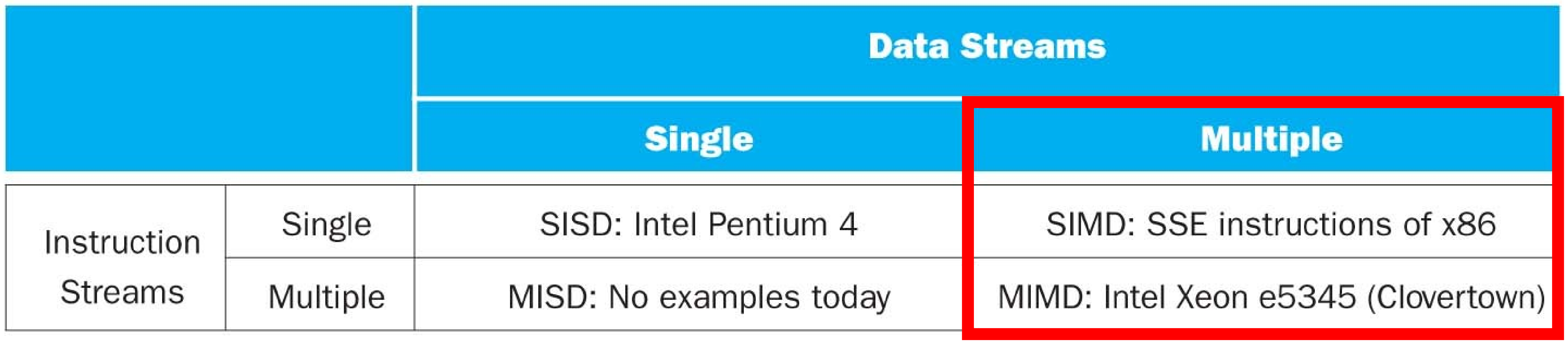
Flynn Taxonomy, 1966



		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 (Clovertown)

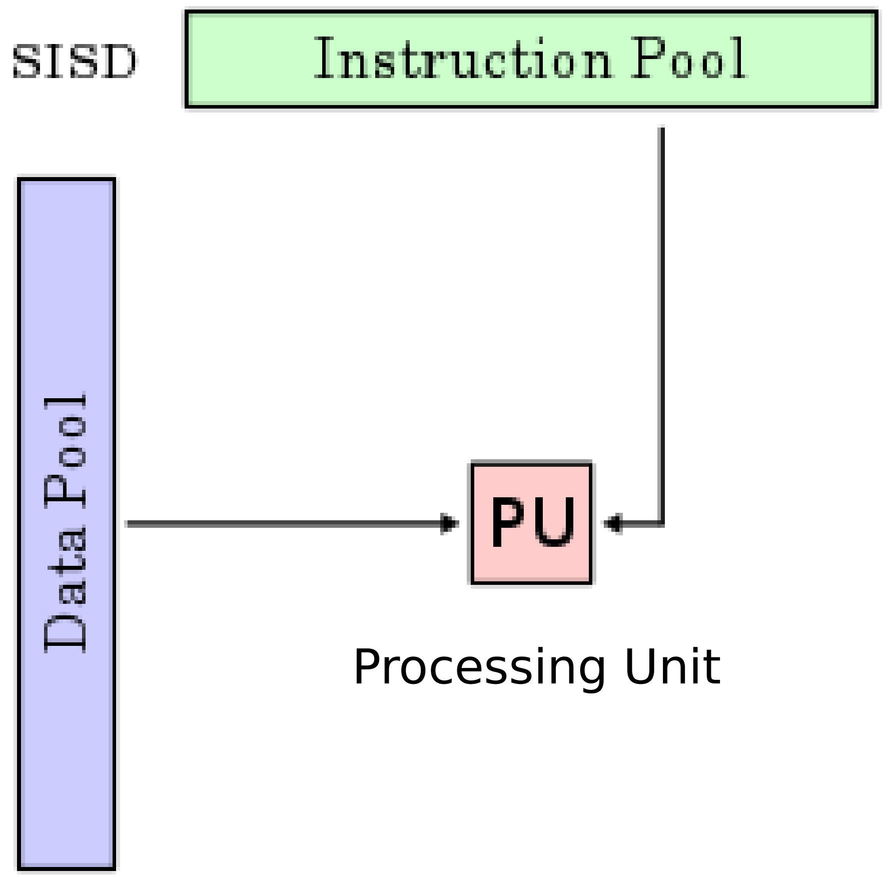
- SIMD and MIMD are currently the most common parallelism in architectures – usually both in same system!
- Most common parallel processing programming style: Single Program Multiple Data (“SPMD”)
  - Single program that runs on all processors of a MIMD
  - Cross-processor execution coordination using synchronization primitives
  - Found in GPUs

Flynn’s Taxonomy

Mike Flynn, “Very High-Speed Computing Systems,” Proc. of IEEE, 1966

- **S**ingle **I**nstruction **S**ingle **D**ata
- **S**ingle **I**nstruction **M**ultiple **D**ata
  - Array Processor
  - Vector Processor
  - GPU
- **M**ultiple **I**nstruction **S**ingle **D**ata
  - Closest form: systolic array processor, streaming processor
- **M**ultiple **I**nstruction **M**ultiple **D**ata
  - Multiprocessor
  - Multithreaded processor

Single-Instruction/Single-Data Stream (SISD)

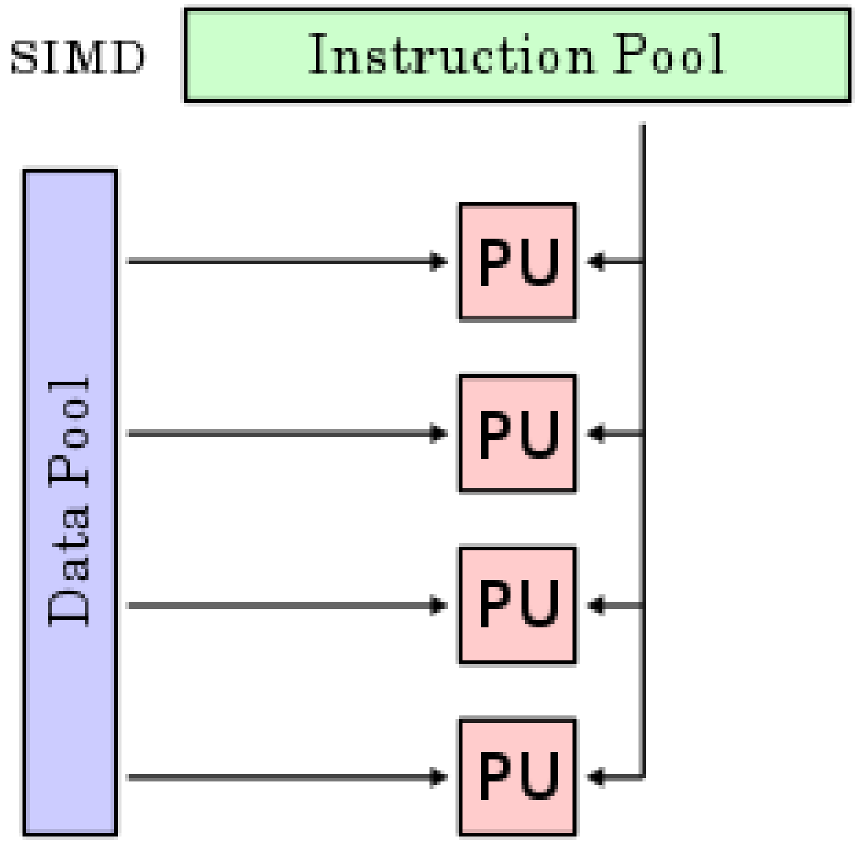


- Sequential computer that exploits no parallelism in either the instruction or data streams. Examples of SISD architecture are traditional uniprocessor machines
  - E.g. our trusted RISC-V pipeline

Instruction Level Parallelism (ILP)

- Improve processor performance by having multiple processor components or functional units simultaneously executing instructions.
- Pipelining - functional units are arranged in *stages*.
- Multiple issue - multiple instructions *might* be simultaneously initiated with resource duplication.
  - VLIW – Very long instruction word - functional units are scheduled at compile time (static scheduling).
  - Superscalar - functional units are scheduled at run-time (dynamic scheduling)

Single-Instruction/Multiple-Data Stream (SIMD or “sim-dee”)



- SIMD computer exploits multiple data streams against a single instruction stream to operations that may be naturally parallelized, e.g., Intel SIMD instruction extensions or NVIDIA Graphics Processing Unit (GPU)

SIMD drawbacks

- All ALUs are required to execute the same instruction, or remain idle.
- In classic design, they must also operate synchronously.
- The ALUs have no instruction storage.
- Efficient for large data parallel problems, but not other types of more complex parallel problems

Vector machines (SIMD)

- Operate on arrays or vectors of data while conventional CPU’s operate on individual data elements or scalars.
  - Also known as *array processors*
- Vector registers.
  - Capable of storing a vector of operands and operating simultaneously on their contents.
- Vectorized and pipelined functional units.
  - The same operation is applied to each element in the vector (or pairs of elements).

Vector machines: Pros and Cons

- Pros
  - Fast and Easy to use.
  - Vectorizing compilers are good at identifying code to exploit.
  - Compilers also can provide information about code that cannot be vectorized.
  - High memory bandwidth.
  - Uses every item in a cache line.
- Cons
  - They don’t handle irregular data structures as well as other parallel architectures.
  - A very finite limit to their ability to handle ever larger problems. (scalability)

VLIW Advantages & Disadvantages (versus Superscalar)

Advantages

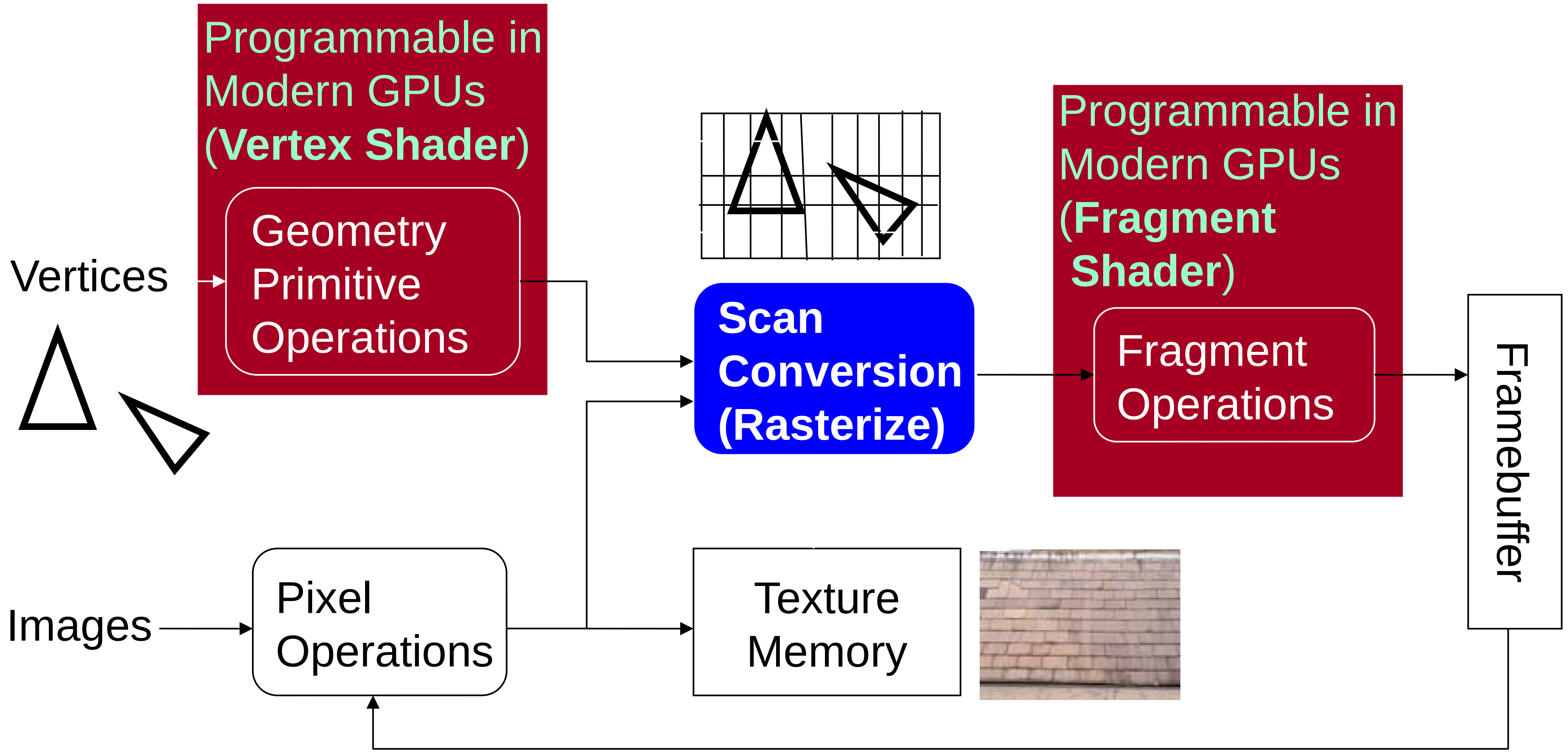
- Simpler hardware (potentially less power hungry)
- E.g. Transmeta Crusoe processor was a VLIW that converted x86 code to VLIW code using dynamic binary translation
- Many processors aimed specifically at digital signal processing (DSP) are VLIW
    - DSP applications usually contain lots of instruction level parallelism in loops that execute many times
- Potentially more scalable
    - Allow more instructions per VLIW bundle and add more functional units

VLIW Advantages & Disadvantages (versus Superscalar)

Disadvantages

- Programmer/compiler complexity and longer compilation times
  - Deep pipelines and long latencies can be confusing (making peak performance elusive)
- Lock step operation, i.e., on hazard all future issues stall until hazard is resolved (hence need for predication)
- Object (binary) code incompatibility
- Needs lots of program memory bandwidth
- Code bloat
  - Nops are a waste of program memory space
  - Speculative code needs additional fix-up code to deal with cases where speculation has gone wrong
  - Register renaming needs free registers, so VLIWs usually have more programmer-visible registers than other architectures
  - Loop unrolling (and software pipelining) to expose more ILP uses more program memory space

GPU Programmable Shaders

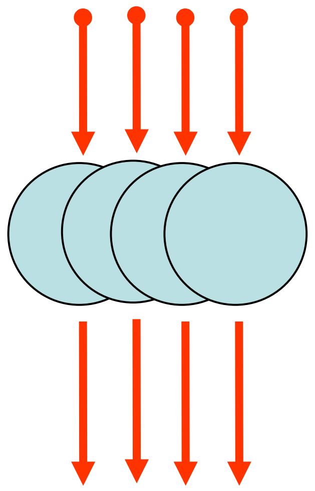

Programmable in Modern GPUs (**Vertex Shader**)
Geometry Primitive Operations
Vertices
Scan Conversion (Rasterize)
Programmable in Modern GPUs (**Fragment Shader**)
Fragment Operations
Framebuffer
Images
Pixel Operations
Texture Memory

Traditional Approach: Fixed function pipeline (state machine)
New Development (2003-): Programmable pipeline

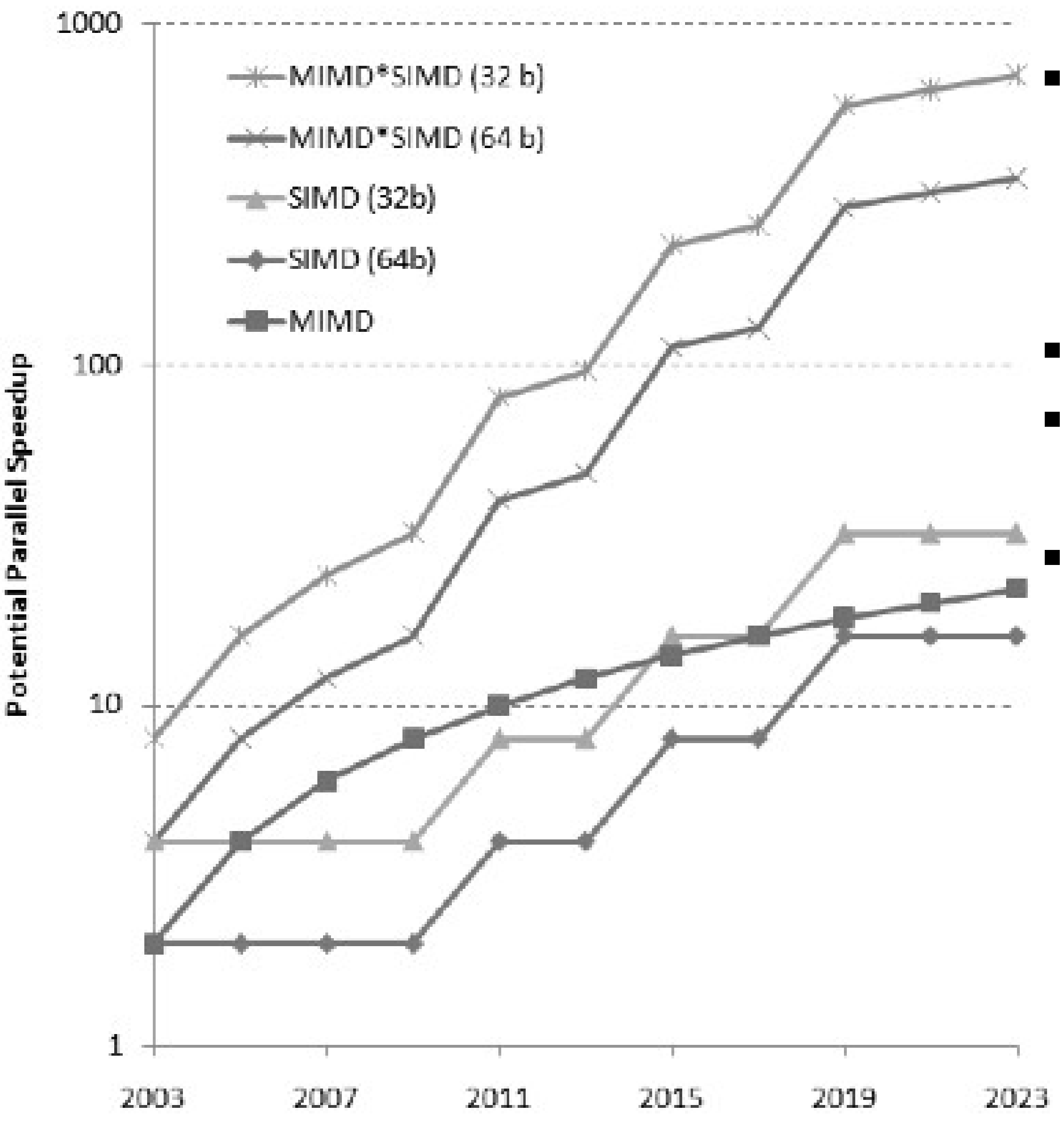
GPU: Data Parallel

- Each fragment shaded independently
  - No dependencies between fragments
    - Temporary registers are zeroed
    - No static variables
    - No Read-Modify-Write textures
  - Multiple “pixel pipes”
- Data Parallelism
  - Support ALU heavy architectures
  - Hide Memory Latency

[Torborg and Kajiya 96, Anderson et al. 97, Igehy et al. 98]



DLP important for conventional CPUs



- Prediction for x86 processors, from Hennessy & Patterson, 5th edition
  - ***Note: Educated guess, not Intel product plans!***
- TLP: 2+ cores / 2 years
- DLP: 2x width / 4 years

- DLP will account for more mainstream parallelism growth than TLP in next decade.
  - SIMD –single-instruction multiple-data (DLP)
  - MIMD- multiple-instruction multiple-data (TLP)

Importance of Data Parallelism

- GPUs are designed for graphics
  - Highly parallel tasks
- Process *independent* vertices & fragments
  - No shared or static data
  - No read-modify-write buffers
- Data-parallel processing
  - GPU architecture is ALU-heavy
  - Performance depends on *arithmetic intensity*
    - Computation / Bandwidth ratio
  - Hide memory latency with more computation

Arithmetic Intensity

Lots of operations per word transferred

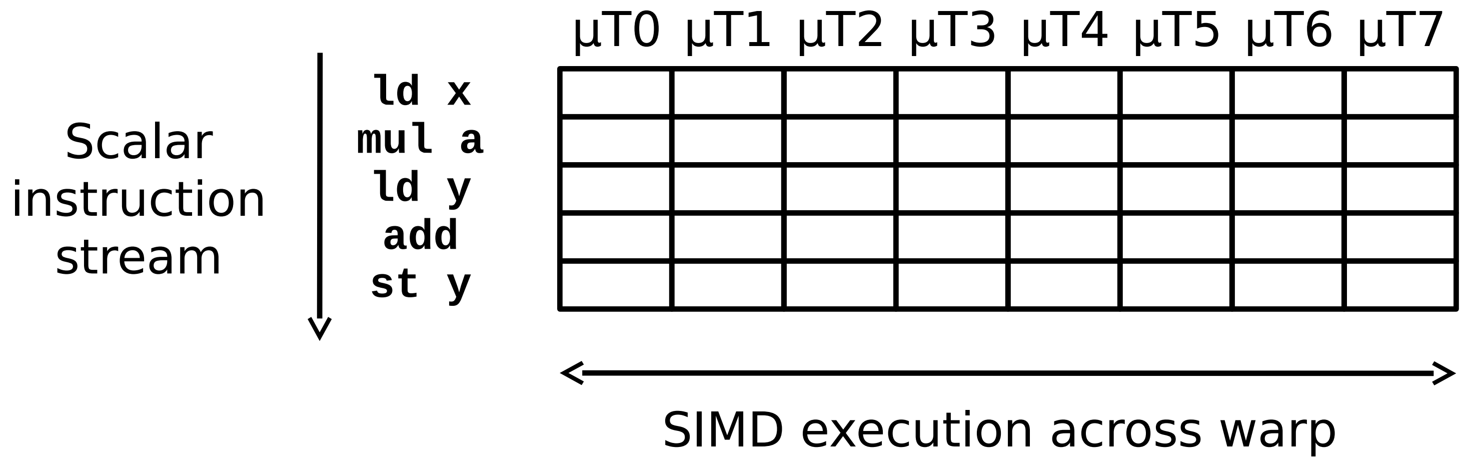
Graphics pipeline

- Vertex
  - Bandwidth: 1 triangle = 32 bytes;
  - Operations: 100-500 f32-ops / triangle
- Rasterization
  - Create 16-32 fragments per triangle
- Fragment
  - Bandwidth: 1 fragment = 10 bytes
  - Operations: 300-1000 i8-ops/fragment

Courtesy of Pat Hanrahan

“Single Instruction, Multiple Thread”

- GPUs use a **SIMT** model, where individual scalar instruction streams for each CUDA thread are grouped together for SIMD execution on hardware (Nvidia groups 32 CUDA threads into a *warp*)



Implications of SIMT model

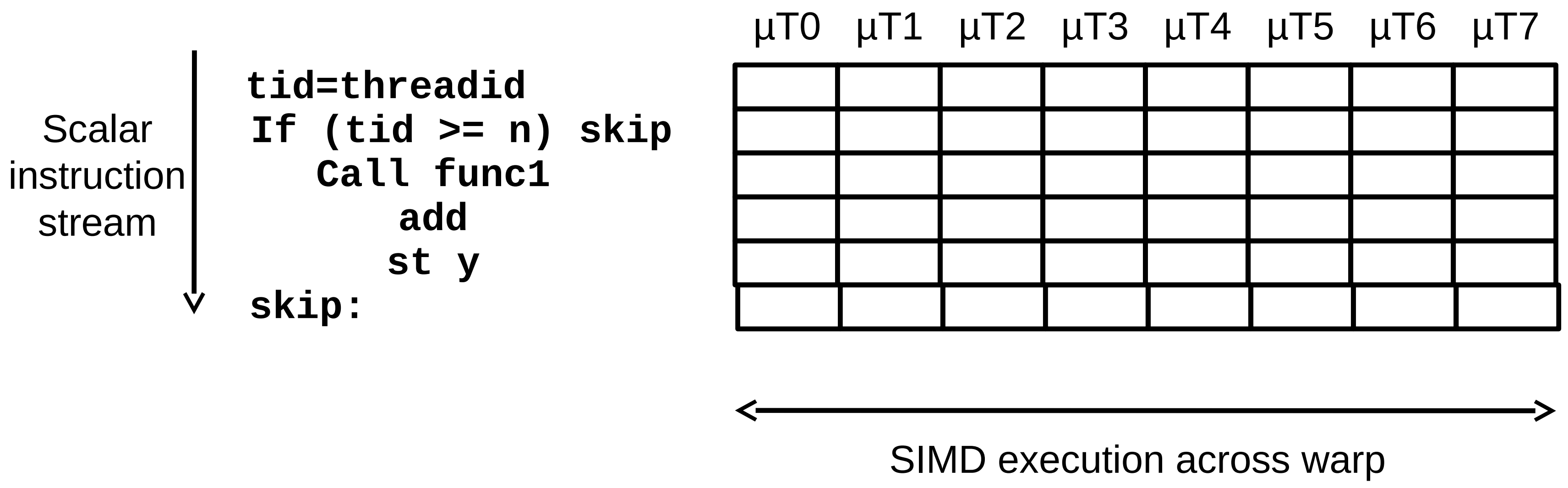
- All “vector” loads and stores are scatter-gather, as individual µthreads perform scalar loads and stores
  - GPU adds hardware to dynamically coalesce individual µthread loads and stores to mimic vector loads and stores
- Every µthread has to perform stripmining calculations redundantly (“am I active?”) as there is no scalar processor equivalent
- Illusion of many independent threads
- But for efficiency, programmer must try and keep µthreads aligned in a SIMD fashion
  - Try and do unit-stride loads and store so memory coalescing kicks in
  - Avoid branch divergence so most instruction slots execute useful work and are not masked off

Conditionals in SIMT model

- Simple if-then-else are compiled into *predicated execution*, equivalent to vector masking
- More complex control flow compiled into branches
- How to execute a vector of branches?

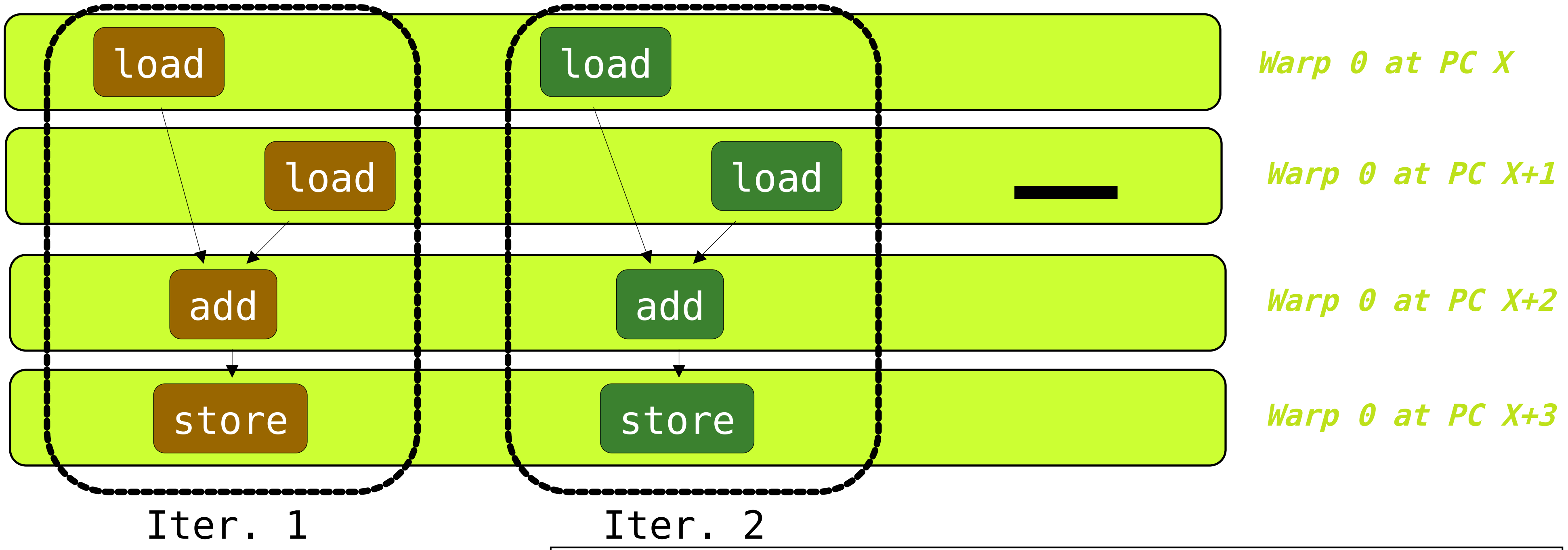
Scalar instruction stream

```
tid=threadid
If (tid >= n) skip
   Call func1
       add
      st y
skip:
```



SPMD on SIMT Machine

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```



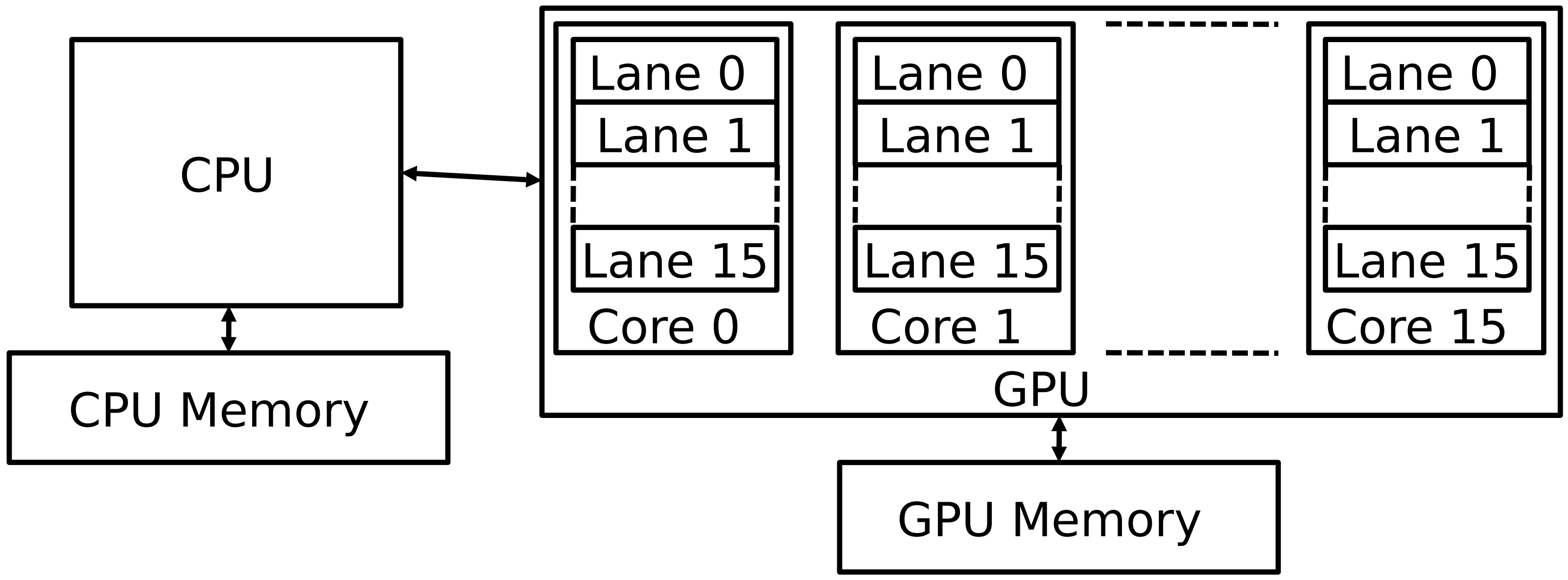
Warp: A set of threads that execute the same instruction (i.e., at the same PC)

This particular model is also called:

SPMD: Single Program Multiple Data

A GPU executes it using the SIMT model:
Single Instruction Multiple Thread

Hardware Execution Model



- GPU is built from multiple parallel cores, each core contains a multithreaded SIMD processor with multiple lanes but with no scalar processor
- CPU sends whole “grid” over to GPU, which distributes thread blocks among cores (each thread block executes on one core)
  - Programmer unaware of number of cores

GPU summary

- Entertainment Industry has driven the economy of GPUs
  - 43 **Billion**$ in 2018
- Moore’s Law in action
- Single-chip designs
- Very Efficient For
  - Fast Parallel Floating Point Processing
  - Single Instruction Multiple Data Operations
  - High Computation per Memory Access
- Not As Efficient For
  - Double Precision
  - Logical Operations on Integer Data
  - Branching-Intensive Operations
  - Random Access, Memory-Intensive Operations

GPU Issues

- Missing Integer & Bit Ops
- Texture Memory Addressing
  - Address conversion burns 3 instr. per array lookup
  - Need large flat texture addressing
- Readback still slow
- Compiler code generator performance
  - Hand code performance critical code
- No native reduction support

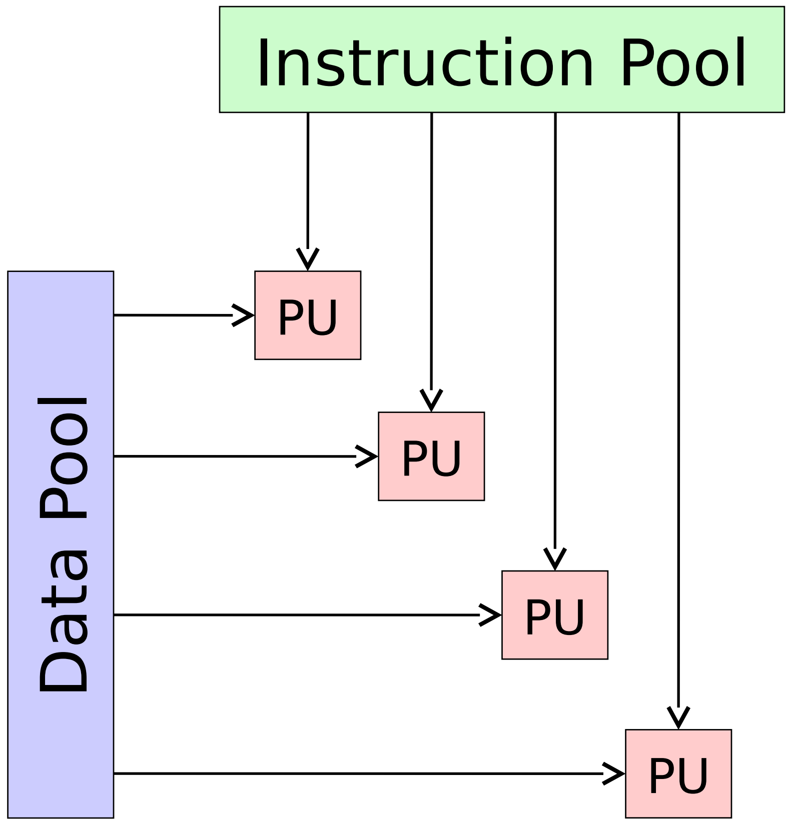
Why Parallel Processing?

- CPU Clock Rates are no longer increasing
  - Technical & economic challenges
    - Advanced cooling technology too expensive or impractical for most applications
    - Energy costs are prohibitive
  - Parallel processing is only path to higher speed

Using Parallelism for Performance

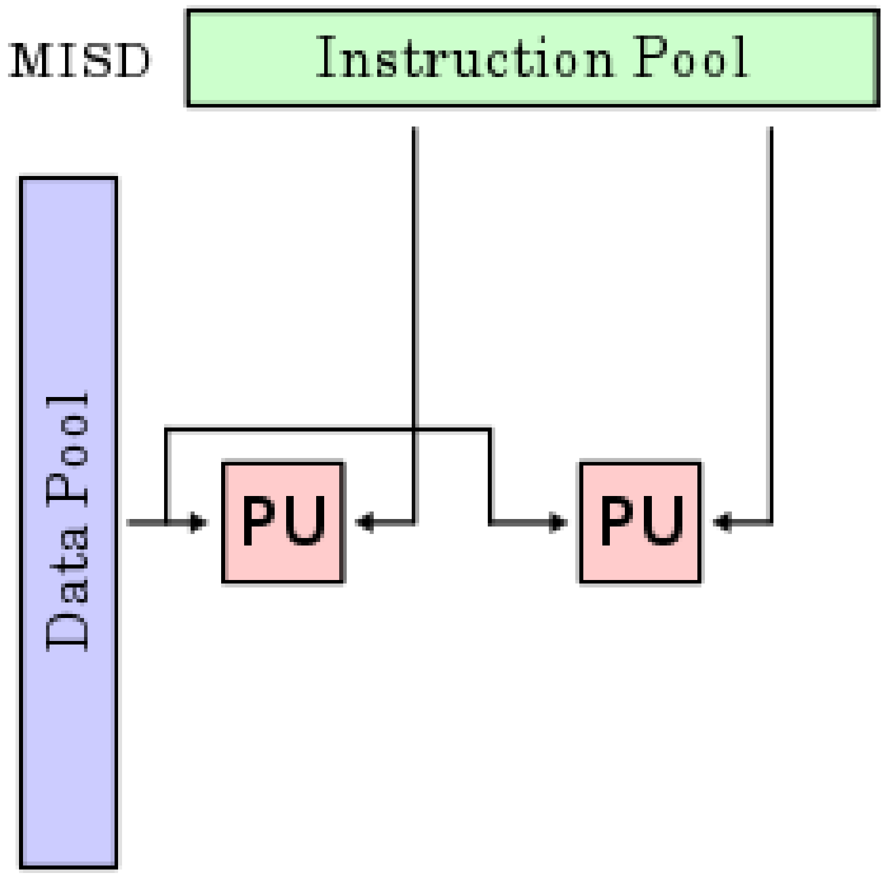
- Two basic ways:
  - Multiprogramming
    - run multiple independent programs in parallel
    - “Easy”
  - Parallel computing
    - run one program faster
    - “Hard”

Multiple-Instruction/Multiple-Data Streams (MIMD or “mim-dee”)



- Multiple autonomous processors simultaneously executing different instructions on different data.
  - MIMD architectures include multicore and Warehouse-Scale Computers

Multiple-Instruction/Single-Data Stream (MISD)



- Multiple-Instruction, Single-Data stream computer that exploits multiple instruction streams against a single data stream.
  - Historical significance

Where is the parallelism?

Different processors take radically different approaches

- CPUs: Instruction-level parallelism
  - Implicit
  - Fine-grain

- GPUs: Thread- & data-level parallelism
  - Explicit
  - Coarse-grain

Whose job to find parallelism?

Different processors take radically different approaches

- CPUs: Hardware dynamically schedules instructions
  - Expensive, complex hardware → Few cores (tens)
  - (Relatively) Easy to write fast software

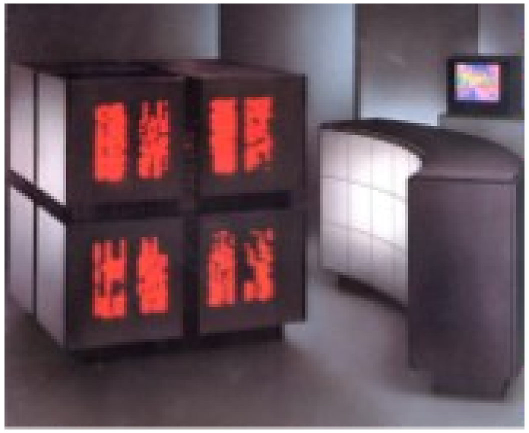
- GPUs: Software makes parallelism explicit
  - Simple, cheap hardware → Many cores (thousands)
  - (Often) Hard to write fast software

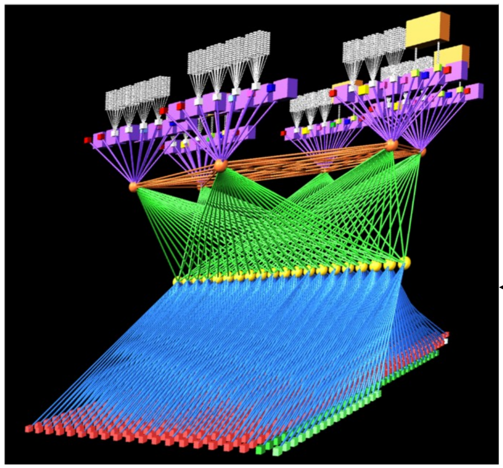
A Brief History of Parallel Computing

- Initial Focus (starting in 1970s): "Supercomputers" for Scientific Computing

C.mmp at CMU (1971)
16 PDP-11 processors

Cray XMP (circa 1984)
4 vector processors


Thinking Machines CM-2 (circa 1987)
65,536 1-bit processors +
2048 floating-point co-processors


Bridges at the Pittsburgh Supercomputer Center

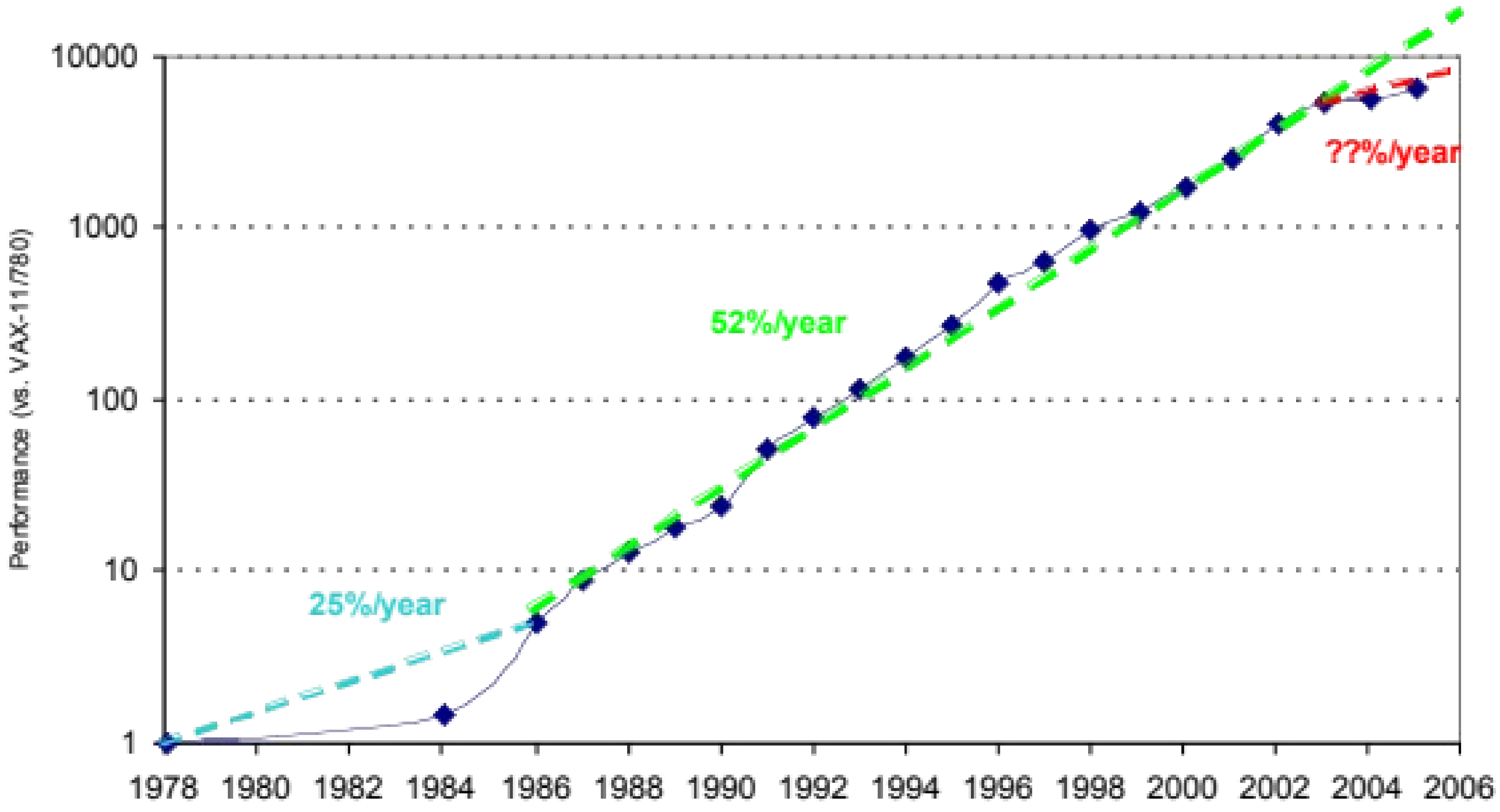
800+ compute nodes
Heterogenous Structure

Parallel Computing History

- Initial Focus (starting in 1970s): “Supercomputers” for Scientific Computing
- Another Driving Application (starting in early ‘90s): Databases
  - Especially, handling millions of transactions per second for web services

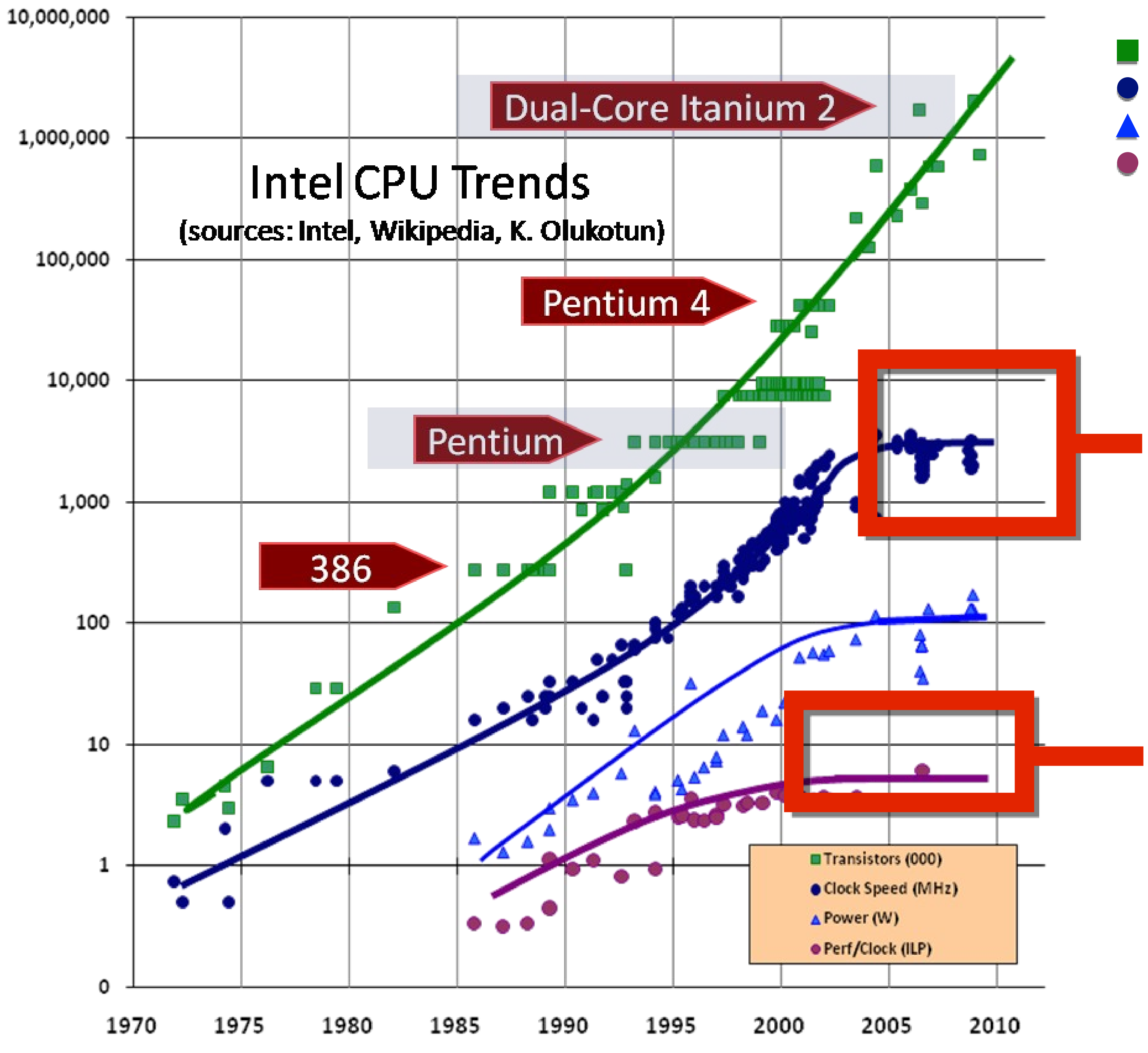
Sun Enterprise 10000
(circa 1997)
16 UltraSPARC-II processors

Uniprocessor Performance



- **VAX : 25%/year 1978 to 1986**
- **RISC + x86: 52%/year 1986 to 2002**
- **RISC + x86: ??%/year 2002 to present**

ILP tapped out + end of frequency scaling



Processor clock rate stops increasing

No further benefit from ILP

The move to multiprocessing

- "… today's processors … are nearing an impasse as technologies approach the speed of light.." David Mitchell, The Transputer: The Time Is Now (1989)
- Transputer had bad timing (Uniprocessor performance)
  → Procrastination rewarded: 2X seq. perf. / 1.5 years
- "We are dedicating all of our future product development to multicore designs. … This is a sea change in computing" Paul Otellini, President, Intel (2005)
- All microprocessor companies switch to MP (2+ CPUs/2 yrs)
  → Procrastination penalized: 2X sequential perf. / 5 yrs
- Even handheld systems moved to multicore
  - Nintendo 3DS, iPhone4S, iPad 3 have two cores each (plus additional specialized cores)
  - Playstation Portable Vita has four cores

Supercomputing

- Today: clusters of multi-core CPUs + GPUs
- Oak Ridge National Laboratory: Summit (#1 supercomputer in world): 4608 nodes, each with 22 core CPU + 2 GPUs

Types of Multiprocessors

- Loosely coupled multiprocessors
  - No shared global memory address space
  - Multicomputer network
    - Network-based multiprocessors
  - Usually programmed via message passing
    - Explicit calls (send, receive) for communication
- Tightly coupled multiprocessors
  - *Shared* global memory address space
  - Traditional multiprocessing: symmetric multiprocessing (SMP)
    - Existing multi-core processors, multithreaded processors
  - Programming model similar to uniprocessors (i.e., multitasking uniprocessor) except
    - Operations on shared data require synchronization

Design Issues for Tightly-coupled Multiprocessors

- Shared memory synchronization
  - How to handle locks, atomic operations
- Cache coherence
  - How to ensure correct operation in the presence of private caches keeping the same memory address cached
- Memory consistency: Ordering of all memory operations
  - What should the programmer expect the hardware to provide?
- Shared resource management
- Communication: Interconnects

Shared Memory

- Multiple execution contexts sharing a single address space
  - Multiple programs (MIMD)
  - Or more frequently: multiple copies of one program (SPMD)
- Implicit (automatic) communication via loads and stores
- Simple software
  - No need for messages, communication happens naturally
  - Supports irregular, dynamic communication patterns: Both DLP and TLP
- Complex hardware
  - Must create a uniform view of memory

Connected vs. Separate Memories

- Separate processor/memory
  - Uniform memory access (UMA): equal latency to all memory
    - Simple software, doesn’t matter where you put data
    - But *lower peak performance*
  - Bus-based UMAs common: *symmetric multi-processors* (SMP)
- Connected processor/memory
  - Non-uniform memory access (NUMA): faster to local memory
    - More *complex software*: where you put data matters
    - Higher peak performance: assuming proper data placement

Shared vs. Point-to-point networks

- Shared network, e.g. bus or crossbar
  - Low latency
  - Low(er) bandwidth: expensive to scale beyond ~16 processors
  - Shared property simplifies cache coherence protocols (today!)
- Point-to-point network: e.g., mesh or ring
  - Longer latency: may need multiple “hops” to communicate
  - Higher bandwidth: scales to 1000s of processors
  - Cache coherence protocols are more complex

Point-to-point networks

- Network topology: organization of network
  - Tradeoff performance (connectivity, latency, bandwidth) ↔ cost
- Router chips
  - Networks that require separate router chips are *indirect*
  - Networks that use processor/memory/router packages are *direct*
    - Fewer components, “Glueless MP”
  - Distinction blurry in the multicore era
- Bus-based systems
- Point-to-point network examples
  - Indirect tree
  - Direct mesh or ring

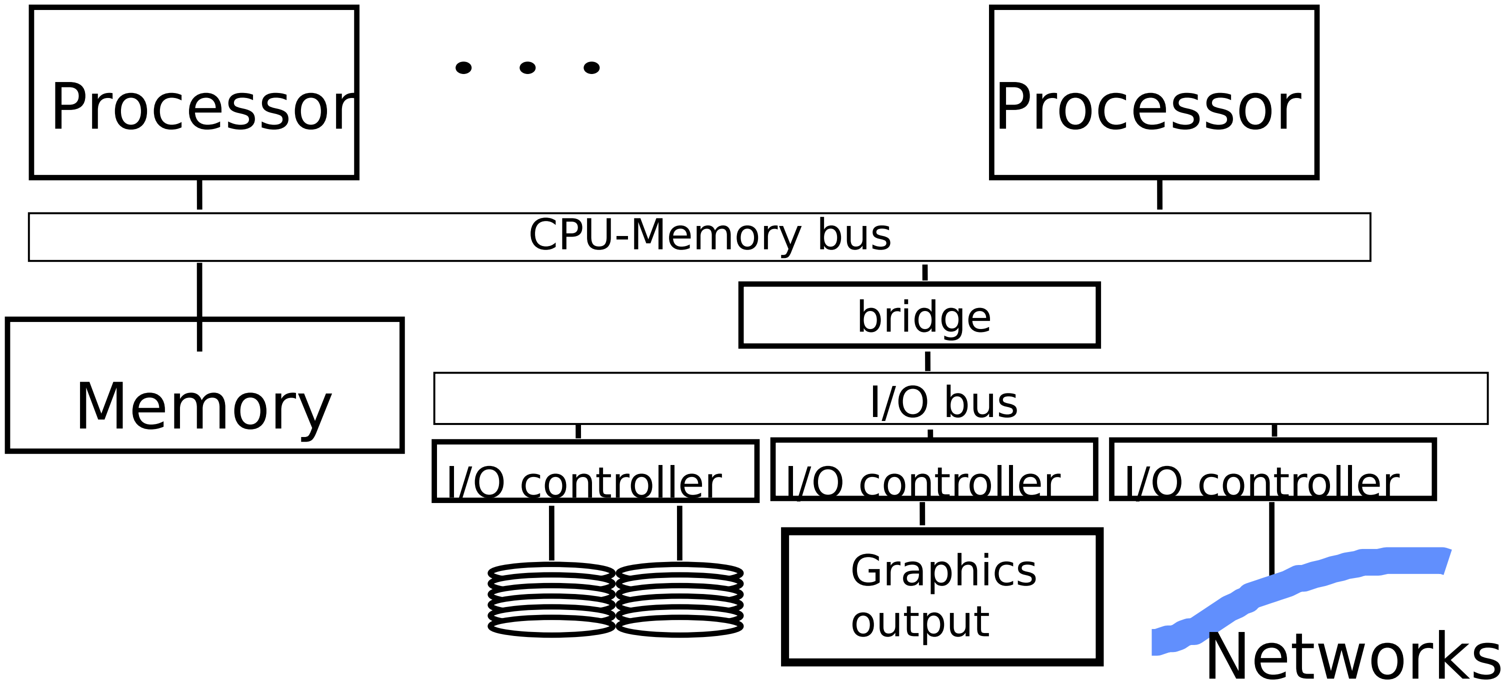
Shared Memory Issues

1) Cache coherence
2) Memory consistency model
3) Synchronization

Memory Consistency vs. Cache Coherence

- *Consistency* is ordering of all memory operations from different processors (i.e., to **different** memory locations)
  - Global ordering of accesses to all memory locations
- *Coherence* is ordering of operations from different processors to the **same** memory location
  - Local ordering of accesses to each cache block

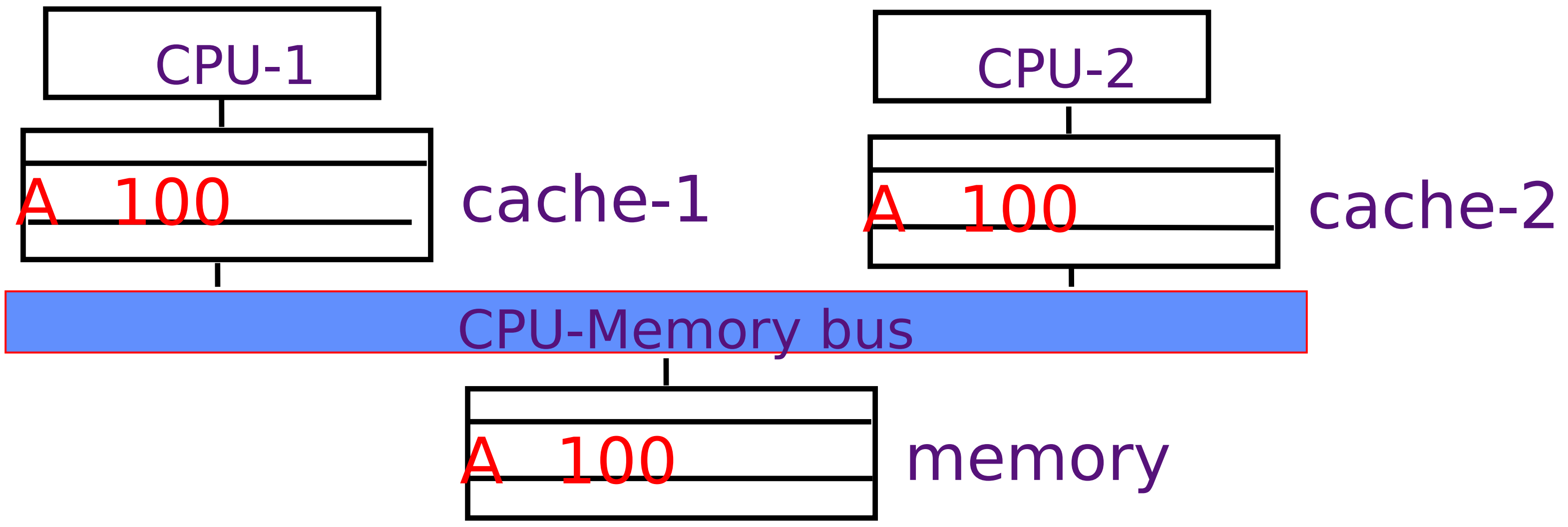
Symmetric multiprocessing



symmetric

- All memory is equally far away from all processors
- Any processor can do any I/O (set up a DMA transfer)

Cache Coherence in SMPs



Suppose CPU-1 updates A to 200.

write-back: memory and cache-2 have stale values

write-through: cache-2 has a stale value

Do these stale values matter?

What is the view of shared memory for programming?

Extreme Solutions

- Problem
  - there are copies of the same memory location
  - a write to one copy needs to be seen correctly by all
- Extreme solutions to consider first

  1) disallow caching of shared variables

  2) allow only one copy of a memory location at a time

  3) allow multiple copies of a memory location, but they must have the same value at all time

Cache Coherence vs. Memory Consistency

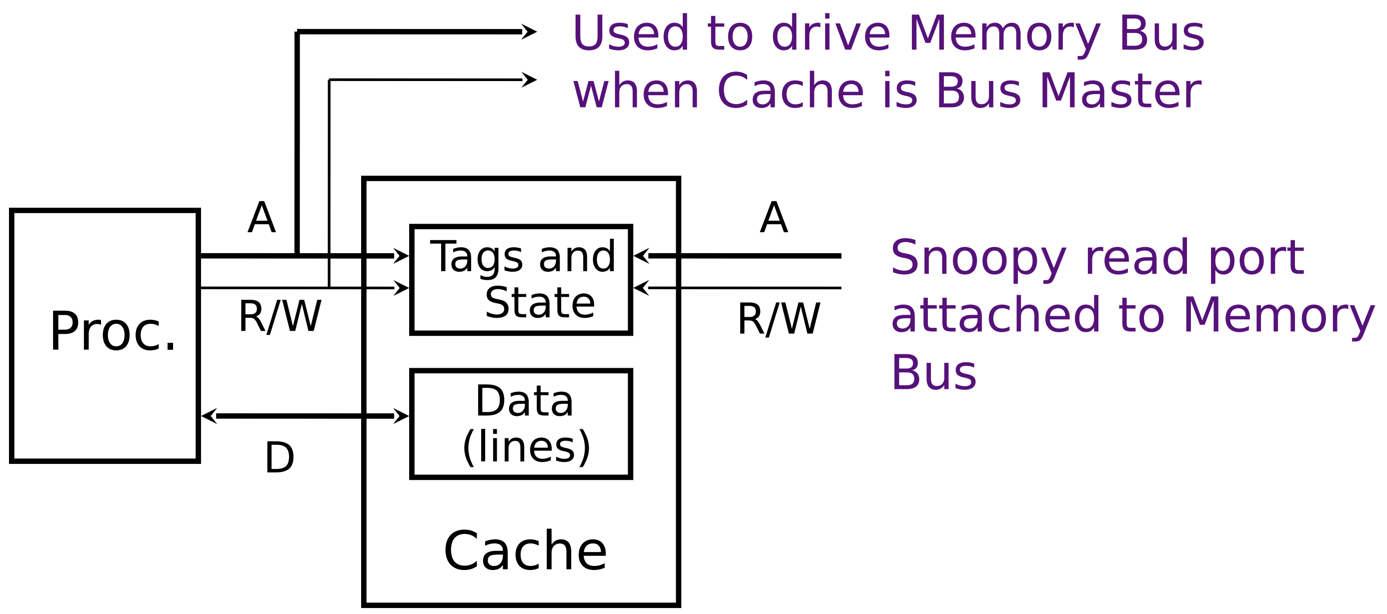
- A cache coherence protocol ensures that all writes by one processor are eventually visible to other processors, for one memory address
  - i.e., updates are not lost
- A memory consistency model gives the rules on when a write by one processor can be observed by a read on another, across different addresses
  - Equivalently, what values can be seen by a load
- A cache coherence protocol is not enough to ensure *sequential consistency*
  - But if sequentially consistent, then caches must be coherent
- Combination of cache coherence protocol plus processor memory reorder buffer used to implement a given architecture’s memory consistency model

Sequential Consistency

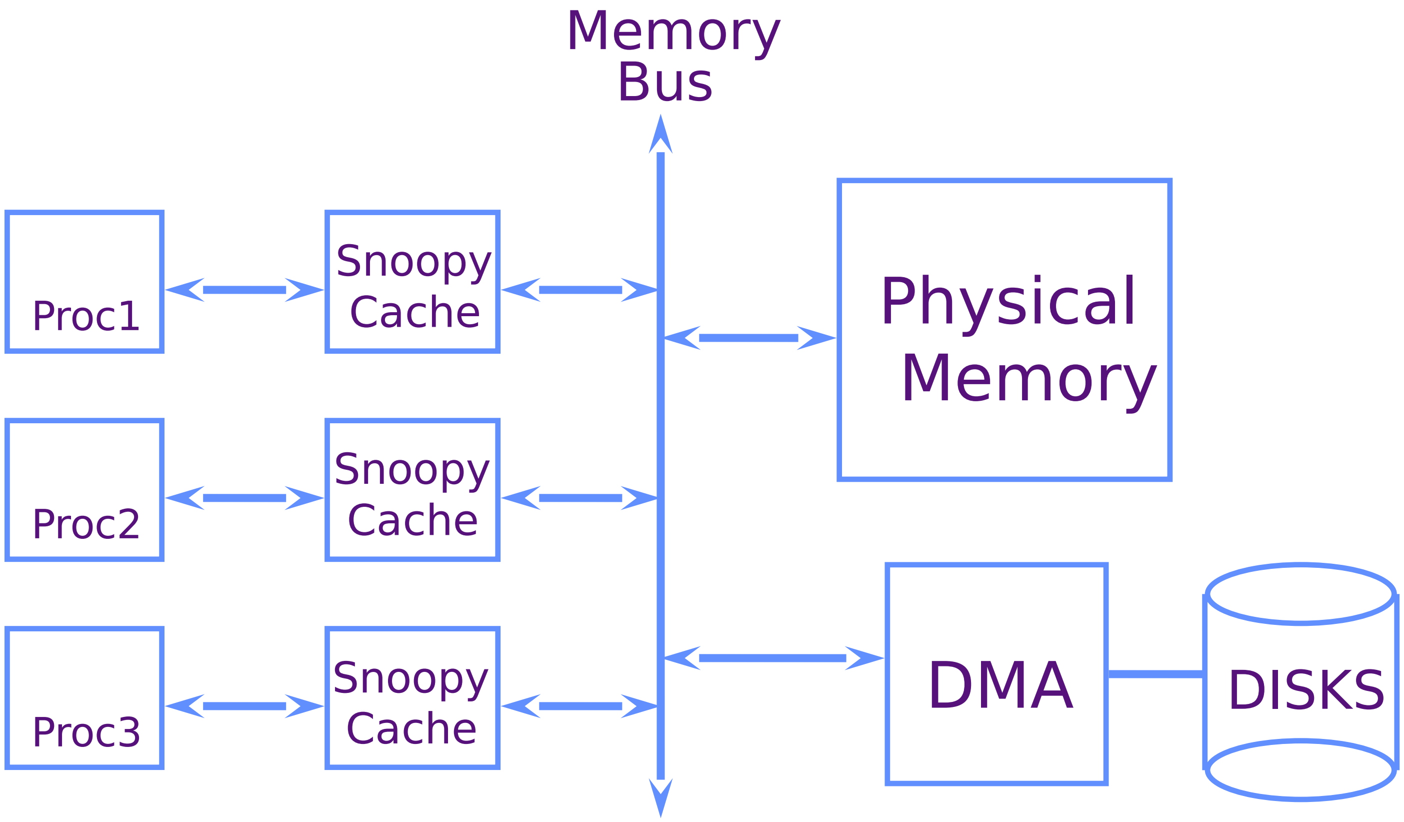
- Memory operations from a processor become visible (to itself and others) in “program order”
- There exists a *total order*, consistent with this partial order - i.e., an interleaving of reads and writes
  - the position at which a write occurs in the hypothetical total order should be the same with respect to all processors
- Said another way: For any possible individual run of a program on multiple processors, should be able to come up with a serial interleaving of all operations that respects
  - Program Order
  - Read-after-write orderings (locally and through interconnect)
  - Also Write-after-read, write-after-write

Snoopy caches (Goodman, 1983)

- Idea: Have cache watch (or snoop upon) bus transfers, and then "do the right thing"
- Snoopy cache tags are dual-ported



Shared Memory Multiprocessor



Use snoopy mechanism to keep all processors' view of memory coherent

Cloud Computing (2000 onward)

- Build out massive centers with many, simple processors, Connected via LANs
- Program using distributed-system models
- “Warehouse computing”