

CS 110

Computer Architecture

Thread-Level Parallelism (TLP)

and OpenMP Intro

Instructor:
Sören Schwertfeger

<https://robotics.shanghaitech.edu.cn/courses/ca>

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley'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

DGEMM Speed Comparison

- Double precision general matrix matrix multiply: DGEMM
- Intel Core i7-5557U CPU @ 3.10 GHz
 - Instructions per clock (mul_pd) 2; Parallel multiplies per instruction 4
 - => 24.8 GFLOPS
- Python:

```
def dgemm(N, a, b, c):  
    for i in range(N):  
        for j in range(N):  
            c[i+j*N] = 0  
            for k in range(N):  
                c[i+j*N] += a[i+k*N] * b[k+j*N]
```

N	Python [Mflops]
32	5.4
160	5.5
480	5.4
960	5.3

- 1 MFLOP = 1 Million floating-point operations per second (fadd, fmul)
- **dgemm(N ...)** takes $2 \cdot N^3$ flops

C versus Python

- $c = a * b$
- a, b, c are $N \times N$ matrices

```
// Scalar; P&H p. 226
void dgemm_scalar(int N, double *a, double *b, double *c) {
    for (int i=0; i<N; i++)
        for (int j=0; j<N; j++) {
            double cij = 0;
            for (int k=0; k<N; k++)
                //      a[i][k] * b[k][j]
                cij += a[i+k*N] * b[k+j*N];
            // c[i][j]
            c[i+j*N] = cij;
        }
}
```

N	C [GFLOPS]	Python [GFLOPS]
32	1.30	0.0054
160	1.30	0.0055
480	1.32	0.0054
960	0.91	0.0053



240x!

Vectorized dgemm

```
// AVX intrinsics; P&H p. 227
void dgemm_avx(int N, double *a, double *b, double *c) {
    // avx operates on 4 doubles in parallel
    for (int i=0; i<N; i+=4) {
        for (int j=0; j<N; j++) {
            // c0 = c[i][j]
            __m256d c0 = {0,0,0,0};
            for (int k=0; k<N; k++) {
                c0 = _mm256_add_pd(
                    c0, // c0 += a[i][k] * b[k][j]
                    _mm256_mul_pd(
                        _mm256_load_pd(a+i+k*N),
                        _mm256_broadcast_sd(b+k+j*N)));
            }
            _mm256_store_pd(c+i+j*N, c0); // c[i,j] = c0
        }
    }
}
```



N	Gflops	
	scalar	avx
32	1.30	4.56
160	1.30	5.47
480	1.32	5.27
960	0.91	3.64

- 4x faster
- Still << theoretical 25 GFLOPS

Loop Unrolling

// Loop unrolling; P&H p. 352

```
const int UNROLL = 4;
```

```
void dgemm_unroll(int n, double *A, double *B, double *C) {  
    for (int i=0; i<n; i+= UNROLL*4) {  
        for (int j=0; j<n; j++) {  
            __m256d c[4];  4 registers  
            for (int x=0; x<UNROLL; x++)  
                c[x] = _mm256_load_pd(C+i+x*4+j*n);  
            for (int k=0; k<n; k++) {  
                __m256d b = _mm256_broadcast_sd(B+k+j*n);  
                for (int x=0; x<UNROLL; x++)  Compiler does the unrolling  
                    c[x] = _mm256_add_pd(c[x],  
                                       _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));  
            }  
            for (int x=0; x<UNROLL; x++)  
                _mm256_store_pd(C+i+x*4+j*n, c[x]);  
        }  
    }  
}
```

N	GFlops		
	scalar	avx	unroll
32	1.30	4.56	12.95
160	1.30	5.47	19.70
480	1.32	5.27	14.50
960	0.91	3.64	6.91 ?

FPU versus Memory Access

- How many floating-point operations does matrix multiply take?
 - $F = 2 \times N^3$ (N^3 multiplies, N^3 adds)
- How many memory load/stores?
 - $M = 3 \times N^2$ (for A, B, C)
- Many more floating-point operations than memory accesses
 - $q = F/M = 2/3 * N$
 - Good, since arithmetic is faster than memory access
 - Let's check the code ...

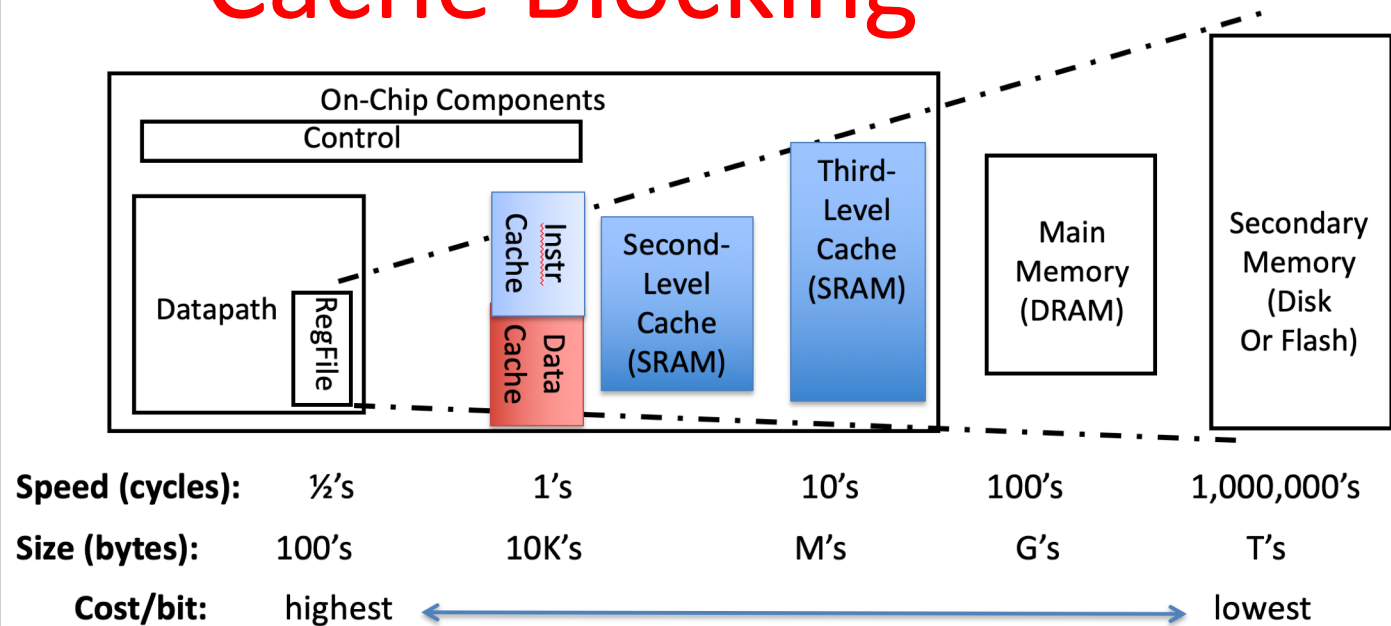
But memory is accessed repeatedly

- $q = F/M = 1.6!$ (1.25 loads and 2 floating-point operations)

Inner loop:

```
for (int k=0; k<N; k++) {  
    c0 = _mm256_add_pd(  
        c0, // c0 += a[i][k] * b[k][j]  
        _mm256_mul_pd(  
            _mm256_load_pd(a+i+k*N),  
            _mm256_broadcast_sd(b+k+j*N)));  
}
```

Cache Blocking



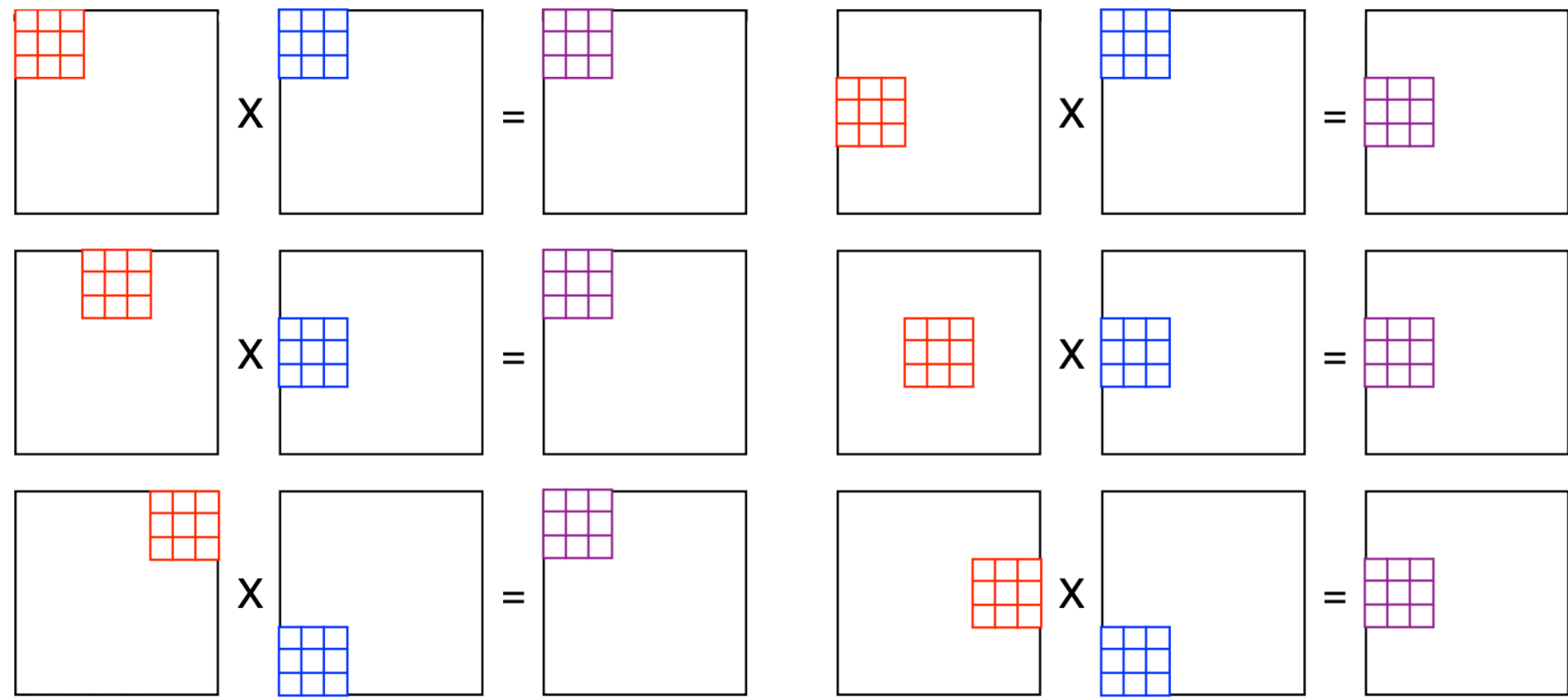
- Where are the operands (A, B, C) stored?
- What happens as N increases?
- Idea: arrange that most accesses are to fast cache!
- Rearrange code to use values loaded in cache many times
- Only “few” accesses to slow main memory (DRAM) per floating point operation

P&H, RISC-V edition p. 465

— -> throughput limited by FP hardware and cache, not slow DRAM

Blocking Matrix Multiply

(divide and conquer: sub-matrix multiplication)



Memory Access Blocking

```
// Cache blocking; P&H p. 556
const int BLOCKSIZE = 32;

void do_block(int n, int si, int sj, int sk, double *A, double *B, double *C) {
    for (int i=si; i<si+BLOCKSIZE; i+=UNROLL*4)
        for (int j=sj; j<sj+BLOCKSIZE; j++) {
            __m256d c[4];
            for (int x=0; x<UNROLL; x++)
                c[x] = _mm256_load_pd(C+i+x*4+j*n);
            for (int k=sk; k<sk+BLOCKSIZE; k++) {
                __m256d b = _mm256_broadcast_sd(B+k+j*n);
                for (int x=0; x<UNROLL; x++)
                    c[x] = _mm256_add_pd(c[x],
                                         _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
            }
            for (int x=0; x<UNROLL; x++)
                _mm256_store_pd(C+i+x*4+j*n, c[x]);
        }
}

void dgemm_block(int n, double* A, double* B, double* C) {
    for(int sj=0; sj<n; sj+=BLOCKSIZE)
        for(int si=0; si<n; si+=BLOCKSIZE)
            for (int sk=0; sk<n; sk += BLOCKSIZE)
                do_block(n, si, sj, sk, A, B, C);
}
```

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

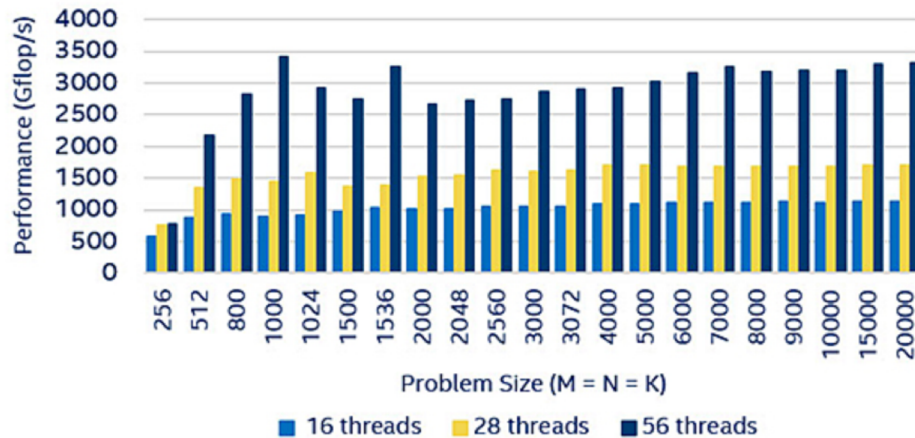
N	Size	GFlops			
		scalar	avx	unroll	blocking
32	3x 8KB	1.30	4.56	12.95	13.80
160	3x 205KB	1.30	5.47	19.70	21.79
480	3x 1.8MB	1.32	5.27	14.50	20.17
960	3x 7.3MB	0.91	3.64	6.91	15.82

Intel Math Kernel Library

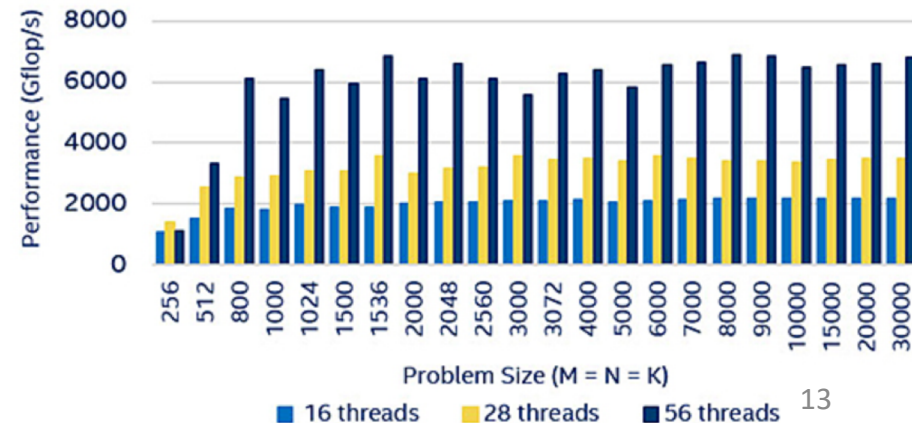
- AVX programming too hard? Use MKL!
 - C/C++ and Fortran for Windows, Linux, macOS
- Knowledge about AVX still very helpful for using MKL (e.g. Cache blocking, ...)
- MKL also for multi-threading...

DGEMM, SGEMM Optimized by Intel® Math Kernel Library on Intel® Xeon® Processor

DGEMM on Intel® Xeon® Platinum 8180 Processor
2.50GHz



SGEMM on Intel® Xeon® Platinum 8180 Processor
2.50 GHz



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

Software

Hardware

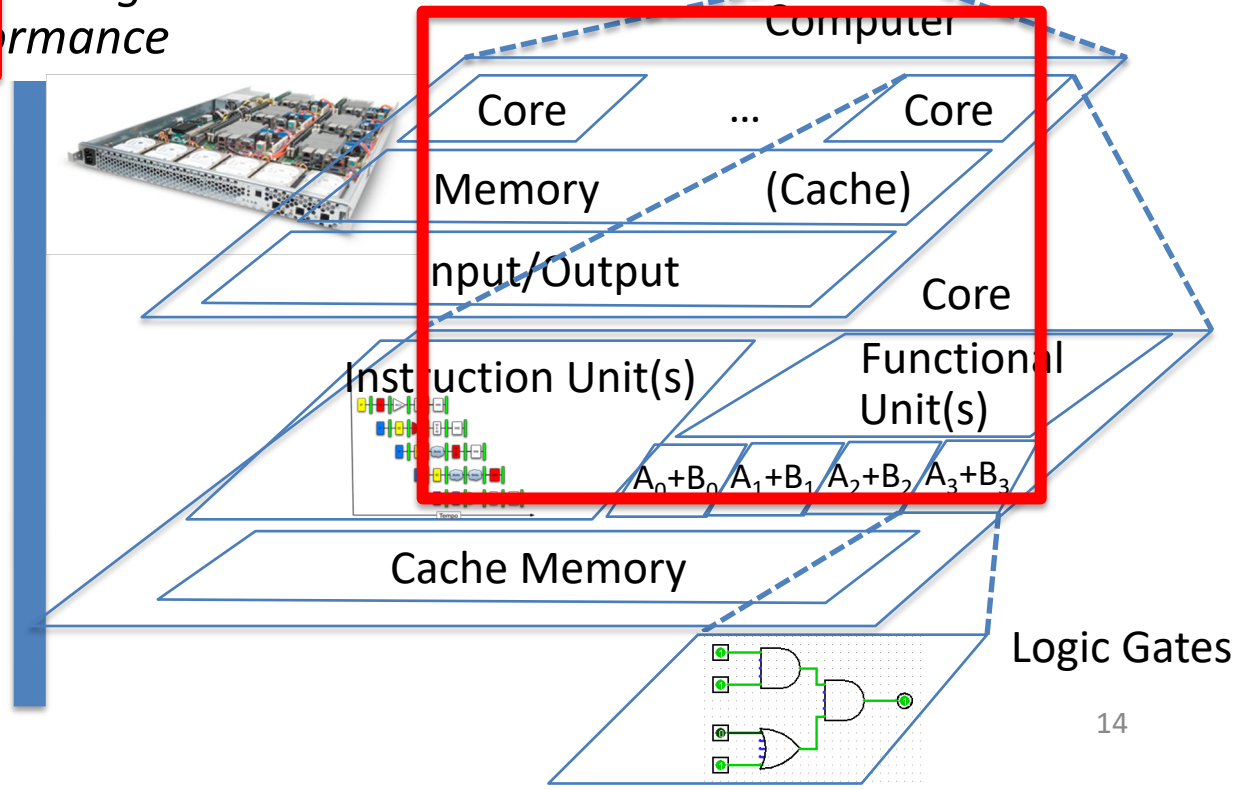
- Parallel Requests
Assigned to computer
e.g., Search "Katz"
- Parallel Threads
Assigned to core
e.g., Lookup, Ads
- Parallel Instructions
>1 instruction @ one time
e.g., 5 pipelined instructions
- Parallel Data
>1 data item @ one time
e.g., Add of 4 pairs of words
- Hardware descriptions
All gates @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

*Harness
Parallelism &
Achieve High
Performance*

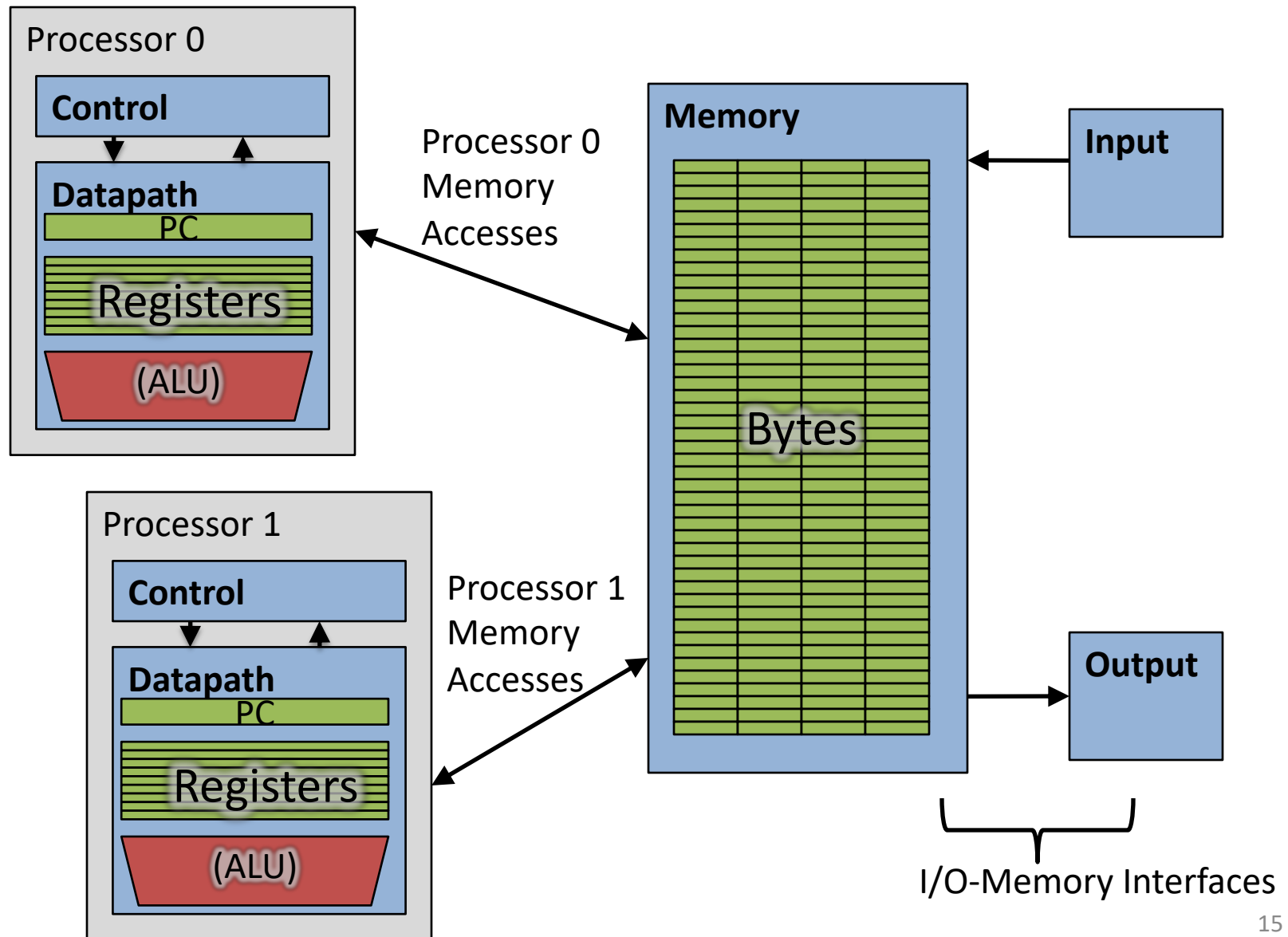
Warehouse
Scale
Computer



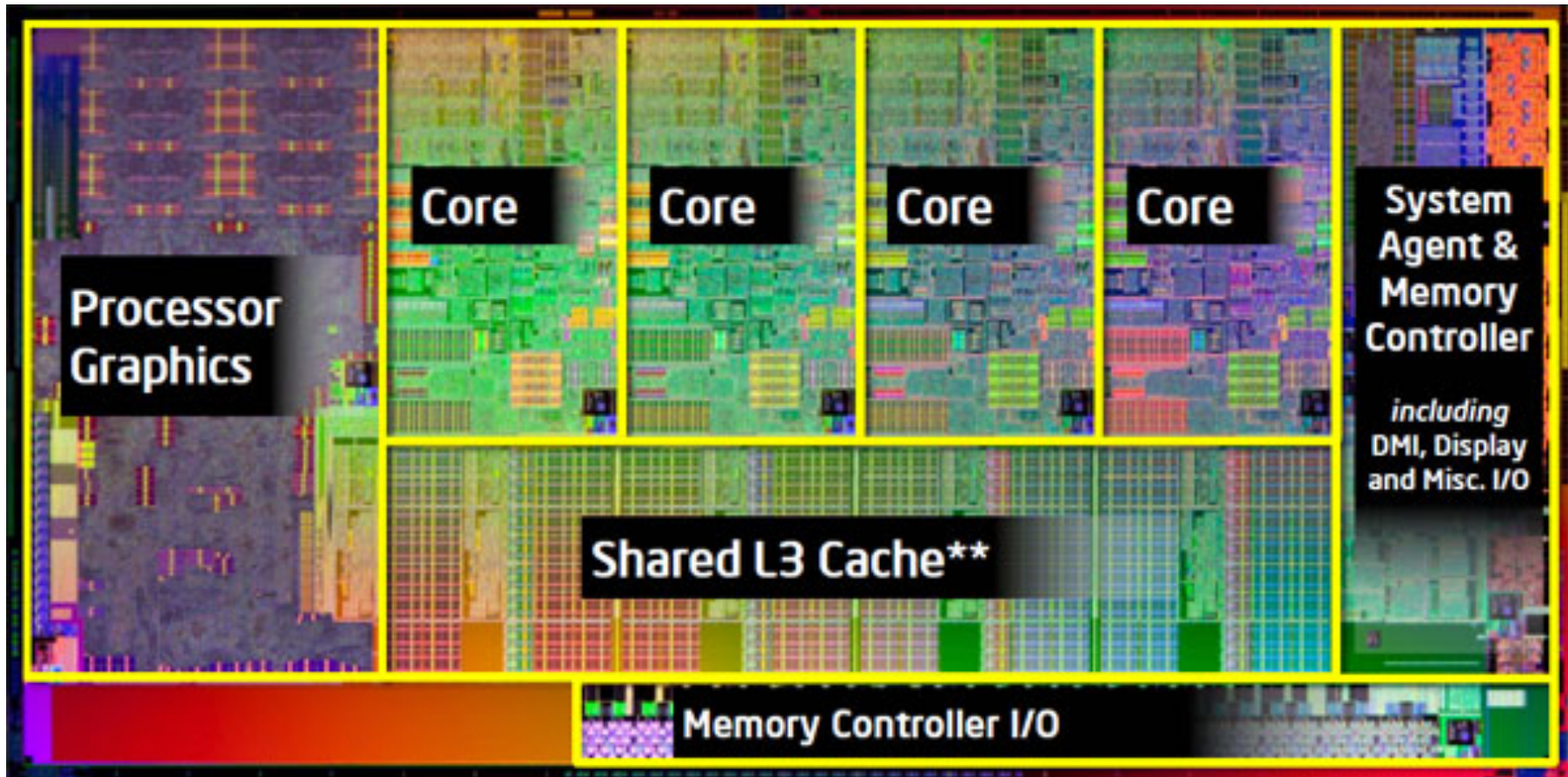
Smart
Phone



Simple Multiprocessor



4 Core Processor with Graphics

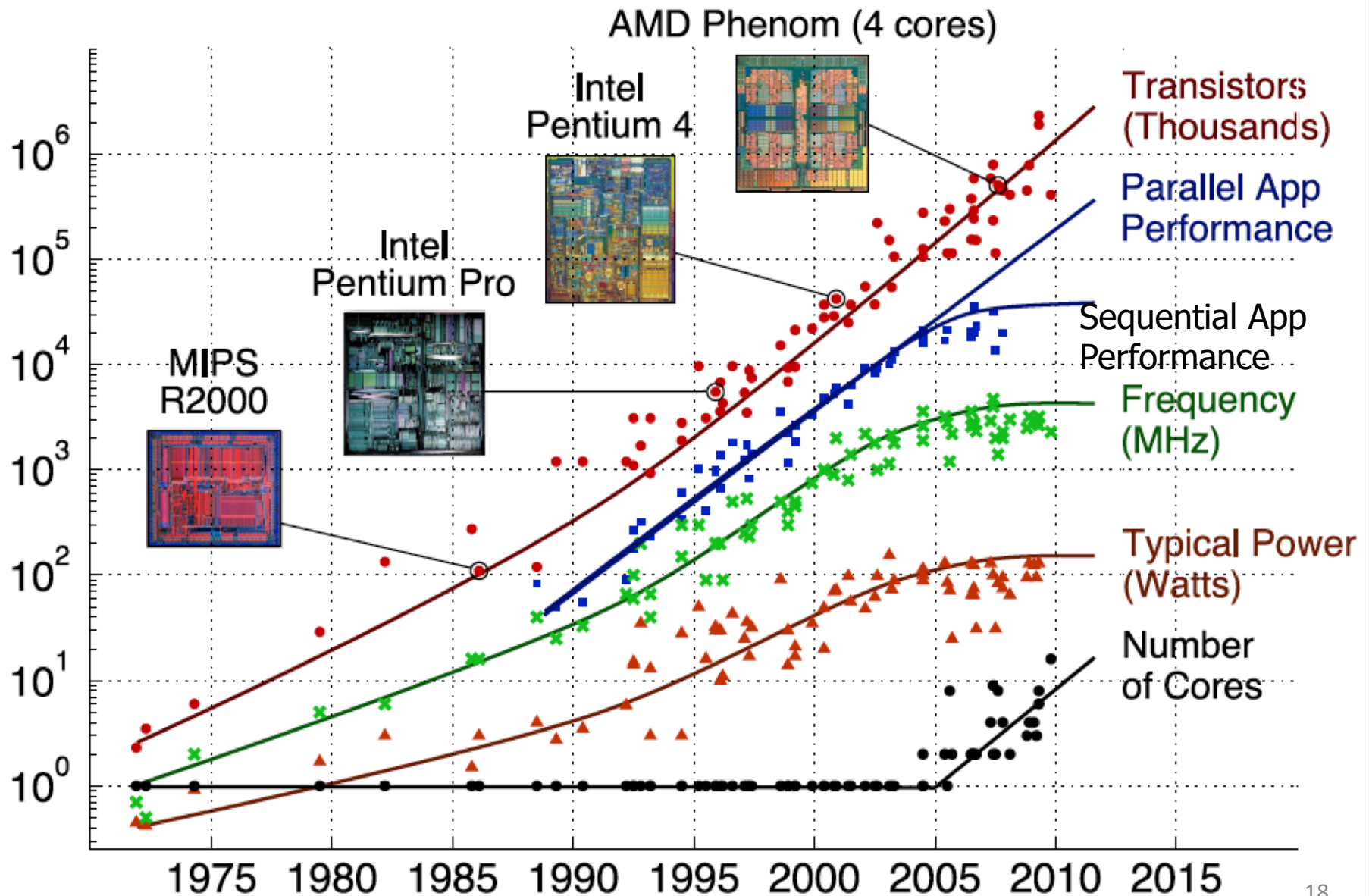


Multiprocessor Execution Model

- 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



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 Zen: 4-32 real cores
- Apple A11: 2 (high performance) + 4 (low power)
- Samsung S9 (Qualcomm Snapdragon): 4 + 4

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

Multiprocessors and You

- Only path to performance is parallelism
 - Clock rates flat or declining
 - SIMD: 2X width every 3-4 years
 - 512b Xeon Processors, 1024b in 2019?
 - 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
- Project 3: fastest code on 10-core computer (SIMD and MIMD!)

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

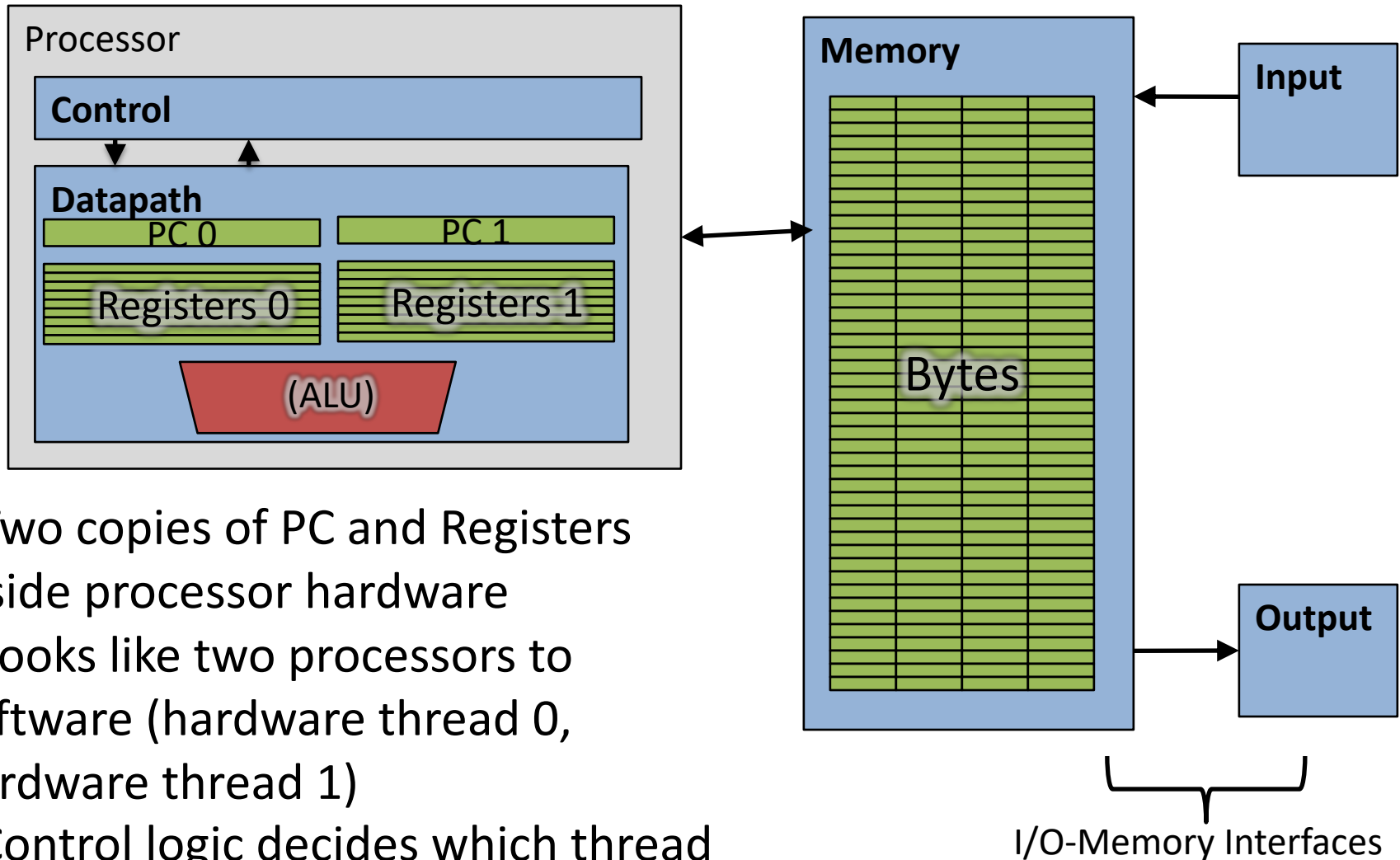
Give the illusion of many active threads by time-multiplexing software threads onto hardware threads

- 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

- 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 (aka Hyperthreading)



Multithreading vs. Multicore

- Multithreading => Better Utilization
 - $\approx 5\%$ 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
 - $\approx 50\%$ more hardware, $\approx 2X$ better performance?
 - Share outer caches (L2\$, L3\$), Memory Controller
- Modern machines do both
 - Multiple cores with multiple threads per core

Sören's MacBook

- `sysctl -a | grep hw\.`

MacBookPro11,3

hw.cachelinesize = 64

...

hw.l1icachesize: 32,768

hw.physicalcpu: 4

hw.l1dcachesize: 32,768

hw.logicalcpu: 8

hw.l2cachesize: 262,144

...

hw.l3cachesize: 6,291,456

hw.cpufrequency =
2,800,000,000

hw.memsize = 17,179,869,184

on Linux:

`cat /proc/cpuinfo`

Sören's iPad Air 2

Apple A8X processor:

- 3 cores!
- L1 \$: 64KB data, 64KB instruction
- L2 \$: 2MB
- L3 \$: 4MB
- Max 1.5GHz clock
- 64bit ARM ISA
- 2 GB RAM

Comparison:

iPad Pro: A9X processor:

- Back to 2 cores...
- L1 \$: 64KB data, 64KB instruction
- L2 \$: 3MB
- No L3 \$: double memory bandwidth...
- 4 or 2 GB RAM

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

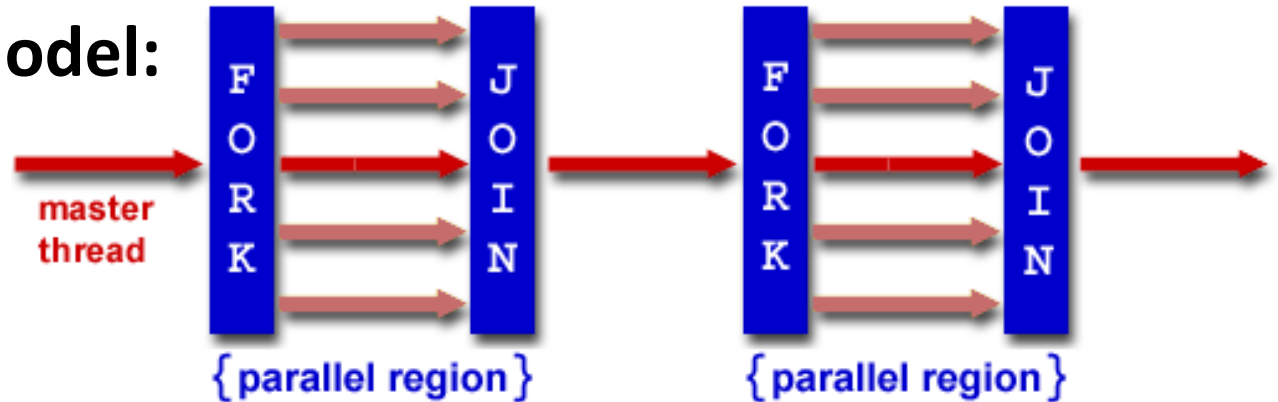
- Multiple threads in a shared memory environment, explicit programming model with full programmer control over parallelization
- **Pros:**
 - Takes advantage of shared memory, programmer need not worry (that much) about data placement
 - Compiler directives are simple and easy to use
 - Legacy serial code does not need to be rewritten
- **Cons:**
 - Code can only be run in shared memory environments
 - Compiler must support OpenMP (e.g. gcc 4.2)

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)`

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!

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 */
}
```

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 Milk


- Your fridge has no milk. 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 milk, and return
- What if the other person gets back while the first person is buying milk?
 - You've just bought twice as much milk 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:

Check lock  Can loop/idle here
if locked

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 \leftrightarrow memory
 - Pair of instr? One for read, one for write

Hardware Synchronization

- Solution:
 - Atomic read/write
 - Read & write in single instruction
 - No other access permitted between read and write
 - Note:
 - Must use *shared memory* (multiprocessing)
- Common implementations:
 - Atomic swap of register \leftrightarrow memory
 - Pair of instructions for “linked” read and write
 - write fails if memory location has been “tampered” with after linked read
- RISC-V has variations of both, but for simplicity we will focus on the former

RISC-V Atomic Memory Operations (AMOs)

- AMOs atomically perform an operation on an operand in memory and set the destination register to the original memory value
- R-Type Instruction Format: Add, And, Or, Swap, Xor, Max, Max Unsigned, Min, Min Unsigned

31	27	26	25	24	20	19	15	14	12	11	7	6	0
funct5		aq	rl	rs2		rs1		funct3		rd		opcode	
5		1	1	5		5		3		5		7	
operation		ordering		src		addr		width		dest		AMO	

Load from address in rs1 to "t"
 rd = "t", i.e., the value in memory
 Store at address in rs1 the calculation
 "t" <operation> rs2
 aq(acquire) and rl(release) to insure *in order* execution

```

amoadd.w rd,rs2,(rs1):
    t = M[x[rs1]];
    x[rd] = t;
    M[x[rs1]] = t + x[rs2]
  
```

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)

```
Try:  li          t0, 1          # Get 1 to set lock
      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)
```

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, ...
 - \approx C: small so easy to learn, but not very high level and it's easy to get into trouble