

Chapter 6

Parallel Processors from Client to Cloud

Introduction

- Goal: connecting multiple computers to get higher performance
 - Multiprocessors
 - Scalability, availability, power efficiency
- Task-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

What We've Already Covered

- §2.11: Parallelism and Instructions
 - Synchronization
- §3.6: Parallelism and Computer Arithmetic
 - Subword Parallelism
- §4.10: Parallelism and Advanced Instruction-Level Parallelism
- §5.10: 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
 - Coordination
 - Communications overhead

Pop Quiz

- Recall Amdahl's Law: $S = 1/((1-p)+p/s)$
- If we have 100 processors, how much must be parallelizable for 90x speedup?
- A: .9
- B: .99
- C: .999
- D: .9999

Amdahl's Law

- Sequential part can limit speedup
- Example: 100 processors, 90× speedup?
 - $T_{\text{new}} = T_{\text{parallelizable}}/100 + T_{\text{sequential}}$
 - $$\text{Speedup} = \frac{1}{(1 - F_{\text{parallelizable}}) + F_{\text{parallelizable}}/100} = 90$$
 - Solving: $F_{\text{parallelizable}} = 0.999$
- 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 = $(10 + 100) \times t_{\text{add}}$
- 10 processors
 - Time = $10 \times t_{\text{add}} + 100/10 \times t_{\text{add}} = 20 \times t_{\text{add}}$
 - Speedup = $110/20 = 5.5$ (55% of potential)
- 100 processors
 - Time = $10 \times t_{\text{add}} + 100/100 \times t_{\text{add}} = 11 \times t_{\text{add}}$
 - 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: $\text{Time} = (10 + 10000) \times t_{\text{add}}$
- 10 processors
 - $\text{Time} = 10 \times t_{\text{add}} + 10000/10 \times t_{\text{add}} = 1010 \times t_{\text{add}}$
 - $\text{Speedup} = 10010/1010 = 9.9$ (99% of potential)
- 100 processors
 - $\text{Time} = 10 \times t_{\text{add}} + 10000/100 \times t_{\text{add}} = 110 \times t_{\text{add}}$
 - $\text{Speedup} = 10010/110 = 91$ (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
 - $\text{Time} = 20 \times t_{\text{add}}$
 - 100 processors, 32×32 matrix
 - $\text{Time} = 10 \times t_{\text{add}} + 1000/100 \times t_{\text{add}} = 20 \times t_{\text{add}}$
 - Constant performance in this example

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
 - Conditional code for different processors

Example: DAXPY ($Y = a \times X + Y$)

■ 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
```

Pop Quiz

- In the DAXPY code, the vector version has improved performance because...
- A: There are no hazards in the vector version
- B: Vector code is always faster
- C: Fewer instructions are fetched and executed
- D: All of the above
- E: None of the above

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

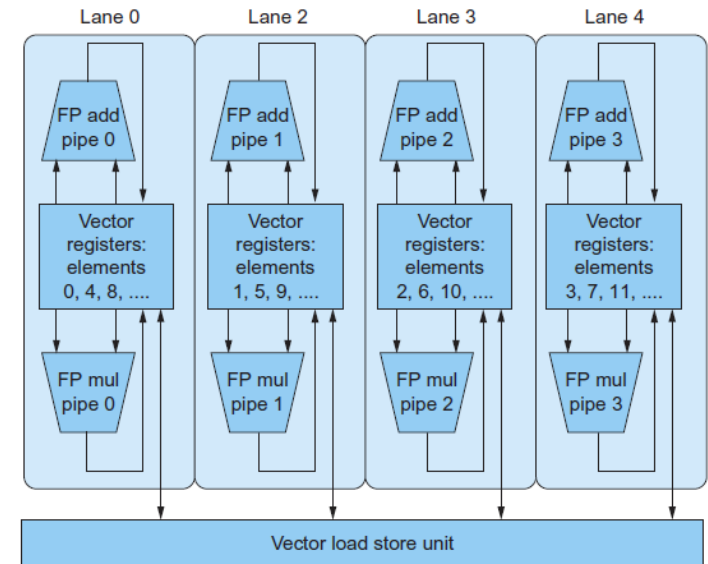
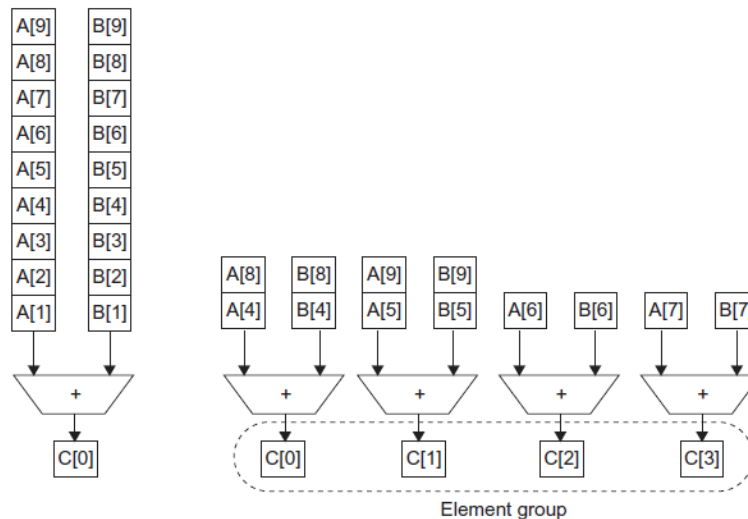
- Vector architectures and compilers
 - Simplify data-parallel programming
 - Explicit statement of absence of loop-carried dependences
 - Reduced checking in hardware
 - Regular access patterns benefit from interleaved and burst memory
 - Avoid control hazards by avoiding loops
- More general than ad-hoc media extensions (such as MMX, SSE)
 - Better match with compiler technology

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 vs. Multimedia Extensions

- Vector instructions have a variable vector width, multimedia extensions have a fixed width
- Vector instructions support strided access, multimedia extensions do not
- Vector units can be combination of pipelined and arrayed functional units:



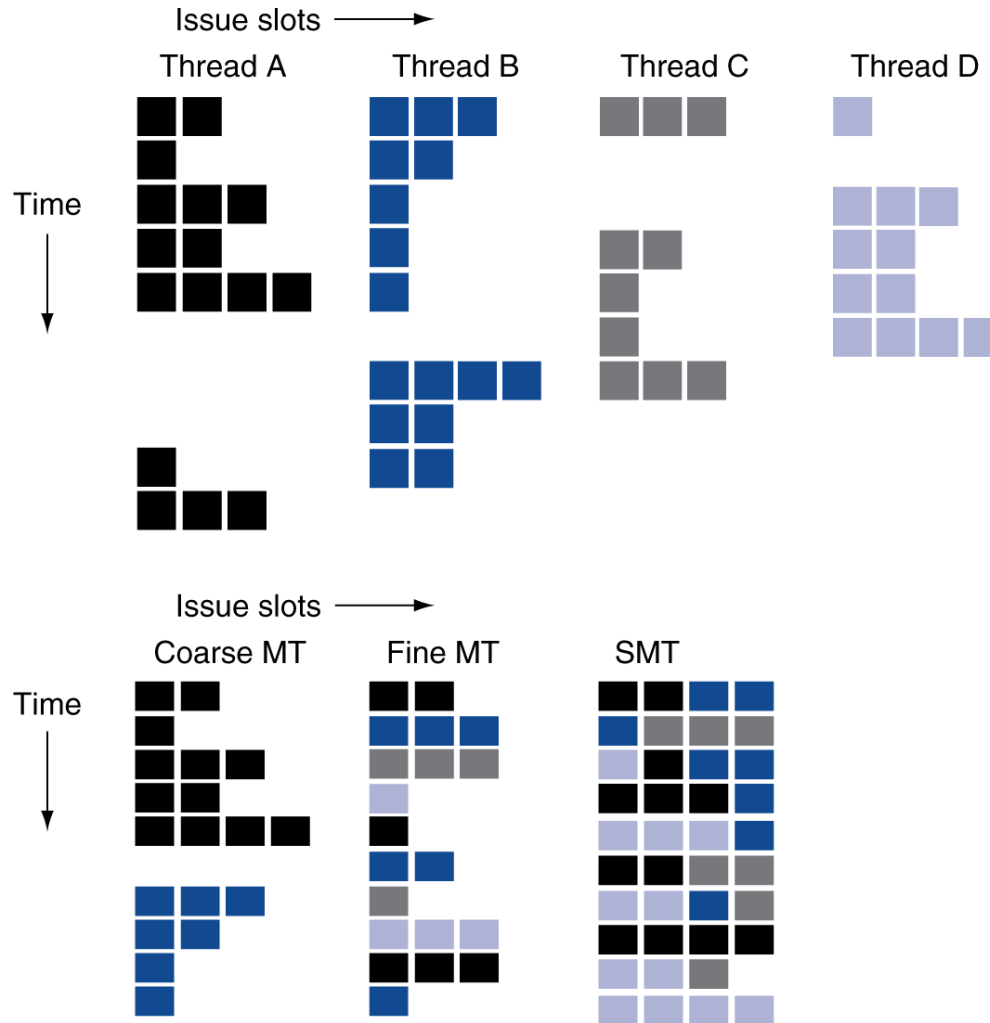
Multithreading

- 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)

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

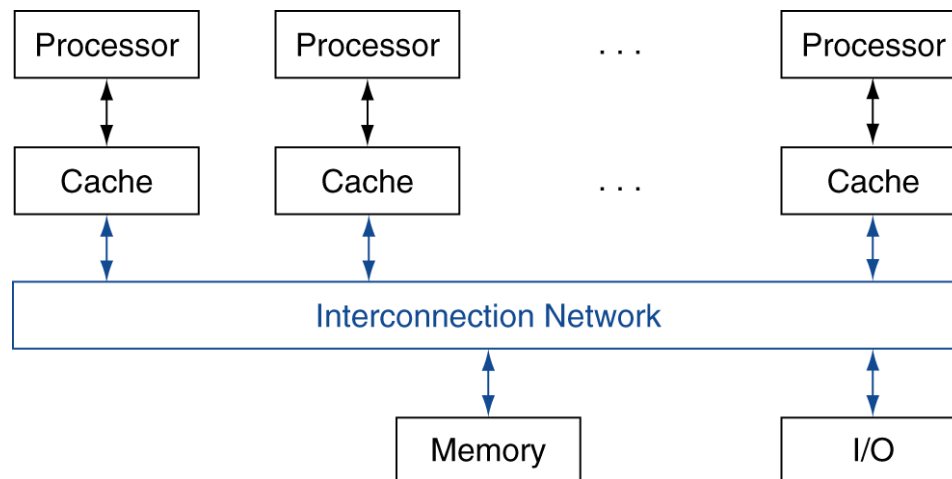


Future of Multithreading

- 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)



Example: Sum Reduction

- Sum 100,000 numbers on 100 processor UMA
 - Each processor has ID: $0 \leq P_n \leq 99$
 - 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];
```

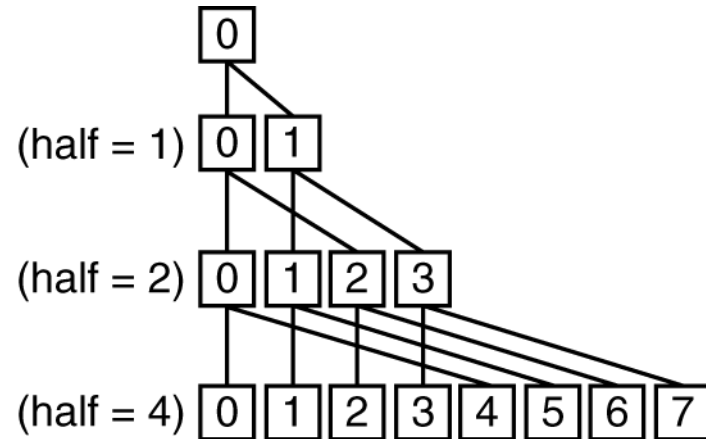
```
        /* Conditional sum needed when half is odd;
```

```
        Processor0 gets missing element */
```

```
    half = half/2; /* dividing line on who sums */
```

```
    if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
```

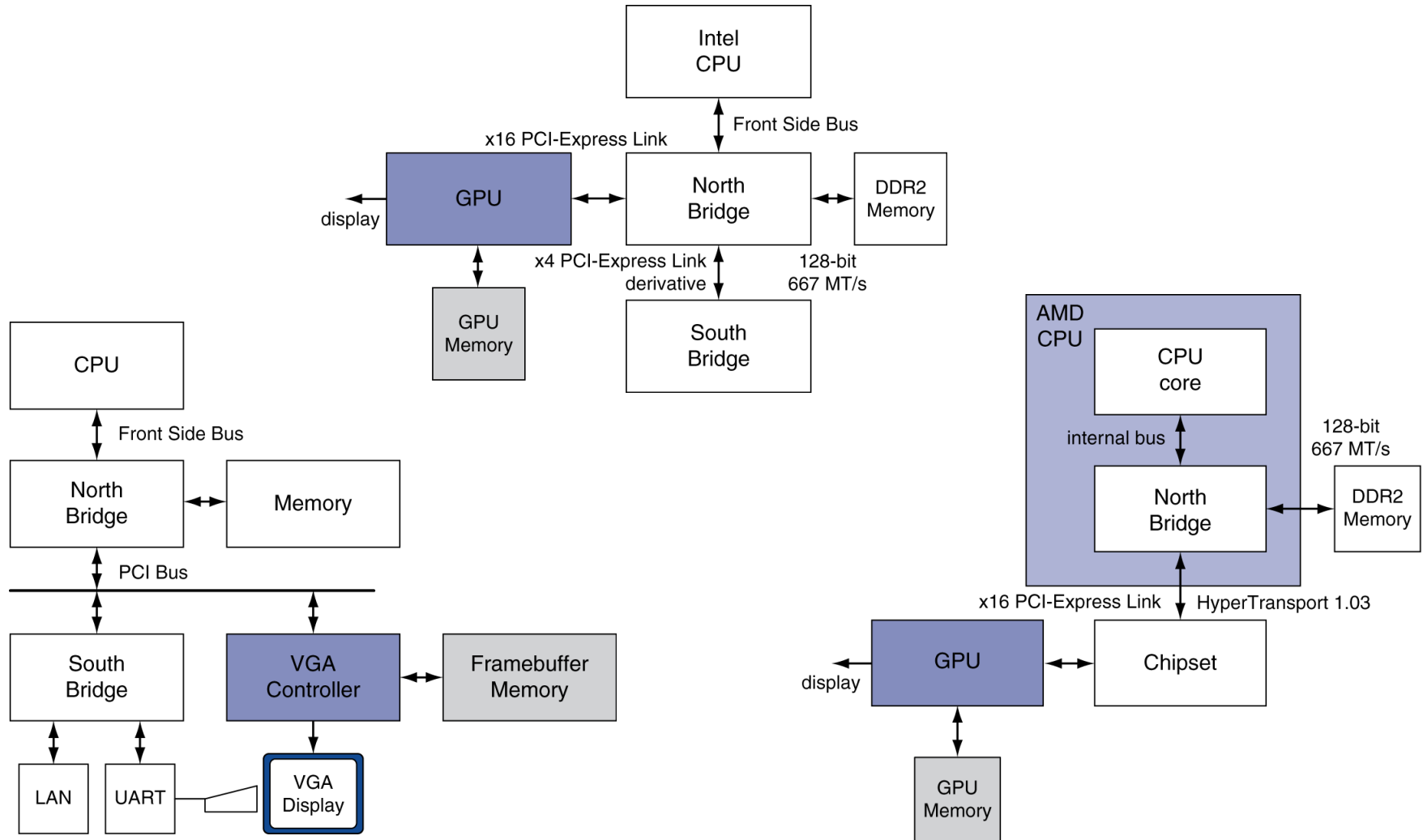
```
until (half == 1);
```



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

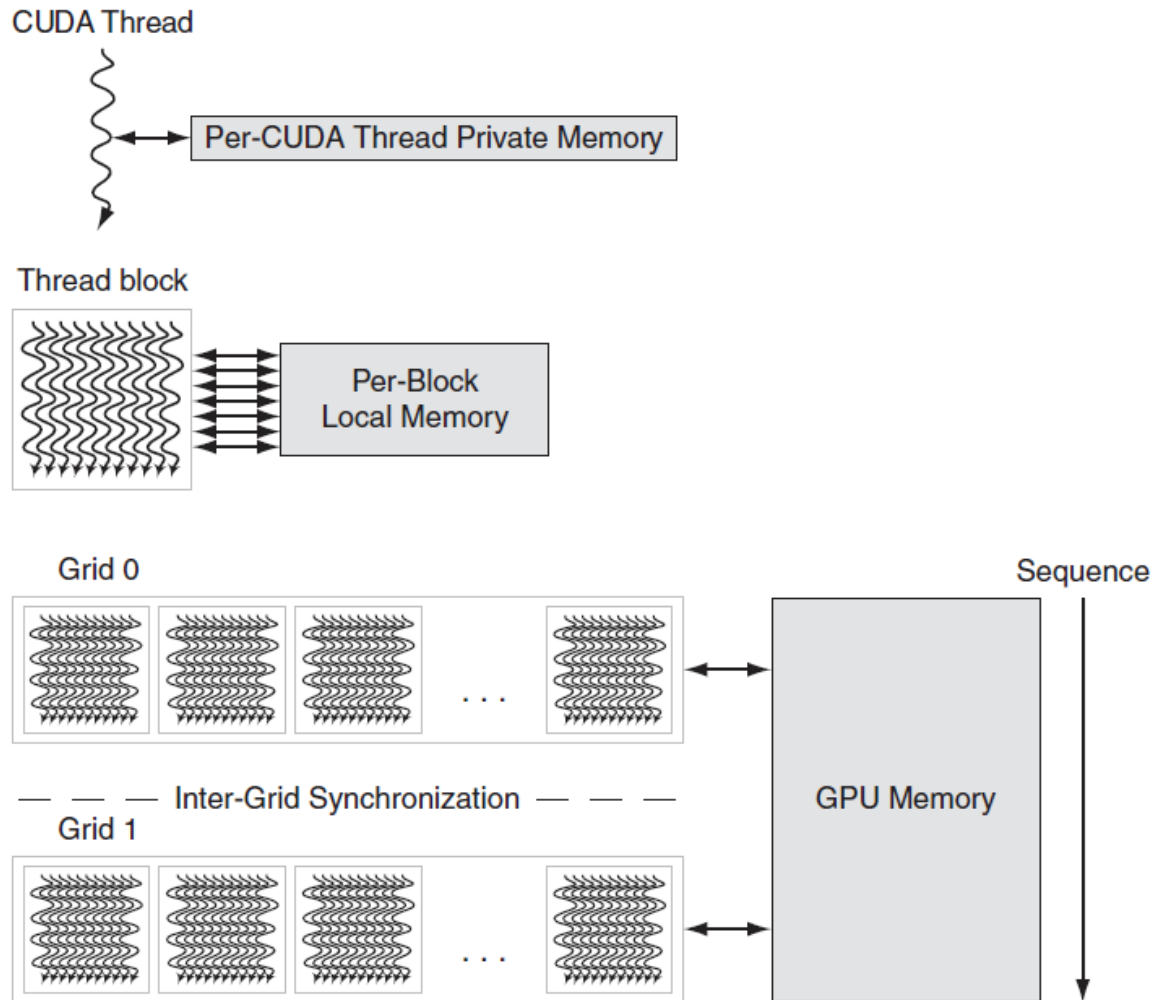
Graphics in the System



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)

GPU Memory Structures



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