CPE 431/531

Chapter 6 – Parallel Processors from Client to Cloud

Dr. Rhonda Kay Gaede

THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

6.1 Motivation

- Why multiprocessors?
  - __performance__
  - __power efficiency__
  - __fault tolerance__

6.2 The Difficulty of Creating Parallel Programs

- The difficulty with parallelism is not the _hardware_ , it's the _software_.
- Why is it difficult to write parallel processing programs that are fast?
  - _scheduling_
  - _load_ _balancing_
  - _synchronization_ _time_
  - _communication_ _overhead_
  - _Amdahl's_ _law_

6.2 Speedup Challenge

- Suppose you want to achieve a speedup of 90 times faster with 100 processors. What percentage of the original computation can be sequential?

$$\frac{ET\ affected}{Amount\ of\ improvement} + \text{ET unaffected} = \text{ET after} \qquad \text{Speedup} = \frac{ET\ before}{ET\ after}$$

$$90 = \frac{ET\ b}{(ET\ b - ET\ aff) + ET\ aff/100}\ \frac{\frac{1}{ETb}}{\frac{1}{ETb}} \qquad \text{fract} = \frac{ET\ aff}{ET\ b}$$

$$90 = \frac{1}{1 - fract + fract/100}$$

90 – 90 fract + 0.9 fract = 1
89 = 89.1 fract
Fract = 0.999

Sequential = 1 – fract = 0.001 = 0.1%

6.2 Speedup Challenge – Bigger Problem

- Suppose you want to perform two sums: one is a sum of 10 scalar variables and one is a matrix sum of a pair of two-dimensional arrays, size 10 by 10. What speedup do you get with 10 versus 40 processors?

Scalar 10t
Matrix 100 r
ET before = 110 t

10 processors
ET after = 10t + 110t/10 = 20t
Speedup = 110t/20t = 5.5
Efficiency = 5.5/10 = 55%

40 processors
ET aff = 10t + 100t/40 = 12.5t
Speedup = 110t/12.5t = 8.8
Efficiency = 8.8/40 = 22%

6.2 Speedup Challenge – Bigger Problem

- Next, calculate the speed-ups assuming the matrices grow to 20 by 20

How many additions? 400t Still doing scalar sum of 10
10 processors
    Original Time = 410t
    Parallel Time = 10t + 400t/10 = 50t
    Speedup = 410t/50t = 8.2
40 processors
    Parallel Time = 10t + 400t/40 = 20t
    Speedup = 410t/20t = 20.5

- Strong scaling – measure speedup with fixed problem
- Weak scaling – program size grows proportionally to the number of processors

6.2 Speedup Challenge: Balancing Load

- To achieve the speed-up of 20.5 on the previous larger problem with 40 processors, we assumed the load was perfectly balanced (each processor did 2.5 % of the work). Instead, show the impact on speed-up if one processor's load is higher than all the rest. Calculate at 5% and 12.5%.

5% load
P5% = 20t
39 processors share 380t
Max (20t, 380t/39) + 10t = 30t
Speedup = 410t/30t = 13.67

12.5% load
P12.5% = 50t
39 processors share 350t
Max (50t, 350t/39) + 10t = 60t
Speedup = 410t/60t = 6.83

6.3 SISD, MIMD, SIMD, SPMD, and Vector

- SISD is the normal case – single instruction, single data.
- MIMD – multiple instruction, multiple data - is _theoretically_ possible but programmers normally write a _single_ _program_ that runs on all processors relying on _conditional_ statements when _different_ processors should execute _different_ sections of code. This style is single _program_, multiple data.
- SIMD – single instruction, multiple data – operate on _vectors_ of data. SIMD needs only _one_ _copy_ of the code that is being simultaneously executed. SIMD works best when dealing with _arrays_ in _for_ loops.
- The _array_ _processors_ that inspired the SIMD category faded into history but two _current_ interpretations of SIMD remain _active_ today.
  - _SIMD in x86 Multimedia Extensions_
  - _Vector_

6.3 x86 Multimedia Extensions

- The most _widely_ used variation of SIMD is the basis of the hundreds of MMX and SSE instructions of the x86 processor. These instructions were added to improve performance of _multimedia_ programs.
  - The hardware allows flexible ALU operations – one 64-bit or two 32-bit or four 16-bit or eight 8-bit
  - Loads and stores are simply as wide as the widest ALU.
  - SSE now supports simultaneous execution of a pair of 64-bit floating-point numbers

6.3 Vector

- An older and more elegant interpretation of SIMD is called a vector architecture, which has been closely identified with Cray Computers.
- Consider Y = a × X + Y

Original

```
      l.d    $f0,a($sp)
      addiu  $t1,$s0,#512
loop: l.d    $f2,0($s0)
      mul.d  $f2,$f2,$f0
      l.d    $f4,0($s1)
      add.d  $f4,$f4,$f2
      s.d    $f4,0($s1)
      addiu  $s0,$s1,#8
      addiu  $s1,$s1,#8
      subu   $t0,$t1,$s0
      bne    $t0,$zero,loop
```

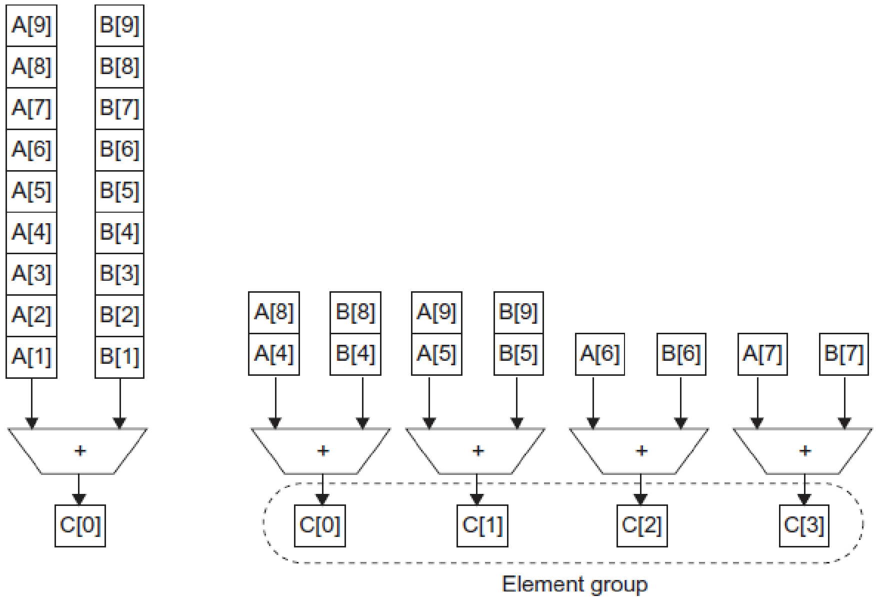
Vector

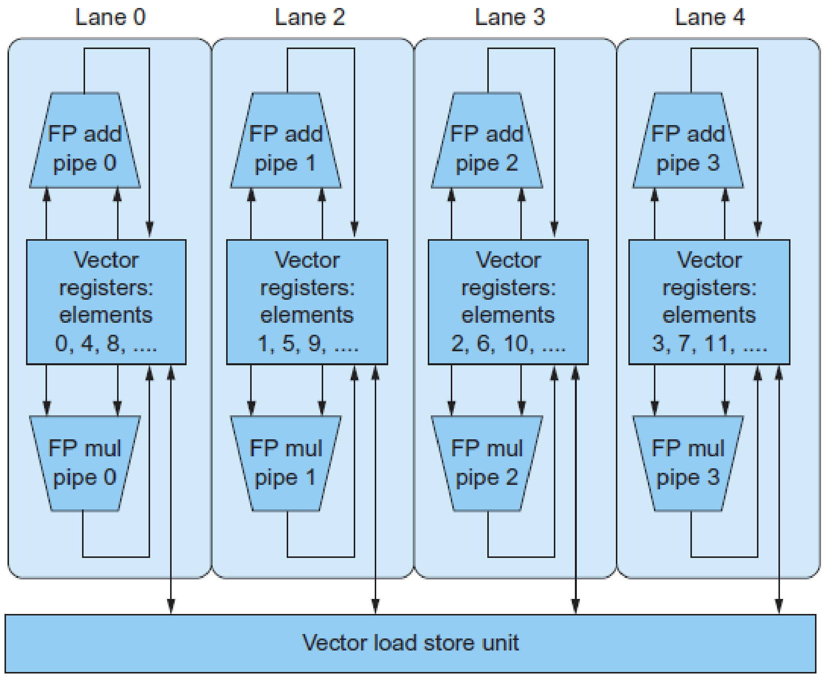
```
l.d      $f0, a($sp)
lv       $v1,0($s0)
mulvs.d  $v2,$v1,$f0
lv       $v3,0($s1)
addv.d   $v4,$v4,$v3
sv       $v4,0($s1)
```

6.3 Comparisons

- Vector versus Scalar
  - A _single_ _vector_ instruction specifies a great deal of work, the instruction _fetch_ and _decode_ bandwidth is greatly reduced.
  - Hardware does not have to check for _data_ _hazards_ _within_ a vector instruction.
  - Vector architectures and compilers have worked well for _data-level_ _parallelism_.
  - Hardware need only check for _data_ hazards _once_ between two vector _instructions_, not _once_ for every _vector_ _element_.
  - Vector instructions that access memory have a _known_ _access_ _pattern_, memory system can be adjusted accordingly.
  - Replacing a _loop_ with a _vector_ _instruction_ reduces _control_ hazards.
- Vector versus Multimedia Extensions
  - Vector specifies the number of operands in _registers_, not in _opcodes_.
  - Vector data transfers need _not_ be _contiguous_.
  - Vector _evolves_ over time more _gracefully_.

6.3 Improving the Performance of Vector



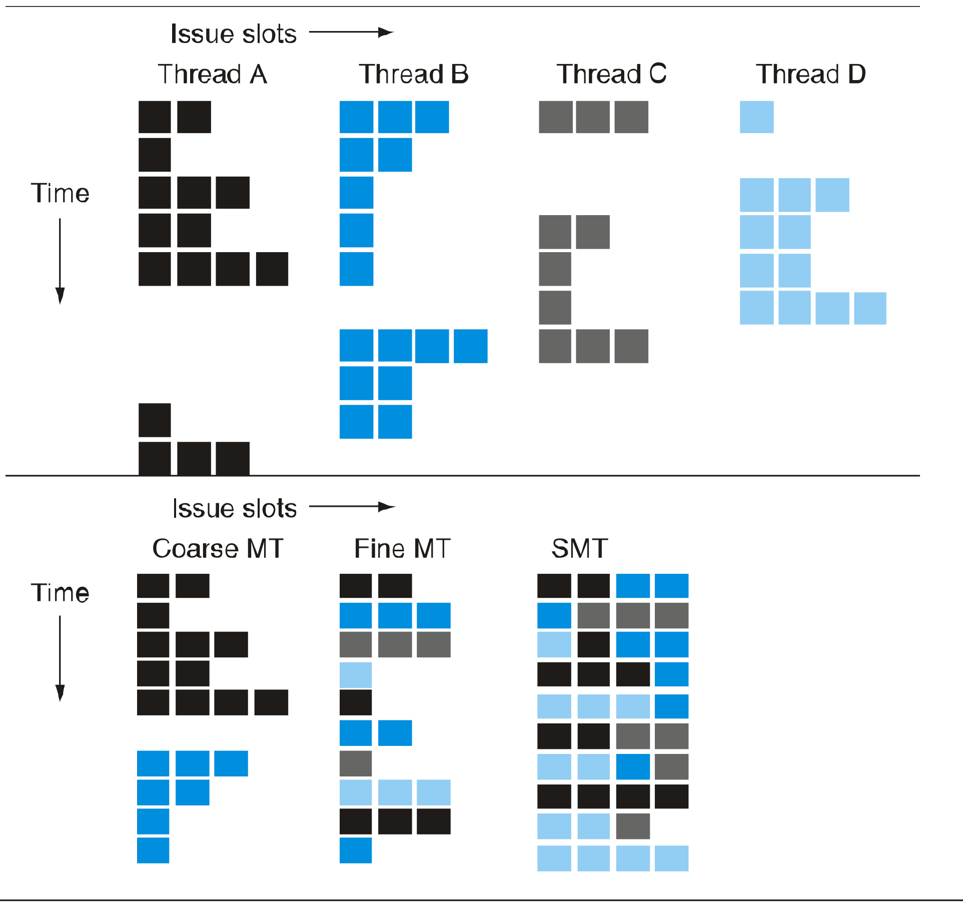

Lane 0
Lane 2
Lane 3
Lane 4
FP add pipe 0
FP add pipe 1
FP add pipe 2
FP add pipe 3
Vector registers: elements 0, 4, 8, ...
Vector registers: elements 1, 5, 9, ...
Vector registers: elements 2, 6, 10, ...
Vector registers: elements 3, 7, 11, ...
FP mul pipe 0
FP mul pipe 1
FP mul pipe 2
FP mul pipe 3
Vector load store unit

6.4 Hardware Multithreading

- Hardware multithreading allows _multiple_ _threads_ to _share_ the _functional_ units of a single processor in _overlapping_ fashion.
- The processor must _duplicate_ the independent _state_ of each _thread_.
- The hardware must support the _ability_ to _change_ to different _thread_ relatively quickly.
- _Fine-grained_ multithreading switches between _threads_ on each _instruction_, often done round robin.
  - Hides _throughput_ losses by doing useful work during _thread_ _stalls_.
  - Inserts _latency_ for threads with _no_ _stalls_.
- _Coarse-grained_ multithreading _switches_ threads only on _costly_ stalls, such as _second-level_ _cache_ misses.
  - It is limited in its ability to overcome _throughput_ losses, especially from _shorter_ stalls. The major problem is pipeline _fill_ _time_.

6.4 Simultaneous Multithreading

- Simultaneous multithreading uses the resources of a _multiple-issue_, _dynamically_ scheduled processor to exploit _thread-level_ parallelism at the same time it exploits _instruction-level_ parallelism.
- The key insight is that multiple-issue processors often have more _functional_ _unit_ _parallelism_ than a _single_ thread can effectively use.
- _Resolution_ of dependences can be handled by the _dynamic_ _scheduling_ capability.



Utilization 19/40 20/40 39/40

6.4 Multithreading Speedup

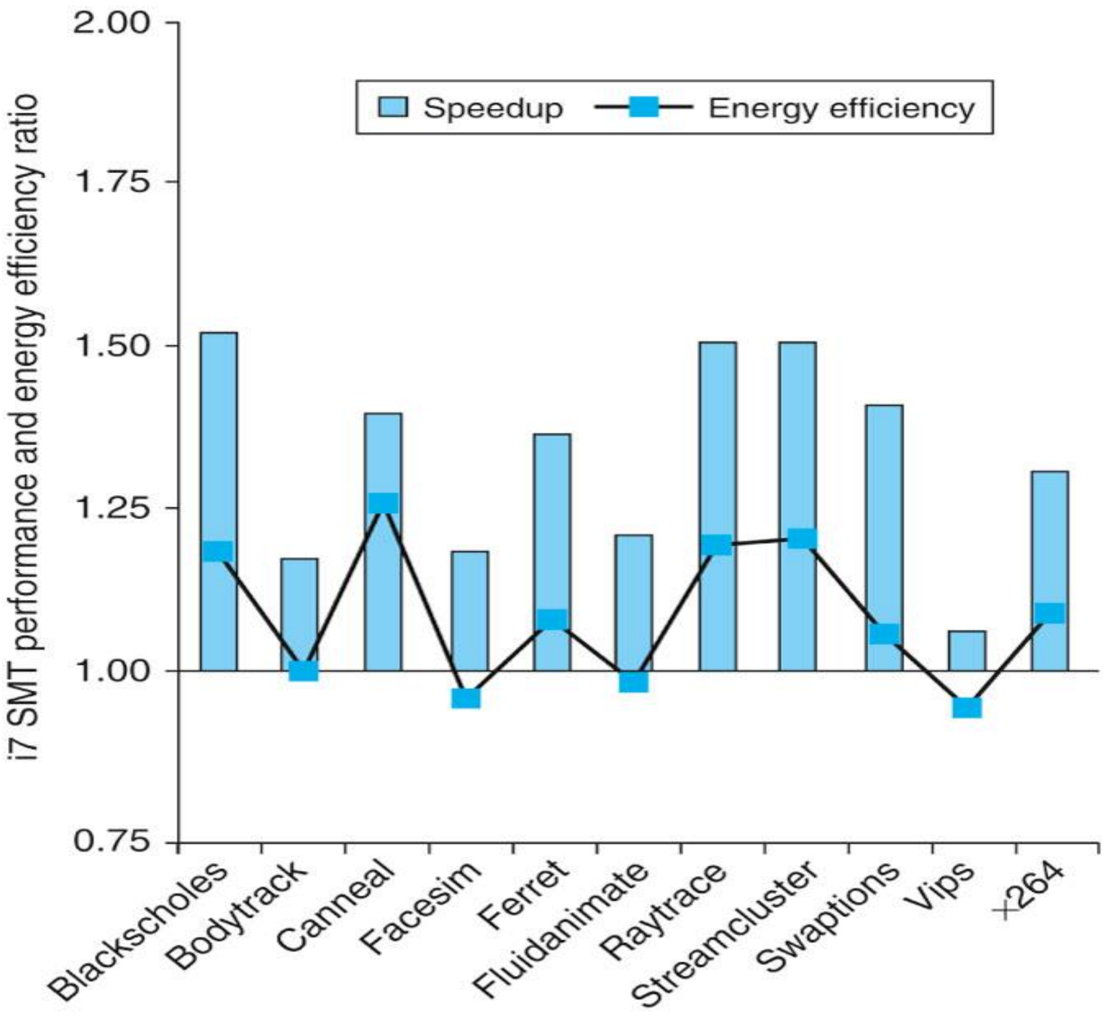
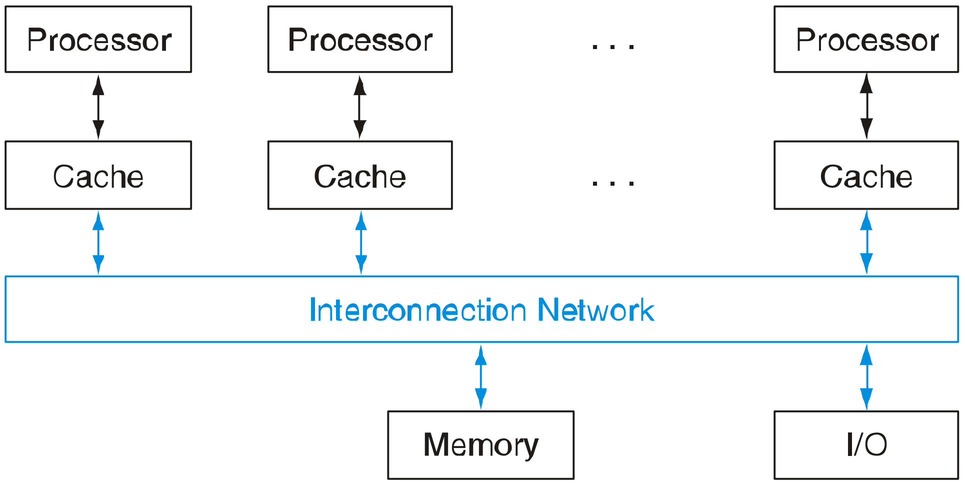


FIGURE 6.6 The speed-up from using multithreading on one core on an i7 processor averages 1.31 for the PARSEC benchmarks (see Section 6.9) and the energy efficiency improvement is 1.07. This data was collected and analyzed by Esmaeilzadeh et. al. [2011].

6.5 Multicore and Other Shared Memory Multiprocessors

- A shared memory multiprocessor (SMP) is one that offers the programmer a _single_ _physical_ _address_ _space_ across all processors
- Processor communicate through _shared_ _variables_ in memory.
- SMPs come in two flavors
  - _UMA_
  - _NUMA_

- Processors need to coordinate when sharing data, this process is called _synchronization_, processors must acquire a _lock_

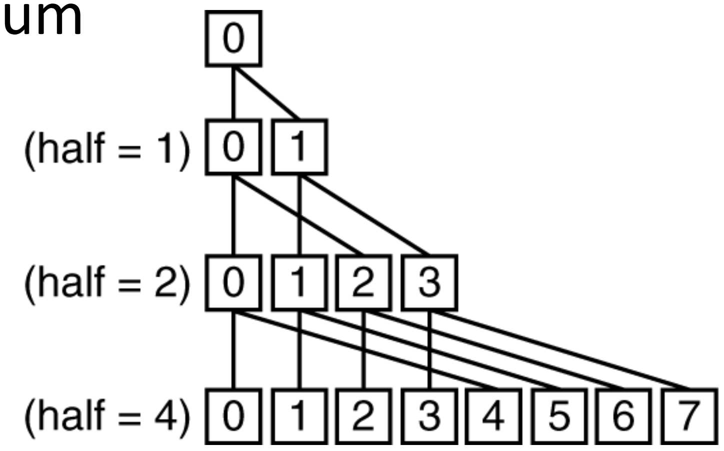


6.5 Shared Address Space Parallel Program (1)

- Suppose we want to sum 64,000 numbers on an SMP with UMA. Let’s assume we have 64 processors.

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

- After execution of this code, there are 64 partial sums
- Need to combine them into single sum
- Do so using a reduction



6.5 Shared Address Space Parallel Program (2)

```
half = 64; /* 64 processors in multiprocessor */
repeat
  synch(); /* wait for partial sum completion */
  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); /* exit with final sum in sum[0] */
```

6.5 A Parallel Programming System

- A limited but popular example is OpenMP
  - OpenMP is an Application Programmer Interface along with a set of compiler directives, environment variables, and runtime library routines.
  - It offers a portable, scalable,and simple programming model for shared memory multiprocessors.
  - Its primary goal is to parallelize loops and to perform reductions.
  - OpenMP extends C using pragmas, commands to the C macro processor

6.5 OpenMP Example

```
Cc –fopenmp foo.c
#define  P 64 /* define a constant */
#pragma omp parallel num_threads(P)

#pragma omp parallel for
for (Pn = 0; Pn < P; Pn +=1)
  for (1000*Pn; i < 1000*(Pn +1); i +=1)
    sum[Pn] += A[i]; /* sum the assigned areas */

#pragma omp parallel for reduction(+ : FinalSum)
for (i = 0; i < P; i += 1)
  FinalSum += sum[i]; /* Reduce to a single number */
```

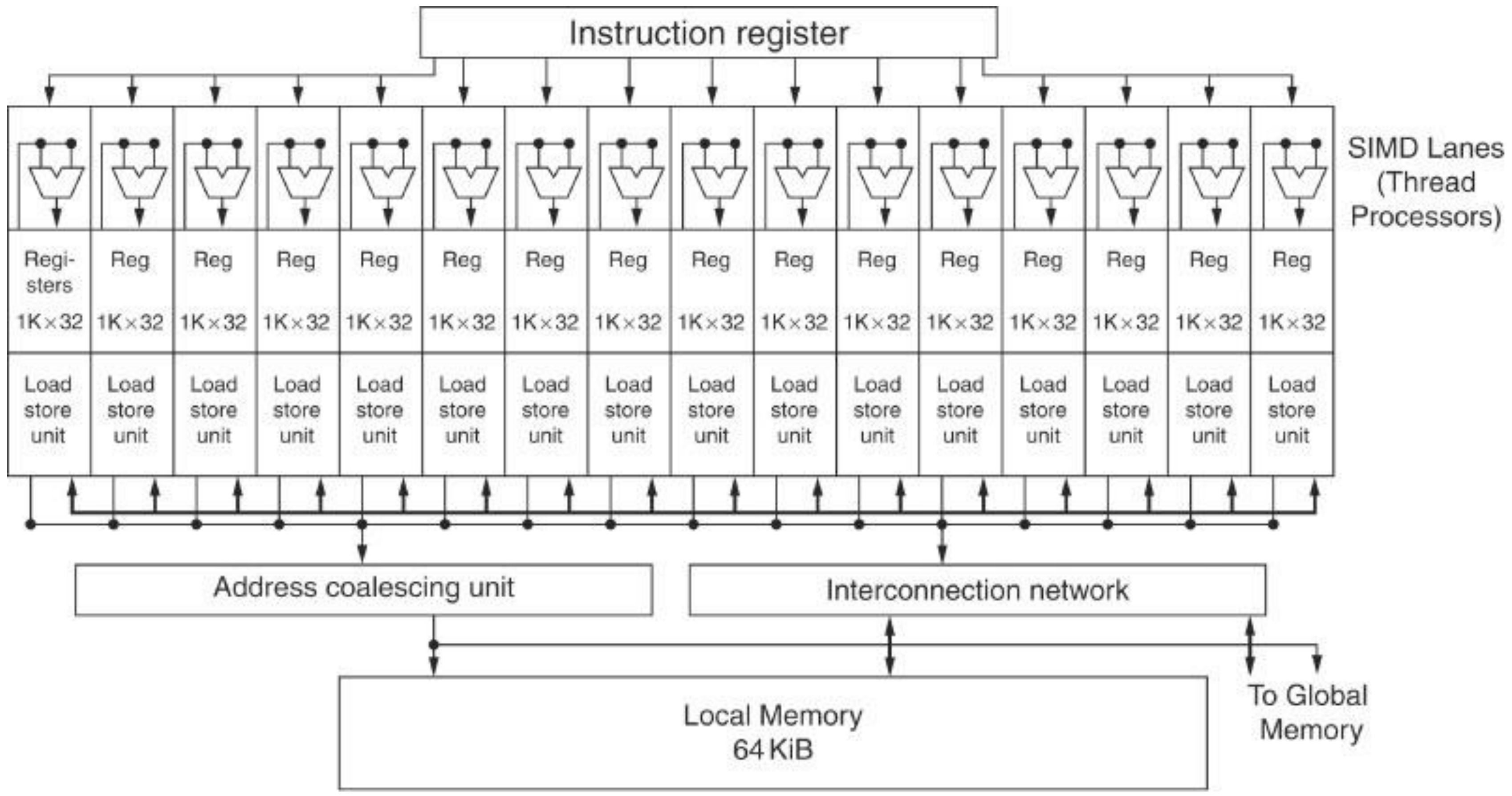
6.6 Introduction to Graphics Processing Units

- A major driving force for improving graphics processing was the gaming industry, a different development community than the one for CPUs.
- Key differences between GPUs and CPUs
  - GPUs are accelerators that supplement a CPU, they don’t have to do everything.
  - GPU problems sizes are typically hundreds of megabytes to gigabytes, but not hundreds of gigabytes to terabytes.
- Different Architecture Features
  - GPUs do not rely on multilevel caches, they rely on having enough threads to hide memory latency.
  - The GPU main memory is oriented towards bandwidth rather than latency.
  - Each GPU processor is more highly multithreaded than a typical CPU, plus they have more processors.

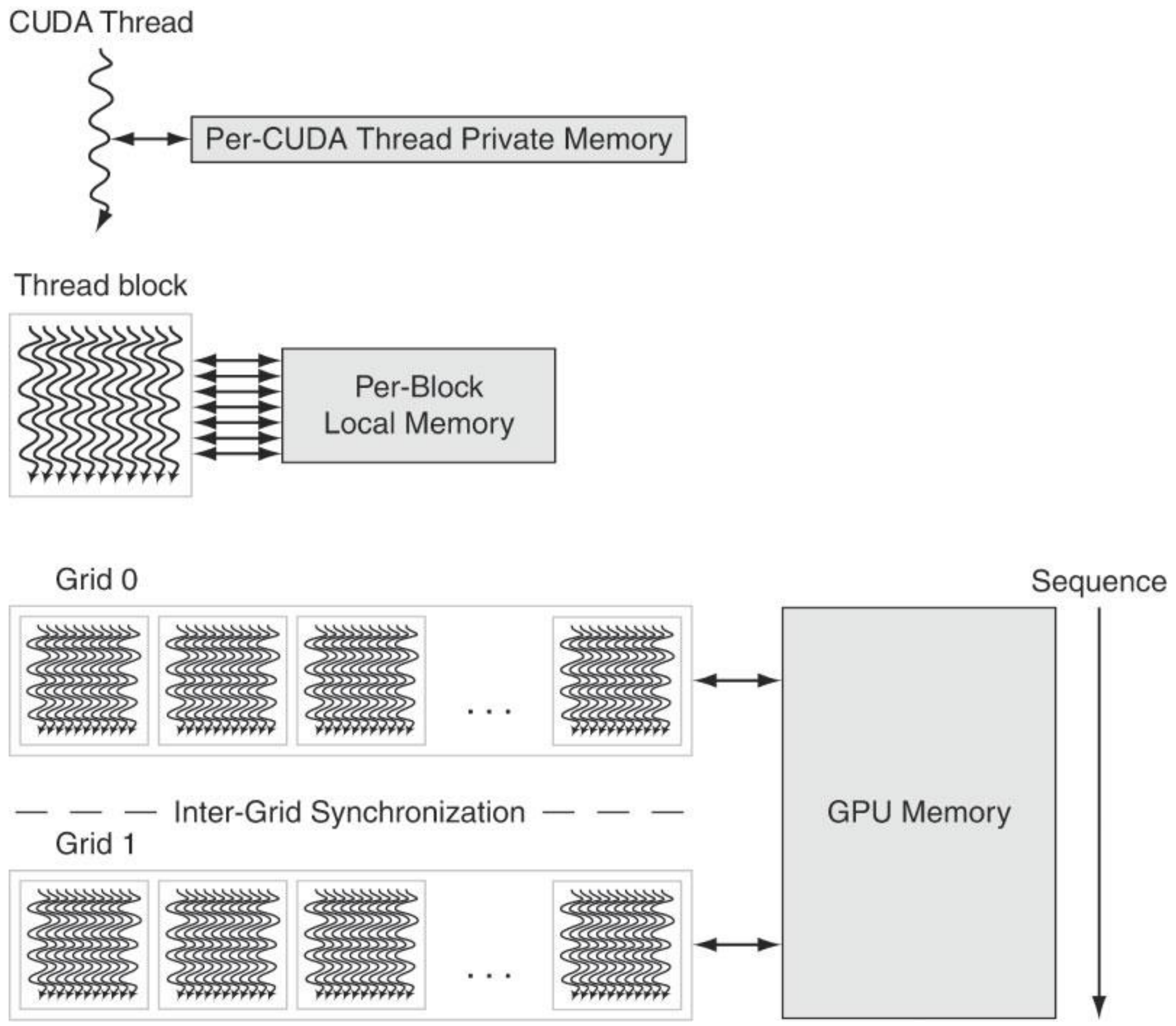
6.6 Programming GPUs

- Initially programmers only had graphics APIs and languages.
  - They developed C-inspired programming languages
  - NVIDIA Compute Unified Device Architecture (CUDA)
  - OpenCL is a multi-company initiative to develop a portable programming language
  - Unifying theme is CUDA thread
  - Compiler and hardware can gang thousands of CUDA threads together to utilize multithreading, MIMD, SIMD, and instruction-level parallelism
  - Threads are blocked together and executed in groups of 32 at a time
  - A multithreaded processor inside a GPU executes these blocks of threads, and a GPU consists of 8 to 32 of these multithreaded processors.

6.6 Block Diagram of SIMD Processor



6.6 GPU Memory Structures

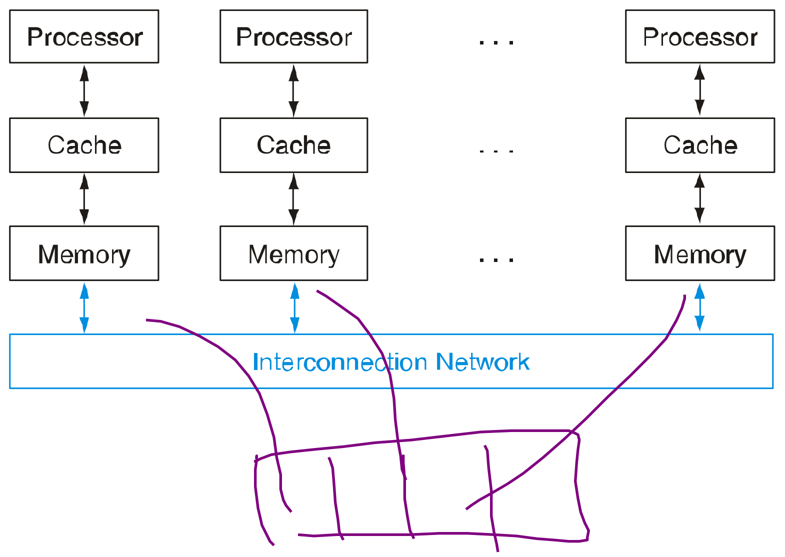


6.6 Putting GPUs into Perspective

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

6.7 Message Passing Multiprocessors

- When an address space is not shared, communication occurs via explicit message passing.
- Communication occurs using send and receive messages.
- Task-level parallelism and applications with little communication do not require shared addressing to run well. Examples include web search, mail servers, and file servers.
- Special purpose inter-connection networks have been used, they provide high performance at much higher cost than LANs.
- More common are clusters that are collections of computers connected via standard networks.



6.7 Message Passing Program

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

6.7 Hardware/Software Interface

- Message passing systems are easier for _hardware_ _[illegible]_ designers to _build_.
- For programmers, there are fewer _side_ _effects_, the communication is explicit, there is no guessing about the _cache_ _coherence_ performance
- However, it's harder to _port_ a _sequential_ program to a message-passing computer.
- Modern systems are a _hybrid_; _multicore_ multiprocessors use _shared_ physical memory and _nodes_ of a _cluster_ communicate with each other using _message_ _passing_.
- The weakness of _separate_ memories for user memory from a parallel programming perspective turns into a strength in system _dependability_.
- Computers can be _replaced_ in a cluster without _bringing_ the system _down_.
- Work can also be more easily _reallocated_ from _failing_ servers.
- Systems can be more easily _expanded_ using _clusters_.

6.7 Warehouse Scale Computers (WSCs)

- The most popular framework for batch processing in a WSC is MapReduce and the open source version, Hadoop, inspired by _Lisp_ functions of the same name.
- WSCs require innovations in _power_ _distribution_, _cooling_, _monitoring_, and _operations_, they are a modern descendant of the 1970s supercomputer.
- The 1970s supercomputer provided the _few_ _companies_ that could afford it, high performance computing for _scientists_ and _engineers_
- Warehouse Scale Computers make it possible for us to have _Internet_ _sensations_ on _YouTube_.

6.7 Warehouse Scale Computers (WSCs)

- WSCS have three major distinctions
  - Ample, easy parallelism - or request-level parallelism
  - Operations Costs Count – not just purchase price, energy, power distribution, and cooling represent more than 30% of the costs of a WSC over 10 years
  - Scale and the Opportunities/Problems Associated with scale
    - 100,000 servers mean volume discounts
    - Time on servers can be sold
    - 100,000 servers mean lots of failures - disks and servers
    - Fault tolerance takes on more importance

6.7 Warehouse Scale Computers (WSCs)

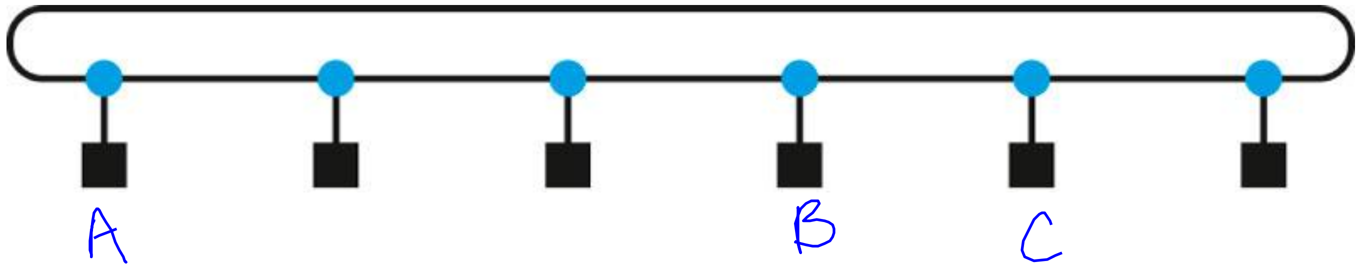
- In _2012_, Amazon Web Services announced the amount of new server capacity it adds _every_ _day_ is sufficient to support all of Amazon in 2003 when it was a company with _6000_ employees and _$5.2 billion_ in annual revenue.
- The growth of cloud computing could be slowed by _security_ concerns, _privacy_ concerns, _standards_ and the _rate_ of _growth_ of Internet bandwidth.
- https://www.youtube.com/watch?v=zRwPSFpLX8I
- https://www.youtube.com/watch?v=XZmGGAbHqa0

6.8 Introduction to Multiprocessor Network Topologies

- The popularity of cloud computing leads to a need for interconnection networks between nodes in a warehouse scale computer.
- The increasing number of cores per chips means we need networks inside a chip as well.
- Network costs include the number of switches, the number of links on a switch to connect to the network, the width per link, and length of the link when the network is mapped onto silicon.
- Network performance includes
  - The latency on an unloaded network to send and receive a message
  - The throughput in terms of the maximum number of messages that can be transmitted in a given time period
  - Delays caused by contention for a portion of the network
  - Variable performance depending on the pattern of communication.
  - Fault tolerance
  - Energy efficiency

6.8 Introduction to Multiprocessor Network Topologies

- Networks are normally drawn as graphs
  - _Edge_ represents _link_
  - _Node_ represents _computers_
  - Links are _bidirectional_
  - Networks consist of _switches_
- First example is a _ring_
  - Capable of _many_ _simultaneous_ transfers



A<=>B 3,3
A<=>C 2,4

max 5

6.8 Network Performance Metrics

- Network Bandwidth - the bandwidth of _each_ _link_ multiplied by the _number_ of _links_ (_best_ _case_)
  - Ring - P times the bandwidth of the link
  - Bus – the bandwidth of the bus

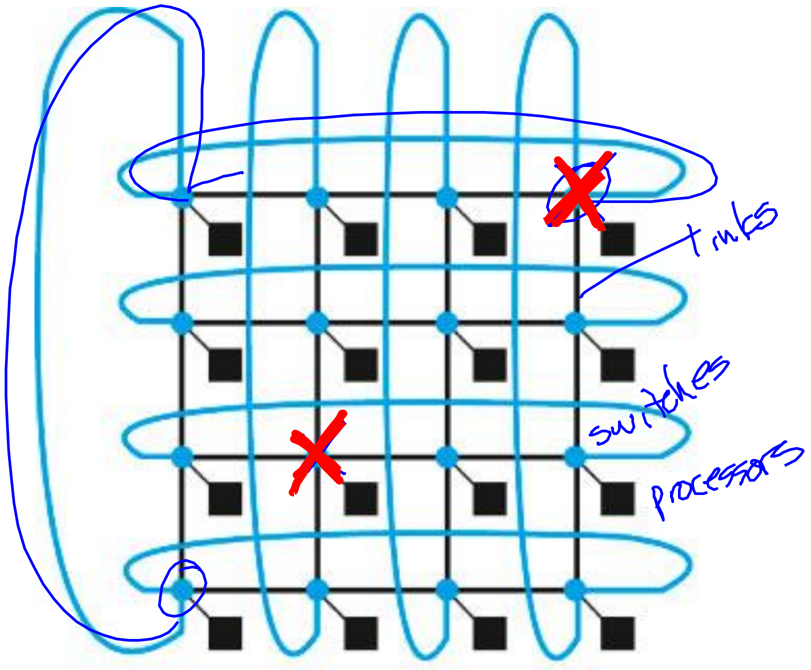
P - number of processors

- Bisection Bandwidth (_worst_ _case_) – the _sum_ of the bandwidth of the links that _cross_ the _divide_ between the two _halves_ of a machine
  - Ring – Two times the link bandwidth
  - Bus – the bandwidth of the bus
  - Some network topologies are not _symmetric_
    - Where do we draw the line?
    - Calculate _all_ _possible_ bisection bandwidths and pick the _smallest_
    - Parallel programs are often _limited_ by the _weakest_ link in the communication chain
- Diameter – _Maximum_ _distance_ between two processors

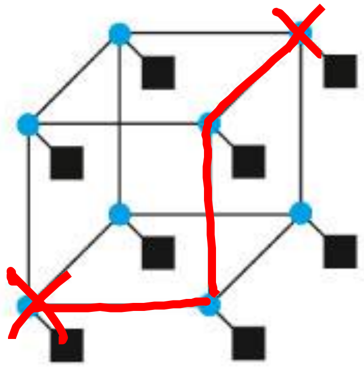
6.8 Network Topologies

- Each processor in a _ring_ network connects to _two_ other processors
  - Diameter of the network _P-1_
- In a _fully_ _connected_ network, every processor has a link to _every_ _other_ processor
  - Network bandwidth – $P \times (P-1)/2$
  - Bisection bandwidth – $(P/2)^2$
  - Diameter _1_
  - High _performance_ at high _cost_
- Many networks have been proposed between these two _extremes_ - their success largely depends on the _communication_ _workload_ of parallel programs
- Only a few have been used in _commercial_ parallel processors

6.8 Commercial Parallel Processor Topologies



a. 2-D grid or mesh of 16 nodes



b. n-cube tree of 8 nodes ($8 = 2^3$ so $n = 3$)