GPU Architecture



- Single-Instruction Multiple-Thread
  - Warp is a group of 32 parallel threads of the same type – vertex, geometry, pixel, or compute
  - SM manages a pool of 24 warps (768 threads)
  - Threads in a warp allow independent branch and execution – branch and converge
  - No need to take care of branch and merging



GeForce 8800 GPU Architecture

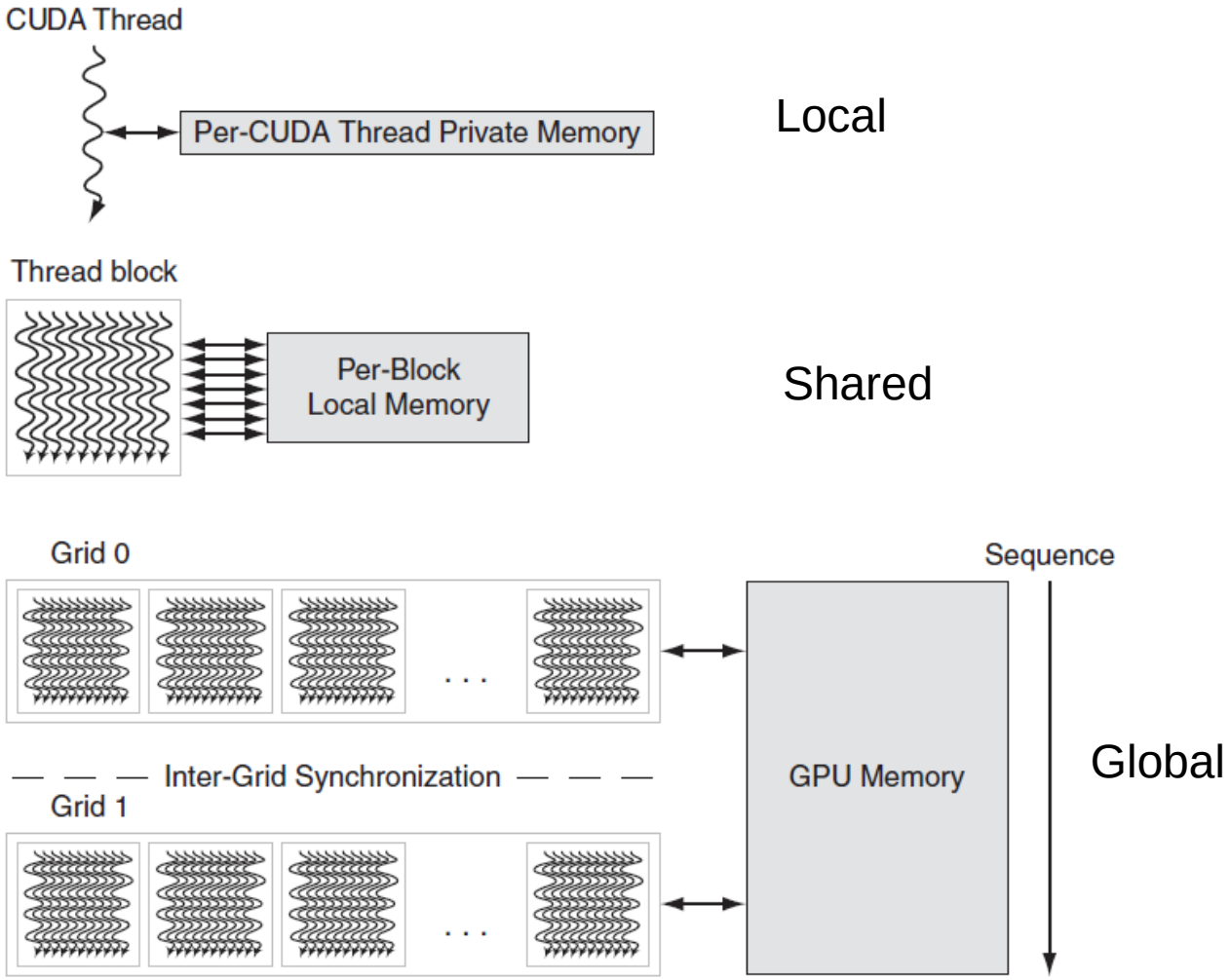
- Memory access
  - Texture unit – fetch texture instruction and filter texture sample from memory
  - The ROP unit writes pixel-fragment output to memory.
  - Computing access – Memory load/store instructions use integer byte addressing with register-plus-offset address
- Memory types
  - Local memory – for per-thread, private, temporary data (implemented in external DRAM);
  - Shared memory – for low-latency access to data shared by cooperating threads in the same SM; and
  - Global memory – for data shared by all threads of a computing application (implemented in external DRAM).
- Fewer memory block accesses.

GeForce 8800 GPU Architecture

- Parallel computing architecture
  - Cooperative thread array (CTA), or a thread block in CUDA terminology
  - Parallel granularity
    - thread—computes result elements selected by its TID;
    - CTA—computes result blocks selected by its CTA ID;
    - grid—computes many result blocks, and sequential grids compute sequentially dependent application steps.

GeForce 8800 GPU Architecture

- Parallel memory access



Volta


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

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**

Putting GPUs into Perspective

Feature	Multicore with SIMD	GPU
SIMD processors	4 to 8	8 to 16
SIMD lanes/processor	2 to 4	8 to 16
Multithreading hardware support for SIMD threads	2 to 4	16 to 32
Typical ratio of single precision to double-precision performance	2:1	2:1
Largest cache size	8 MB	0.75 MB
Size of memory address	64-bit	64-bit
Size of main memory	8 GB to 256 GB	4 GB to 6 GB
Memory protection at level of page	Yes	Yes
Demand paging	Yes	No
Integrated scalar processor/SIMD processor	Yes	No
Cache coherent	Yes	No

Guide to GPU Terms

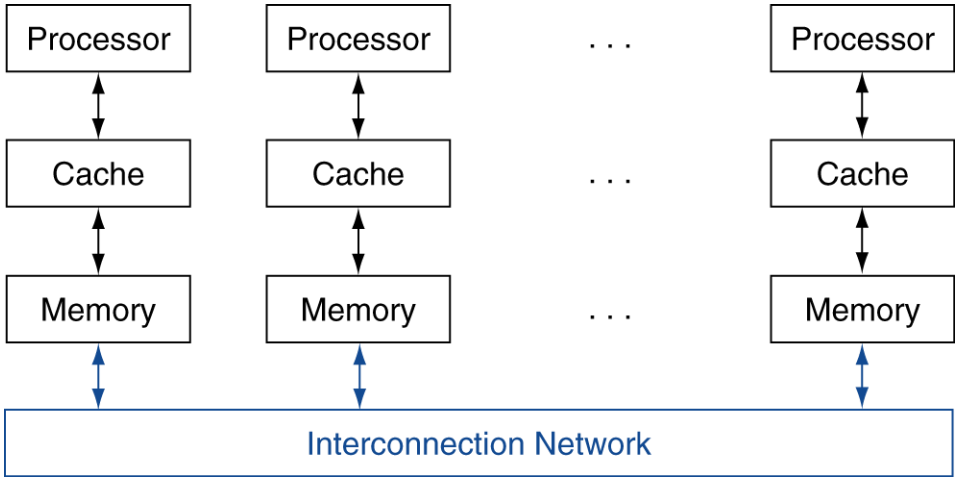
Type	More descriptive name	Closest old term outside of GPUs	Official CUDA/ NVIDIA GPU term	Book definition
Program abstractions	Vectorizable Loop	Vectorizable Loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel.
	Body of Vectorized Loop	Body of a (Strip-Mined) Vectorized Loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via Local Memory.
	Sequence of SIMD Lane Operations	One iteration of a Scalar Loop	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register.
Machine object	A Thread of SIMD Instructions	Thread of Vector Instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask.
	SIMD Instruction	Vector Instruction	PTX Instruction	A single SIMD instruction executed across SIMD Lanes.

Guide to GPU Terms

Type	More descriptive name	Closest old term outside of GPUs	Official CUDA/ NVIDIA GPU term	Book definition
Processing hardware	Multithreaded SIMD Processor	(Multithreaded) Vector Processor	Streaming Multiprocessor	A multithreaded SIMD Processor executes threads of SIMD instructions, independent of other SIMD Processors.
	Thread Block Scheduler	Scalar Processor	Giga Thread Engine	Assigns multiple Thread Blocks (bodies of vectorized loop) to multithreaded SIMD Processors.
	SIMD Thread Scheduler	Thread scheduler in a Multithreaded CPU	Warp Scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution.
	SIMD Lane	Vector lane	Thread Processor	A SIMD Lane executes the operations in a thread of SIMD instructions on a single element. Results stored depending on mask.
Memory hardware	GPU Memory	Main Memory	Global Memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU.
	Local Memory	Local Memory	Shared Memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors.
	SIMD Lane Registers	Vector Lane Registers	Thread Processor Registers	Registers in a single SIMD Lane allocated across a full thread block (body of vectorized loop).

Message Passing

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



Hardware/Software Issues

- Hardware
  - Message passing for communication is much easier for hardware design
- Software
  - Cache coherence for communication is implicit for software

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, dependability, scalable, affordable

Loosely Coupled Clusters

- Problems
  - Administration cost (prefer virtual machines)
  - Low interconnect bandwidth
    - c.f. processor/memory bandwidth on an SMP
- Warehouse-Scale Computers
  - Up to 100000 servers
  - The order of $150M for center house with special electrical and cooling infrastructure

Sum Reduction (Again)

- Sum 100,000 on 100 processors
- First distribute 1000 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

Warehouse-Scale Computers

- Ample, easy parallelism
- Operational costs count
- Scale and the opportunities/problems associated with scale
- Maintenance issue – disk but not processor, fault tolerance issue rises due to the AFR of 2%

Grid Computing

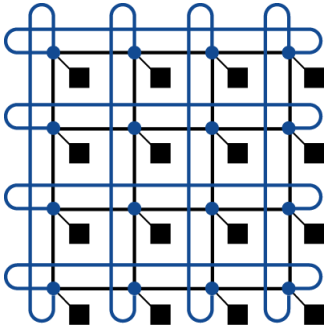
- Separate computers interconnected by long-haul networks
  - E.g., Internet connections
  - Work units farmed out, results sent back
- Can make use of idle time on PCs
  - E.g., SETI@home, World Community Grid
  - Over 5 million computer users in more than 200 countries offer SETI@home 257 TeraFLOPS.

Interconnection Networks

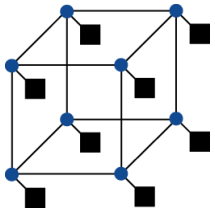
- Network topologies
  - Arrangements of processors, switches, and links

Bus

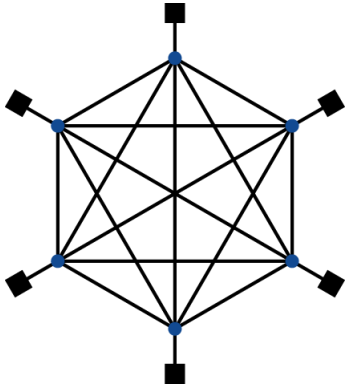
Ring



2D Mesh



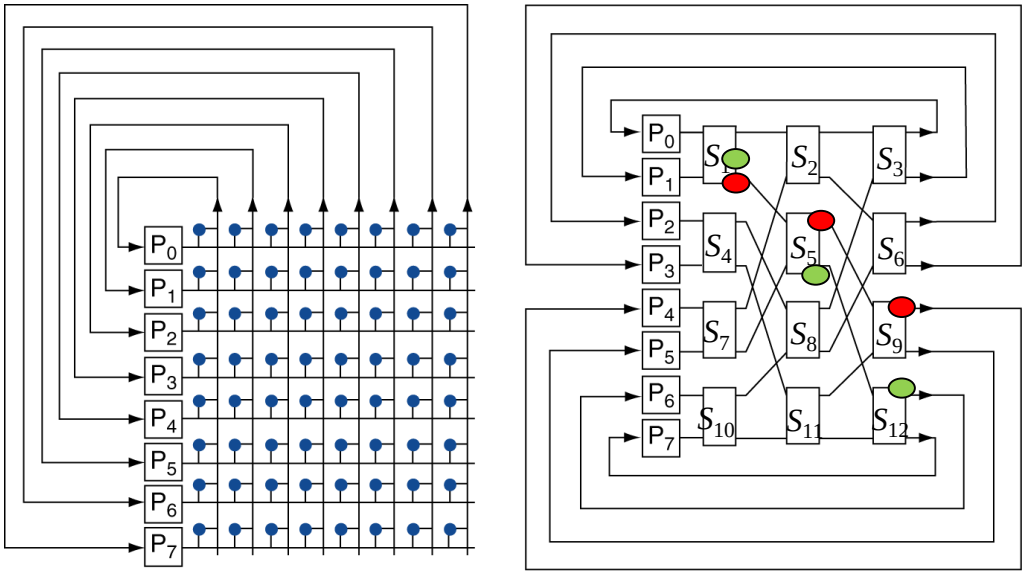
N-cube (N = 3)



Fully connected

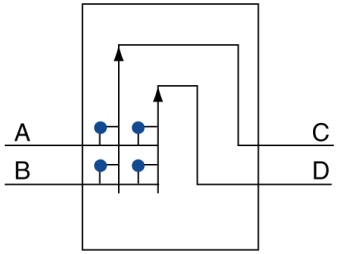
Multistage Networks

- Multistage network



a. Crossbar

b. Omega network



c. Omega network switch box

Network Characteristics

- Performance
  - Latency per message (unloaded network)
  - Throughput
    - Link bandwidth
    - Total network bandwidth
    - Bisection bandwidth
  - Congestion delays (depending on traffic)
- Reliability
- Cost
- Power
- Routability in silicon

Parallel Benchmarks

- Linpack: matrix linear algebra
- SPECrate: parallel run of SPEC CPU programs
  - Job-level parallelism
- SPLASH: Stanford Parallel Applications for Shared Memory
  - Mix of kernels and applications, strong scaling
- NAS (NASA Advanced Supercomputing) suite
  - computational fluid dynamics kernels
- PARSEC (Princeton Application Repository for Shared Memory Computers) suite
  - Multithreaded applications using Pthreads and OpenMP

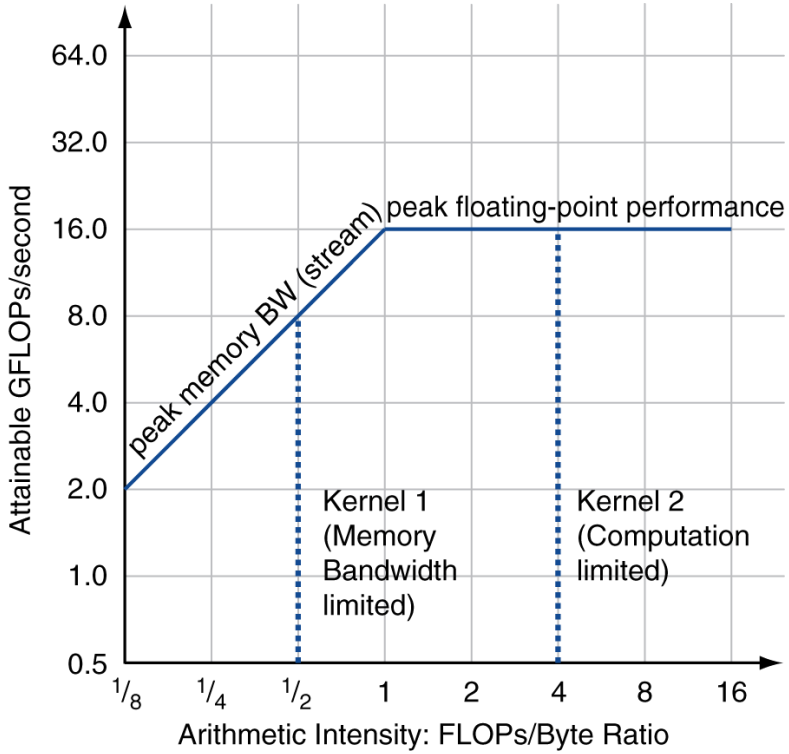
Code or Applications?

- Traditional benchmarks
  - Fixed code and data sets
- Parallel programming is evolving
  - Should algorithms, programming languages, and tools be part of the system?
  - Compare systems, provided they implement a given application
  - E.g., Linpack, Berkeley Design Patterns
- Would foster innovation in approaches to parallelism

Modeling Performance

- Assume performance metric of interest is achievable GFLOPs/sec
  - Measured using computational kernels from Berkeley Design Patterns
- Arithmetic intensity of a kernel
  - FLOPs per byte of memory accessed
- For a given computer, determine
  - Peak GFLOPS (from data sheet)
  - Peak memory bytes/sec (using Stream benchmark)

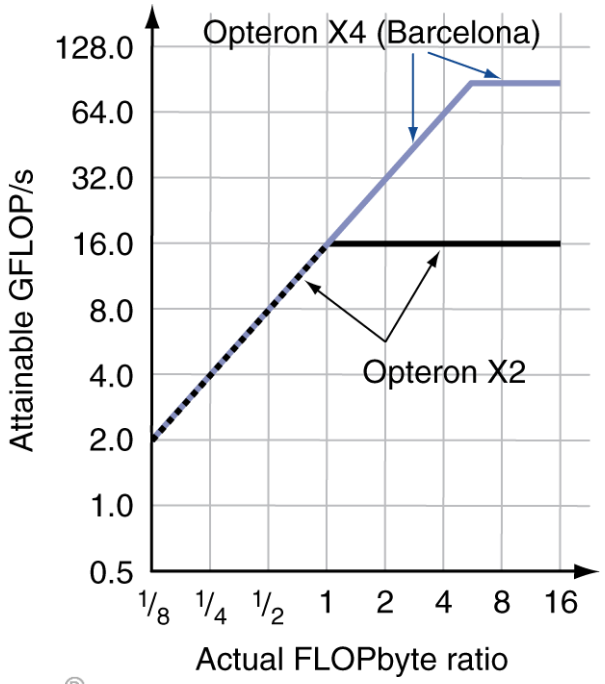
Roofline Diagram



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

Comparing Systems

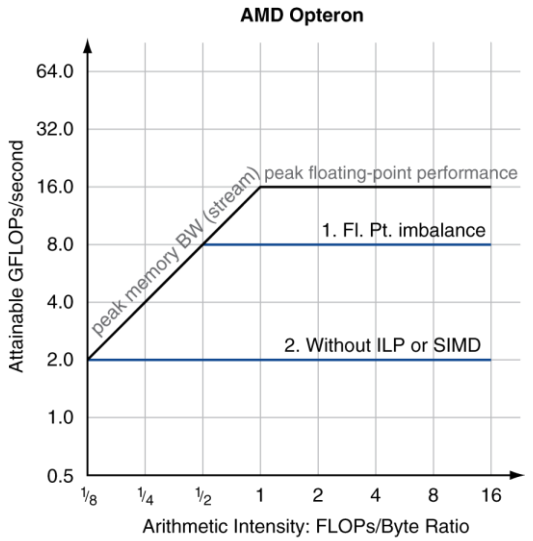
- Example: Opteron X2 vs. Opteron X4
  - 2-core vs. 4-core, 2× FP performance/core, 2.2GHz vs. 2.3GHz
  - Same memory system

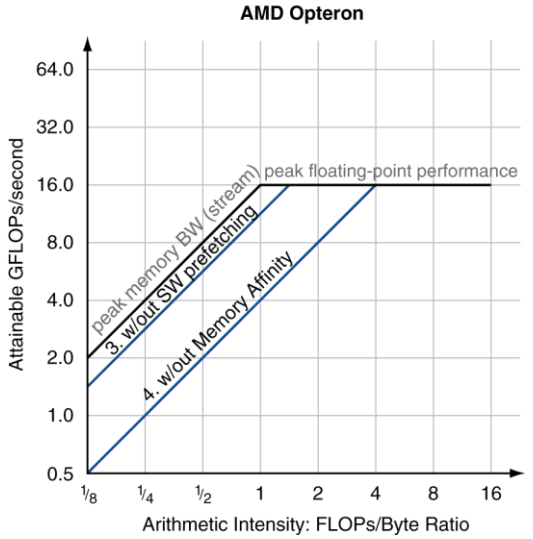


- To get higher performance on X4 than X2
  - Need high arithmetic intensity
  - Or working set must fit in X4's 2MB L-3 cache

Optimizing Performance

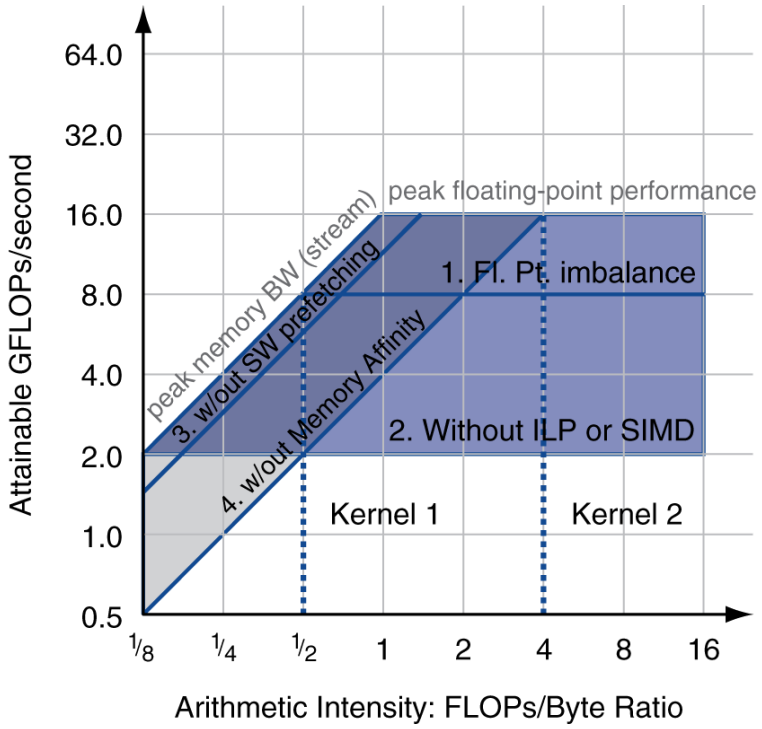
- Optimize FP performance
  - Balance adds & multiplies
  - Improve superscalar ILP and use of SIMD instructions
- Optimize memory usage
  - Software prefetch
    - Avoid load stalls
  - Memory affinity
    - Avoid non-local data accesses





Optimizing Performance

- Choice of optimization depends on arithmetic intensity of code

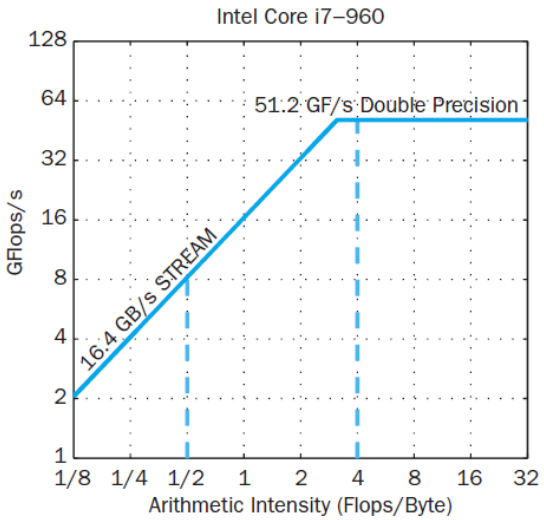


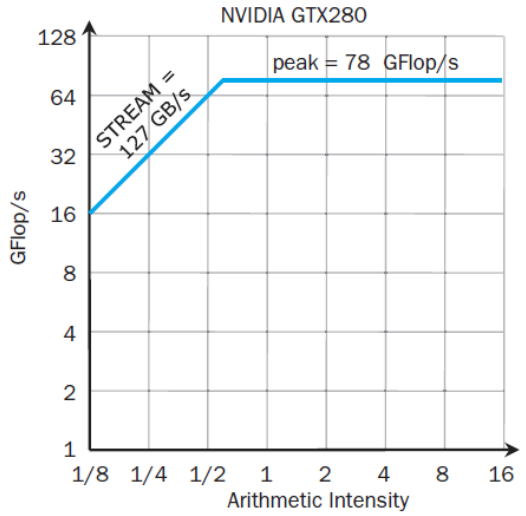
- Arithmetic intensity is not always fixed
  - May scale with problem size
  - Caching reduces memory accesses
    - Increases arithmetic intensity

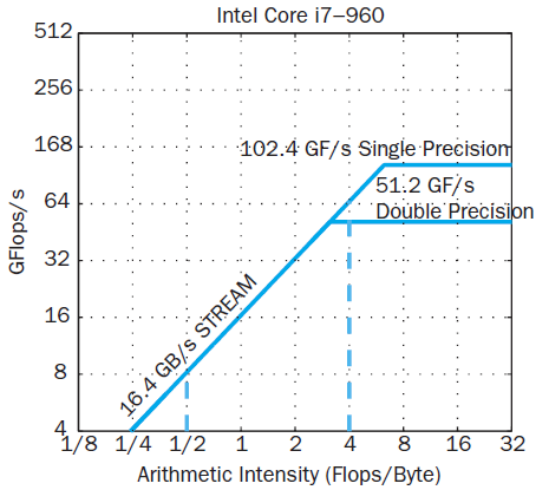
i7-960 vs. NVIDIA Tesla 280/480

	Core i7-960	GTX 280	GTX 480	Ratio 280/i7	Ratio 480/i7
Number of processing elements (cores or SMs)	4	30	15	7.5	3.8
Clock frequency (GHz)	3.2	1.3	1.4	0.41	0.44
Die size	263	576	520	2.2	2.0
Technology	Intel 45 nm	TCMS 65 nm	TCMS 40 nm	1.6	1.0
Power (chip, not module)	130	130	167	1.0	1.3
Transistors	700 M	1400 M	3100 M	2.0	4.4
Memory brandwith (GBytes/sec)	32	141	177	4.4	5.5
Single frecision SIMD width	4	8	32	2.0	8.0
Dobule precision SIMD with	2	1	16	0.5	8.0
Peak Single frecision scalar FLOPS (GFLOP/sec)	26	117	63	4.6	2.5
Peak Single frecision s SIMD FLOPS (GFLOP/Sec)	102	311 to 933	515 to 1344	3.0-9.1	6.6-13.1
(SP 1 add or multiply)	N.A.	(311)	(515)	(3.0)	(6.6)
(SP 1 instruction fused)	N.A	(622)	(1344)	(6.1)	(13.1)
(face SP dual issue fused)	N.A	(933)	N.A	(9.1)	–
Peal double frecision SIMD FLOPS (GFLOP/sec)	51	78	515	1.5	10.1

Rooflines


Intel Core i7–960
128
64
32
16
8
4
2
1
GFlops/s
51.2 GF/s Double Precision
16.4 GB/s STREAM
1/8 1/4 1/2 1 2 4 8 16 32
Arithmetic Intensity (Flops/Byte)


NVIDIA GTX280
128
64
32
16
8
4
2
1
GFlop/s
peak = 78 GFlop/s
STREAM = 127 GB/s
1/8 1/4 1/2 1 2 4 8 16
Arithmetic Intensity


Intel Core i7–960
512
256
168
64
32
16
8
4
GFlops/s
102.4 GF/s Single Precision
51.2 GF/s
Double Precision
16.4 GB/s STREAM
1/8 1/4 1/2 1 2 4 8 16 32
Arithmetic Intensity (Flops/Byte)

Benchmarks

Kernel	Units	Core i7-960	GTX 280	GTX 280/ i7-960
SGEMM	GFLOP/sec	94	364	3.9
MC	Billion paths/sec	0.8	1.4	1.8
Conv	Million pixels/sec	1250	3500	2.8
FFT	GFLOP/sec	71.4	213	3.0
SAXPY	GBytes/sec	16.8	88.8	5.3
LBM	Million lookups/sec	85	426	5.0
Solv	Frames/sec	103	52	0.5
SpMV	GFLOP/sec	4.9	9.1	1.9
GJK	Frames/sec	67	1020	15.2
Sort	Million elements/sec	250	198	0.8
RC	Frames/sec	5	8.1	1.6
Search	Million queries/sec	50	90	1.8
Hist	Million pixels/sec	1517	2583	1.7
Bilat	Million pixels/sec	83	475	5.7

Performance Summary

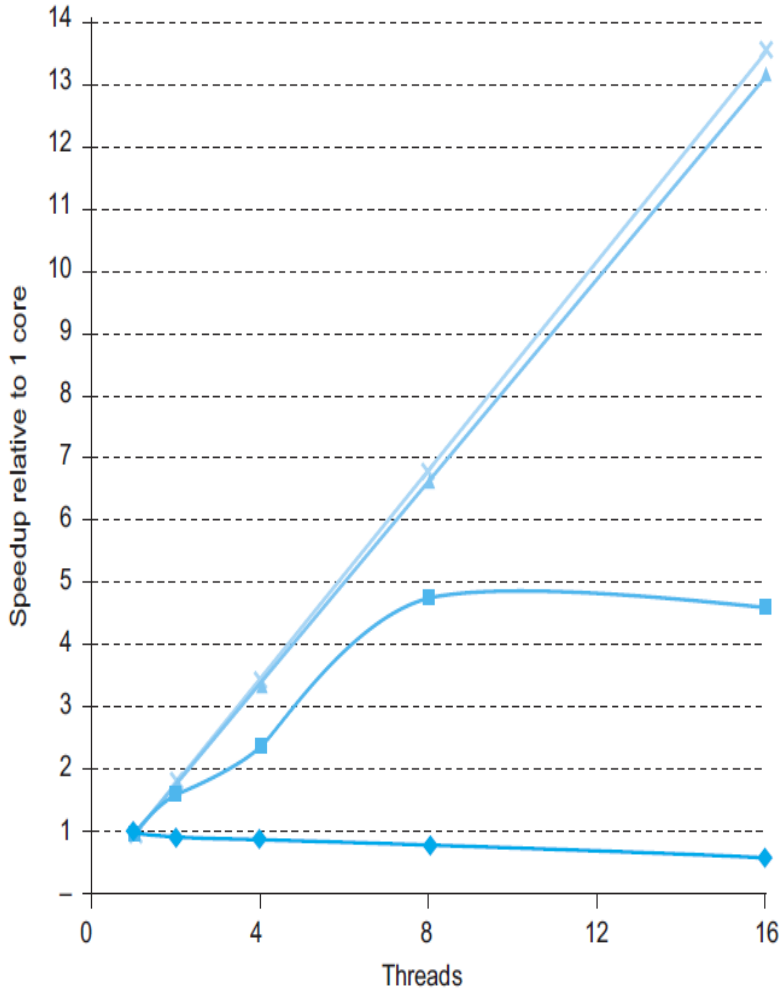
- GPU (480) has 4.4 X the memory bandwidth
  - Benefits memory bound kernels
- GPU has 13.1 X the single precision throughout, 2.5 X the double precision throughput
  - Benefits FP compute bound kernels
- CPU cache prevents some kernels from becoming memory bound when they otherwise would on GPU
- GPUs offer scatter-gather, which assists with kernels with stridden data
- Lack of synchronization and memory consistency support on GPU limits performance for some kernels

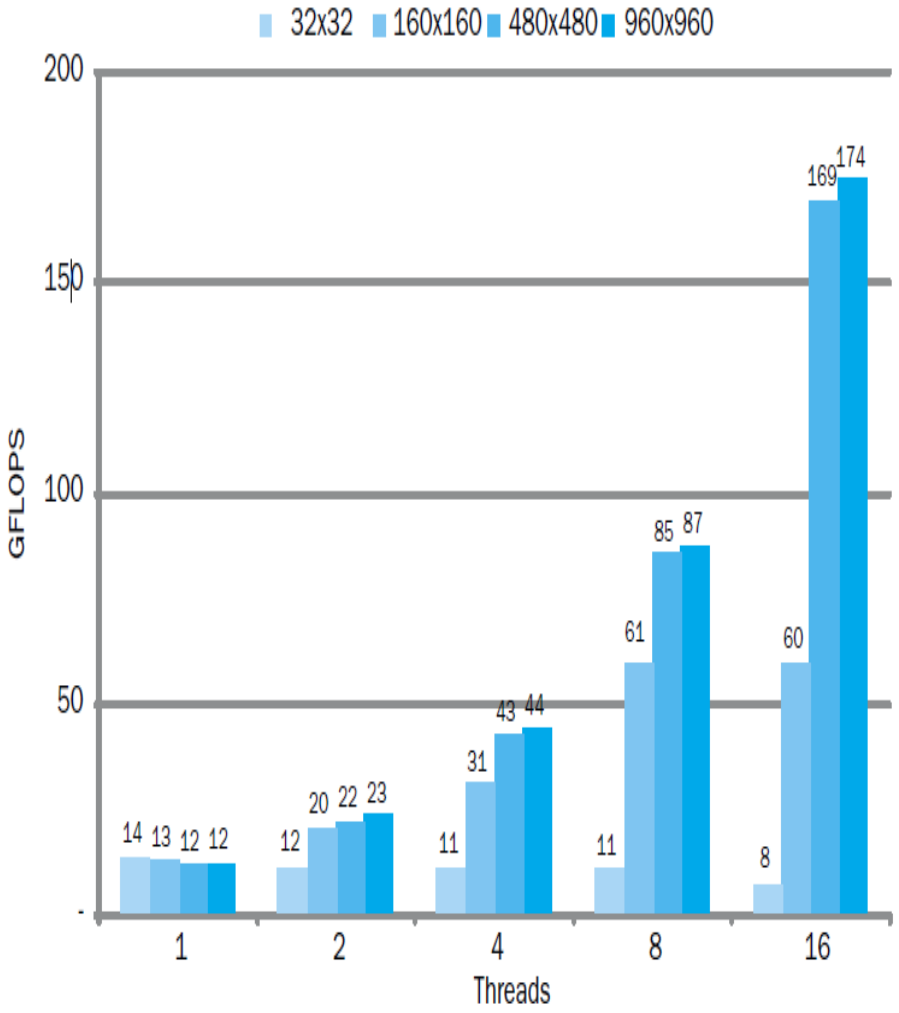
Multi-threading DGEMM

- Use OpenMP:

```
void dgemm (int n, double* A, double* B, double* C)
{
#pragma omp parallel for
 for ( int sj = 0; sj < n; sj += BLOCKSIZE )
  for ( int si = 0; si < n; si += BLOCKSIZE )
   for ( int sk = 0; sk < n; sk += BLOCKSIZE )
    do_block(n, si, sj, sk, A, B, C);
}
```

Multithreaded DGEMM





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

Pitfalls

- Not developing the software to take account of a multiprocessor architecture
  - Example: using a single lock for a shared composite resource
    - Serializes accesses, even if they could be done in parallel
    - Use finer-granularity locking

Concluding Remarks

- Goal: higher performance by using multiple processors
- Difficulties
  - Developing parallel software
  - Devising appropriate architectures
- SaaS importance is growing and clusters are a good match
- Performance per dollar and performance per Joule drive both mobile and WSC

Concluding Remarks (con't)

- SIMD and vector operations match multimedia applications and are easy to program