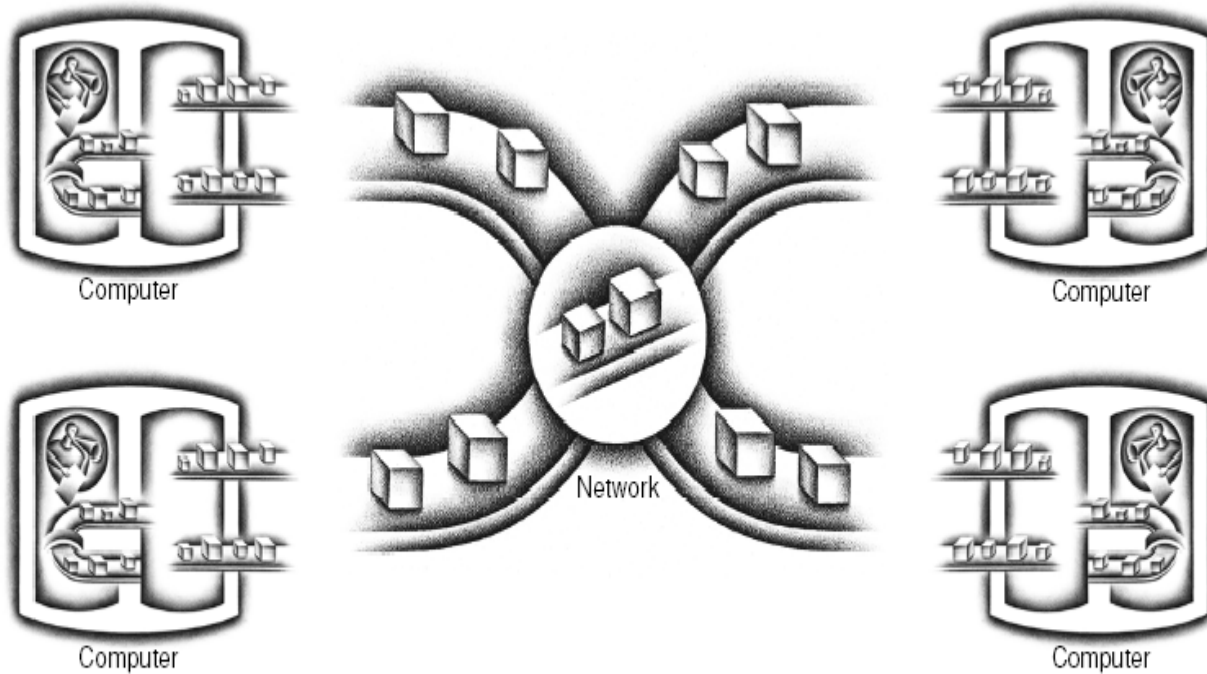


Parallel Processors from Client to Cloud



Parallel Computers

- **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

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

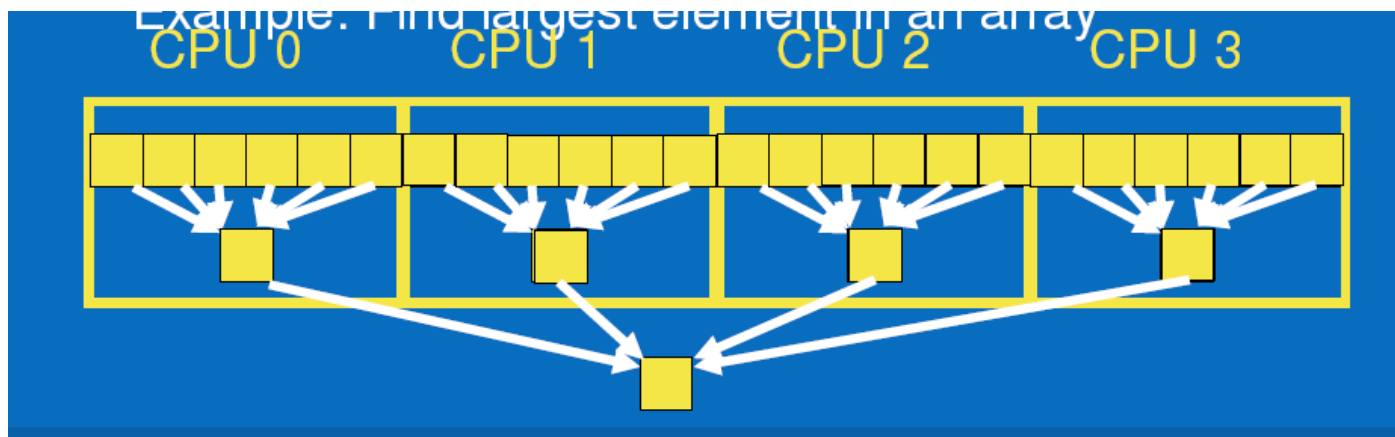
- Sequential part can limit speedup
- Example: 100 processors, 90× speedup?
 - $T_{\text{new}} = T_{\text{parallelizable}}/100 + T_{\text{sequential}}$
 - $$\text{Speedup} = \frac{1}{(1 - F_{\text{parallelizable}}) + F_{\text{parallelizable}}/100} = 90$$
 - Solving: $F_{\text{parallelizable}} = 0.999$
- Need sequential part to be 0.1% of original time

Parallelization Strategy

- Data decomposition
- Task decomposition
- Objective
 - Minimize the communication overheads as much as possible

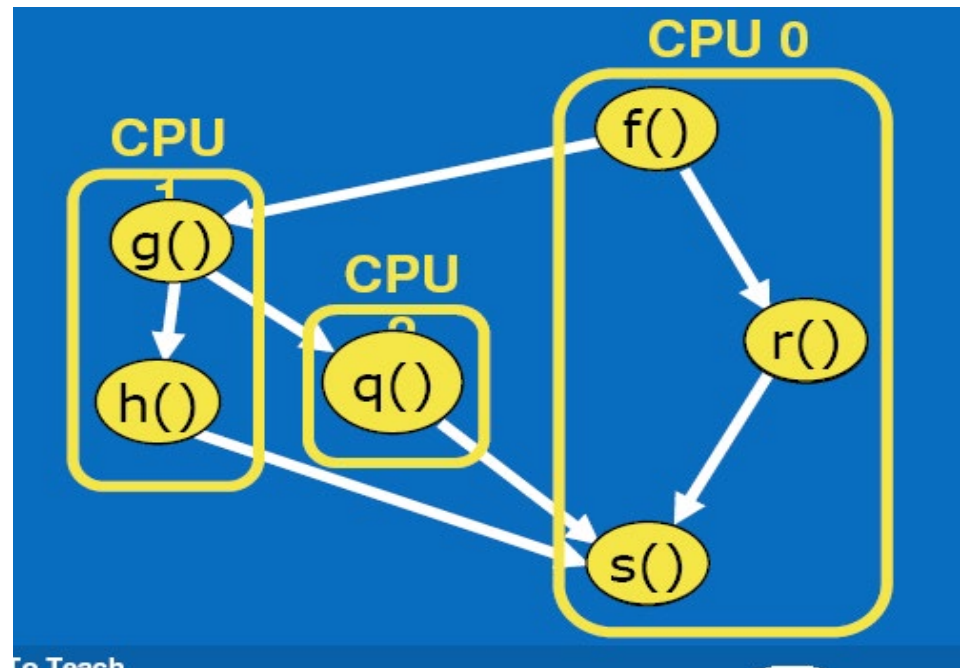
Data Decomposition

- **Decide how data elements should be divided among processors**
- **Decide which tasks each processor should be doing**
- **Example: Find the largest element in an array**



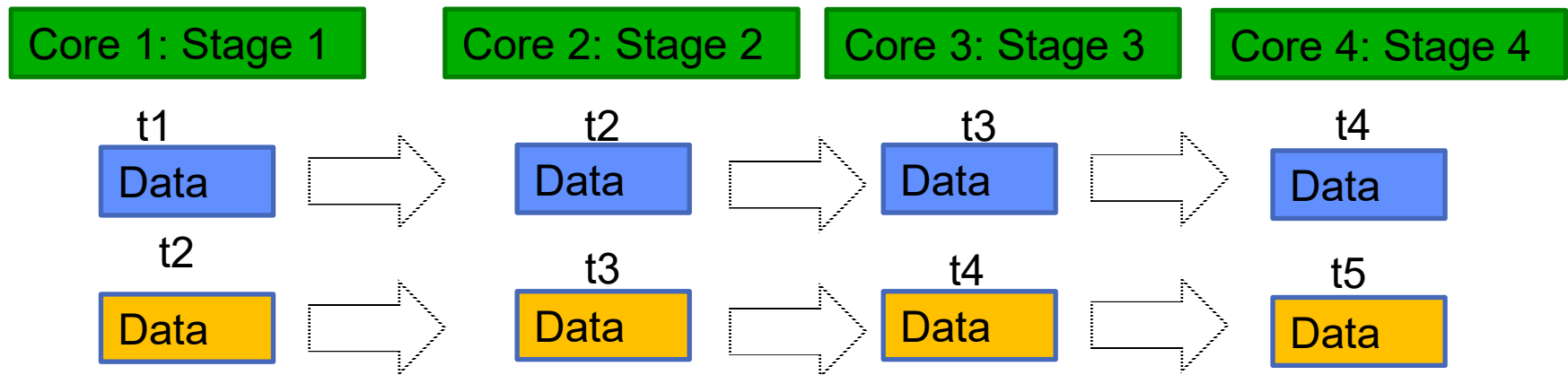
Task Decomposition

- Divide tasks among processors
- Decide which data elements are going to be accessed (read and/or written) by which processors
- Example



Pipelining

- Special kind of task parallelism



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)

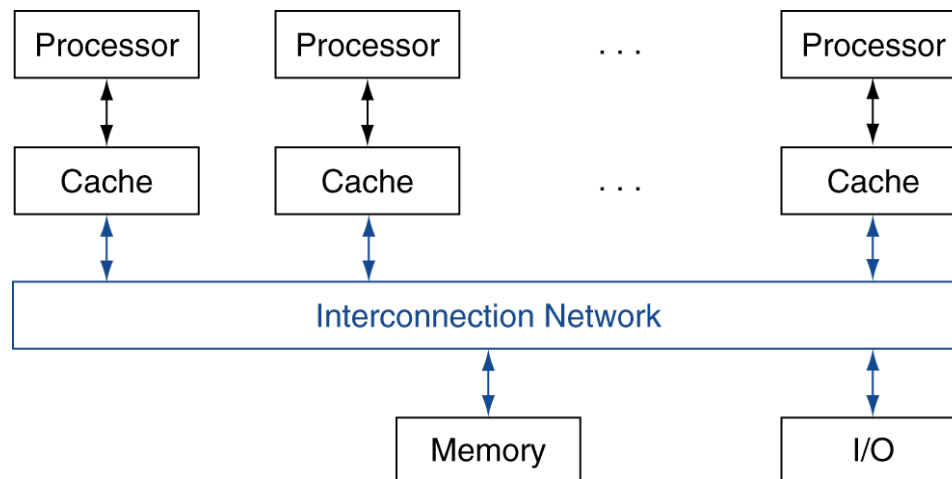
- What if matrix size is 100×100 ?
- Single processor: Time = $(10 + 10000) \times t_{\text{add}}$
- 10 processors
 - Time = $10 \times t_{\text{add}} + 10000/10 \times t_{\text{add}} = 1010 \times t_{\text{add}}$
 - Speedup = $10010/1010 = 9.9$ (99% of potential)
- 100 processors
 - Time = $10 \times t_{\text{add}} + 10000/100 \times t_{\text{add}} = 110 \times t_{\text{add}}$
 - Speedup = $10010/110 = 91$ (91% of potential)
- Assuming load balanced

Strong vs Weak Scaling

- **Strong scaling: problem size fixed**
- **Weak scaling: problem size proportional to number of processors**

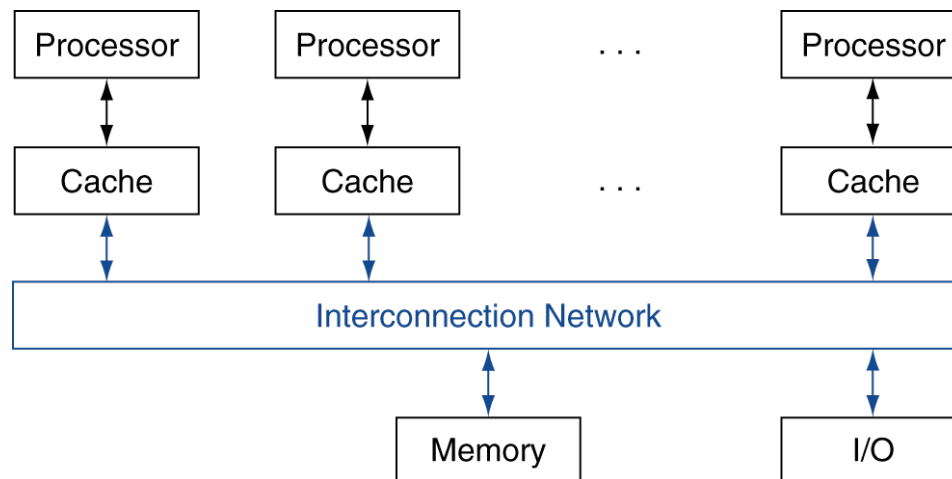
Shared Memory Multiprocessor

- **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)



Cache Coherency

- **Traffic per processor and the bus bandwidth determine the # of processors**
- **Caches can lower bus traffic**
 - **Cache coherency problem**



Cache Coherency

Time	Event	\$ A	\$ B	X (memory)
0				1
1	CPU A reads X	1		1
2	CPU B reads X	1	1	1
3	CPU A stores 0 into X	0	1	0

Cache Coherency Protocol

- **Snooping Solution (Snoopy Bus):**
 - Send all requests for data to all processors
 - Processors snoop to see if they have a copy and respond accordingly
 - Requires broadcast, since caching information is at processors
 - Works well with bus (natural broadcast medium)
 - Dominates for small scale machines (most of the market)

Basic Snoopy Protocols

- **Write Invalidate Protocol:**
 - Multiple readers, single writer
 - Write to shared data: an invalidate is sent to all caches which snoop and *invalidate* any copies
 - Read Miss:
 - » Write-through: memory is always up-to-date
 - » Write-back: snoop in caches to find most recent copy
- **Write Update Protocol:**
 - Write to shared data: broadcast on bus, processors snoop, and *update* copies
 - Read miss: memory is always up-to-date
- **What happens if two processors try to write to the same shared data word in the same clock cycle?**
 - Write serialization: bus serializes requests

Basic Snoopy Protocols

• Invalidation

Processor activity	Bus activity	Contents of CPU A' cache	Contents of CPU B's cache	Contexts of memory location X
				0
CPU A reads X	Cache miss for X	0		
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Invalidation for X	1		0
CPU B reads X	Cache miss for X	1	1	1

• Update

Processor activity	Bus activity	Contents of CPU A' cache	Contents of CPU B's cache	Contexts of memory location X
				0
CPU A reads X	Cache miss for X	0		
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Write broadcast of X	1	1	1
CPU B reads X		1	1	1

Basic Snoopy Protocols

- **Write Invalidate versus Broadcast:**
 - Invalidate requires one transaction per write-run
 - Invalidate uses spatial locality: one transaction per block
 - Update has lower latency between write and read
 - Update: BW (increased) vs. latency (decreased) tradeoff

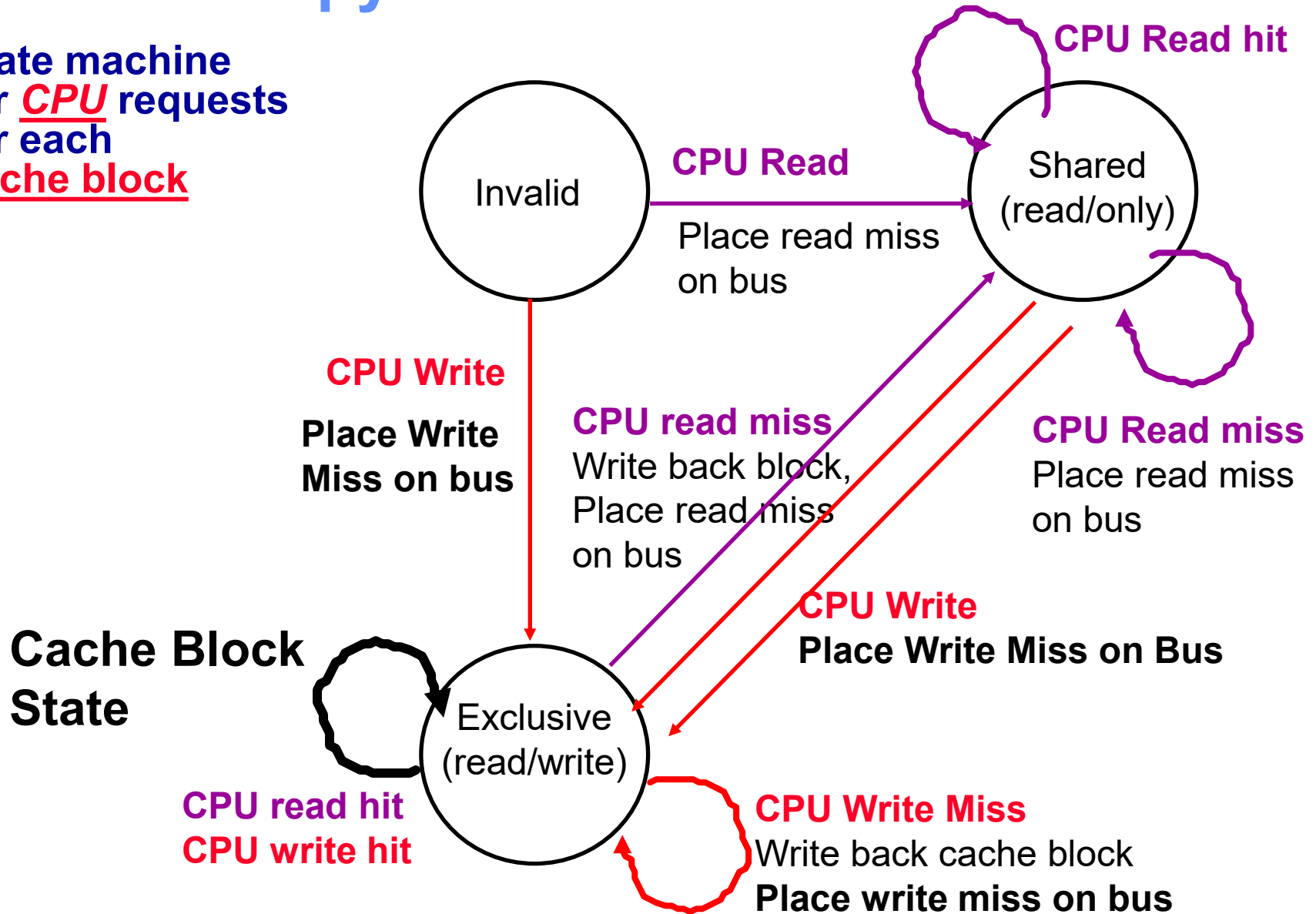
Invalidate protocol is more popular than update !

An Example Snoopy Protocol

- **Invalidation protocol, write-back cache**
- **Each block of memory is in one state:**
 - Clean in all caches and up-to-date in memory
 - OR Dirty in exactly one cache
 - OR Not in any caches
- **Each cache block is in one state:**
 - Shared: block can be read
 - OR Exclusive: cache has only copy, its writeable, and dirty
 - OR Invalid: block contains no data
- **Read misses: cause all caches to snoop**
- **Writes to clean line are treated as misses (or write invalidate)**

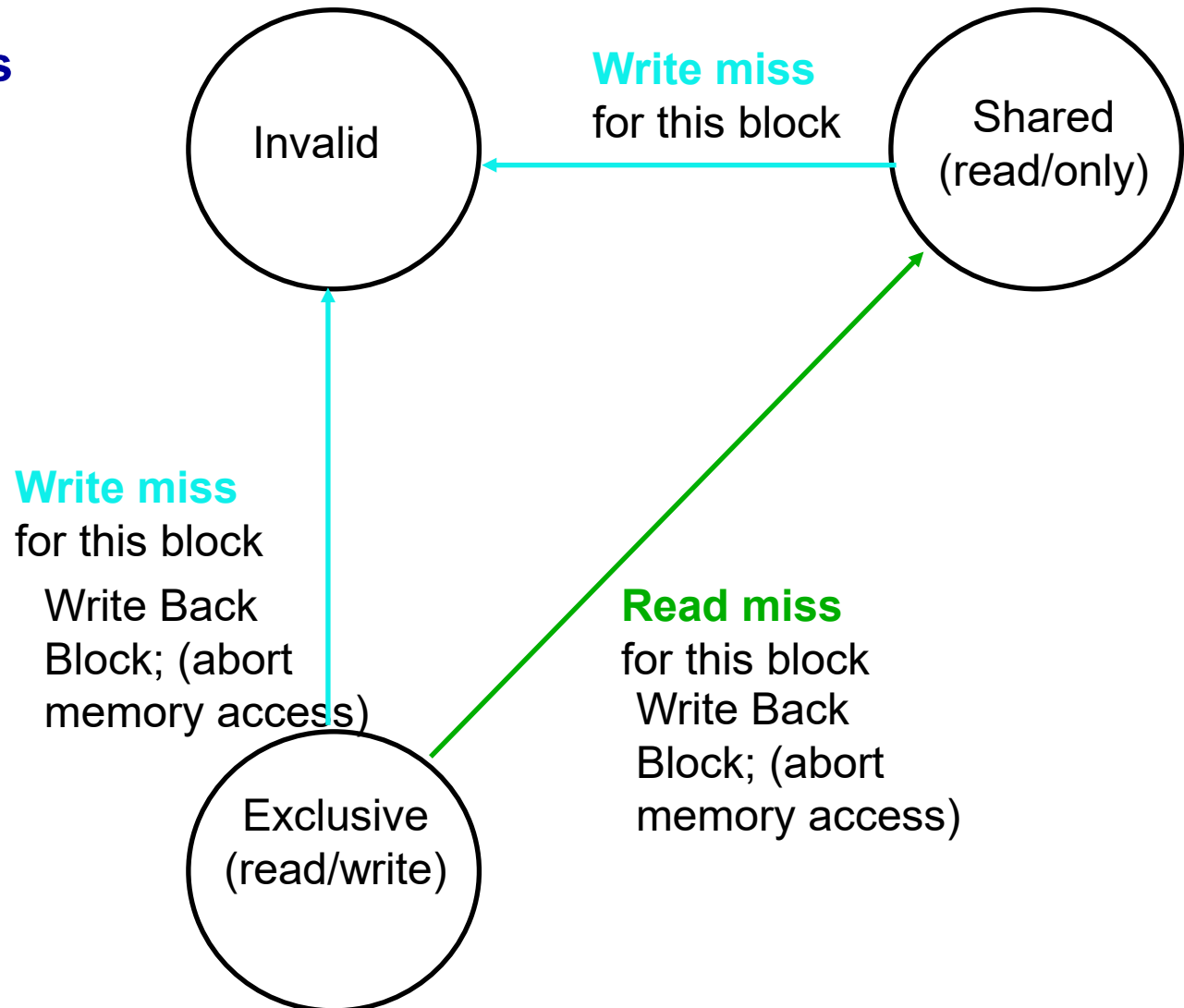
Snoopy-Cache State Machine-I

- State machine for **CPU** requests for each cache block



Snoopy-Cache State Machine-II

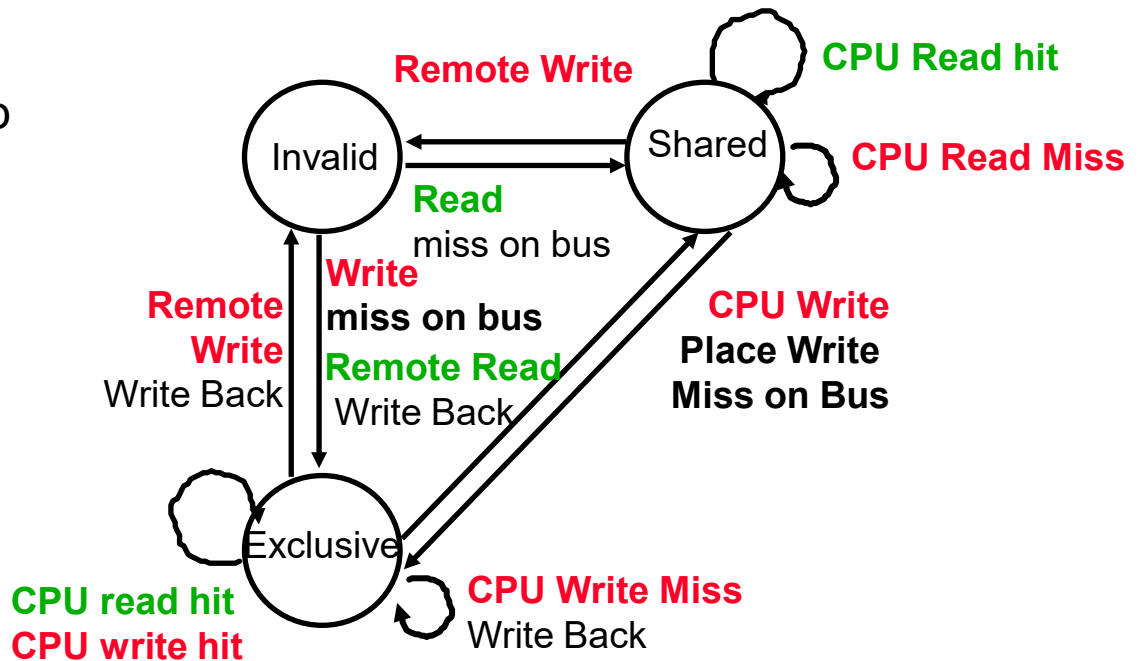
- State machine for bus requests for each cache block



Example

	Processor 1			Processor 2			Bus			Memory		
	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1												
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2

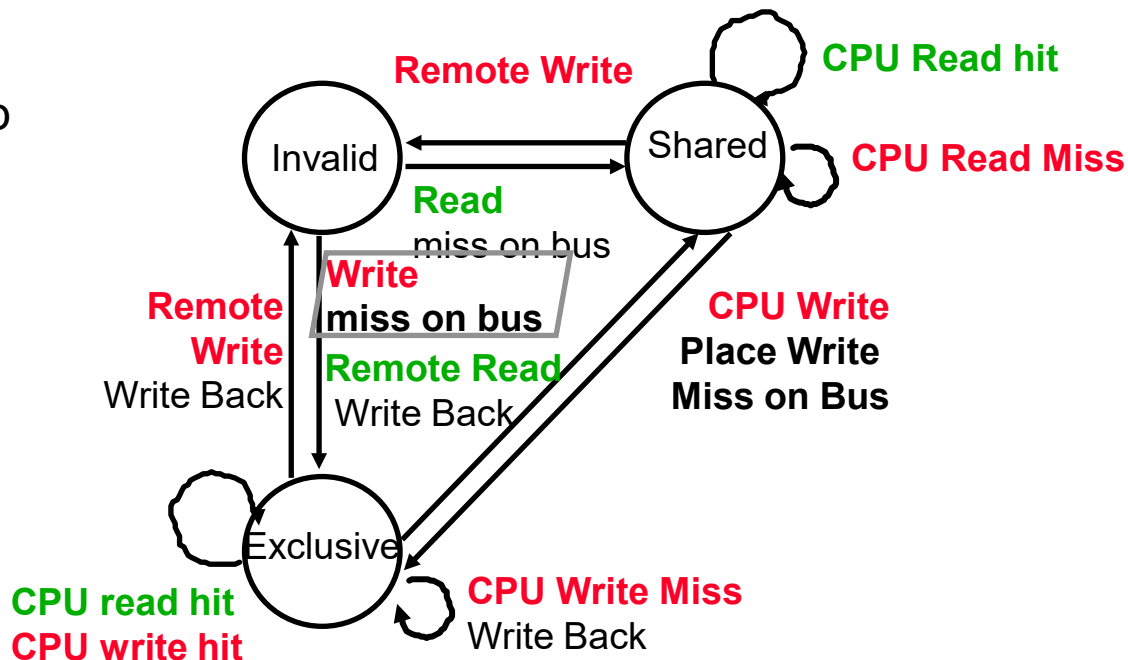


Example: Step 1

[illegible]

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but $A1 \neq A2$.

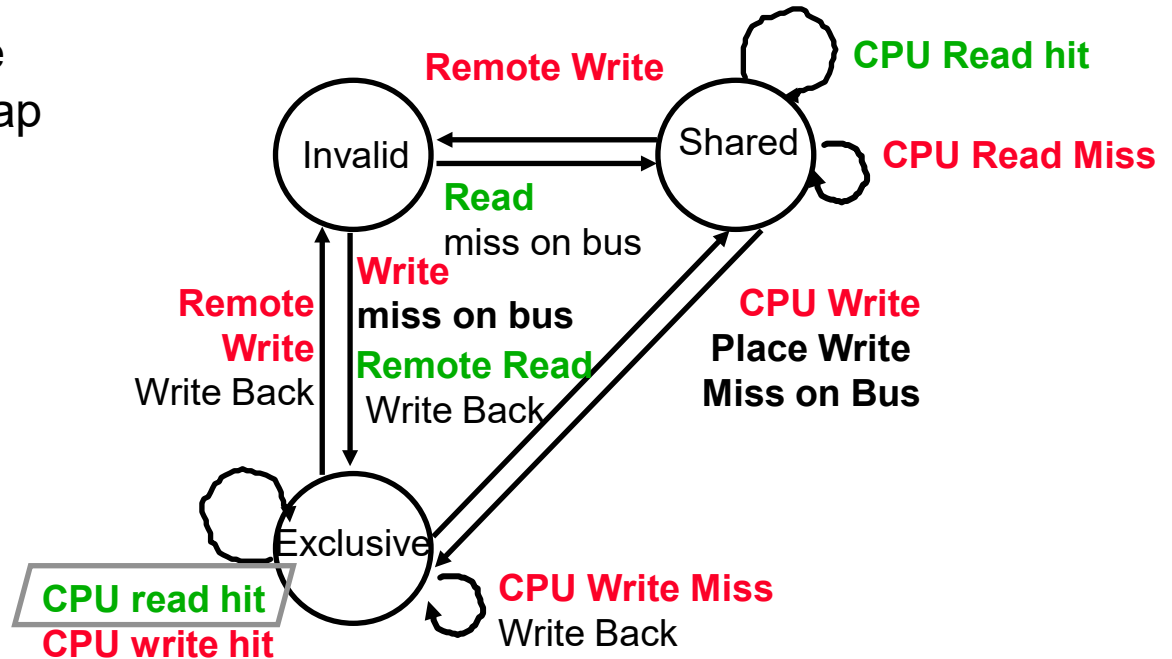
Active arrow =



Example: Step 2

	P1			P2			Bus			Memory		
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	<u>Excl.</u>	<u>A1</u>	<u>10</u>				<u>WrMs</u>	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

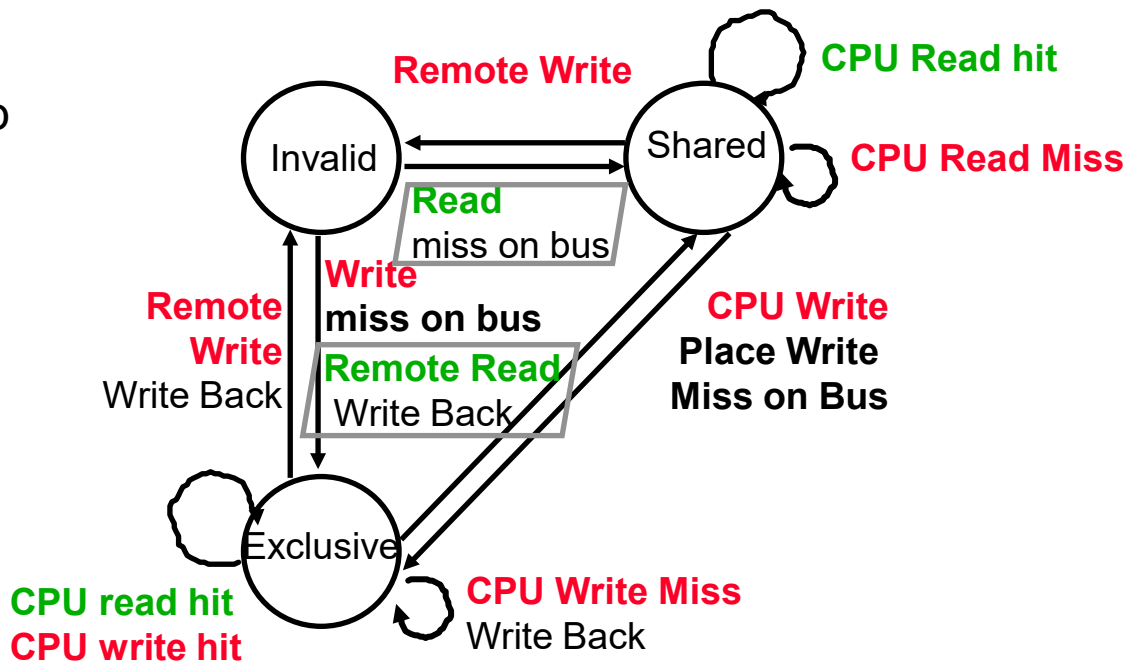
Assumes initial cache state is invalid and A1 and A2 map to same cache block, but $A1 \neq A2$



Example: Step 3

[illegible]

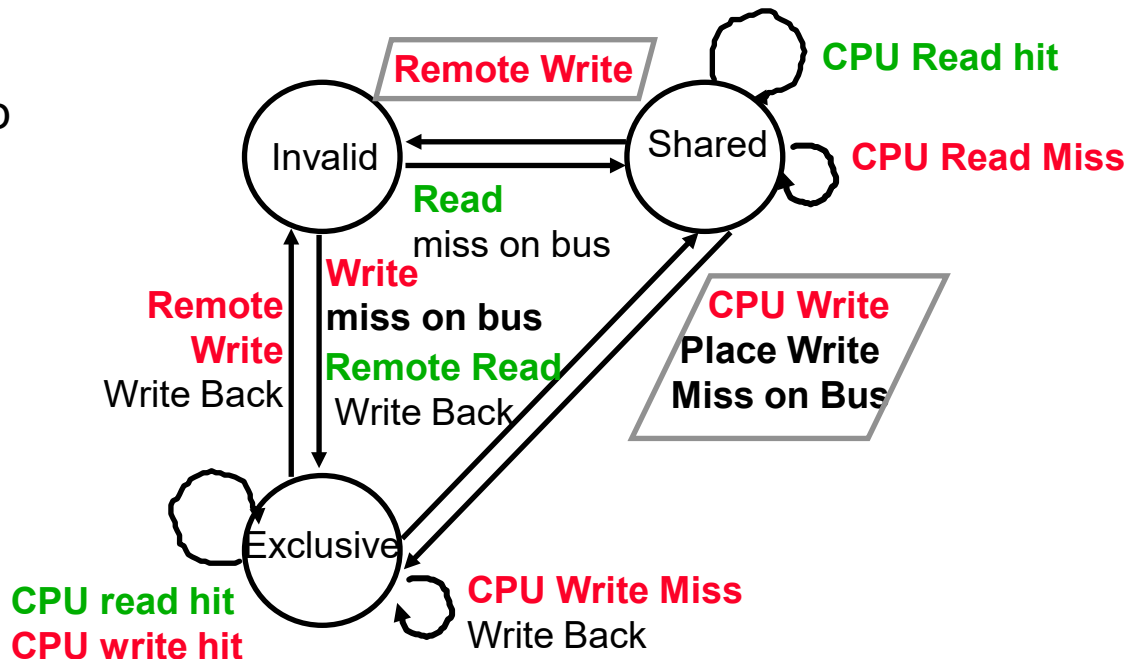
Assumes initial cache state is invalid and A1 and A2 map to same cache block, but $A1 \neq A2$.



Example: Step 4

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	<u>Excl.</u>	<u>A1</u>	<u>10</u>				<u>WrMs</u>	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				<u>Shar.</u>	<u>A1</u>		<u>RdMs</u>	P2	A1			
	<u>Shar.</u>	A1	10				<u>WrBk</u>	P1	A1	10	<u>A1</u>	<u>10</u>
				Shar.	A1	<u>10</u>	<u>RdDa</u>	P2	A1	10	A1	10
P2: Write 20 to A1	<u>Inv.</u>			<u>Excl.</u>	A1	<u>20</u>	<u>WrMs</u>	P2	A1		A1	10
P2: Write 40 to A2												-
												-

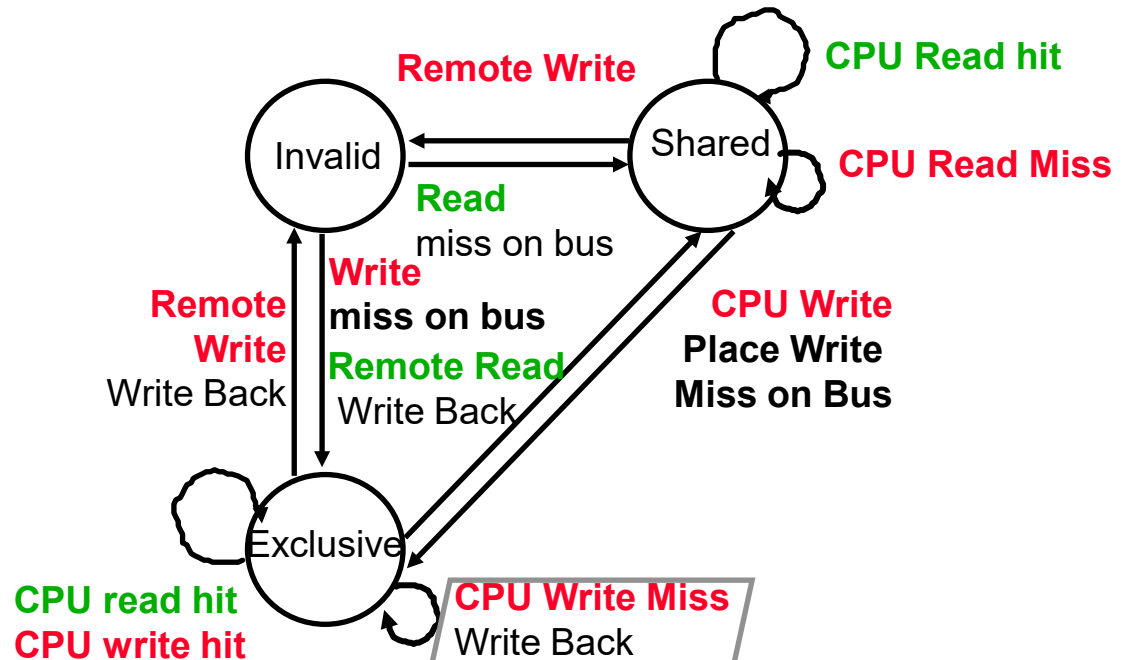
Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2



Example: Step 5

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1	<u>Excl.</u>	<u>A1</u>	<u>10</u>				<u>WrMs</u>	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				<u>Shar.</u>	<u>A1</u>		<u>RdMs</u>	P2	A1			
	<u>Shar.</u>	A1	10				<u>WrBk</u>	P1	A1	10	<u>A1</u>	<u>10</u>
				Shar.	A1	<u>10</u>	<u>RdDa</u>	P2	A1	10	A1	10
P2: Write 20 to A1	<u>Inv.</u>			<u>Excl.</u>	A1	<u>20</u>	<u>WrMs</u>	P2	A1		A1	10
P2: Write 40 to A2							<u>WrMs</u>	P2	A2		A1	10
				Excl.	<u>A2</u>	<u>40</u>	<u>WrBk</u>	P2	A1	20	<u>A1</u>	<u>20</u>

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but $A1 \neq A2$



Coherency Misses: 4th C

Joins Compulsory, Capacity, Conflict

- 1. True sharing misses arise from the communication of data through the cache coherence mechanism**
 - Invalidates due to 1st write to shared block
 - Reads by another CPU of modified block in different cache
 - Miss would still occur if block size were 1 word
- 2. False sharing misses when a block is invalidated because some word in the block, other than the one being read, is written into**
 - Invalidation does not cause a new value to be communicated, but only causes an extra cache miss
 - Block is shared, but no word in block is actually shared
⇒ miss would not occur if block size were 1 word

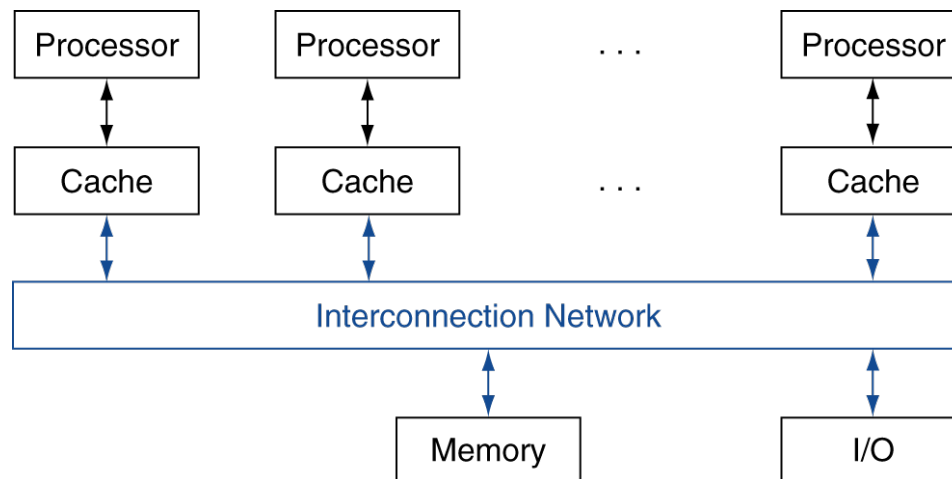
Example: True vs. False Sharing vs. Hit?

- Assume x1 and x2 in same cache block.
P1 and P2 both read x1 and x2 before.

Time	P1	P2	True, False, Hit? Why?
1	Write x1		True miss; invalidate x1 in P2
2		Read x2	False miss; x1 irrelevant to P2
3	Write x1		False miss; x1 irrelevant to P2
4		Write x2	False miss; x1 irrelevant to P2
5	Read x2		True miss; invalidate x2 in P1

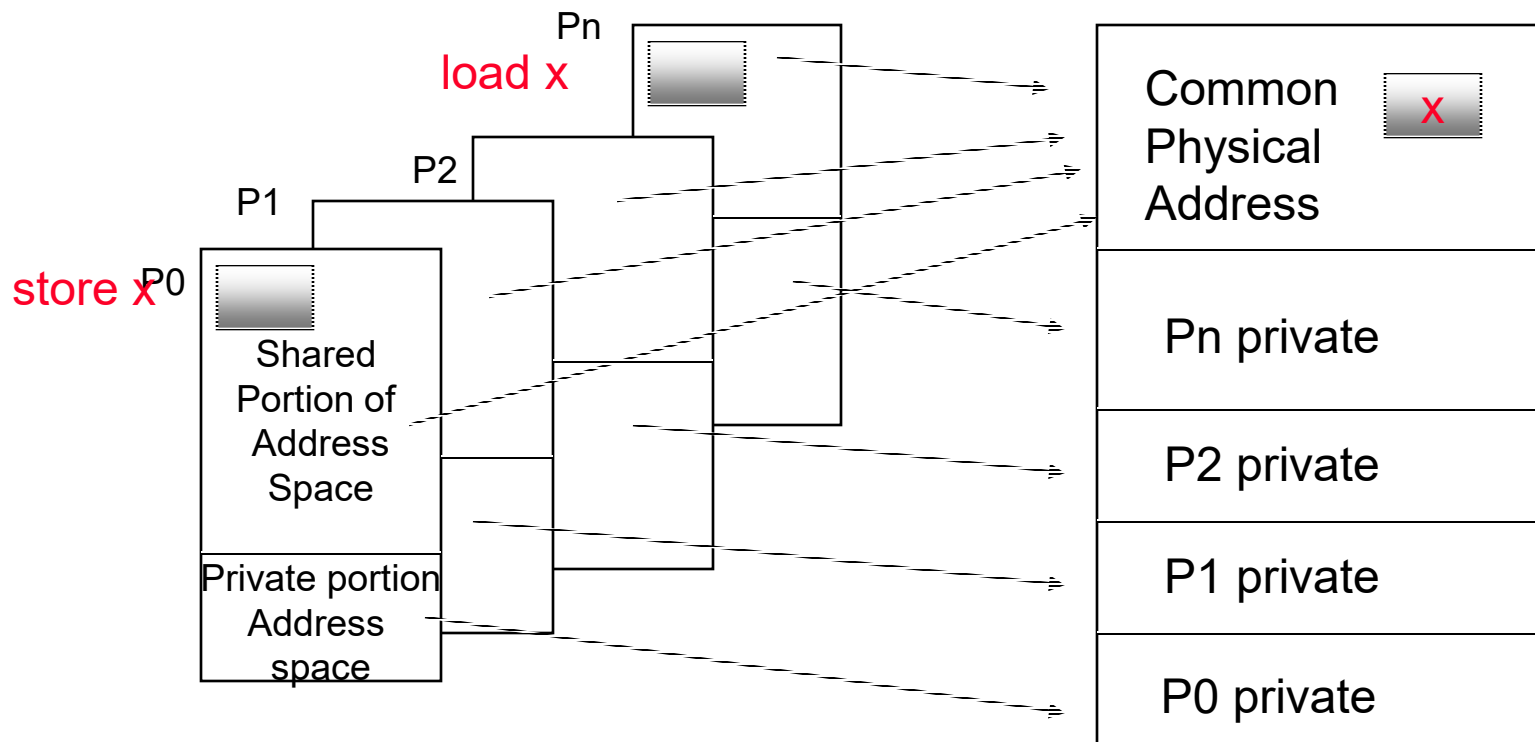
Shared Memory

- **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)



Communication Models

- **Single Address Space: load/store**

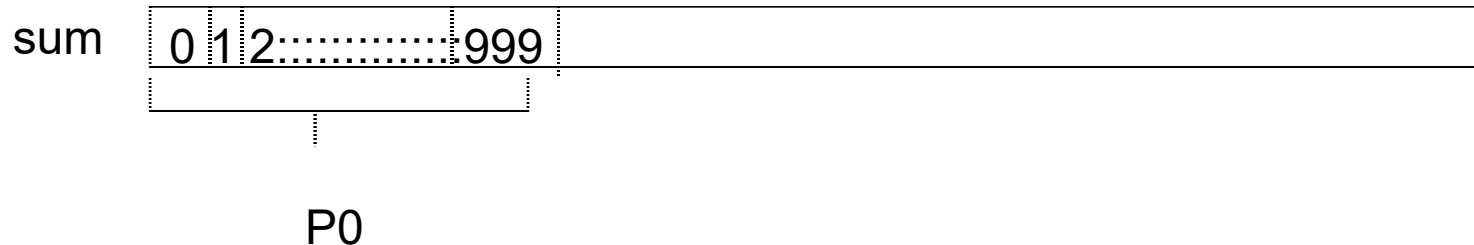


Program Example – Single-Address Space

- sum 100,000 numbers & 100 processors (load & store)

First Step: each processor (P_n) sums his subset of numbers

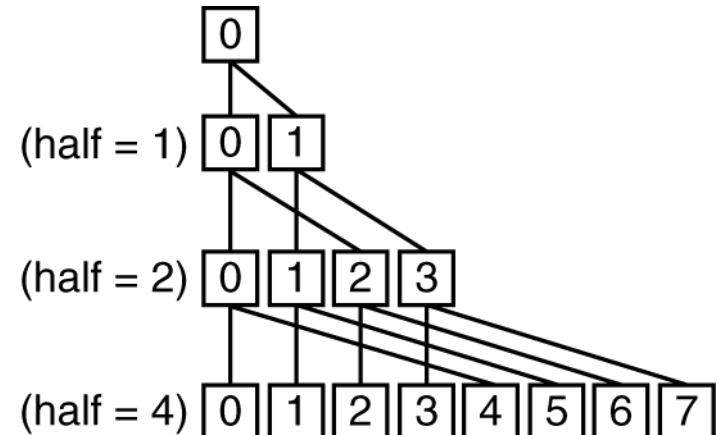
```
sum[Pn] = 0;      → Sum is shared variable  
for (i = 1000*Pn; i < 1000*(Pn+1); i = i + 1)  
    sum[Pn] = sum[Pn] + A[i]; /* sum the assigned areas*/
```



Program Example – Single-Address Space

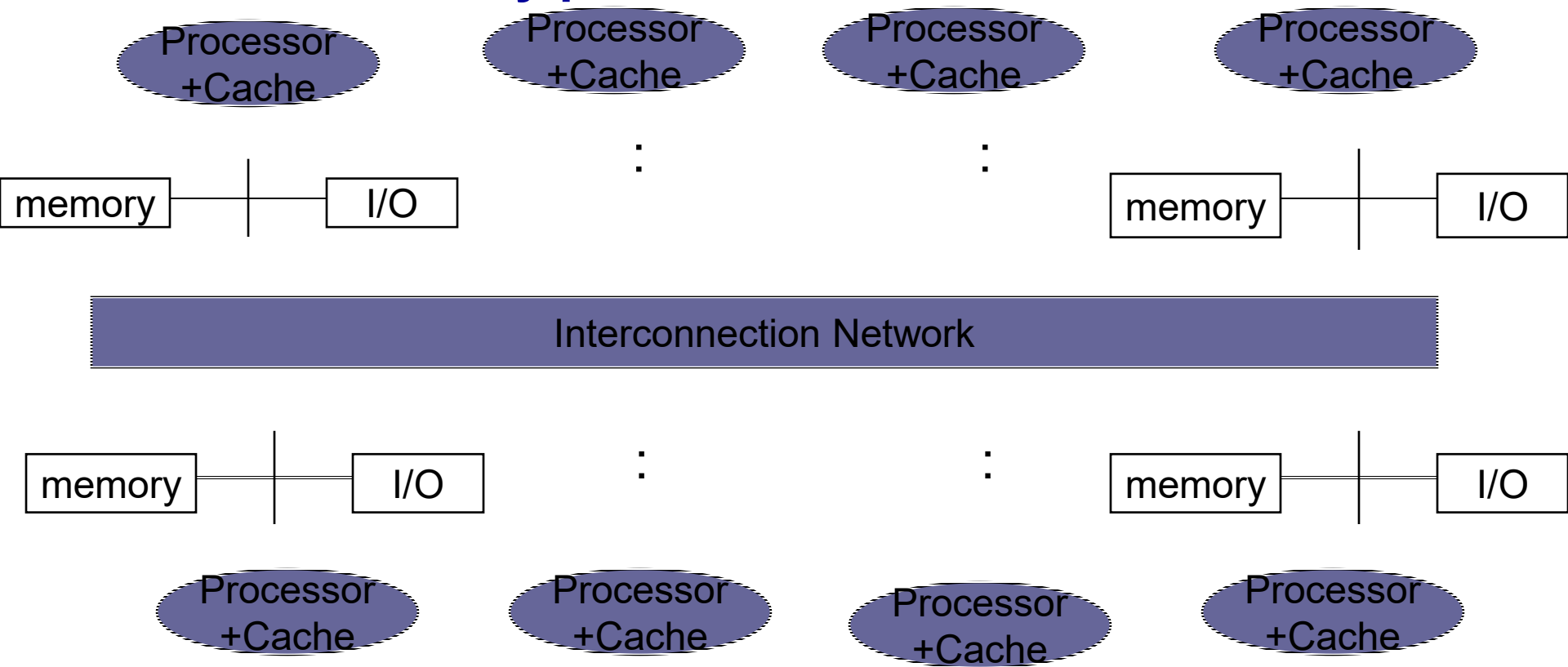
Second Step: Add partial sums via divide-and-conquer

```
half = 100; /* 100 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] */
```



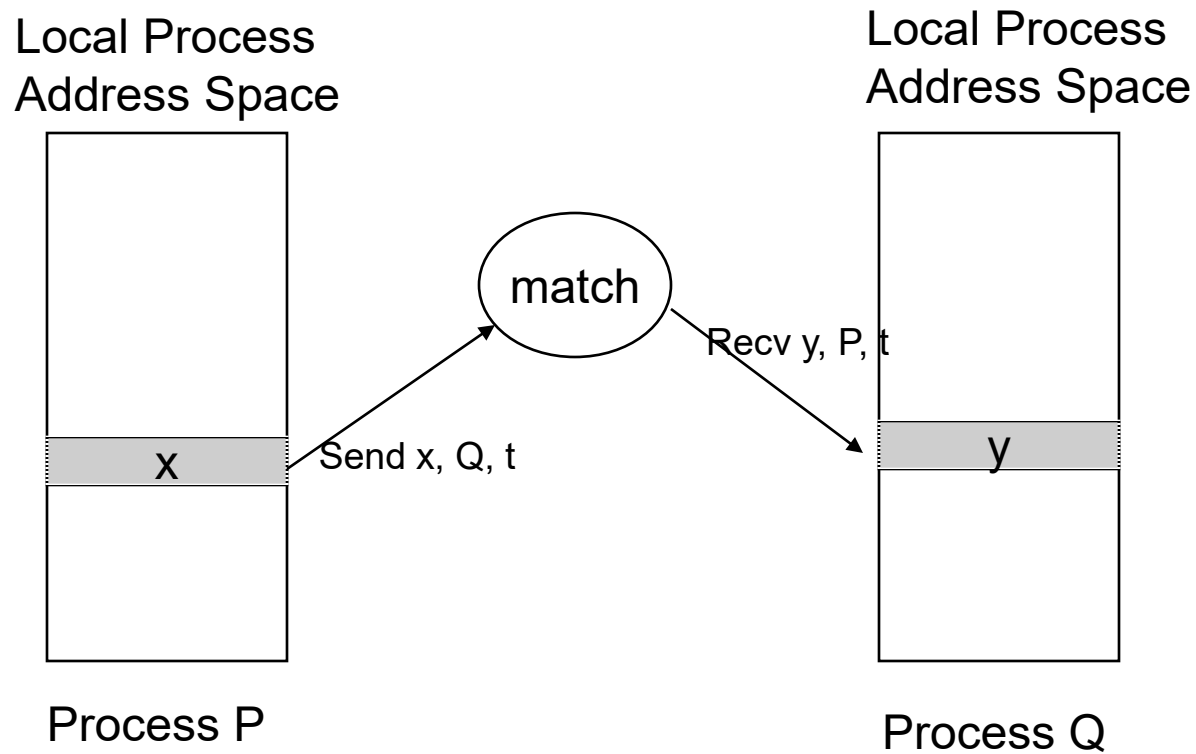
Message Passing Multiprocessors

- **Clusters:** Collections of computers connected via I/O over standard network switches to form a message-passing multiprocessors
- **NUMA:** Non-Uniform Memory Access/Directory-based cache coherency protocol



Communication Models

- **Multiple address spaces: Message Passing**



Parallel Program – Message Passing

- **sum 100,000 numbers & 100 processors (send & receive)**

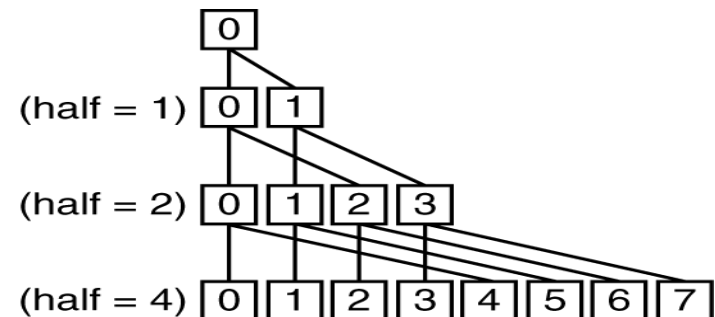
First Step: each processor (P_n) sums his subset of numbers

```
sum = 0;  
for (i = 0; i < 1000; i = i + 1) /* loop over each array */  
    sum = sum + A1[i]; /* sum the local arrays */
```

Parallel Program – Message Passing

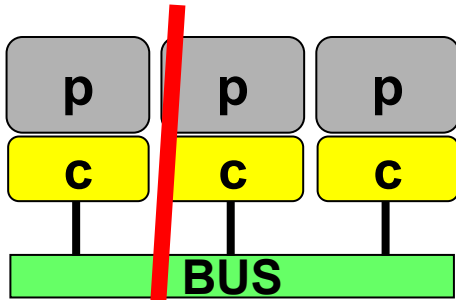
Second Step: Add partial sums via divide-and-conquer

```
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 */
```



Bisection Bandwidth is Important

Bus Multicore



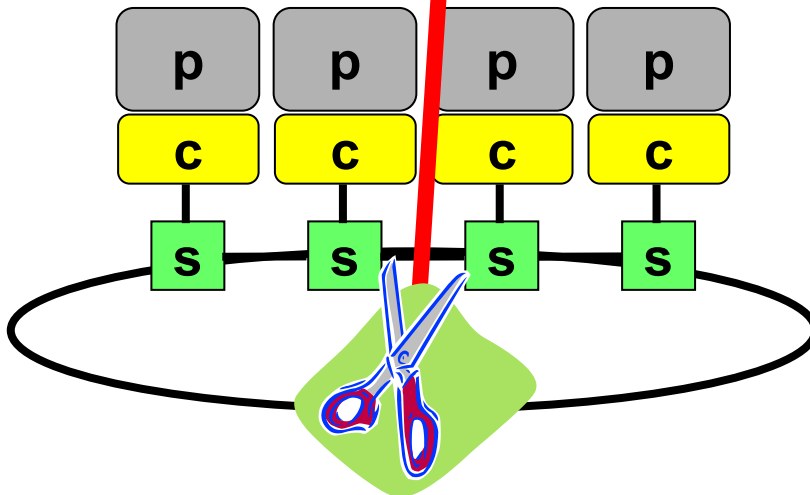
Total network bandwidth =

$$\text{bandwidth-per-link} \times \text{link_no}$$

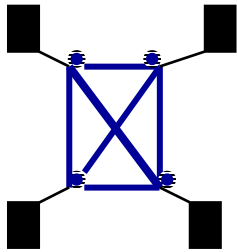
Bisection bandwidth =

the bandwidth between two equal parts of a multiprocessor

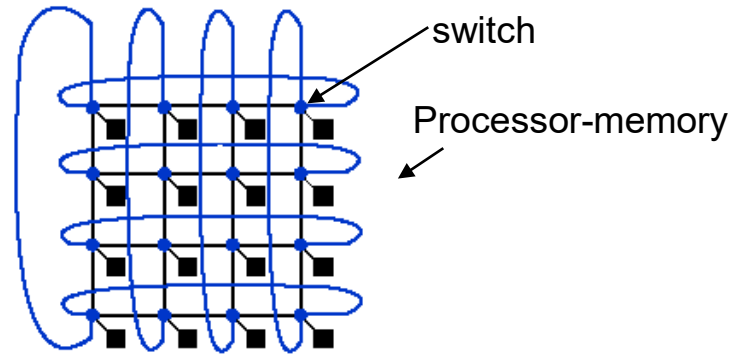
Ring Multicore



Network Topology



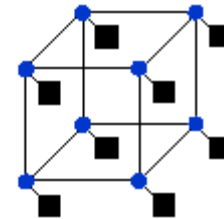
Fully-connected



2D torus

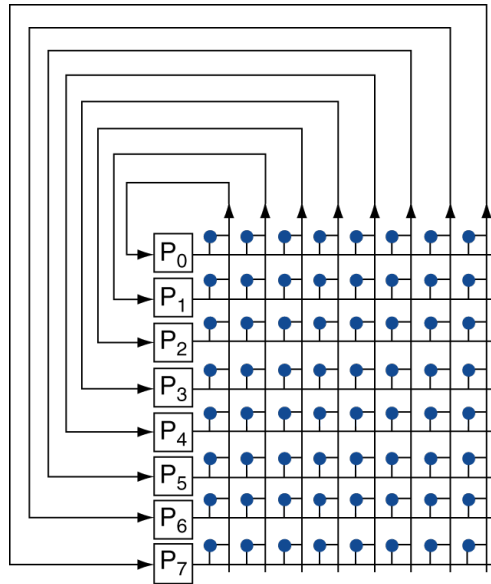


Ring

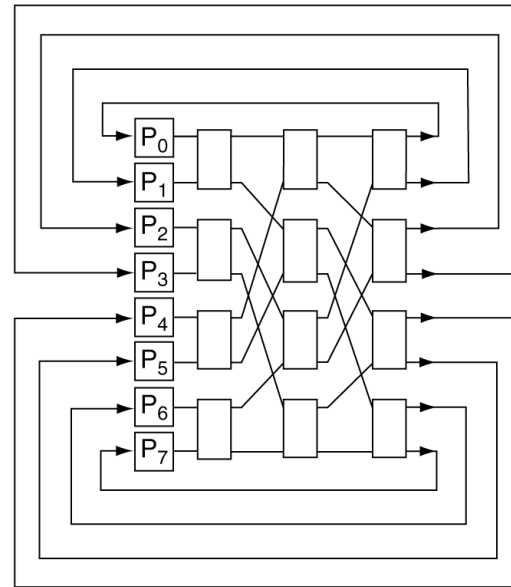


Cube

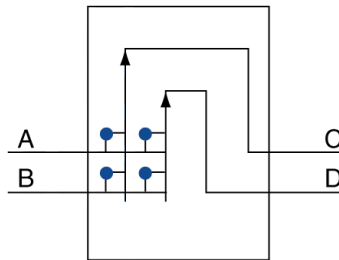
Multistage Networks



a. Crossbar



b. Omega network

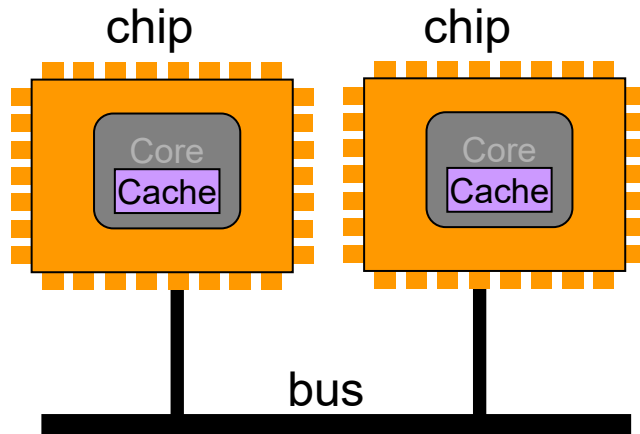


c. Omega network switch box

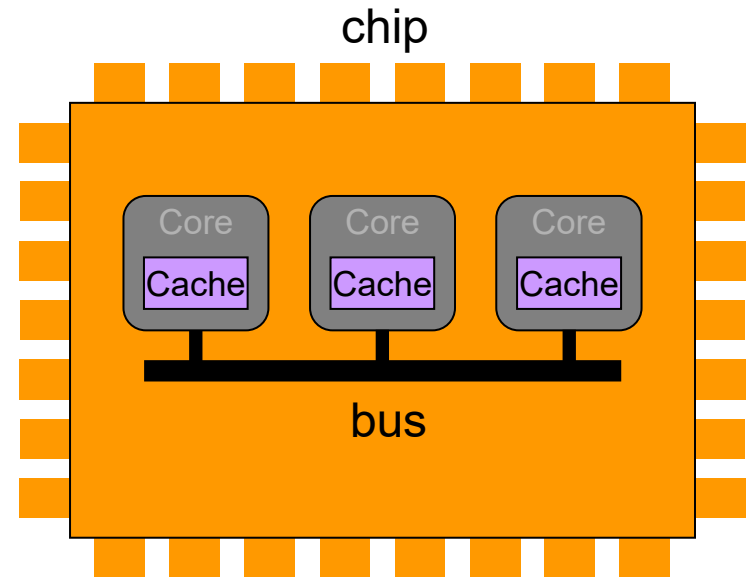
Network Characteristics

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

What is multi-core?



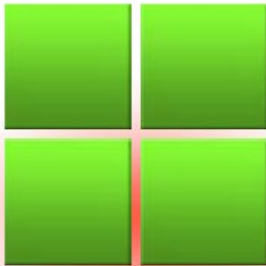
Off-chip bus



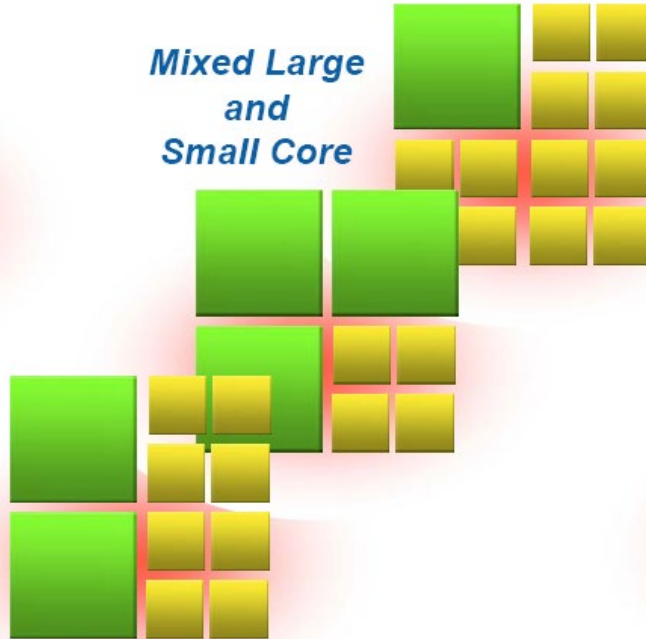
On-chip bus

From Multicore to Manycore

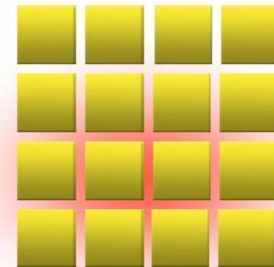
All Large Core



*Mixed Large
and
Small Core*

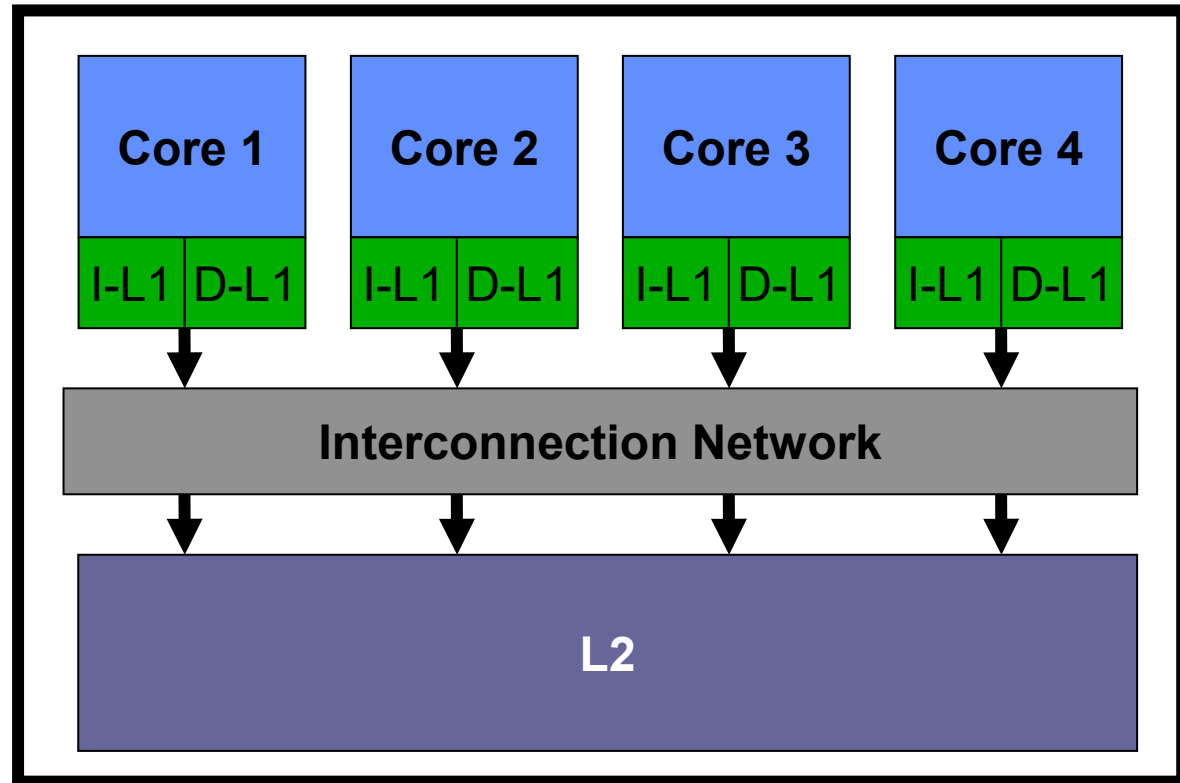
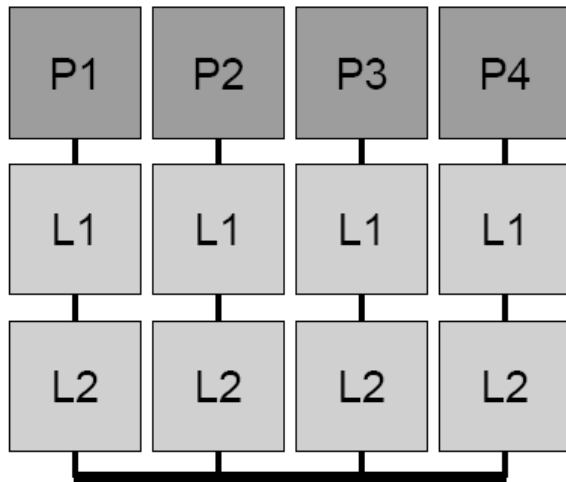


All Small Core



Basic CMP Architecture

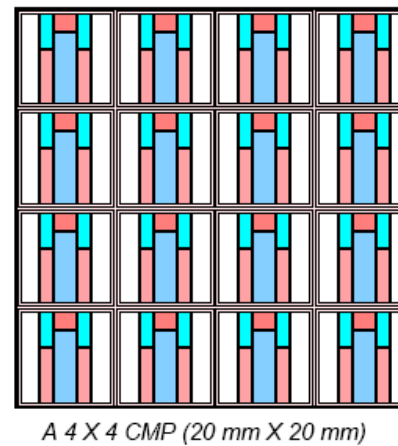
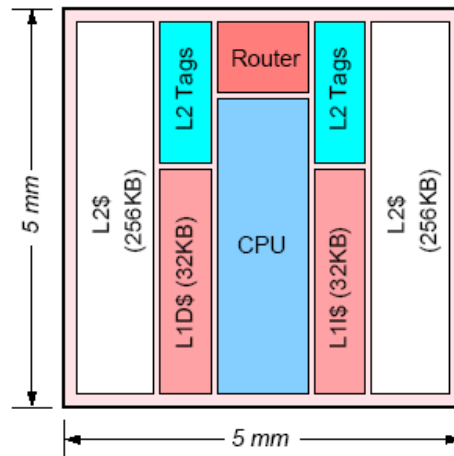
- L1 caches are always private to a core
- L2 caches can be private or shared – which is better?



Scalable CMP Architecture

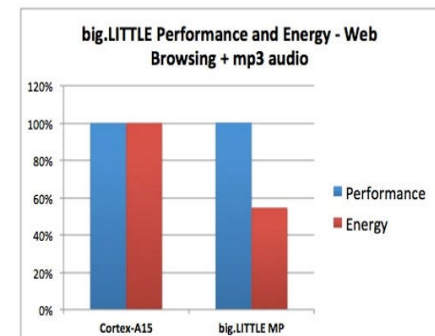
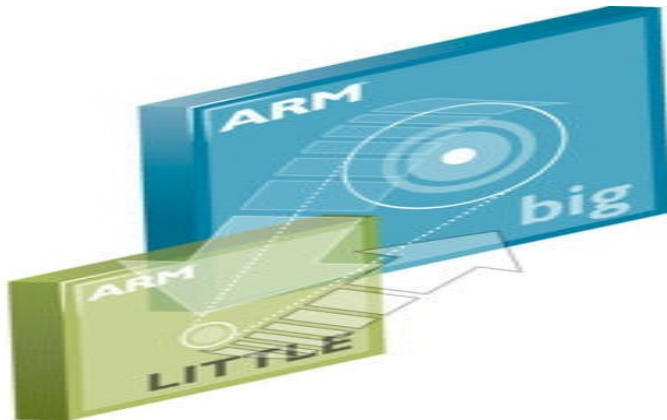
- **Tiled CMP**

- Each tile includes processor, L1, L2, and router
- Physically distributed last level cache



ARM Big.Little Technology

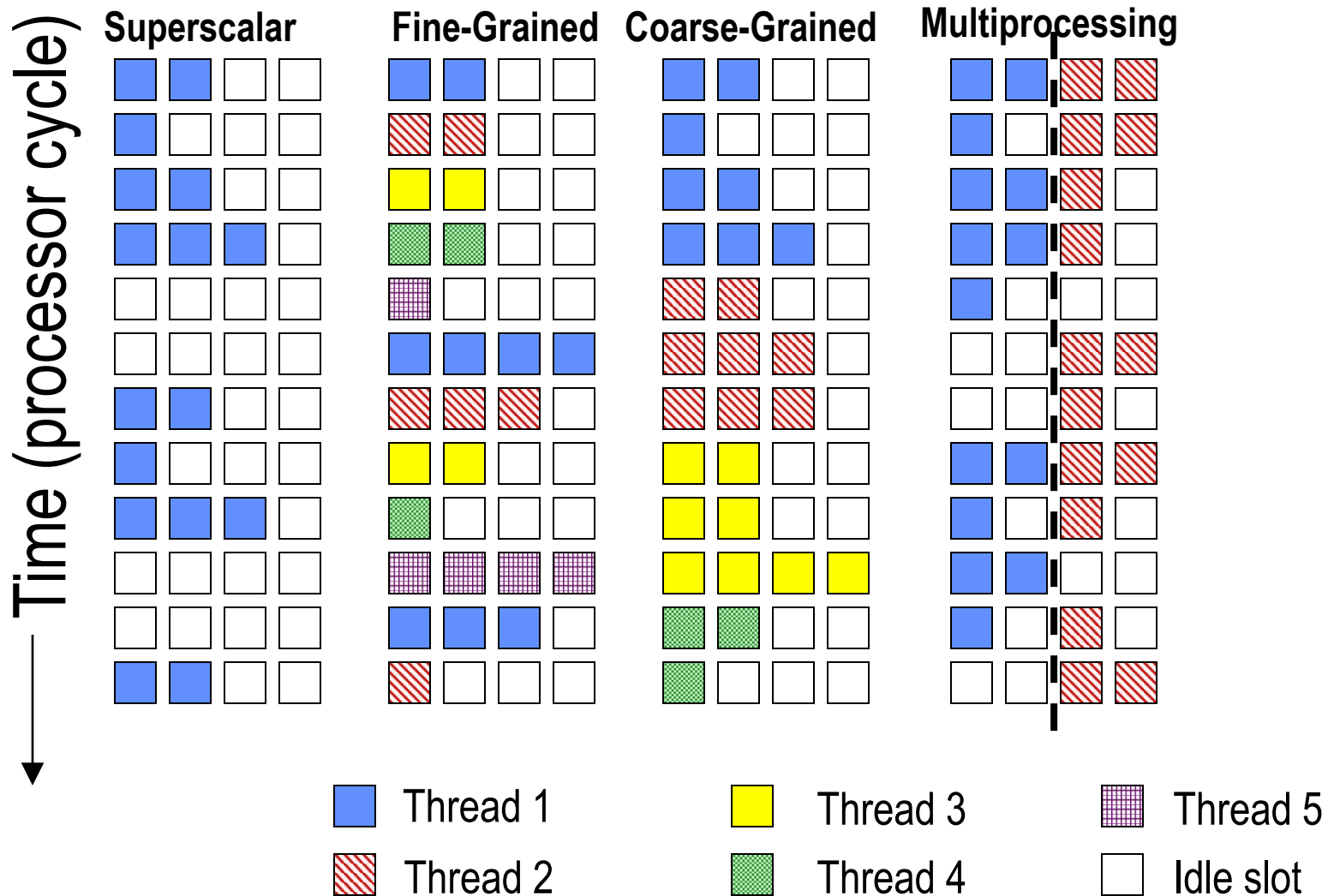
- ARM big.LITTLE processing is designed to deliver the vision of **the right processor for the right job.**
- In current big.LITTLE system implementations a **'big' ARM Cortex™-A15 processor** is paired with a **'LITTLE' Cortex™-A7 processor** to create a system that can accomplish both high intensity and low intensity tasks in the most energy efficient manner
 - Cortex-A15 – heavy workloads
 - Cortex-A7 - light workloads, like operating system activities, user interface and other always on, always connected tasks.



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)

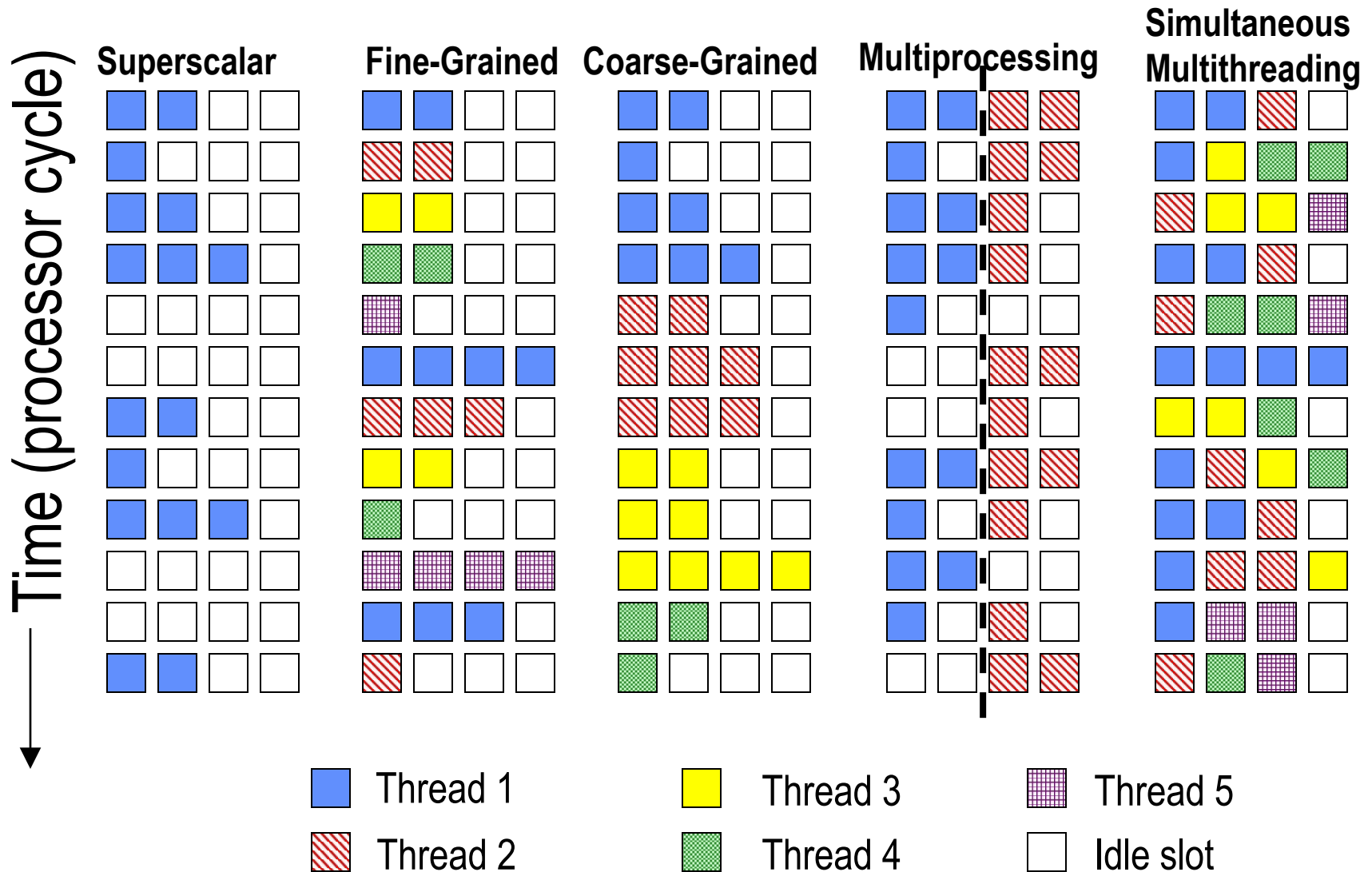
Multithreaded Categories



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

Multithreaded Categories



Computing Device Classification: Instruction and Data Streams


		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

SIMD

- SIMD architectures can exploit significant data-level parallelism for:
 - matrix-oriented scientific computing
 - media-oriented image and sound processors

$$\begin{array}{c} 4 \times 2 \text{ matrix} \\ \begin{bmatrix} a_{11} & a_{12} \\ \cdot & \cdot \\ a_{31} & a_{32} \\ \cdot & \cdot \end{bmatrix} \end{array}
 \begin{array}{c} 2 \times 3 \text{ matrix} \\ \begin{bmatrix} \cdot & b_{12} & b_{13} \\ \cdot & b_{22} & b_{23} \end{bmatrix} \end{array}
 =
 \begin{array}{c} 4 \times 3 \text{ matrix} \\ \begin{bmatrix} \cdot & x_{12} & x_{13} \\ \cdot & \cdot & \cdot \\ \cdot & x_{32} & x_{33} \\ \cdot & \cdot & \cdot \end{bmatrix} \end{array}$$

A	B
	
C	D
AxC	BxD

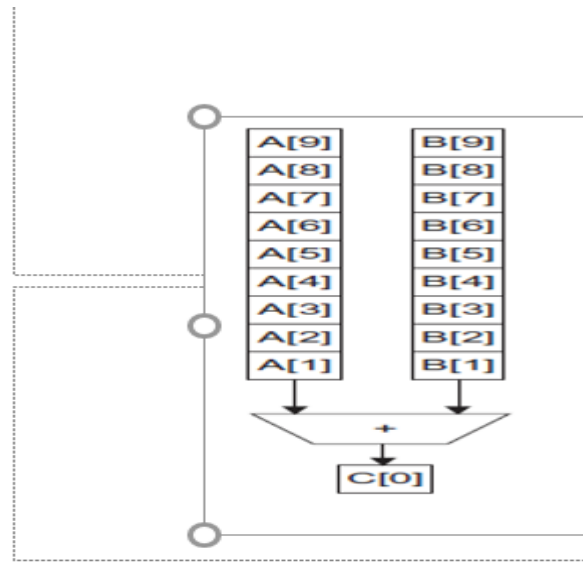
- SIMD is more energy efficient than MIMD
 - Only needs to fetch one instruction per data operation
 - Makes SIMD attractive for personal mobile devices
- SIMD allows programmer to continue to think sequentially

SIMD Extensions

- **Media applications operate on data types narrower than the native word size**
 - 4-byte registers – R,G,B (byte per pixel)
- **Implementations:**
 - Intel MMX (1996)
 - » Eight 8-bit integer ops or four 16-bit integer ops
 - Streaming SIMD Extensions (SSE) (1999)
 - » Eight 16-bit integer ops
 - » Four 32-bit integer/fp ops or two 64-bit integer/fp ops
 - Advanced Vector Extensions (2010)
 - » Four 64-bit integer/fp ops
 - Operands must be consecutive and aligned memory locations

Vector Architectures

- Basic idea:
 - Read sets of data elements into “vector registers” (Gather)
 - Operate on those registers
 - Highly pipelined function units
 - Disperse the results back into memory (Scatter)



Vector Extension to RISC-V

- v0 to v31: 64×64 -bit element registers
- Vector instructions
 - fl`d.v`, fs`d.v`: load/store vector
 - fadd`.d.v`: add vectors of double
 - fadd`.d.vs`: add scalar to each element of vector of double
- Significantly reduces instruction-fetch bandwidth

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

Conventional RISC-V code:

```
fld    f0,a(x3)      // load scalar a
addi   x5,x19,512     // end of array x
loop:  fld    f1,0(x19) // load x[i]
      fmul.d  f1,f1,f0  // a * x[i]
      fld    f2,0(x20) // load y[i]
      fadd.d  f2,f2,f1  // a * x[i] + y[i]
      fsd    f2,0(x20)  // store y[i]
      addi   x19,x19,8  // increment index to x
      addi   x20,x20,8  // increment index to y
      bltu   x19,x5,loop // repeat if not done
```

Vector RISC-V code:

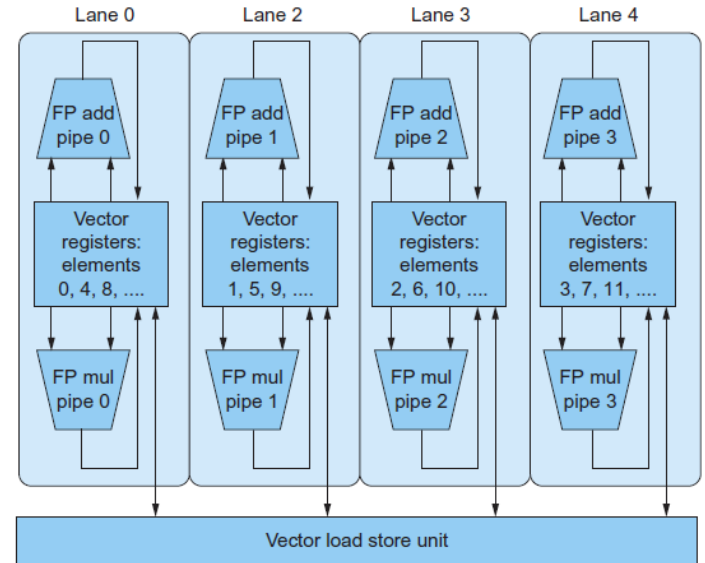
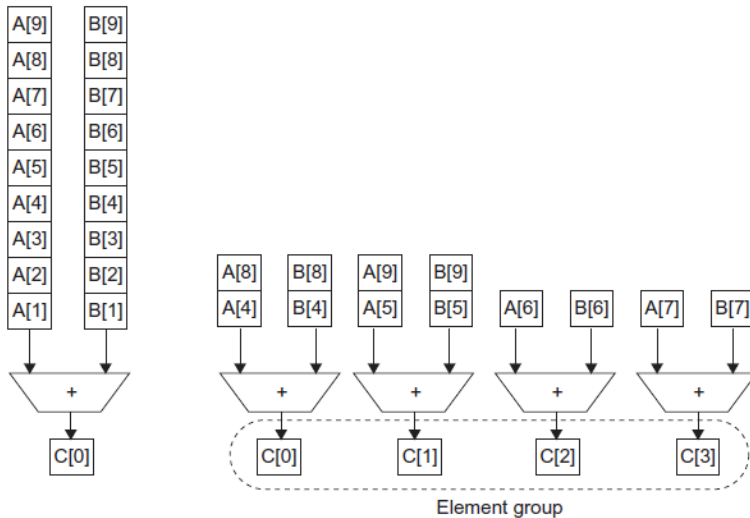
```
fld    f0,a(x3)      // load scalar a
fld.v   v0,0(x19)     // load vector x
fmul.d.vs v0,v0,f0    // vector-scalar multiply
fld.v   v1,0(x20)     // load vector y
fadd.d.v v1,v1,v0     // vector-vector add
fsd.v   v1,0(x20)     // store vector y
```


Vector vs. Scalar

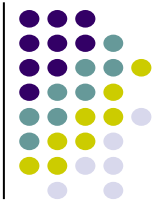
- **Vector architectures and compilers**
 - Simplify data-parallel programming
 - Explicit statement of absence of loop-carried dependences
 - » Reduced checking in hardware
 - Regular access patterns benefit from interleaved and burst memory
 - Avoid control hazards by avoiding loops
- **More general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

Multiple-Lane Vector units

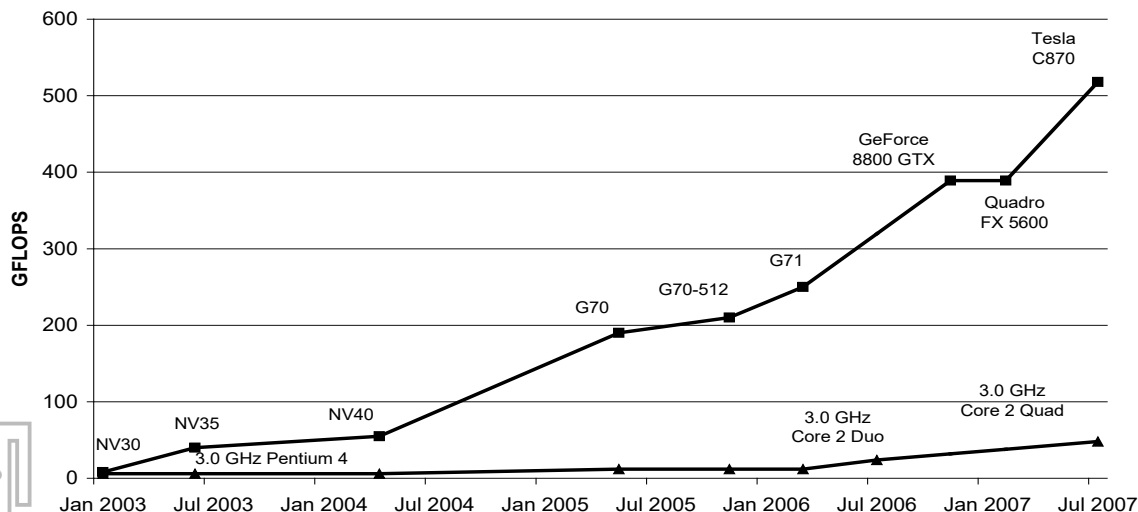
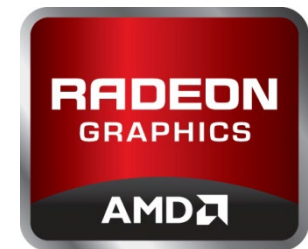
- Vector units can be combination of pipelined and arrayed functional units:



What is GPU?



- Graphics Processing Units
- GPU is a device to compute massive vertices, pixels, and general purpose data
- Feature
 - High availability
 - High computing performance
 - Low price of computing capability



GPU's History and Evolution



- **Early History**

- In early 90's, graphics are only performed by a video graphics array (VGA) controller
- In 1997, VGA controllers start to incorporate 3D acceleration functions
- In 2000, the term GPU is coined to denote that the graphics devices had become a processor

- **GPU Evolution**

- **Fixed-Function → Programmable**

- 1999, NVIDIA *Geforce 256*, Fixed-function vertex transform and pixel pipeline
- 2001, NVIDIA *Geforce 3*, 1st programmable vertex processor
- 2002, ATI *Radeon 9700*, 1st programmable pixel(fragment) processor

- **Non-unified Processor → Unified Processor**

- 2005, Microsoft *XBOX 360*, 1st unified shader architecture

- **Tesla GPU series released in 2007**

- **Fermi Architecture released in 2009**

- **Kepler Architecture released in 2012**

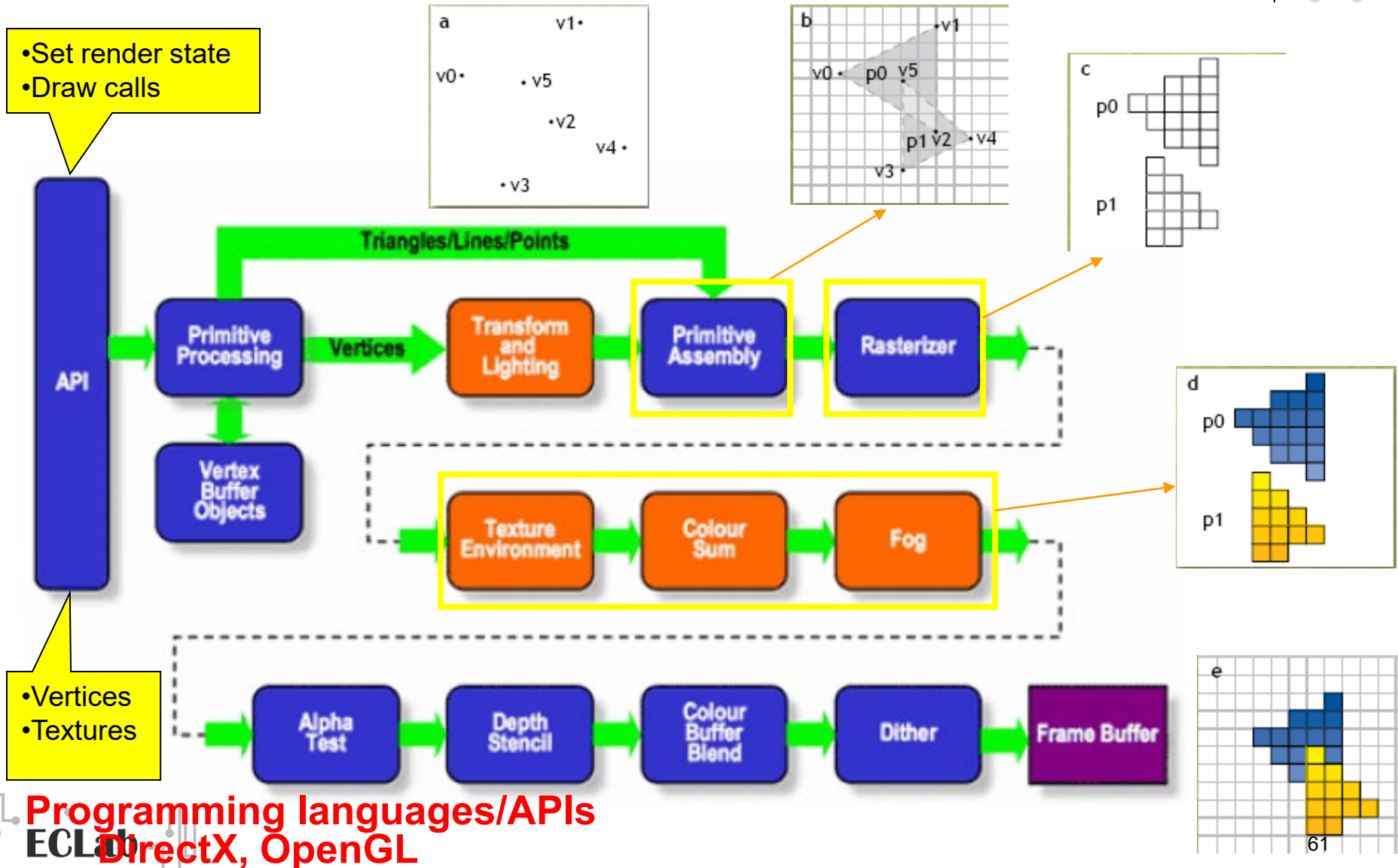
- **Maxwell Architecture released in 2014**

- **Pascal Architecture released in 2016 (16 nm FinFET process)**

- **Volta Architecture released in 2017 – Tensor Core for AI (12 nm process)**

- **Turing Architecture released in 2018**

Fixed-function 3D Graphics Pipeline

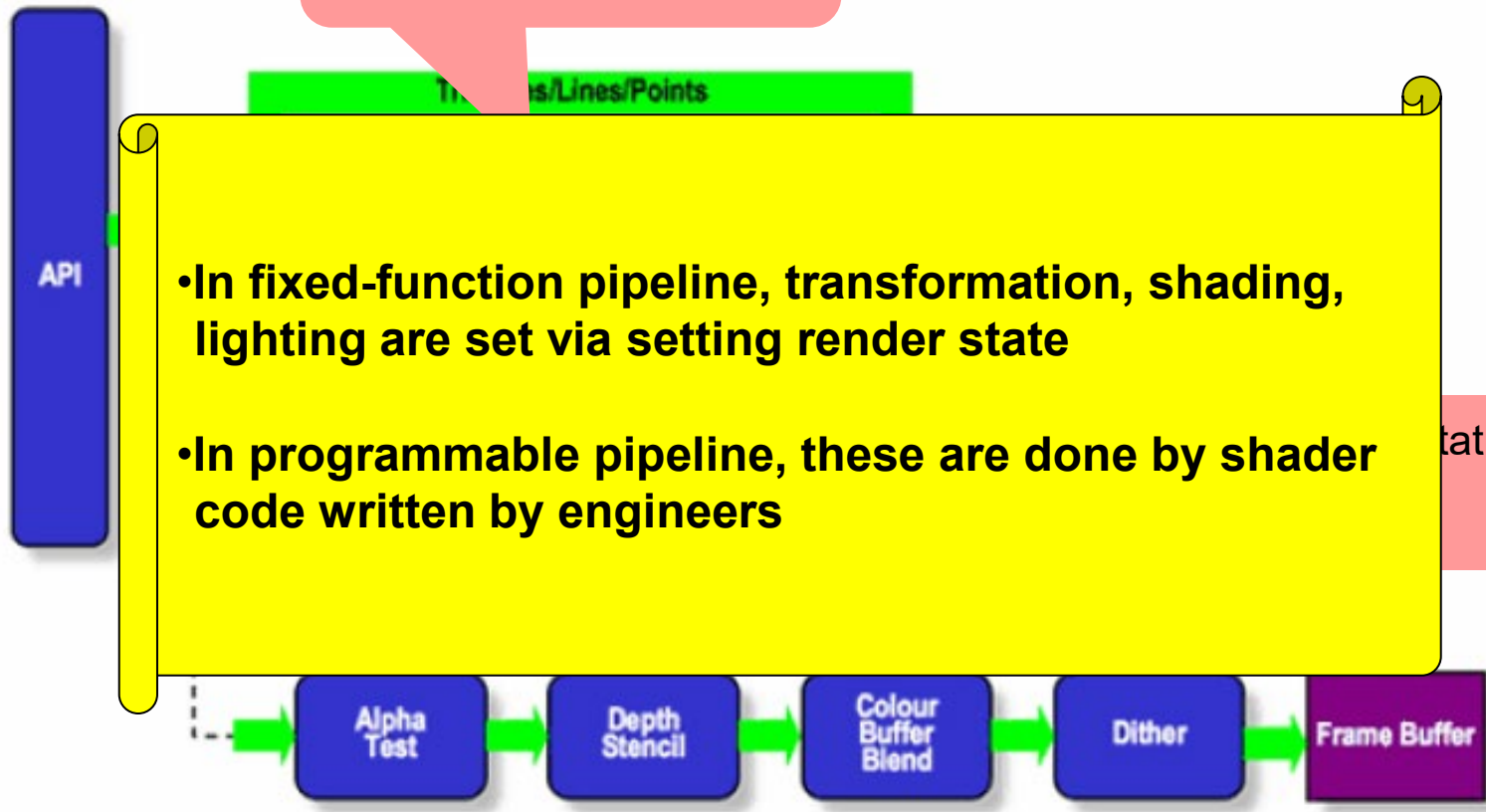


Programmable 3D Graphics Pipeline



High Level Shader Language
(HLSL)

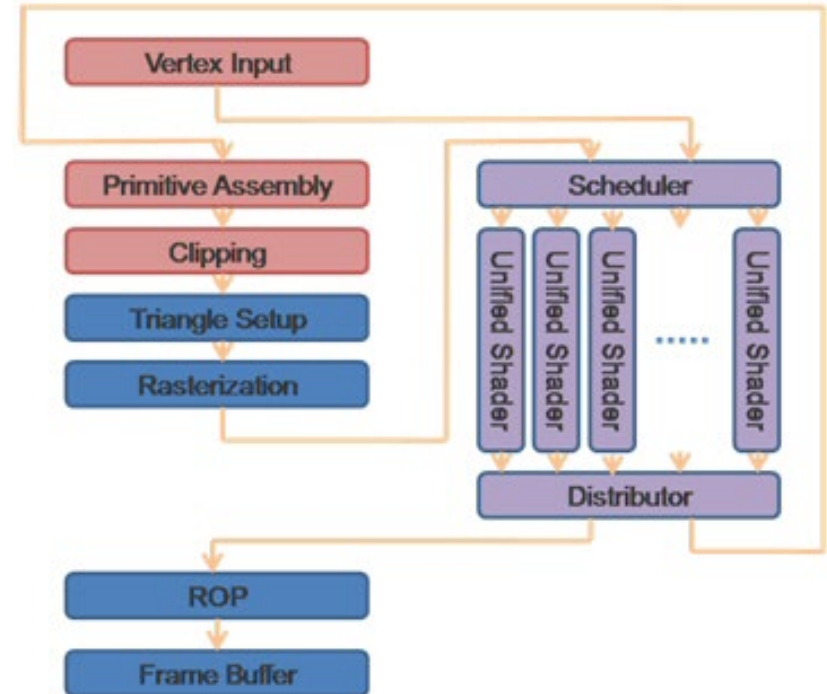
- Vertex computation



Unified Shader Architecture



- Use the same shader processors for all types of computation
 - Vertex threads
 - Pixel threads
 - Computation threads
- Advantage
 - Better resource utilization
 - Lower hardware complexity



Modern GPUs: A Computing Device



- GPUs have orders of magnitudes more computing power than CPU
- General-purpose tasks with high-degree of data level parallelism running on GPU outperform those on CPU
=> General-Purpose computing on GPU (**GPGPU**)
- GPGPU programming models
 - NVIDIA's CUDA
 - AMD's StreamSDK
 - OpenCL

GPGPU Performance	
Medical Imaging	300X
Molecular Dynamics	150X
SPICE	130X
Fourier Transform	130X
Fluid Dynamics	100X

Fundamental Architectural Differences between CPU & GPU

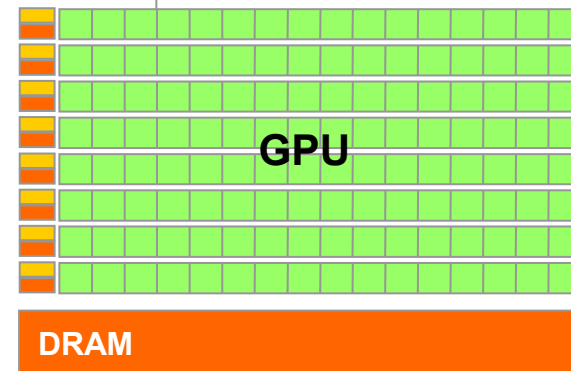
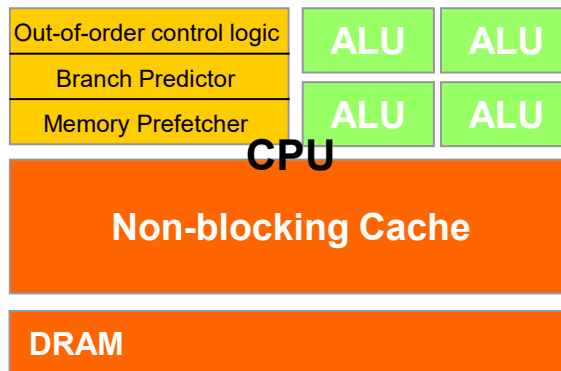


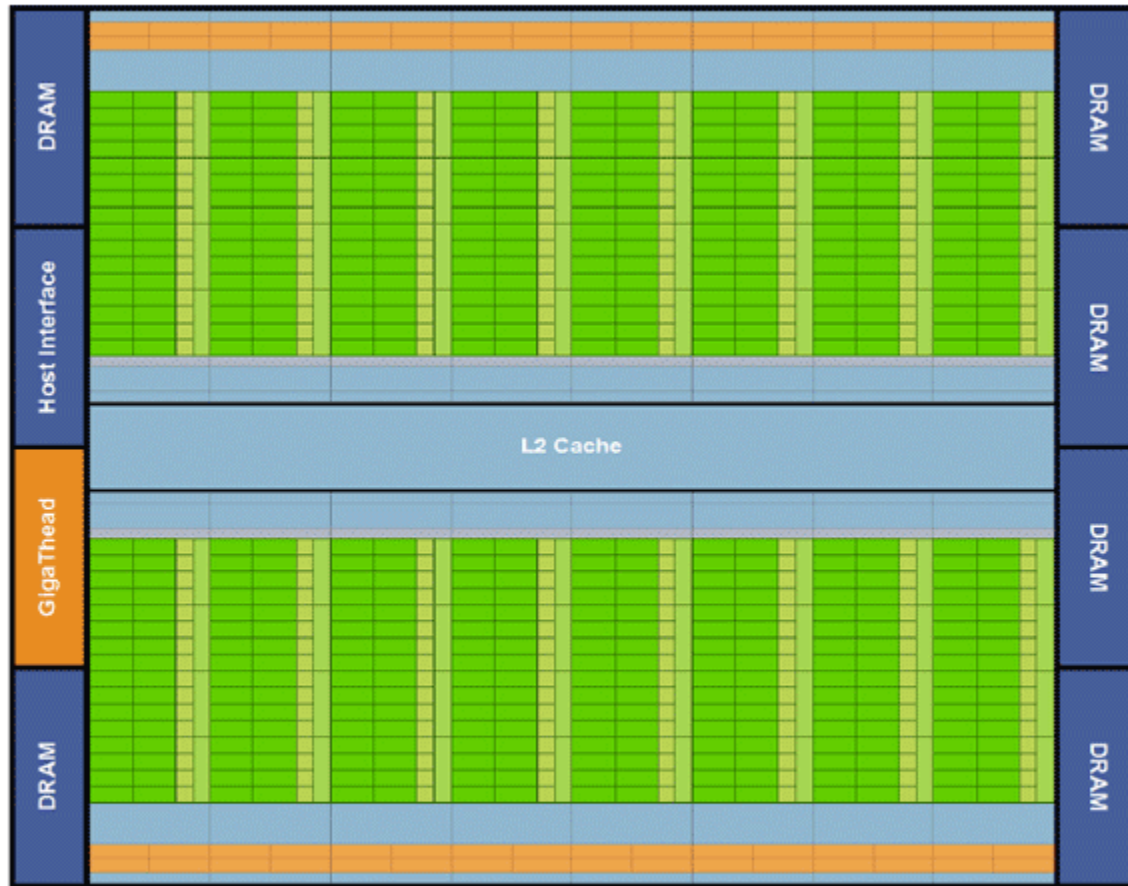
- **Multi-core CPU**

- Coarse-grain, heavyweight threads
- Memory latency is resolved through large on-chip caches & out-of-order execution

- **Modern GPU**

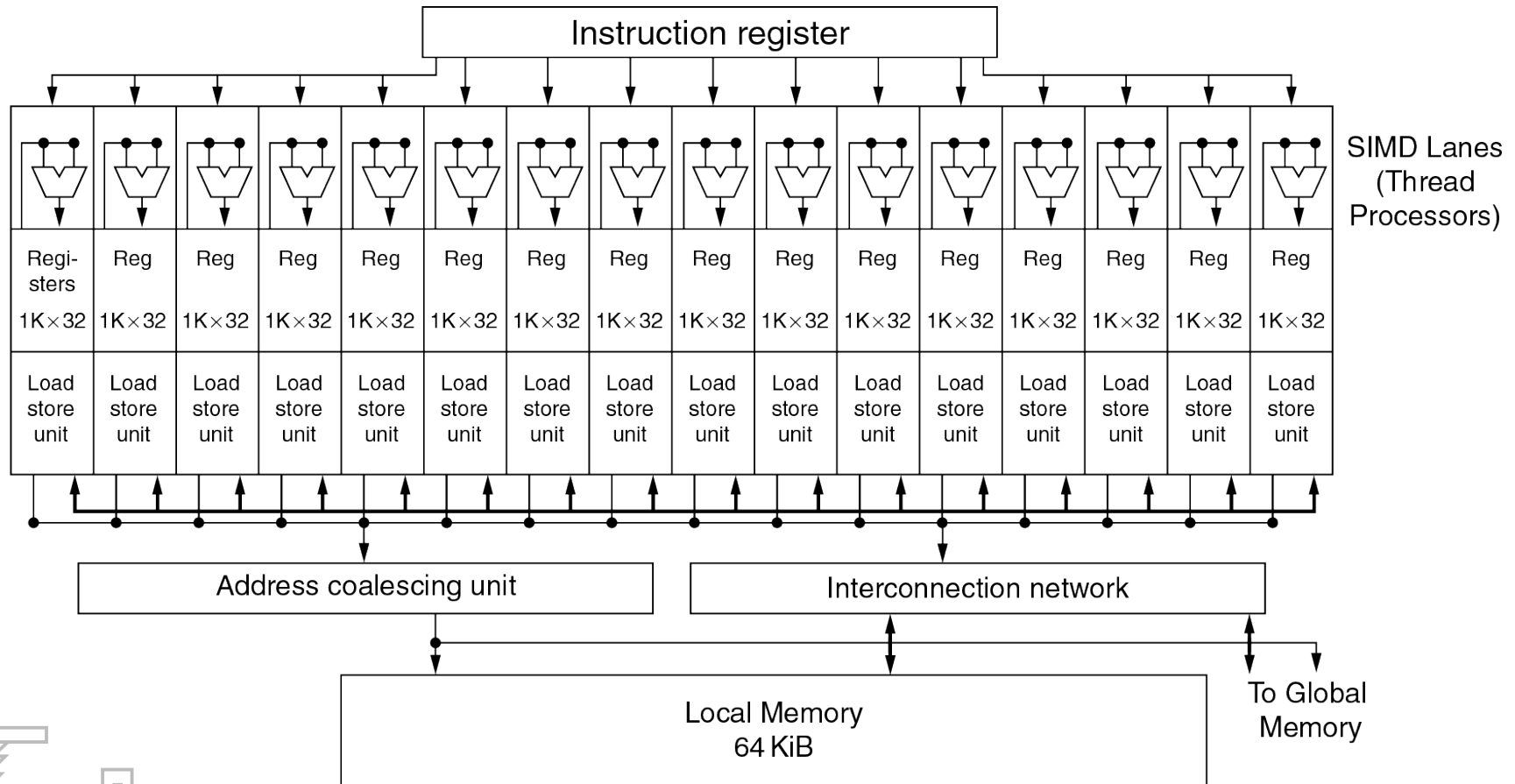
- Fine-grain, lightweight threads
- Exploit thread-level parallelism for hiding latency





Fermi's 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).

SIMD processor



SIMT Execution Model of GPUs

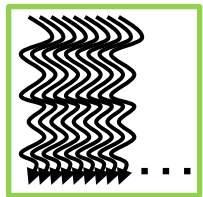


- **SIMT (Single Instruction Multiple Threads)**

- **Warp**

- A group of threads (pixel, vertex, compute...)
- Basic scheduling/execution unit
- Common PC value

Thread block



1 ~ x thread ID

1 ~ 32 thread ID



Warp 1

33 ~ 64 thread ID



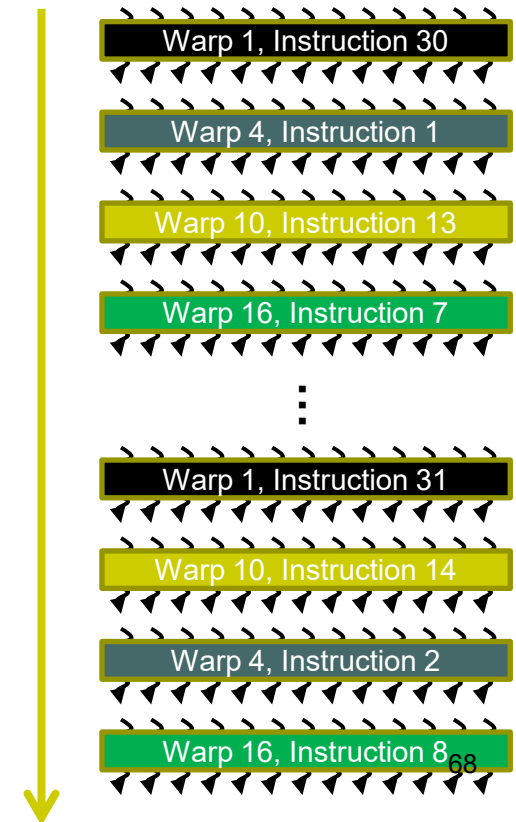
Warp 2

⋮

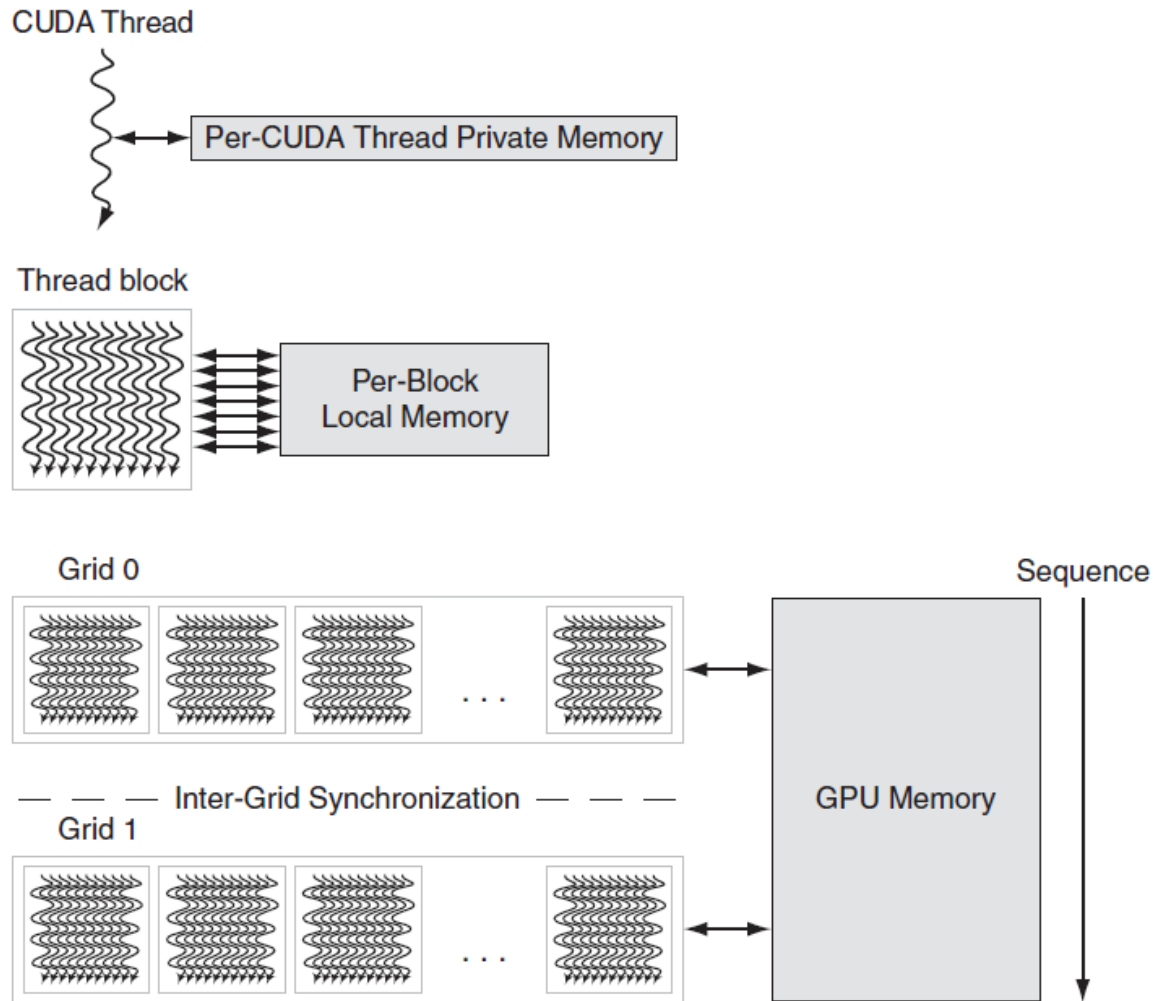


Warp n

Time



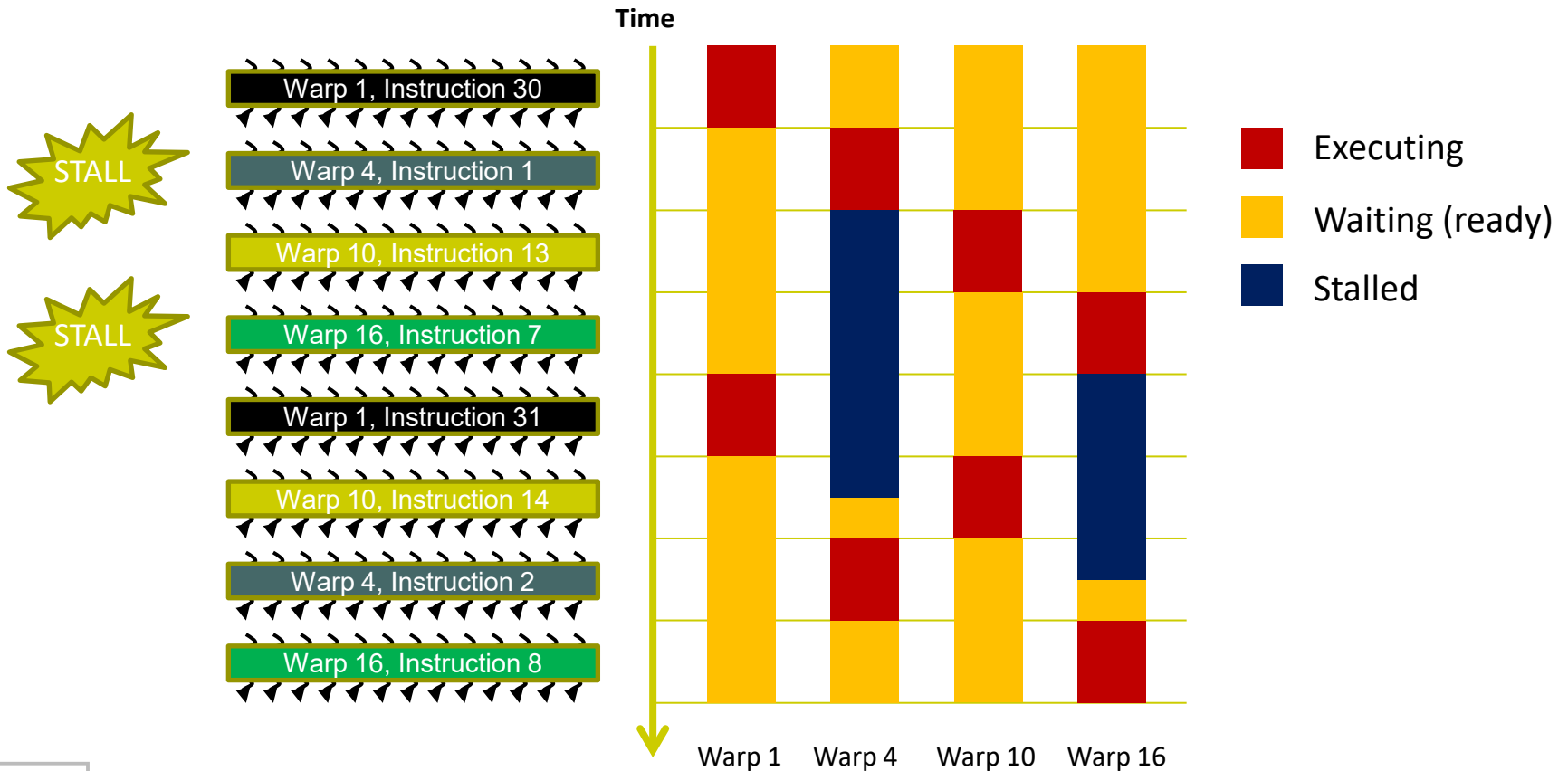
GPU Memory Structures



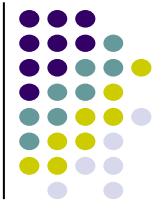
Latency Hiding



- Interleaved warp execution



Volta



$D =$

$$\begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix}$$

FP16 or FP32

FP16

$$\begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix}$$

FP16

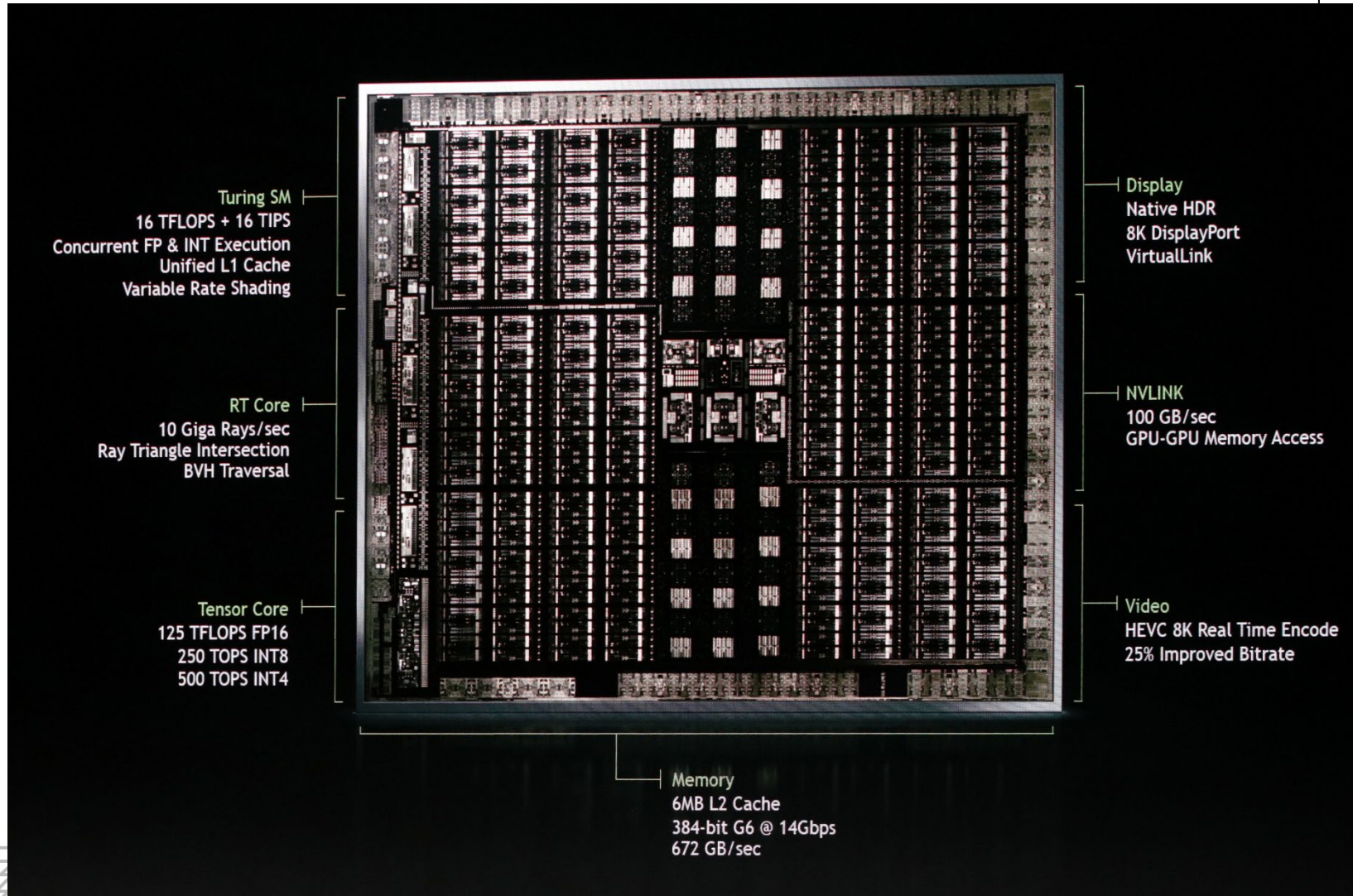
+

$$\begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

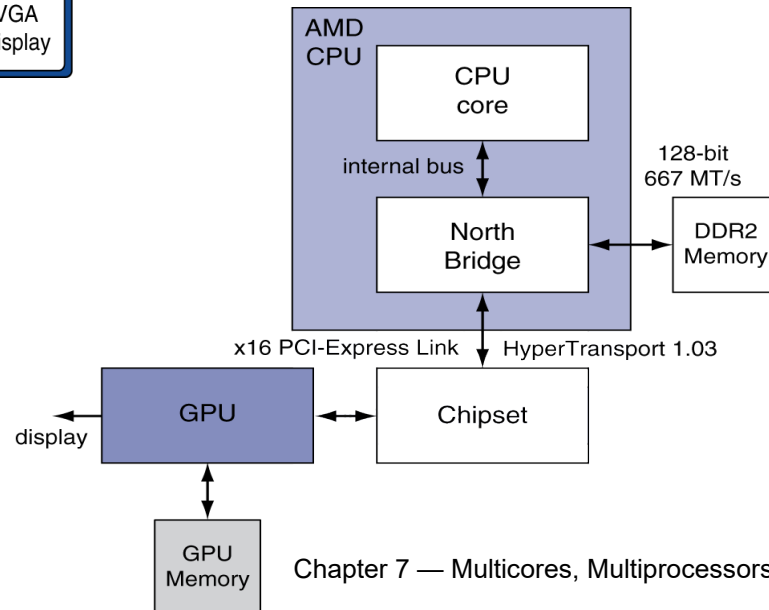
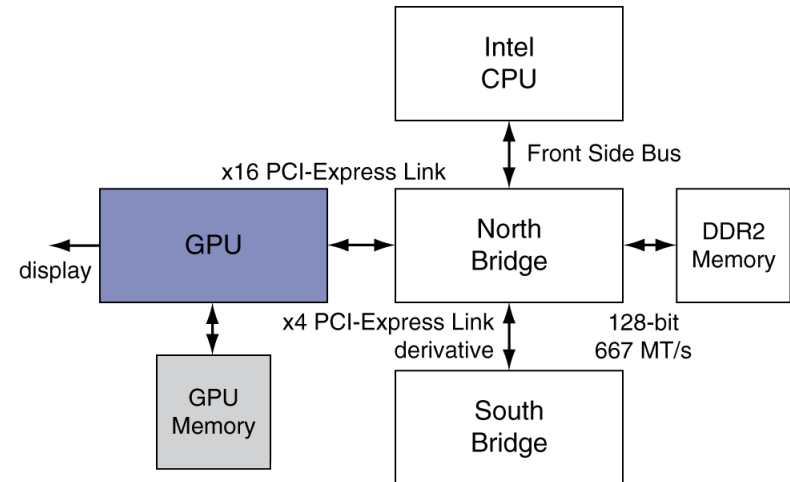
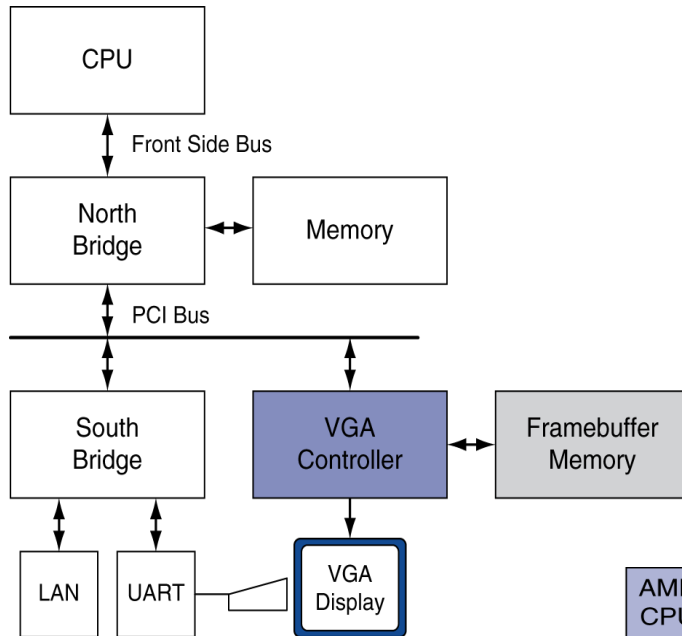
FP16 or FP32



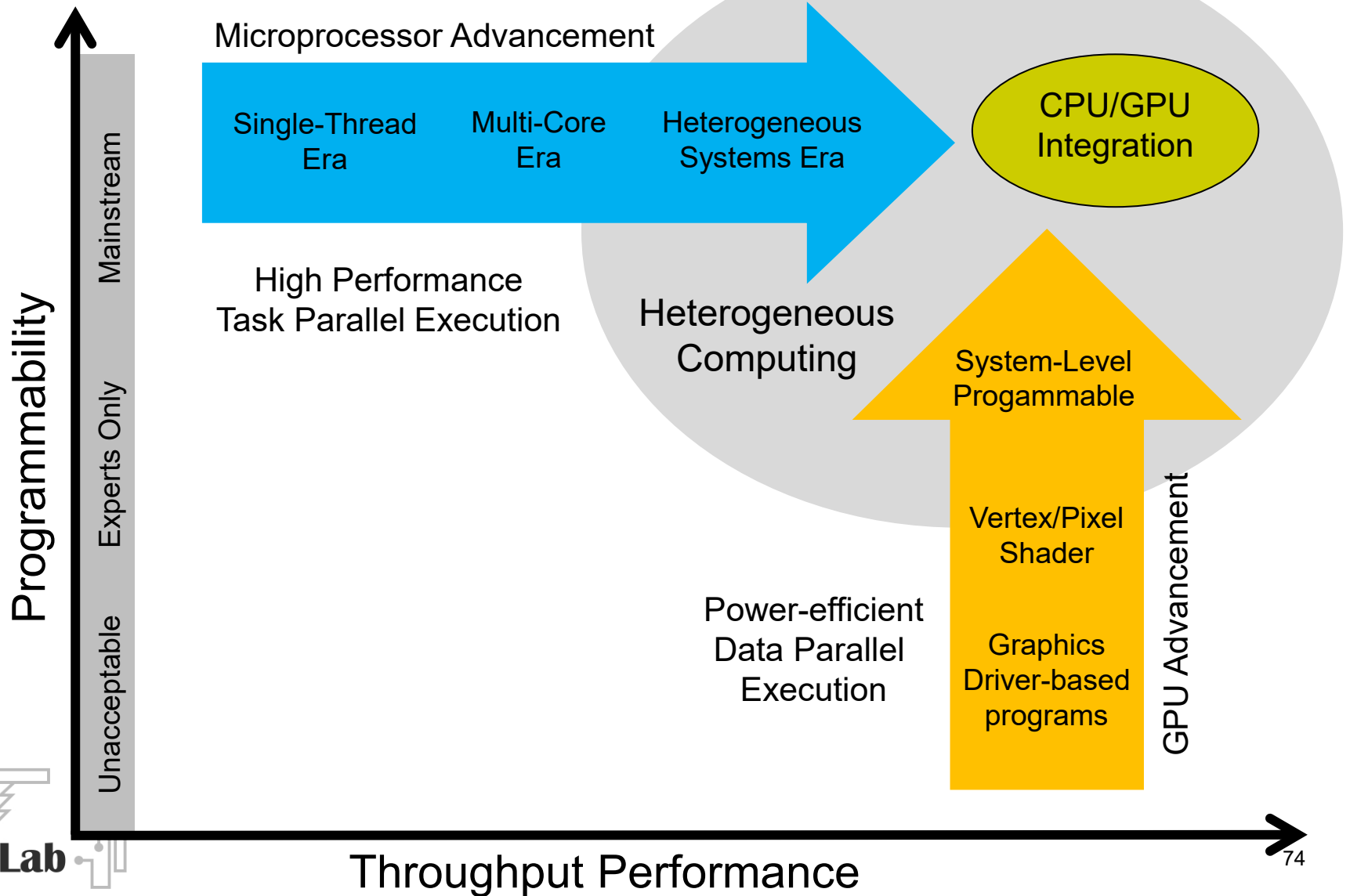
Turing



Graphics in the System



CPU/GPU Integration: CPU's Advancement Meets GPU's



Heterogeneous Computing ~ 2011

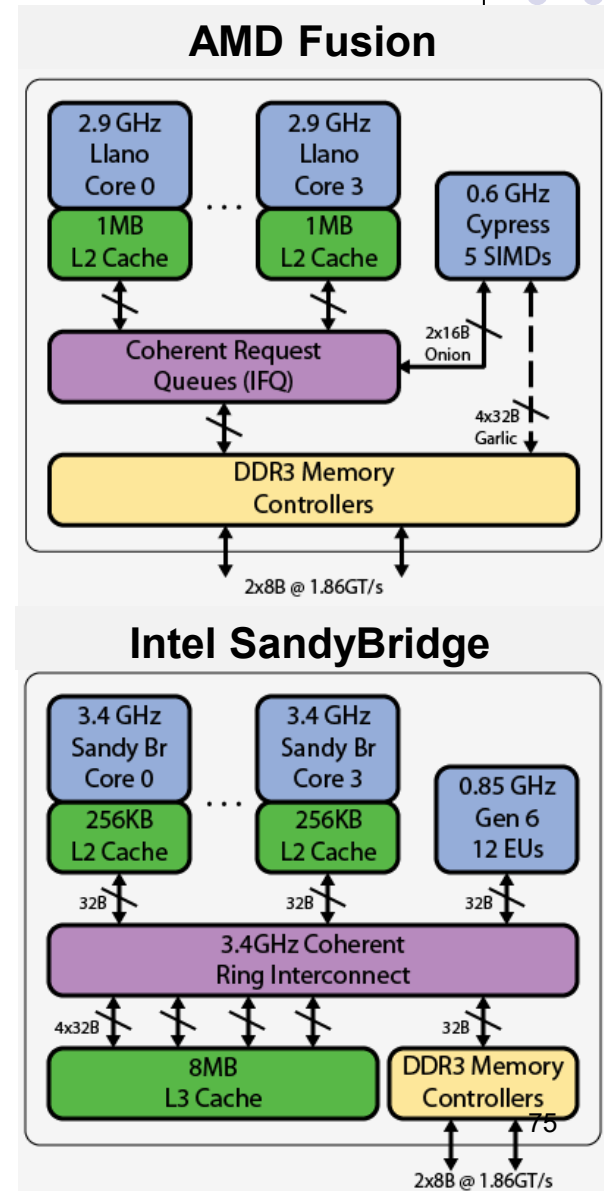


- **Intel SandyBridge**

- Shared last-level cache (LLC) and main memory

- **AMD Fusion APU (Accelerated Processing Unit)**

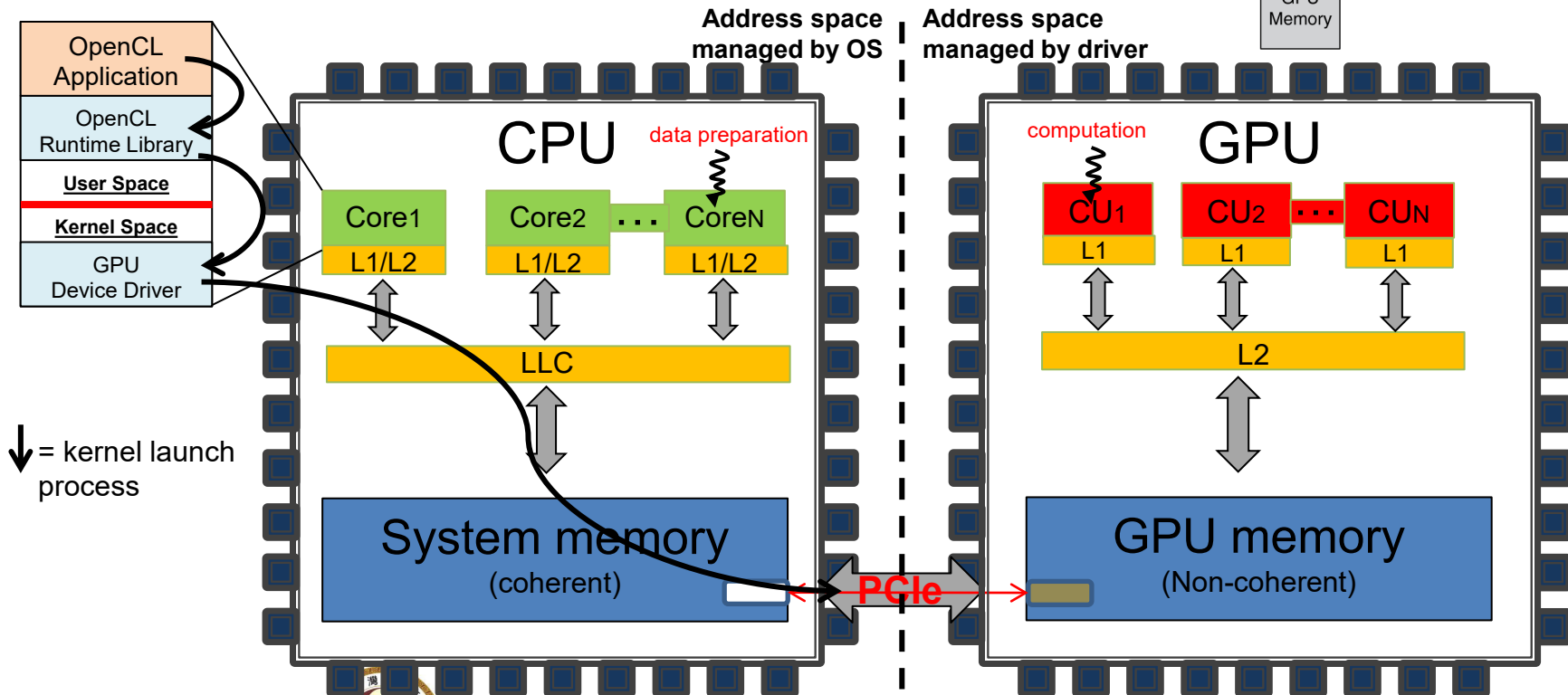
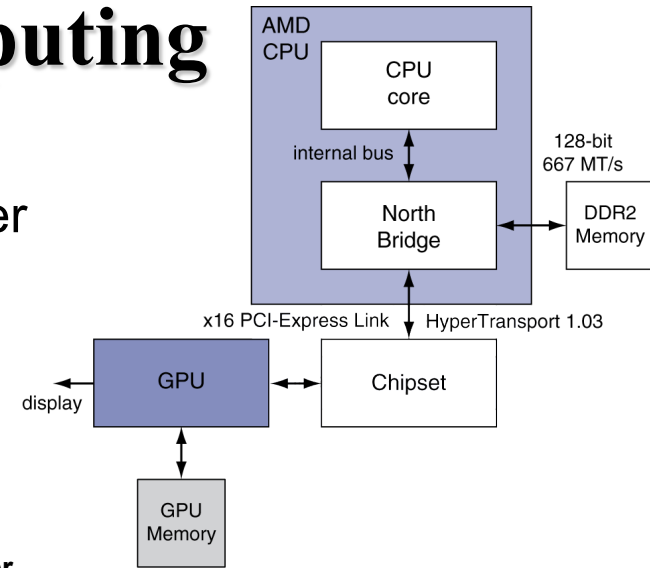
- Shared main memory



Evolution of Heterogeneous Computing

■ Dedicated GPU

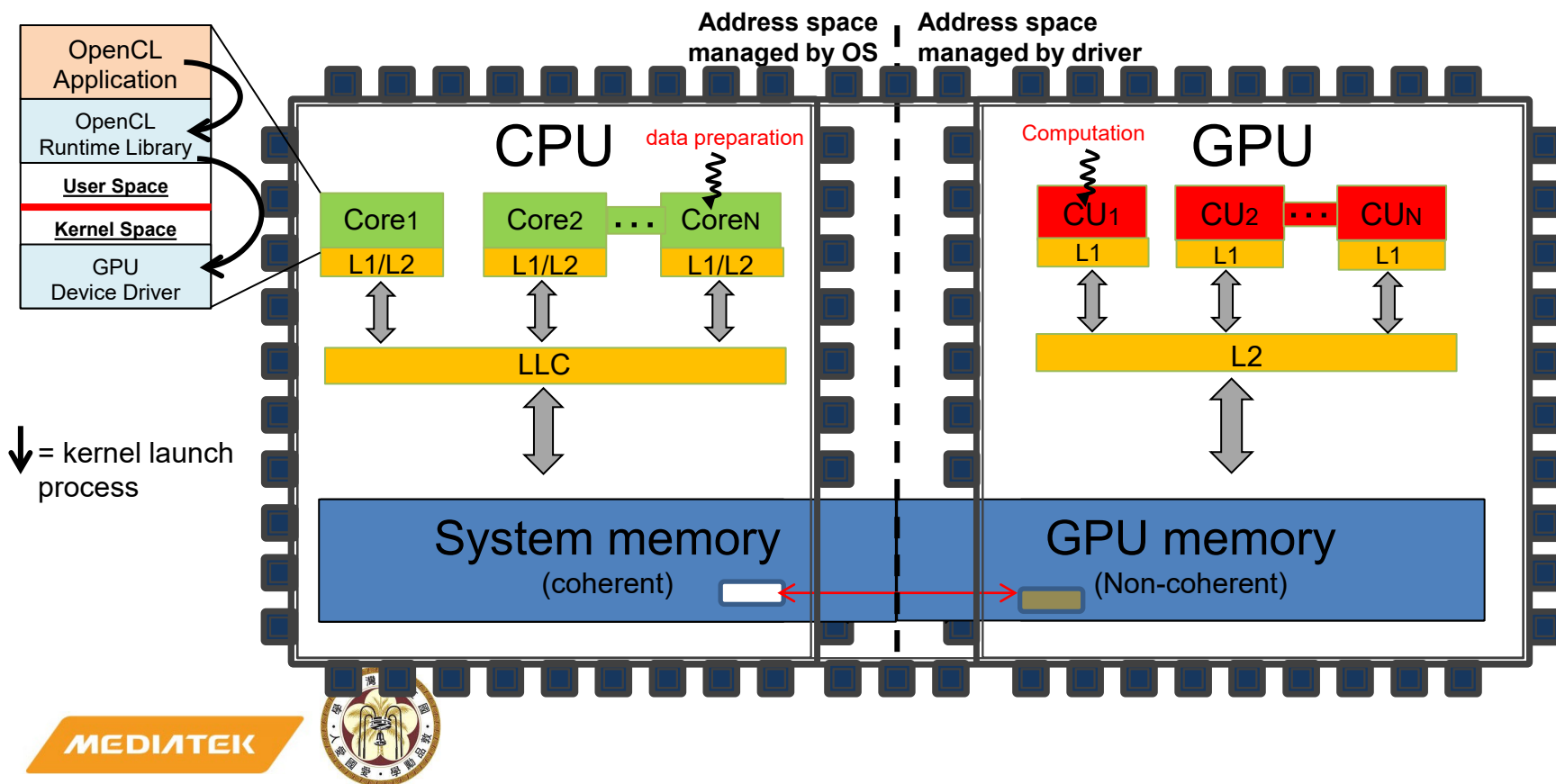
- GPU kernel is launched through the device driver
- Separate CPU/GPU address space
- Separate system/GPU memory
- Data copy between CPU/GPU via PCIe



Evolution of Heterogeneous Computing

■ Integrated GPU architecture

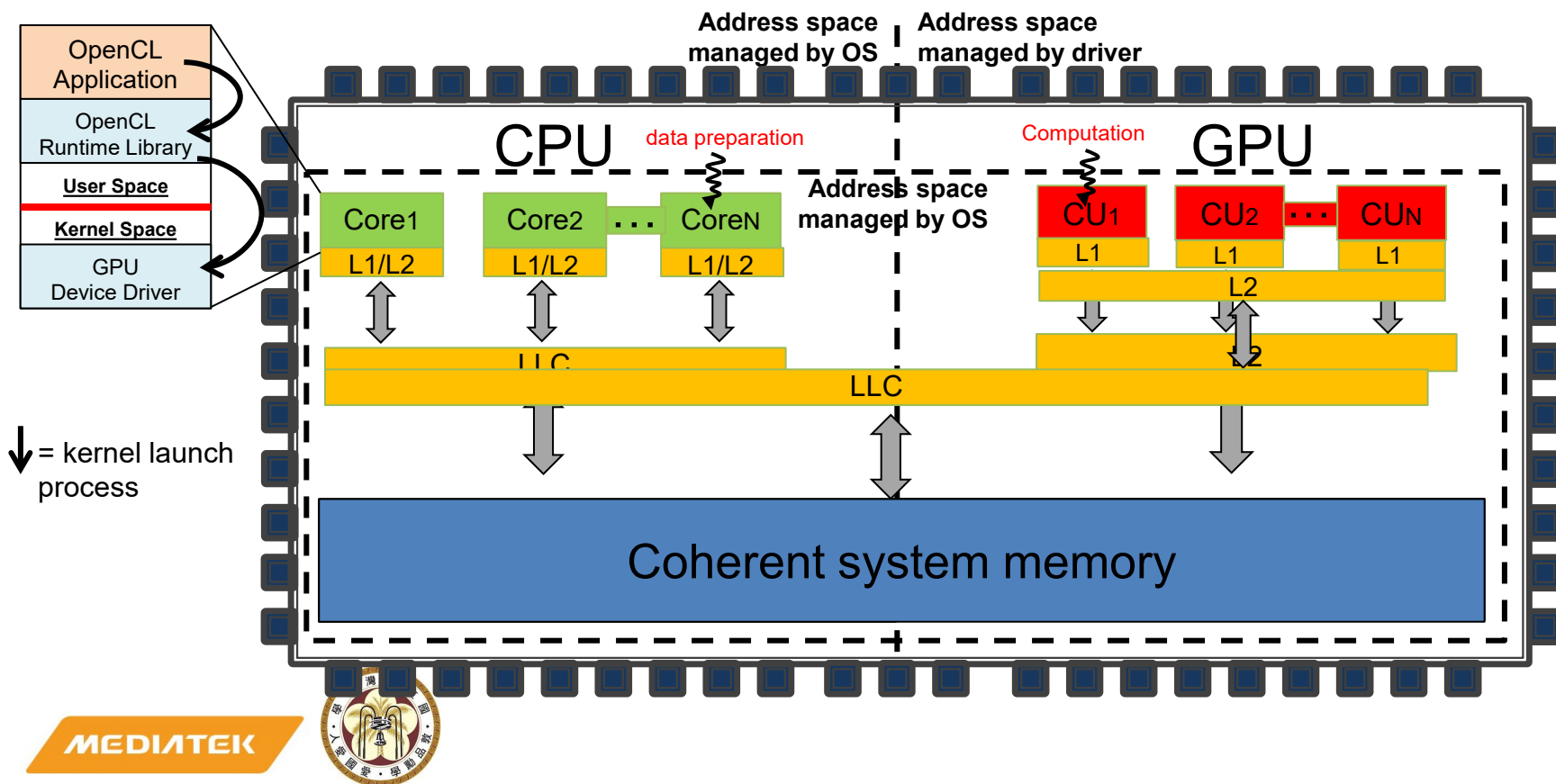
- GPU kernel is launched through the device driver
- Separate CPU/GPU address space
- Separate system/GPU memory
- Data copy between CPU/GPU *via memory bus*



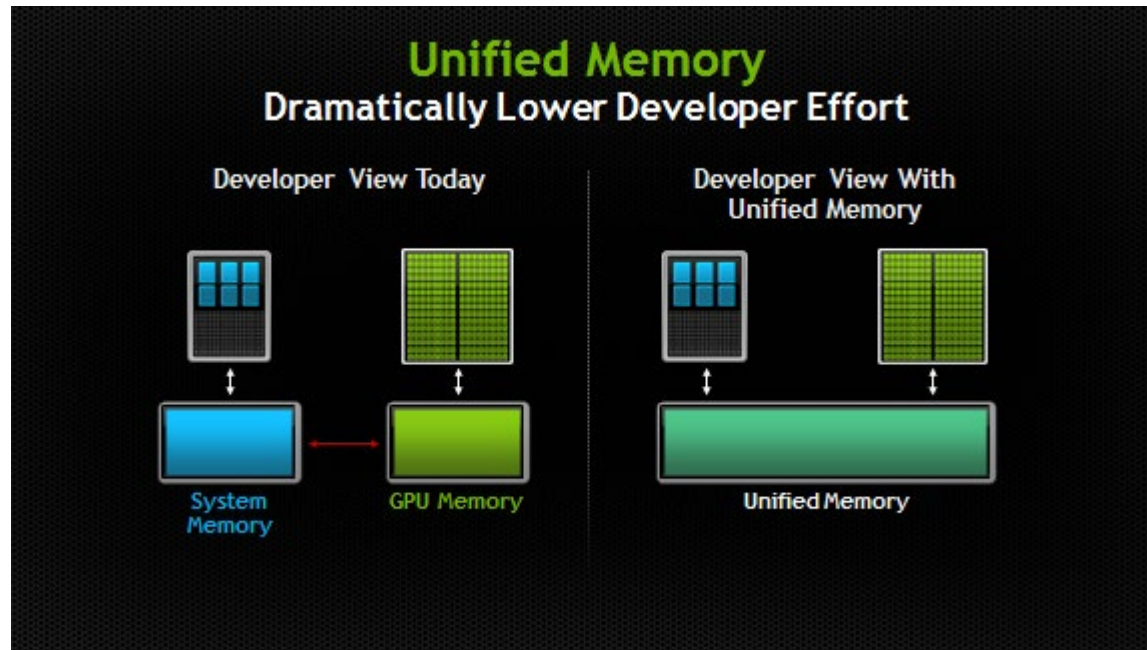
Evolution of Heterogeneous Computing

■ Integrated GPU architecture

- GPU kernel is launched through the device driver
- Unified CPU/GPU address space (managed by OS)
- Unified system/GPU memory
- No data copy - data can be retrieved by pointer passing



CUDA Unified Memory



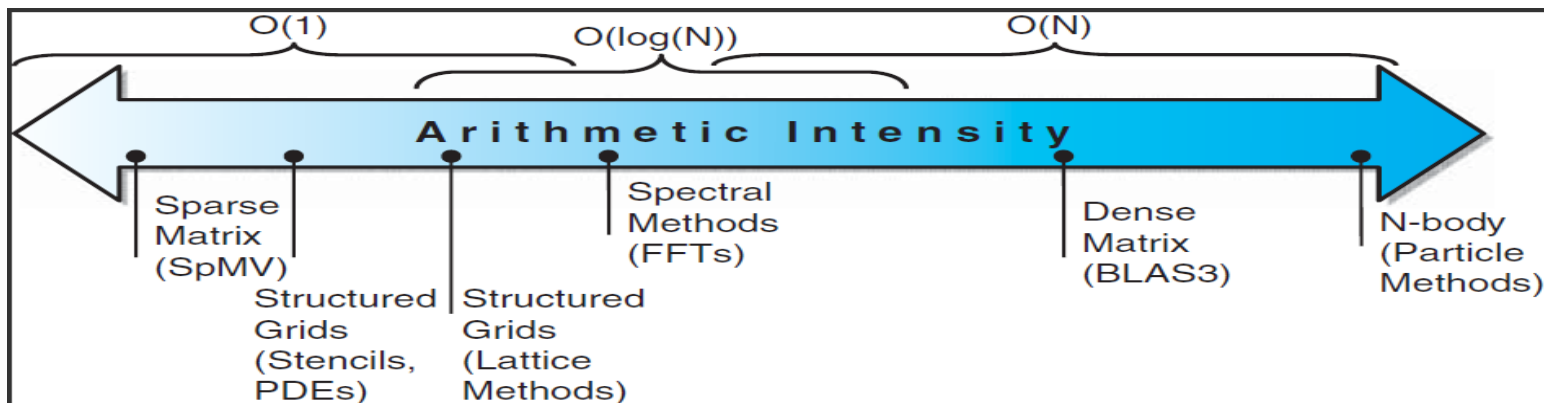
Parallel Benchmarks

- **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

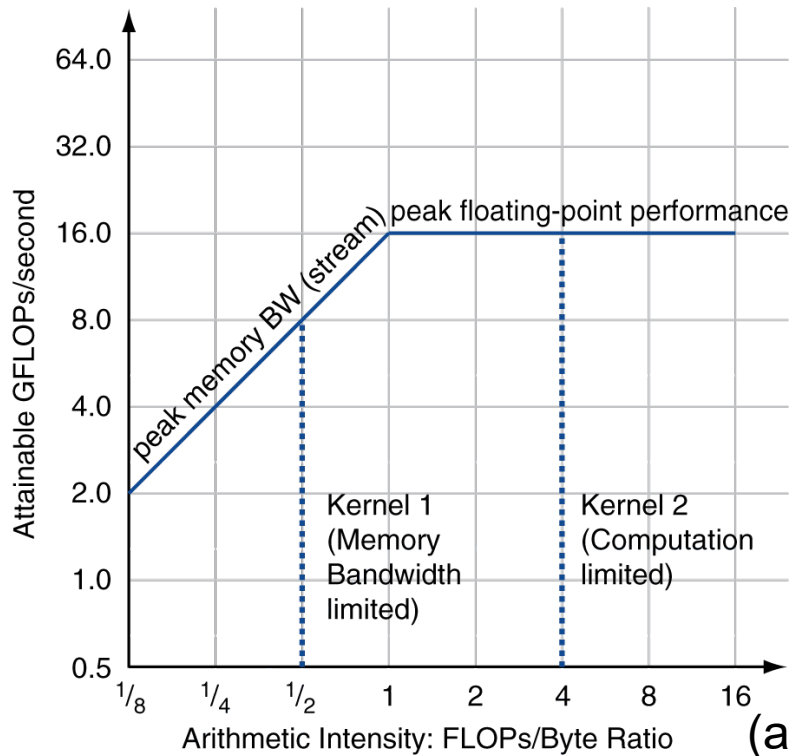


Modeling Performance~ Roofline Model

- Target performance metric
 - Achievable GFLOPs/sec
- Hardware: for a given computer, determine
 - Peak GFLOPS (from data sheet)
 - Peak memory bytes/sec (using Stream benchmark)
- Software: Arithmetic intensity of a kernel
 - FLOPs per byte of memory accessed



Roofline : A Simple Performance Model



$$\frac{\text{Floating-Point Ops/sec}}{\text{Floating-Point Ops/ byte}} = \text{Bytes/sec}$$

Arithmetic Intensity

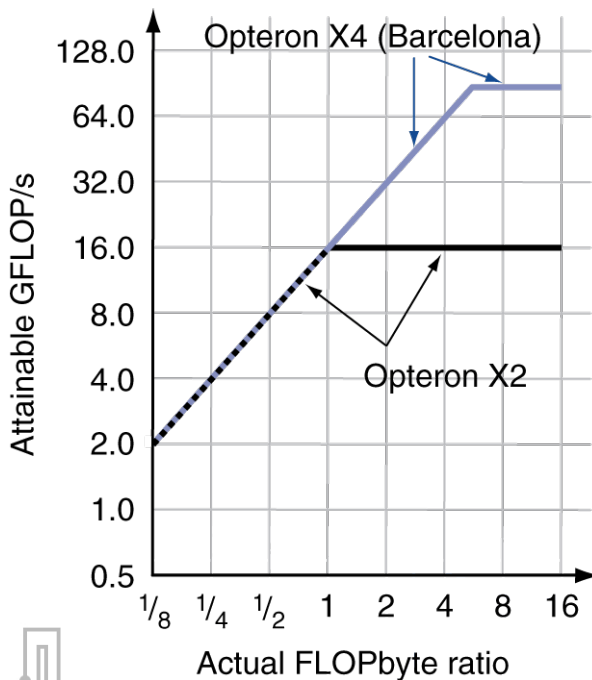
(assuming 16GB/sec peak bandwidth)

$$\text{Attainable GFLOPs/sec} = \text{Min} (\text{Peak Memory BW} \times \text{Arithmetic Intensity}, \text{Peak FP Performance})$$

Comparing Systems



- **Example: Opteron X2 vs. Opteron X4**
 - 2-core vs. 4-core, 2x 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