CS 110
Computer Architecture
Lecture 20:
Thread-Level Parallelism (TLP) and OpenMP Intro

Instructors:

Sören Schwertfeger & Chundong Wang

https://robotics.shanghaitech.edu.cn/courses/ca/21s/

School of Information Science and Technology

ShanghaiTech University

Slides based on UC Berkeley's CS61C

Review

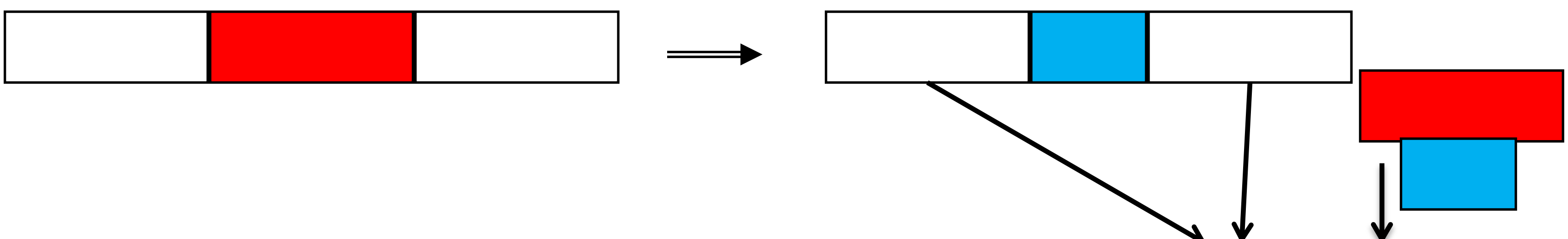
- Amdahl's Law: Serial sections limit speedup
- Flynn Taxonomy
- Intel SSE SIMD Instructions
  - Exploit data-level parallelism in loops
  - One instruction fetch that operates on multiple operands simultaneously
  - 128-bit XMM registers
- SSE Instructions in C
  - Embed the SSE machine instructions directly into C programs through use of Intrinsics
  - Achieve efficiency beyond that of optimizing compiler

Big Idea: Amdahl's (Heartbreaking) Law

- Speedup due to enhancement E is

$$\text{Speedup w/ E} = \frac{\text{Exec time w/o E}}{\text{Exec time w/ E}}$$

- Suppose that enhancement E accelerates a fraction F (F <1) of the task by a factor S (S>1) and the remainder of the task is unaffected



Execution Time w/ E = Execution Time w/o E x [(1-F) + F/S]

Speedup w/ E = 1 / [(1-F) + F/S]

Last Lecture: DGEMM Performance

- Intel i7-5557U theoretical limit (AVX2): 24.8 GFLOPS
- Cache:
  - L3: 4 MB 16-way set associative shared cache
  - L2: 2 x 256 KB 8-way set associative caches
  - L1 Cache: 2 x 32KB 8-way set associative caches (2x: D & I)
- Maximum memory bandwidth (GB/s): 29.9

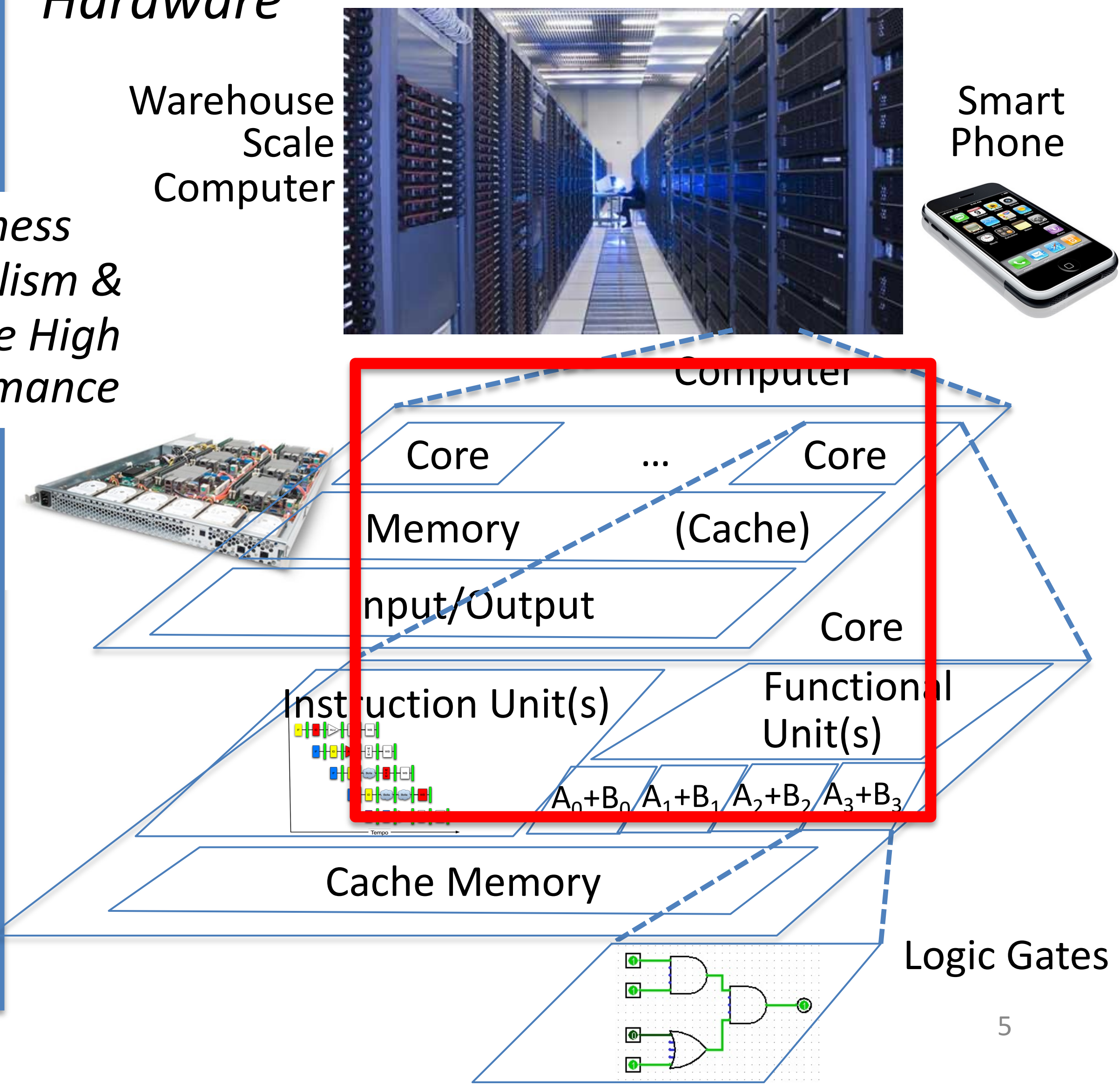
N	Size	GFlops			
		scalar	avx	unroll	blocking
32	3x 8KiB	1.30	4.56	12.95	13.80
160	3x 200KiB	1.30	5.47	19.70	21.79
480	3x 1.8MiB	1.32	5.27	14.50	20.17
960	3x 7.2MiB	0.91	3.64	6.91	15.82

New-School Machine Structures (It’s a bit more complicated!)

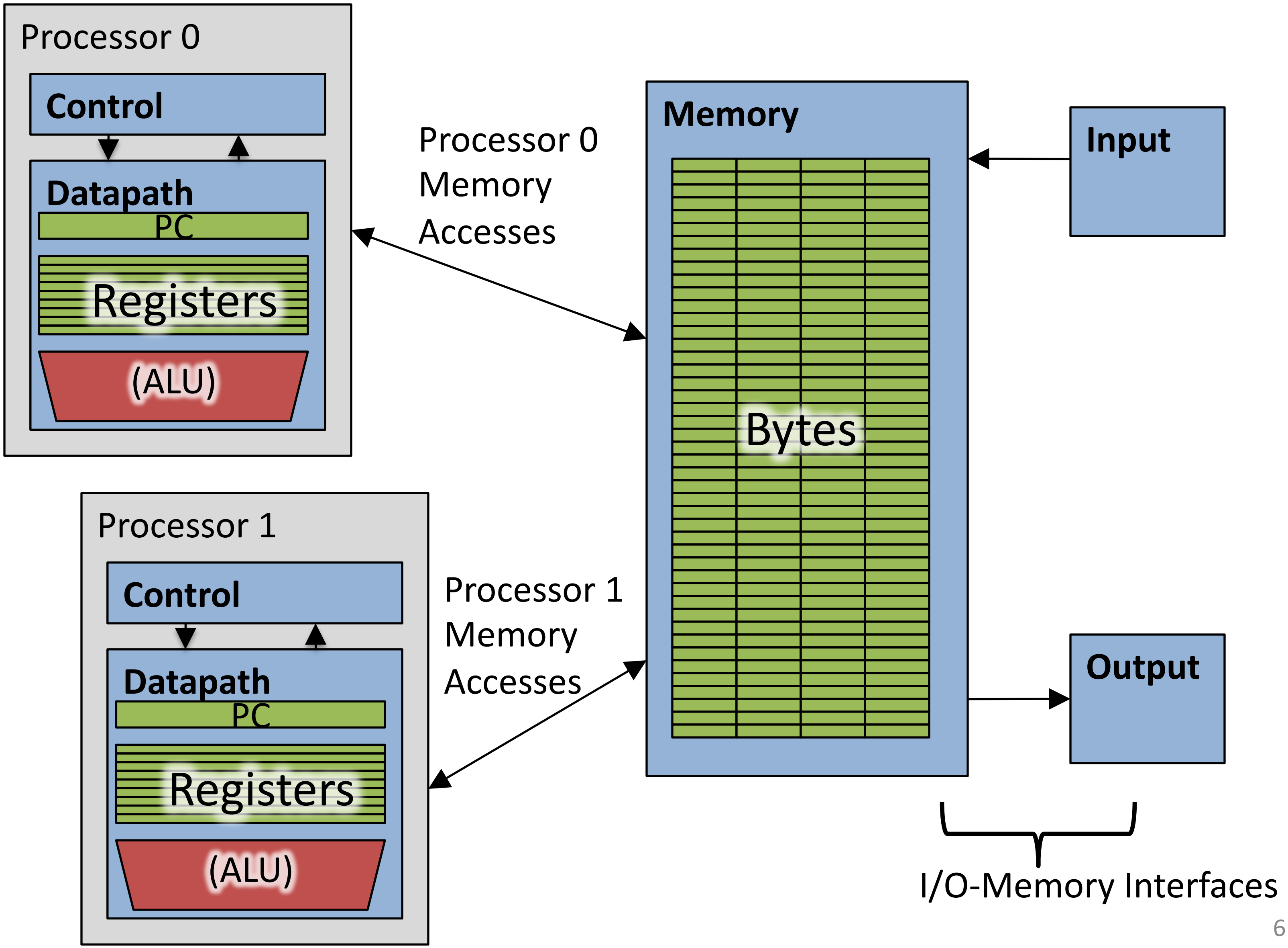
Software | *Hardware*

- Parallel Requests
  Assigned to computer
  e.g., Search “Katz”
- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads
- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words
- Hardware descriptions
  All gates @ one time
- Programming Languages

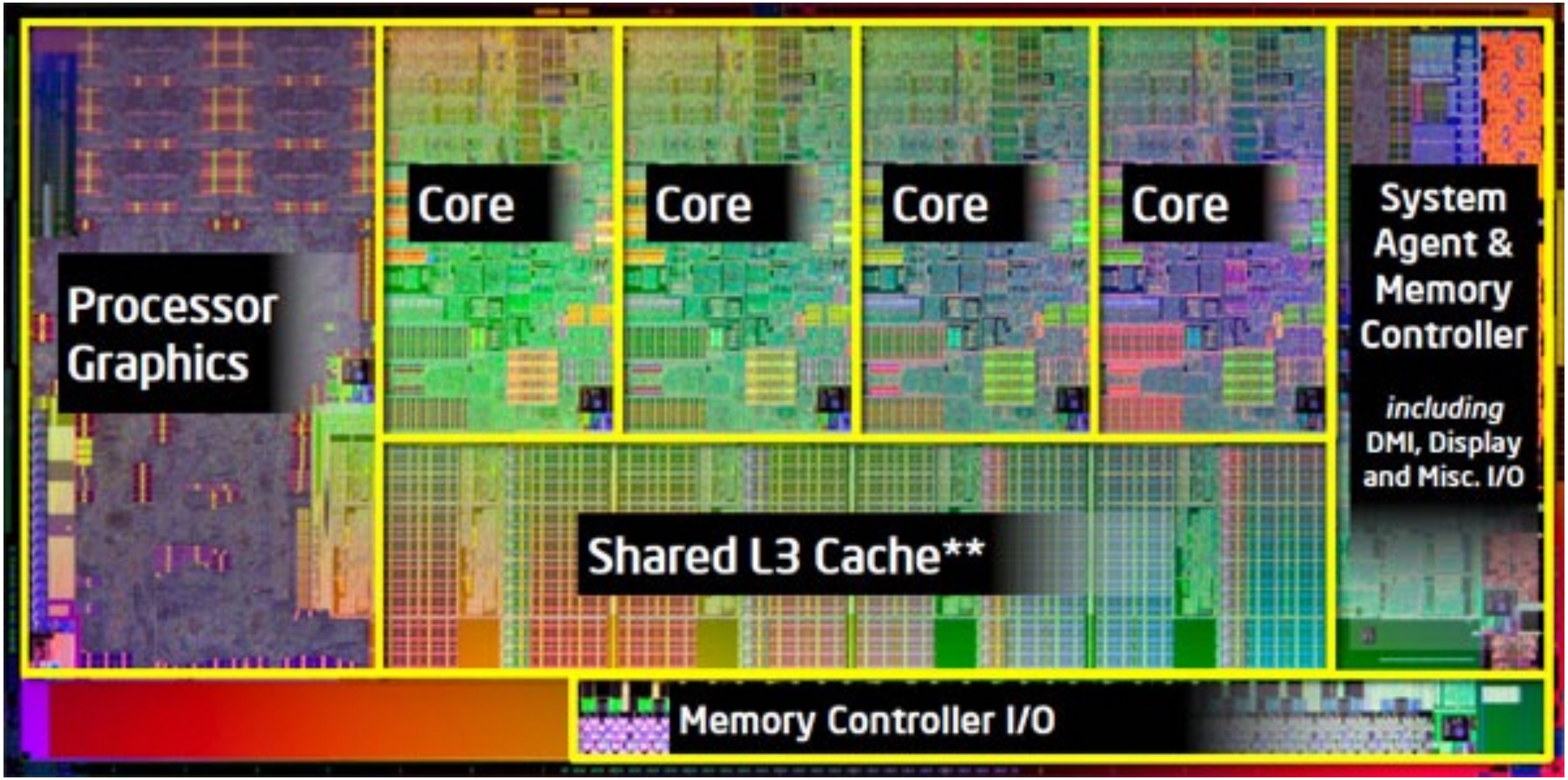
Harness Parallelism & Achieve High Performance



Simple Multiprocessor



4 Core Processor with Graphics

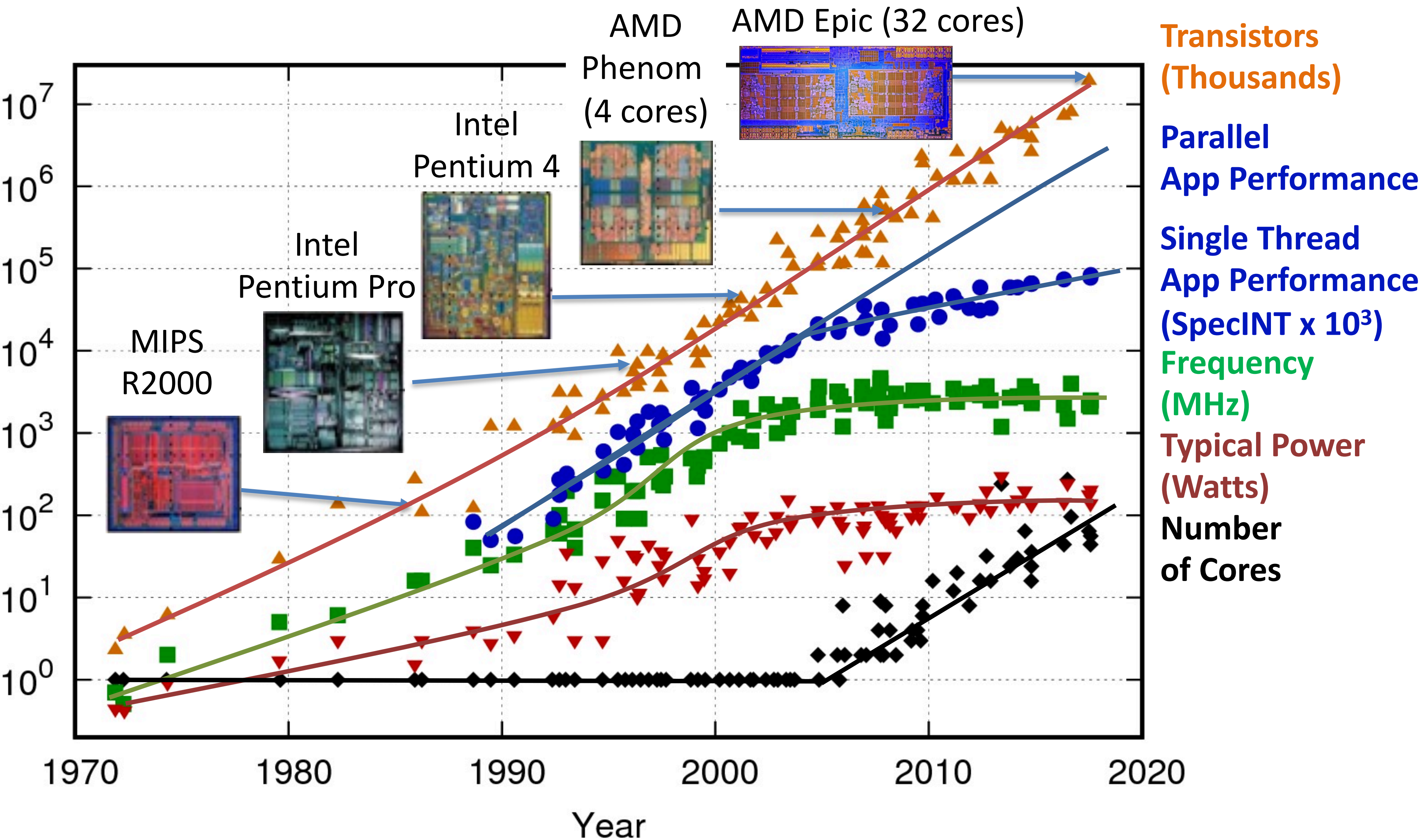


Multiprocessor Execution Model

- Each processor has its own PC and executes an independent stream of instructions (MIMD)
- Different processors can access the same memory space
  - Processors can communicate via shared memory by storing/loading to/from common locations
- Two ways to use a multiprocessor:
  1. Deliver high throughput for independent jobs via job-level parallelism
  2. **Improve the run time of a single program that has been specially crafted to run on a multiprocessor - a parallel-processing program**

Use term *core* for processor ("Multicore") because "Multiprocessor Microprocessor" too redundant

Transition to Multicore



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Current Multi-Core CPUs

- Intel Core i7: 4-10 real cores
- Intel Core i9: 10-18 real cores
- Intel Xeon Platinum: 16, 24, 26, 28 real cores
- AMD Epyc 2: 8 - <u>64</u> real cores
- Apple A13: 2 (high performance) + 4 (low power) Apple designed ARM CPUs
- Samsung S20 (Samsung Exynos 990): 1 + 3 + 4

1 x ARM Cortex-A77	2.85GHz	512kB L2$
3 x ARM Cortex-A77	2.4GHz	256kB L2$
4 x ARM Cortex-A55	1.8GHz	128kB L2$

Parallelism the Only Path to Higher Performance

- Sequential processor performance not expected to increase much, and might go down
- If want apps with more capability, have to embrace parallel processing (SIMD and MIMD)
- In mobile systems, use multiple cores and GPUs
- In warehouse-scale computers, use multiple nodes, and all the MIMD/SIMD capability of each node

Comparing Types of Parallelism…

- SIMD-type parallelism (Data Parallel)
  - A SIMD-favorable problem can map easily to a MIMD-type fabric
  - SIMD-type fabrics generally offer a much higher ***throughput per $***
    - Much simpler control logic
    - Classic example: Graphics cards are massive supercomputers compared to the CPU: TeraFLOPS rather than gigaflops
- MIMD-type parallelism (data-dependent Branches!)
  - A MIMD-favorable problem ***will not map easily*** to a SIMD-type fabric
  - E.g.: some problems work well on GPU (e.g. Deep Learning). Others NOT (e.g. compiler)

Multiprocessors and You

- Only path to performance is parallelism
  - Clock rates flat or declining
  - CPI generally flat
  - SIMD:
    - 2011: 256b Intel & AMD
    - 2016: 512b Intel & Fujitsu A64FX
    - X: 1024b specified – no CPU planned yet
    - GPUs: massive SIMD
  - MIMD: Add 2 cores every 2 years: 2, 4, 6, 8, 10, …
- Key challenge is to craft parallel programs that have high performance on multiprocessors as the number of processors increase – i.e., that scale
  - Scheduling, load balancing, time for synchronization, overhead for communication

Threads

- *Thread:* a sequential flow of instructions that performs some task
- Each thread has a PC + processor registers and accesses the shared memory
- Each processor provides one (or more) *hardware* threads that actively execute instructions
- Operating system multiplexes multiple *software* threads onto the available hardware threads

Operating System Threads

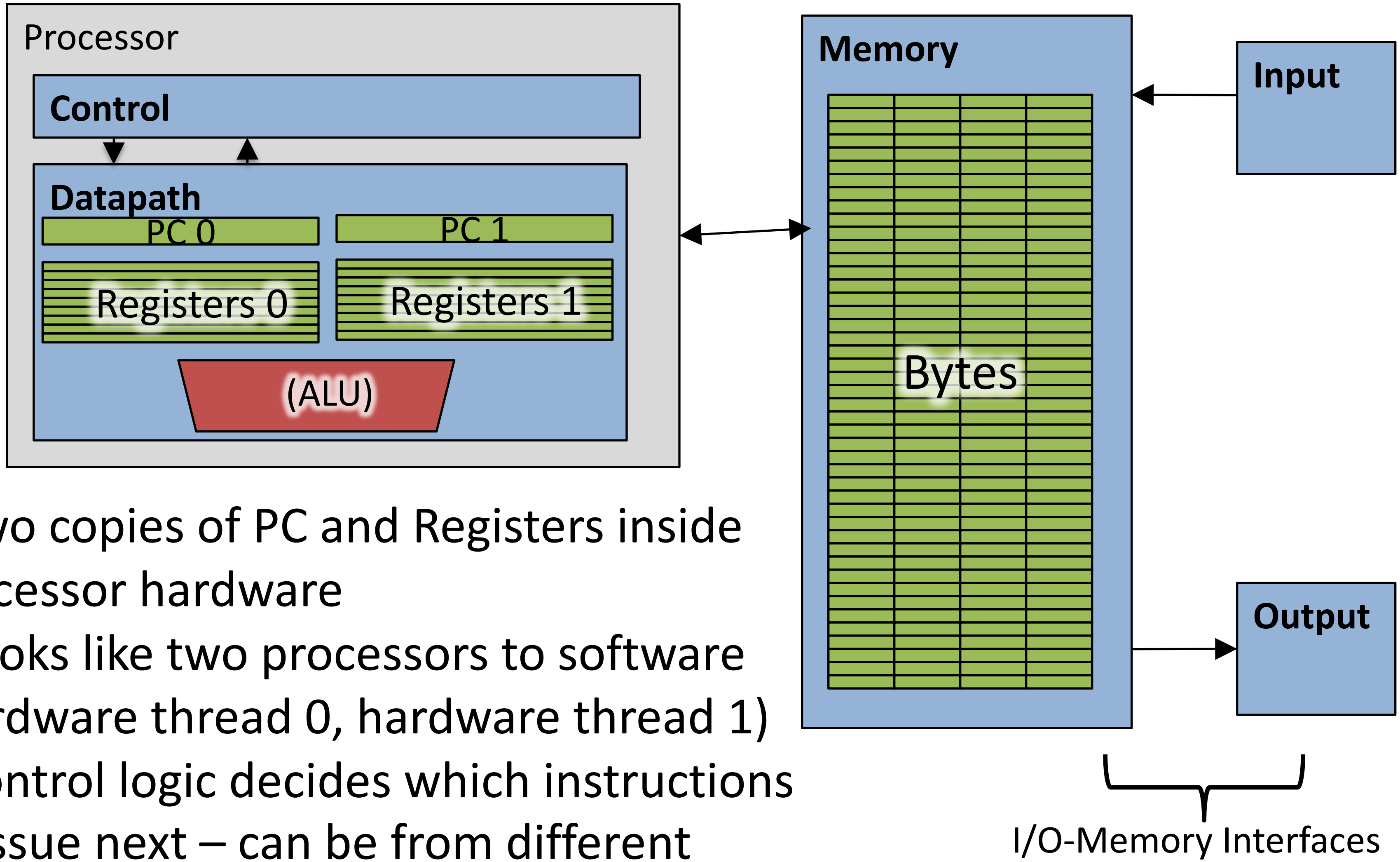
Give the illusion of many active threads by time-multiplexing software threads onto hardware threads

- Remove a software thread from a hardware thread by interrupting its execution and saving its registers and PC into memory
  - Also if one thread is blocked waiting for network access or user input
- Can make a different software thread active by loading its registers into a hardware thread's registers and jumping to its saved PC

Hardware Multithreading (Hyperthreading)

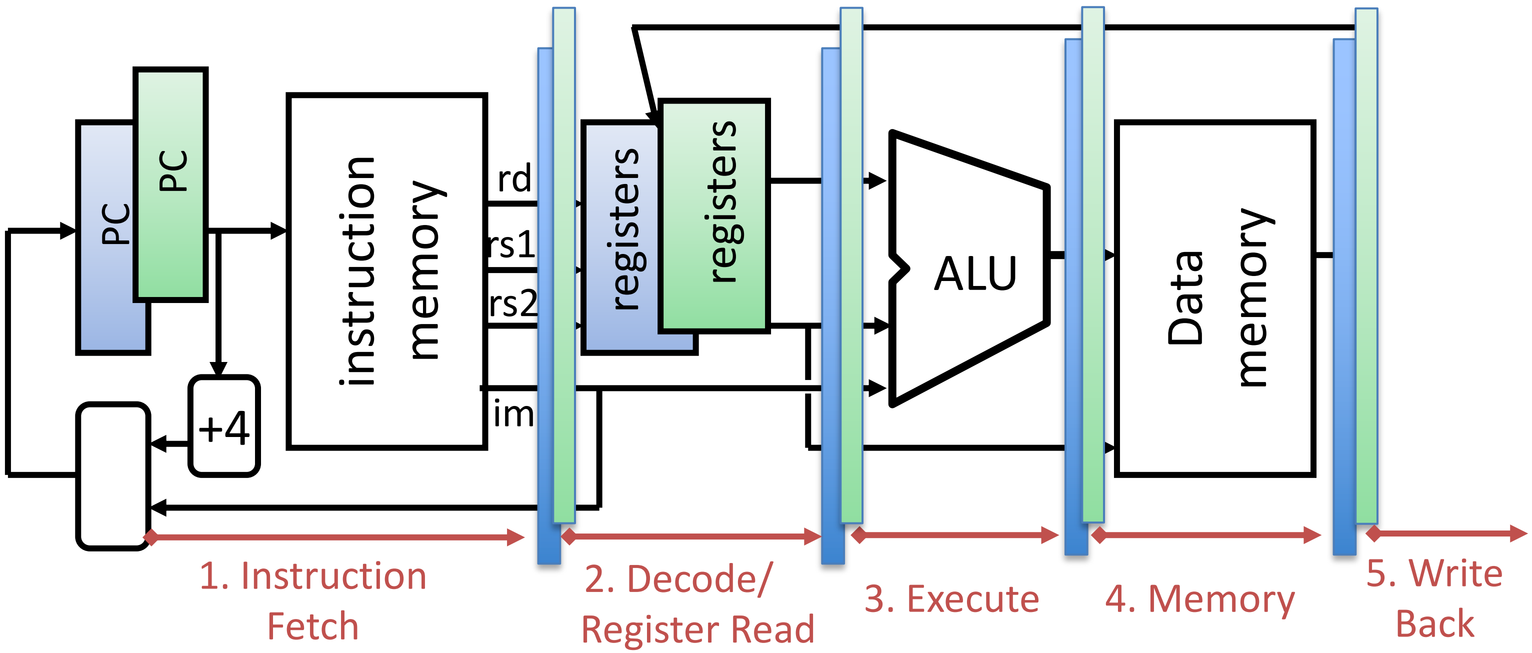
- Basic idea: Processor resources are expensive and should not be left idle
  - Long memory latency to memory on cache miss?
- Hardware switches threads to bring in other useful work while waiting for cache miss
  - Cost of thread context switch must be much less than cache miss latency
- Put in redundant hardware so don’t have to save context on every thread switch:
  - PC, Registers
- Attractive for apps with abundant TLP
  - Commercial multi-user workloads

Hardware Multithreading (Hyperthreading)



- Two copies of PC and Registers inside processor hardware
- Looks like two processors to software (hardware thread 0, hardware thread 1)
- Control logic decides which instructions to issue next – can be from different threads!

Hyper-threading (simplified)



- Duplicate all elements that hold the state (registers)
- Use the same CL blocks
- Use muxes to select which state to use every clock cycle
- => run 2 independent processes
  - No Hazards: registers different; different control flow; memory different; Threads: memory hazard should be solved by software (locking, mutex, …)
- Speedup?
  - No obvious speedup; Complex pipeline: make use of CL blocks in case of unavailable resources (e.g. wait for memory)

Multithreading vs. Multicore

- Multithreading => Better Utilization
  - ≈5% more hardware, 1.10X better performance?
  - Share integer adders, floating-point units, all caches (L1 I$, L1 D$, L2$, L3$), Memory Controller
- Multicore => Duplicate Processors
  - ≈50% more hardware, ≈2X better performance?
  - Share outer caches (L2$, L3$), Memory Controller
- Modern machines do both
  - Multiple cores with multiple threads per core

Sören's MacBook

Chundong's should be much better ☺

- `sysctl -a | grep hw\.`

MacBookPro11,3

…

hw.physicalcpu: 4

hw.logicalcpu: 8

…

hw.cpufrequency = 2,800,000,000

hw.memsize = 17,179,869,184

hw.cachelinesize = 64

hw.l1icachesize: 32,768

hw.l1dcachesize: 32,768

hw.l2cachesize: 262,144

hw.l3cachesize: 6,291,456

on Linux:

`cat /proc/cpuinfo`

Chundong's Workstation

```
wangc@HP:~$ lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
Address sizes:       36 bits physical, 48 bits virtual
CPU(s):              16
On-line CPU(s) list: 0-15
Thread(s) per core:  2
Core(s) per socket:  8
Socket(s):           1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               158
Model name:          Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz
Stepping:            13
CPU MHz:             3600.000
CPU max MHz:         3600.0000
BogoMIPS:            7200.00
Virtualization:      VT-x
```

100s of (Mostly Dead) Parallel Programming Languages

ActorScript	Concurrent Pascal	JoCaml	Orc
Ada	Concurrent ML	Join	Oz
Afnix	Concurrent Haskell	Java	Pict
Alef	Curry	Joule	Reia
Alice	CUDA	Joyce	SALSA
APL	E	LabVIEW	Scala
Axum	Eiffel	Limbo	SISAL
Chapel	Erlang	Linda	SR
Cilk	Fortan 90	MultiLisp	Stackless Python
Clean	Go	Modula-3	SuperPascal
Clojure	Io	Occam	VHDL
Concurrent C	Janus	occam-π	XC

OpenMP

- OpenMP is a language extension used for multi-threaded, shared-memory parallelism
  - Compiler Directives (inserted into source code)
  - Runtime Library Routines (called from your code)
  - Environment Variables (set in your shell)
- Portable
- Standardized
- Easy to compile: `cc –fopenmp name.c`

Shared Memory Model with Explicit Thread-based Parallelism

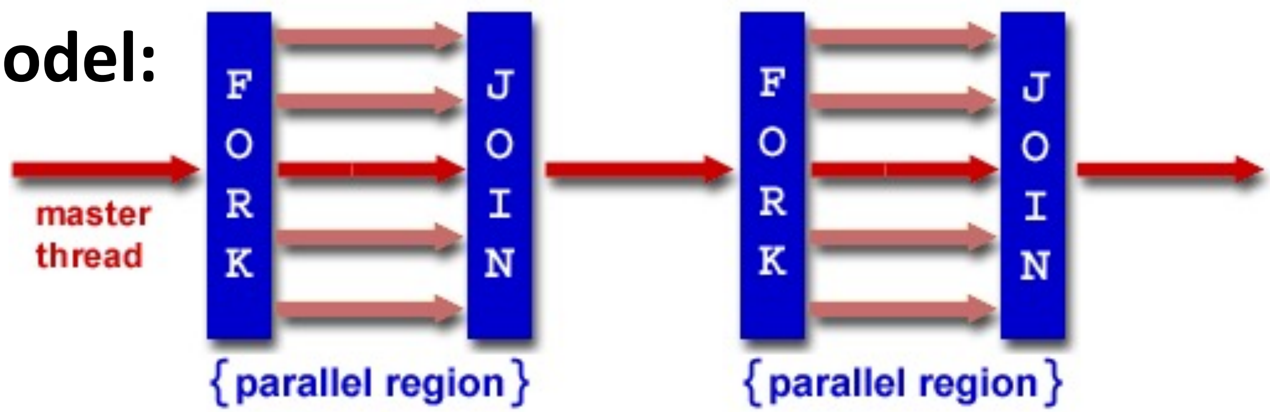
- Multiple threads in a shared memory environment, explicit programming model with full programmer control over parallelization
- **Pros:**
  - Takes advantage of shared memory, programmer need not worry (that much) about data placement
  - Compiler directives are simple and easy to use
  - Legacy serial code does not need to be rewritten
- **Cons:**
  - Code can only be run in shared memory environments
  - Compiler must support OpenMP

OpenMP in CS110

- OpenMP is built on top of C, so you don't have to learn a whole new programming language
  - Make sure to add `#include <omp.h>`
  - Compile with flag: `gcc -fopenmp`
  - Mostly just a few lines of code to learn
- You will NOT become experts at OpenMP
  - Use slides as reference, will learn to use in lab
- **Key ideas:**
  - Shared vs. Private variables
  - OpenMP directives for parallelization, work sharing, synchronization

OpenMP Programming Model

- **Fork - Join Model:**



- OpenMP programs begin as single process (*master thread*) and executes sequentially until the first parallel region construct is encountered
  - *FORK:* Master thread then creates a team of parallel threads
  - Statements in program that are enclosed by the parallel region construct are executed in parallel among the various threads
  - *JOIN:* When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread

OpenMP Extends C with Pragmas

- *Pragmas* are a preprocessor mechanism C provides for language extensions
- Commonly implemented pragmas: structure packing, symbol aliasing, floating point exception modes (not covered)
- Good mechanism for OpenMP because compilers that don't recognize a pragma are supposed to ignore them
  - Runs on sequential computer even with embedded pragmas

`parallel` Pragma and Scope

- Basic OpenMP construct for parallelization:

```
#pragma omp parallel
{
   /* code goes here */
}
```

This is annoying, but curly brace MUST go on separate line from #pragma

  - *Each* thread runs a copy of code within the block
  - Thread scheduling is *non-deterministic*
- OpenMP default is *shared* variables
  - To make private, need to declare with pragma:

```
#pragma omp parallel private (x)
```

Thread Creation

- How many threads will OpenMP create?
- Defined by **OMP_NUM_THREADS** environment variable (or code procedure call)
  - Set this variable to the maximum number of threads you want OpenMP to use
  - Usually equals the number of physical cores * number of threads/core in the underlying hardware on which the program is run

What Kind of Threads?

- OpenMP threads are operating system (software) threads.
- OS will multiplex requested OpenMP threads onto available hardware threads.
- Hopefully each gets a real hardware thread to run on, so no OS-level time-multiplexing.
- But other tasks on machine can also use hardware threads!
  - And you may want more threads than hardware if you have a lot of I/O so that while waiting for I/O other threads can run

OMP_NUM_THREADS

- OpenMP intrinsic to set number of threads:

```
omp_set_num_threads(x);
```

- OpenMP intrinsic to get number of threads:

```
num_th = omp_get_num_threads();
```

- OpenMP intrinsic to get Thread ID number:

```
th_ID = omp_get_thread_num();
```

Parallel Hello World

```
#include <stdio.h>
#include <omp.h>
int main () {
  int nthreads, tid;
  /* Fork team of threads with private var tid */
  #pragma omp parallel private(tid)
  {
    tid = omp_get_thread_num(); /* get thread id */
    printf("Hello World from thread = %d\n", tid);
    if (tid == 0) {
     /* Only master thread does this */
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
  }  /* All threads join master and terminate */
}
```

Results

```
wangc@HP:~/TT$ gcc omp.c -O3 -o p -fopenmp
wangc@HP:~/TT$ ./p
Hello World from thread = 6
Hello World from thread = 11
Hello World from thread = 12
Hello World from thread = 7
Hello World from thread = 15
Hello World from thread = 0
Number of threads = 16
Hello World from thread = 2
Hello World from thread = 10
Hello World from thread = 9
Hello World from thread = 5
Hello World from thread = 14
Hello World from thread = 4
Hello World from thread = 8
Hello World from thread = 13
Hello World from thread = 3
Hello World from thread = 1
```

```
wangc@HP:~/TT$ ./p
Hello World from thread = 13
Hello World from thread = 0
Number of threads = 16
Hello World from thread = 7
Hello World from thread = 6
Hello World from thread = 11
Hello World from thread = 1
Hello World from thread = 9
Hello World from thread = 3
Hello World from thread = 15
Hello World from thread = 8
Hello World from thread = 4
Hello World from thread = 10
Hello World from thread = 2
Hello World from thread = 14
Hello World from thread = 12
Hello World from thread = 5
```

omp_set_num_threads

```
#include <stdio.h>
#include <omp.h>
int main () {
  int nthreads, tid;
  omp_set_num_threads(8); // Newly-added here.
  /* Fork team of threads with private var tid */
  #pragma omp parallel private(tid)
  {
    tid = omp_get_thread_num(); /* get thread id */
    printf("Hello World from thread = %d\n", tid);
    if (tid == 0) {
     /* Only master thread does this */
     nthreads = omp_get_num_threads();
     printf("Number of threads = %d\n", nthreads);
    }
  }  /* All threads join master and terminate */
}
```

omp_set_num_threads

```
wangc@HP:~/TT$ gcc omp.c -o p -O3 -fopenmp
wangc@HP:~/TT$ ./p
Hello World from thread = 1
Hello World from thread = 7
Hello World from thread = 6
Hello World from thread = 5
Hello World from thread = 0
Number of threads = 8
Hello World from thread = 2
Hello World from thread = 4
Hello World from thread = 3
wangc@HP:~/TT$ ./p
Hello World from thread = 3
Hello World from thread = 0
Number of threads = 8
Hello World from thread = 1
Hello World from thread = 2
Hello World from thread = 7
Hello World from thread = 5
Hello World from thread = 6
Hello World from thread = 4
wangc@HP:~/TT$ ./p
Hello World from thread = 2
Hello World from thread = 7
Hello World from thread = 5
Hello World from thread = 3
Hello World from thread = 0
Number of threads = 8
Hello World from thread = 6
Hello World from thread = 4
Hello World from thread = 1
wangc@HP:~/TT$
```

Data Races and Synchronization

- Two memory accesses form a *data race* if from different threads to same location, and at least one is a write, and they occur one after another
- If there is a data race, result of program can vary depending on chance (which thread first?)
- Avoid data races by synchronizing writing and reading to get deterministic behavior
- Synchronization done by user-level routines that rely on hardware synchronization instructions
- (more later)

Analogy: Buying Yogurt

- Your fridge has no yogurt. You and your roommate will return from classes at some point and check the fridge
- Whoever gets home first will check the fridge, go and buy yogurt, and return
- What if the other person gets back while the first person is buying yogurt?
  - You've just bought twice as much yogurt as you need!
- It would've helped to have left a note…

Lock Synchronization (1/2)

- Use a "Lock" to grant access to a region (*critical section*) so that only one thread can operate at a time
  - Need all processors to be able to access the lock, so use a location in shared memory as *the lock*
- Processors read lock and either wait (if locked) or set lock and go into critical section
  - **0** means lock is free / open / unlocked / lock off
  - **1** means lock is set / closed / locked / lock on

Lock Synchronization (2/2)

- Pseudocode:

Can loop/idle here if locked

```
Check lock
Set the lock
Critical section
(e.g. change shared variables)
Unset the lock
```

Possible Lock Implementation

- Lock (a.k.a. busy wait)

```
Get_lock:                       # s0 -> addr of lock
        addiu t1,zero,1         # t1 = Locked value
Loop:   lw    t0,0(s0)          # load lock
        bne   t0,zero,Loop      # loop if locked
Lock:   sw    t1,0(s0)          # Unlocked, so lock
```

- Unlock

```
Unlock:
    sw zero,0(s0)
```

- Any problems with this?

Possible Lock Problem

- Thread 1

```
       addiu t1,zero,1
Loop:  lw  t0,0(s0)

       bne t0,zero,Loop

Lock:  sw t1,0(s0)
```

- Thread 2

```
       addiu t1,zero,1
Loop:  lw t0,0(s0)

       bne t0,zero,Loop

Lock:  sw t1,0(s0)
```

Time

Both threads think they have set the lock!
Exclusive access not guaranteed!

Hardware Synchronization

- Hardware support required to prevent an interloper (another thread) from changing the value
  - *Atomic* read/write memory operation
  - No other access to the location allowed between the read and write
- How best to implement in software?
  - Single instr? Atomic swap of register ↔ memory
  - Pair of instr? One for read, one for write
- Needed even on uniprocessor systems
  - Interrupts can happen: can trigger thread context switches...

RISC-V: Two solutions!

- Option 1: Read/Write Pairs
  - Pair of instructions for “linked” read and write
  - Load reserved and Store conditional
  - No other access permitted between read and write
    - Must use *shared memory* (<u>multi</u>processing)

- Option 2: Atomic Memory Operations
  - Atomic swap of register ↔ memory

Read/Write Pairs

- Load reserved: **lr rd, rs**
  - Load the word pointed to by **rs** into **rd**, and add a reservation
- Store conditional: **sc rd, rs1, rs2**
  - Store the value in **rs2** into the memory location pointed to by **rs1**, only if the reservation is still valid and set the status in **rd**
    - Returns 0 (success) if location has not changed since the **lr**
    - Returns nonzero (failure) if location has changed: ***Actual store will not take place***

Synchronization in RISC-V Example

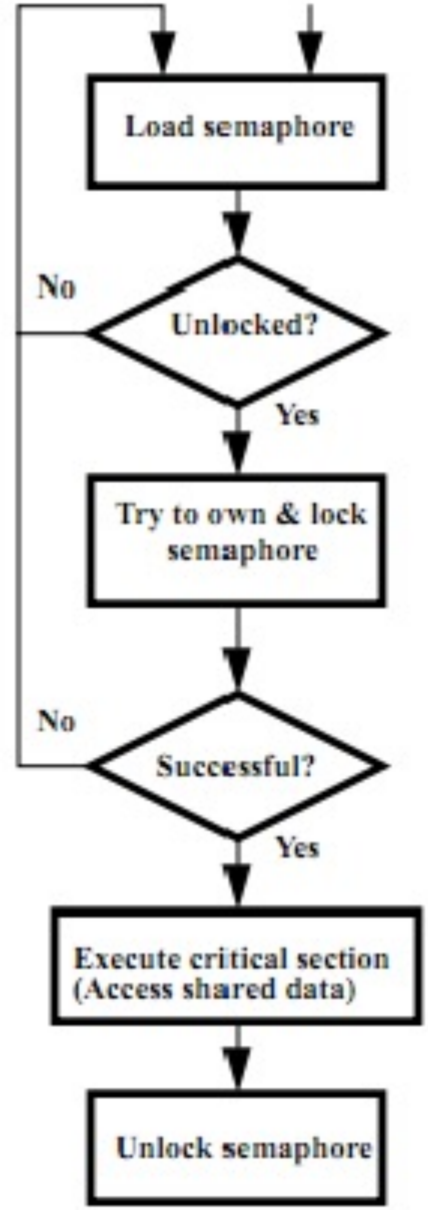
- Atomic swap (to test/set lock variable)
- Exchange contents of register and memory:
  s4 ↔ Mem(s1)

```
try:
   lr   t1, s1         #load reserved
   sc   t0, s1, s4     #store conditional
   bne  t0, x0, try    #loop if sc fails
   add  s4, x0, t1     #load value in s4
```

sc would fail if another threads executes **sc** here

Test-and-Set

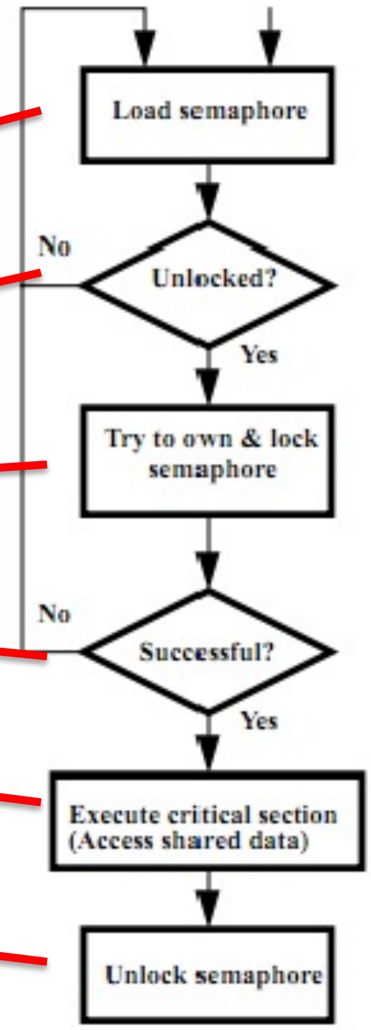
- In a single atomic operation:
  - ***Test*** to see if a memory location is set (contains a 1)
  - ***Set*** it (to 1) if it isn't (it contained a zero when tested)
    - Otherwise indicate that the Set failed, so the program can try again
  - While accessing, no other instruction can modify the memory location, including other Test-and-Set instructions
- Useful for implementing lock operations



Test-and-Set in RSIC-V using lr/sc

- Example: RISC-V sequence for implementing a T&S at (s1)

```
        li t2, 1
Try:
        lr t1, s1
        bne t1, x0, Try
        sc t0, s1, t2
        bne t0, x0, Try
Locked:
        # critical section
Unlock:
        sw x0,0(s1)
```



Option 2: RISC-V Atomic Memory Operations (AMOs)

- Encoded with an R-type instruction format
  - swap, add, and, or, xor, max, min
  - **AMOSWAP** `rd, rs2, (rs1)`
  - **AMOADD** `rd, rs2, (rs1)`
- Take the value pointed to by `rs1`
  - Load it into `rd`
  - Apply the operation to that value with the contents in `rs2`
    - If `rs2==rd`, use the old value in `rd`
  - Store the result back to where `rs1` is pointed to
- This allows atomic swap as a primitive
  - It also allows "reduction operations" that are common to be efficiently implemented

aq(acquire) and rl(release) to insure *in order* execution

RISC-V Critical Section

- Assume that the lock is in memory location stored in register a0
- The lock is "set" if it is 1; it is "free" if it is 0 (it's initial value)

```
      li            t0, 1          # Get 1 to set lock
Try:  amoswap.w.aq  t1, t0, (a0)   # t1 gets old lock value
                                   # while we set it to 1
      bnez          t1, Try        # if it was already 1, another
                                   # thread has the lock,
                                   # so we need to try again
         … critical section goes here …
      amoswap.w.rl  x0, x0, (a0)   # store 0 in lock to release
```

Lock Synchronization

Broken Synchronization

```
while (lock != 0) ;

lock = 1;

// critical section

lock = 0;
```

Fix (lock is at location (a0))

```
      li             t0, 1
Try:  amoswap.w.aq t1, t0, (a0)
      bnez         t1, Try
Locked:

      # critical section

Unlock:
      amoswap.w.rl  x0, x0, (a0)
```

How to use

- Don’t implement yourself!
- Use according library – e.g.:
  - pthread
  - C++:
    - std::thread C++11 https://en.cppreference.com/w/cpp/thread
    - std::jthread C++20
    - std::mutex; std::lock_guard; std::scoped_lock; std::shared_lock
    - std::condition_variable; std::counting_semaphore; std::latch; std::barrier
    - std::promise; std::future
  - Qt QThread
  - OpenMP

And in Conclusion, …

- Sequential software is slow software
  - SIMD and MIMD only path to higher performance
- Multithreading increases utilization, Multicore more processors (MIMD)
- OpenMP as simple parallel extension to C
  - Threads, Parallel for, private, critical sections, …
  - ≈ C: small so easy to learn, but not very high level and it's easy to get into trouble