

FU05 Computer Architecture

14. Parallel Processors (並列プロセッサ)

Ben Abdallah Abderazek

E-mail: benab@u-aizu.ac.jp

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

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

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: Time = $(10 + 10000) \times t_{\text{add}}$
- 10 processors
 - Time = $10 \times t_{\text{add}} + 10000/10 \times t_{\text{add}} = 1010 \times t_{\text{add}}$
 - Speedup = $10010/1010 = 9.9$ (99% of potential)
- 100 processors
 - Time = $10 \times t_{\text{add}} + 10000/100 \times t_{\text{add}} = 110 \times t_{\text{add}}$
 - 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 = $20 \times t_{\text{add}}$
 - 100 processors, 32×32 matrix
 - 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

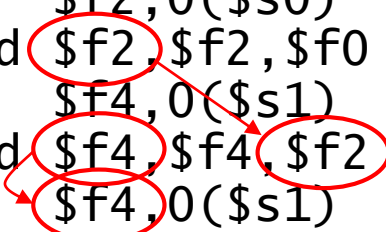
| | | Data Streams | |
|---------------------|----------|-----------------------------------|---|
| | | Single | Multiple |
| Instruction Streams | Single | SISD: Intel Pentium 4 | SIMD: SSE instructions of x86 |
| | Multiple | MISD: No examples today | MIMD: Intel Xeon e5345 |

- 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 x(i)
      l.d    $f4,0($s1)          ;load y(i)
      add.d  $f4,$f4,$f2          ;a x 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
```

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

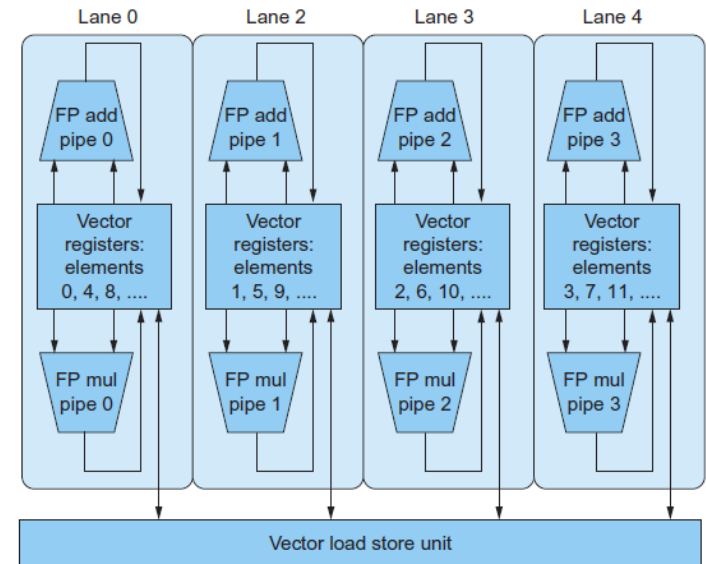
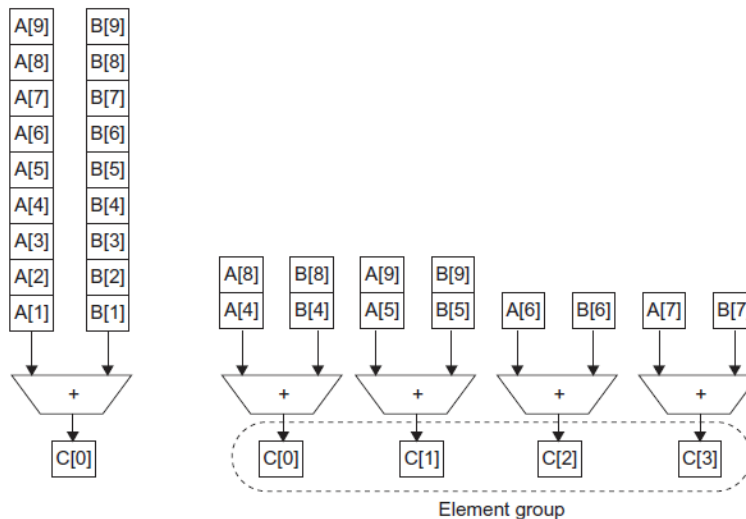
- Vector architectures and compilers
 - Simplify data-parallel programming
 - Explicit statement of absence of loop-carried dependences
 - Reduced checking in hardware
 - Regular access patterns benefit from interleaved and burst memory
 - Avoid control hazards by avoiding loops
- More general than ad-hoc 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:



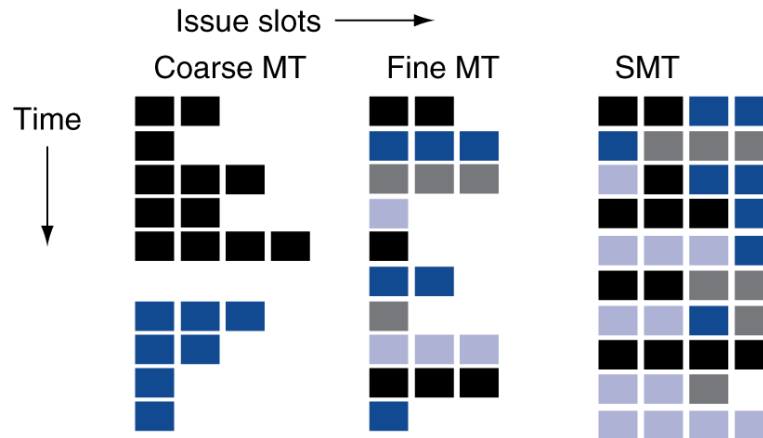
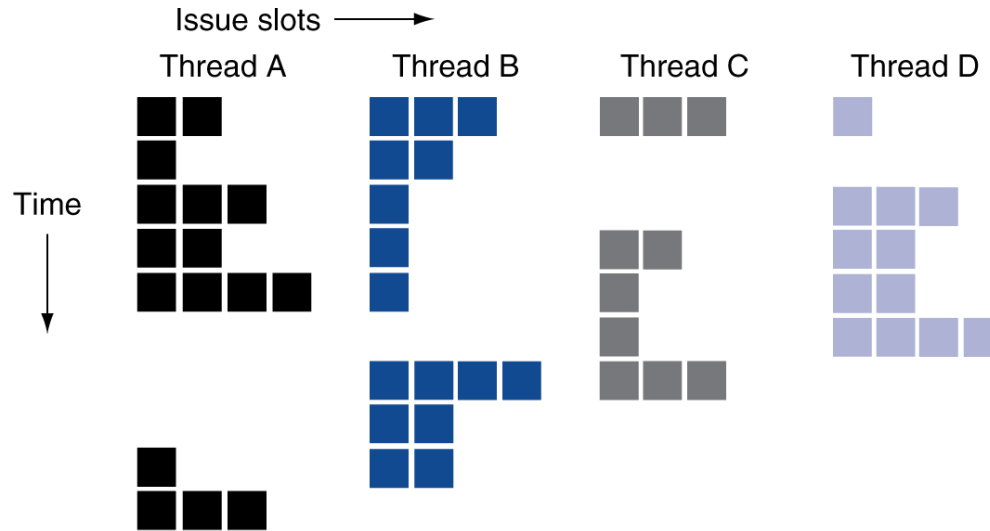
Multithreading

- 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

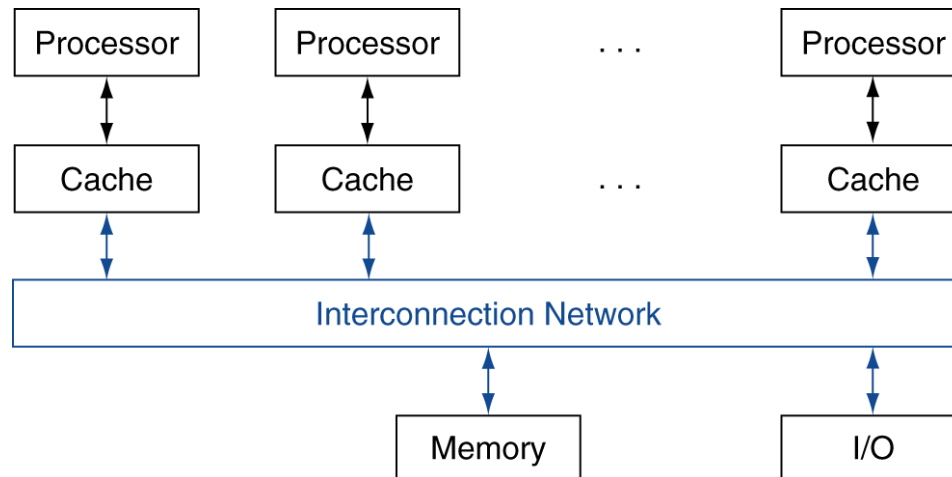


Future of Multithreading

- 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];
```

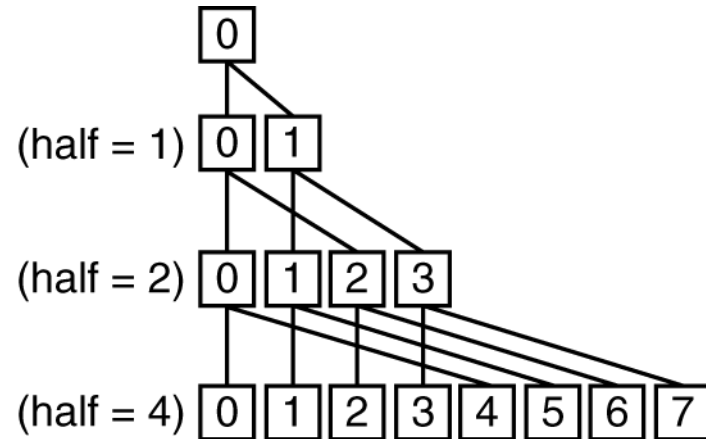
```
        /* Conditional sum needed when half is odd;
```

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

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

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

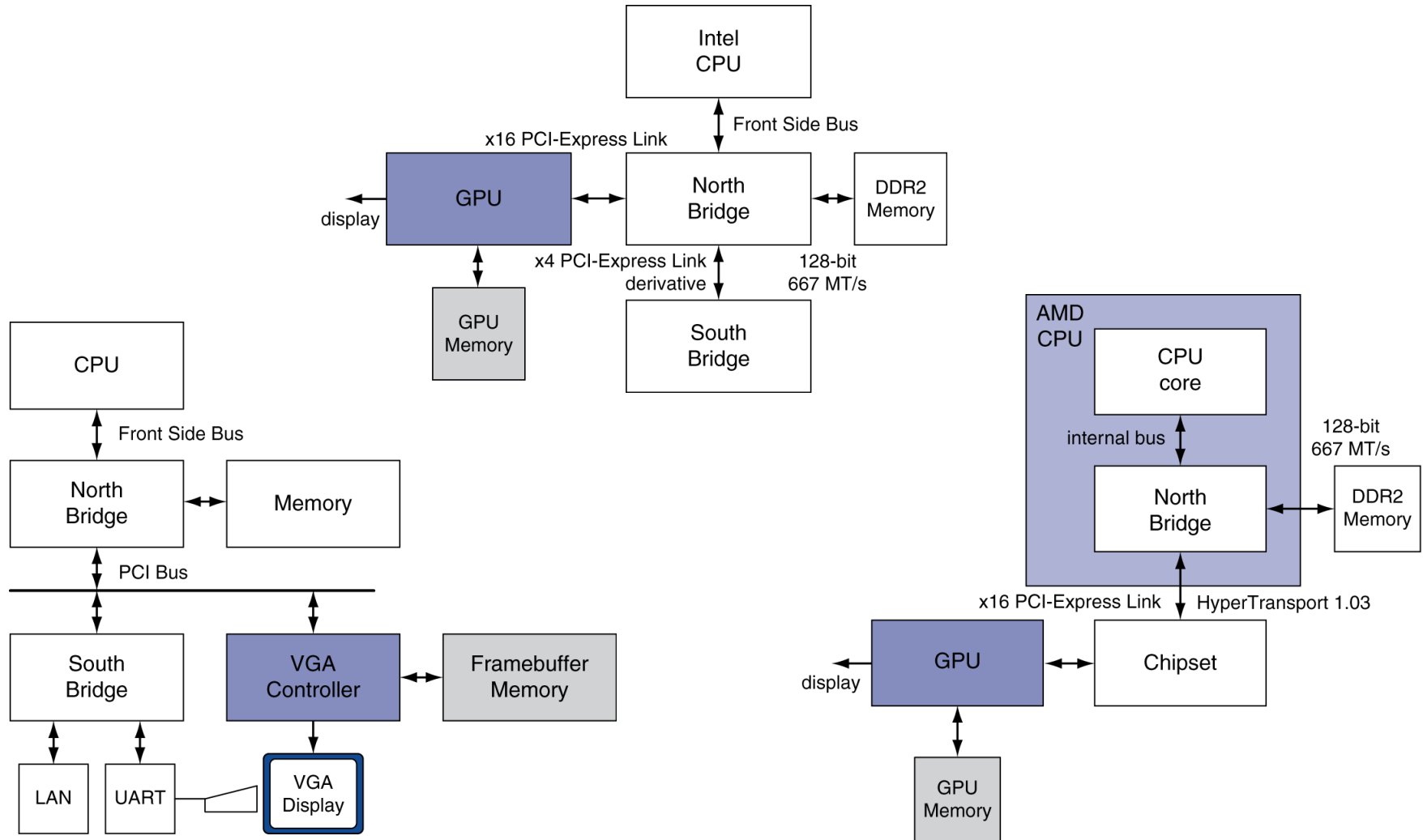
```
until (half == 1);
```



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

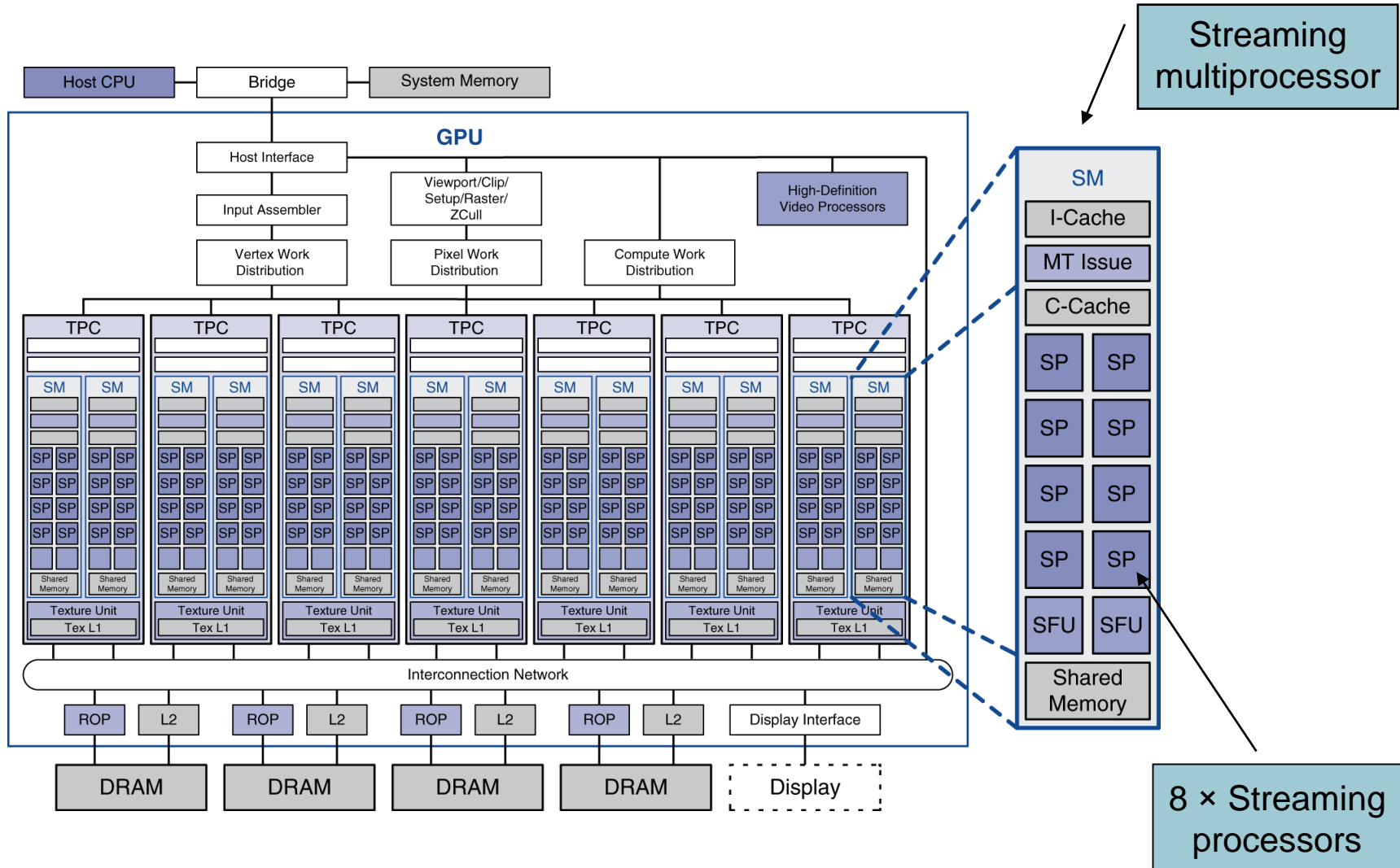
Graphics in the System



GPU Architectures

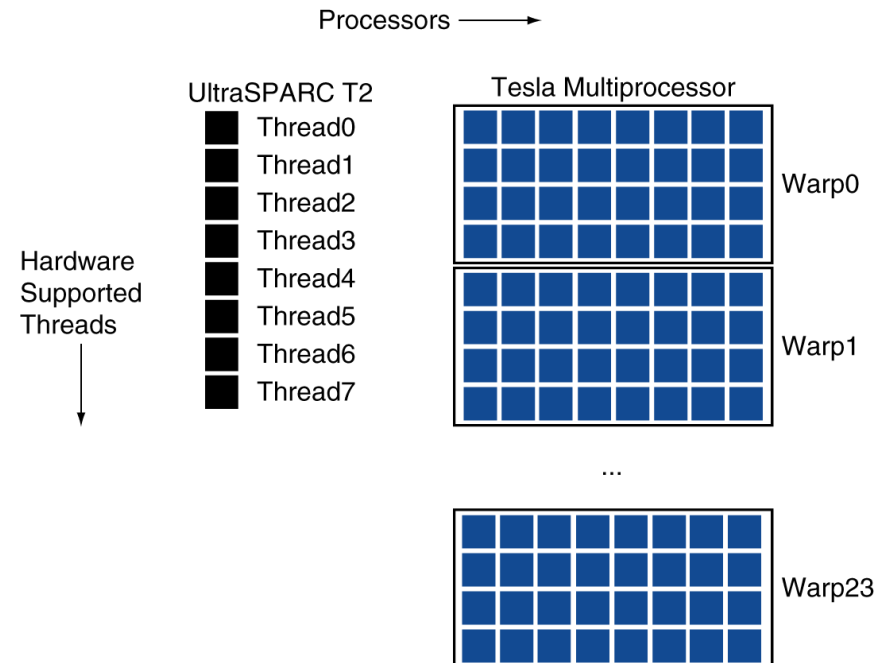
- Processing is highly data-parallel
 - GPUs are highly multithreaded
 - Use thread switching to hide memory latency
 - Less reliance on multi-level caches
 - Graphics memory is wide and high-bandwidth
- Trend toward general purpose GPUs
 - Heterogeneous CPU/GPU systems
 - CPU for sequential code, GPU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

Example: NVIDIA Tesla



Example: NVIDIA Tesla

- Streaming Processors
 - Single-precision FP and integer units
 - Each SP is fine-grained multithreaded
- Warp: group of 32 threads
 - Executed in parallel, SIMD style
 - 8 SPs
× 4 clock cycles
 - Hardware contexts for 24 warps
 - Registers, PCs, ...

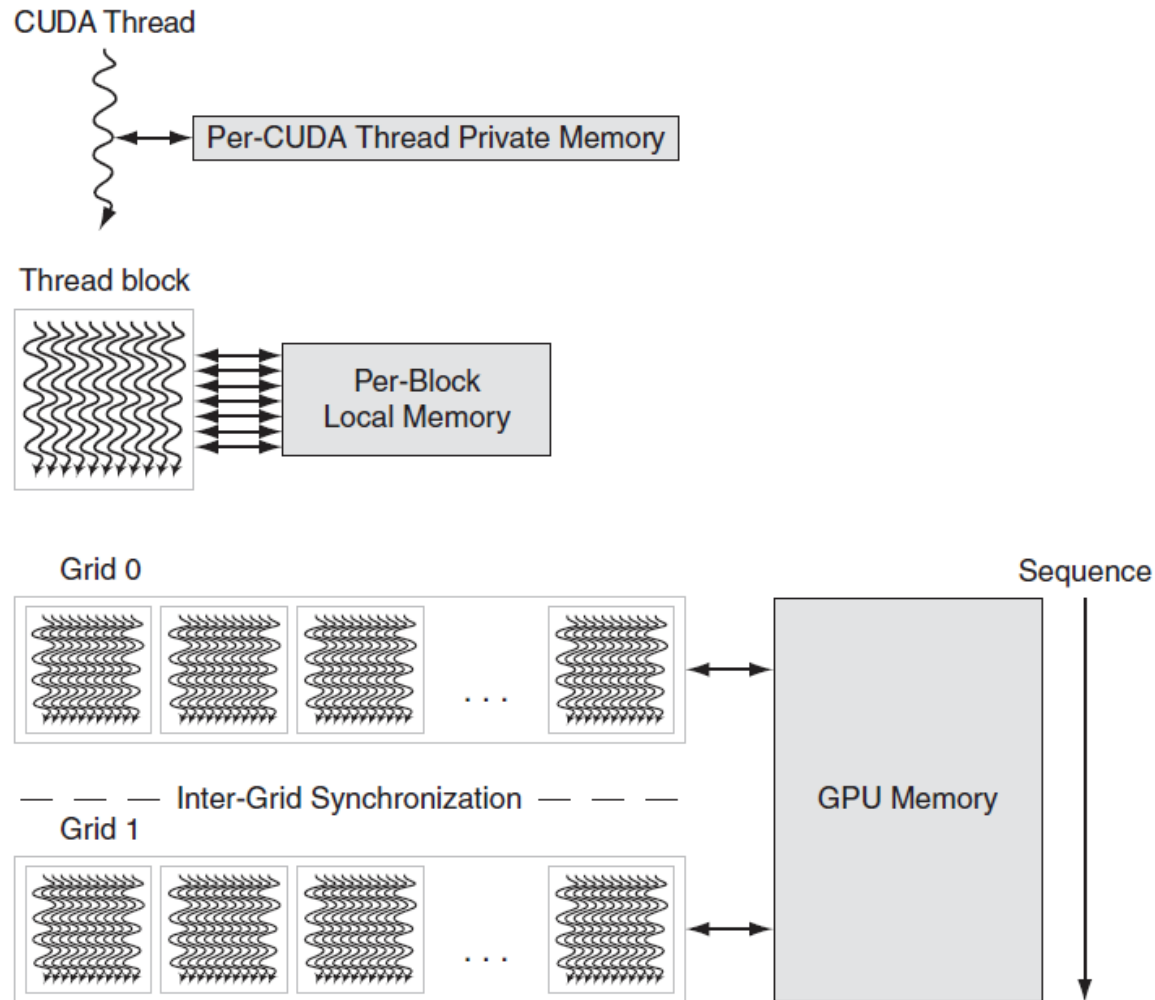


Classifying GPUs

- Don't fit nicely into SIMD/MIMD model
 - Conditional execution in a thread allows an illusion of MIMD
 - But with performance degradation
 - 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 |

GPU Memory Structures



Putting GPUs into Perspective

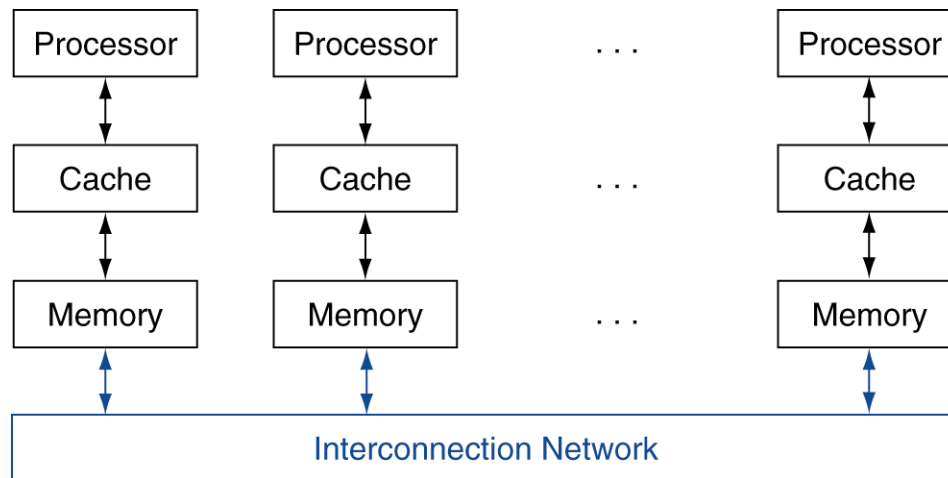
| Feature | Multicore with SIMD | GPU |
|---|---------------------|--------------|
| SIMD processors | 4 to 8 | 8 to 16 |
| SIMD lanes/processor | 2 to 4 | 8 to 16 |
| Multithreading hardware support for SIMD threads | 2 to 4 | 16 to 32 |
| Typical ratio of single precision to double-precision performance | 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 |

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. |
| 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



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
 - E.g., SETI@home, World Community Grid

Interconnection Networks

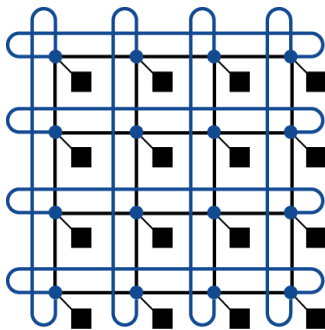
- Network topologies
 - Arrangements of processors, switches, and links



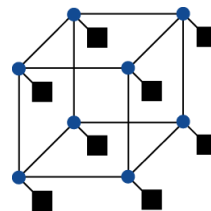
Bus



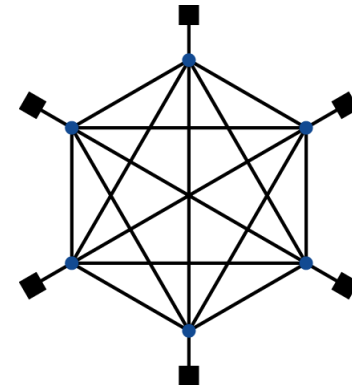
Ring



2D Mesh

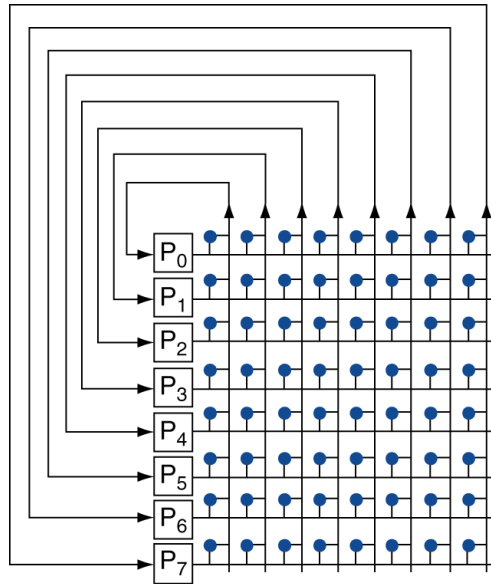


N-cube ($N = 3$)

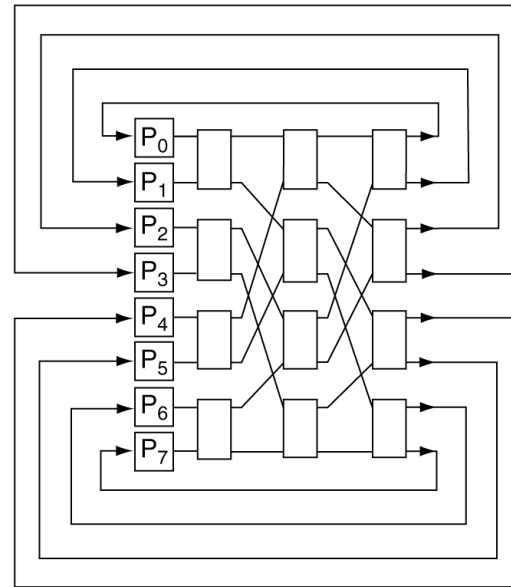


Fully connected

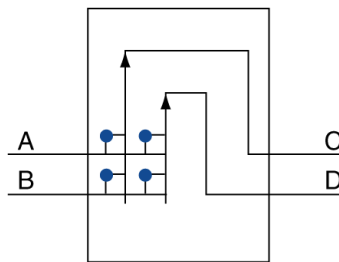
Multistage Networks



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)
- 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

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