

# Parallelism 2

# Performance: Added SIMD intrinsics to Matrix Multiply

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
- But still << theoretical 25 GFLOPS!

# Amdahl's Law\* applied to `dgemm`

- Measured `dgemm` performance
  - Peak 5.5 GFLOPS
  - Large matrices 3.6 GFLOPS
  - Processor 24.8 GFLOPS
- Why are we not getting (close to) 25 GFLOPS?
  - Something else (not floating-point ALU) is limiting performance!
  - But what? Possible culprits:
    - Cache
    - Hazards
    - Let's look at both!

# “Vectorized” dgemm: Pipeline Hazards

```
// 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
        }
    }
}
```



“add\_pd” depends on result of “mult\_pd” which depends on “load\_pd”

# Loop Unrolling

- On high performance processors, optimizing compilers performs “loop unrolling” operation to expose more parallelism and improve performance:

```
for(i=0; i<N; i++)
    x[i] = x[i] + s;

```

- Could become:

```
for(i=0; i<N; i+=4) {
    x[i]      = x[i] + s;
    x[i+1]    = x[i+1] + s;
    x[i+2]    = x[i+2] + s;
    x[i+3]    = x[i+3] + s;
}
```

1. Expose data-level parallelism for vector (SIMD) instructions or super-scalar multiple instruction issue
2. Mix pipeline with unrelated operations to help with reduce hazards
3. Reduce loop “overhead”
4. Makes code size larger

# 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]);
        }
    }
}
```

How do you verify that the generated code is actually unrolled?

# Performance

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

WOW!

?

# FPU versus Memory Access

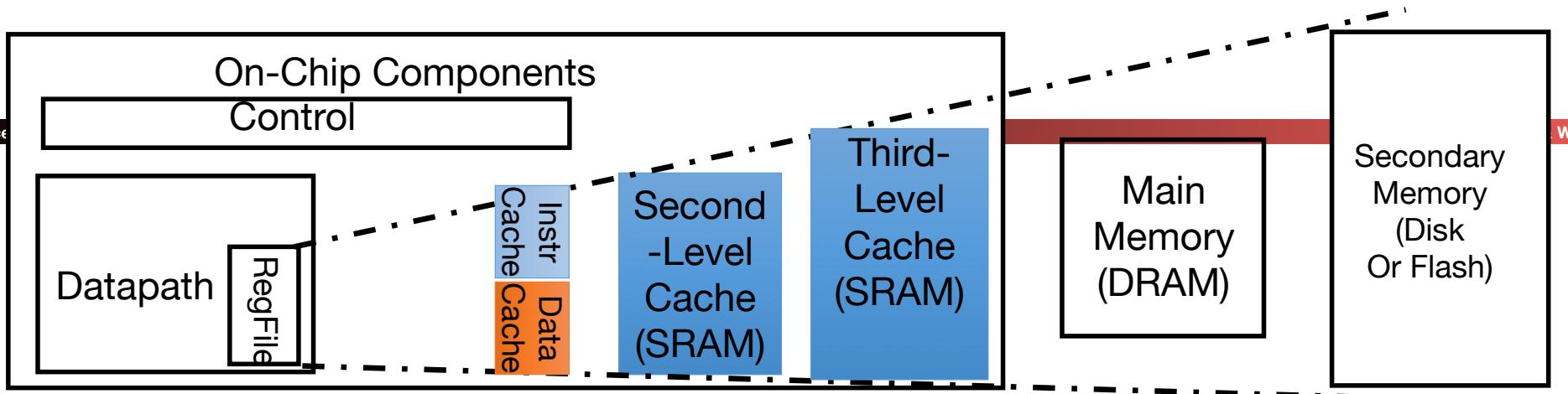
- How many floating-point operations does matrix multiply take?
  - $F = 2 \times N^3$  ( $N^3$  multiplies,  $N^3$  adds)
- How many memory load/stores must occur?
  - $M = 3 \times N^2$  (for A, B, C)
- Many more floating-point operations than memory accesses
  - $q = F/M = 2/3 * N$
  - Good, since 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)  
(and really, 2 loads if you think in terms of cost...)

## 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)));
}
```



<b>Speed (cycles):</b>	½'s	1's	10's	100's-1000	1,000,000's
<b>Size (bytes):</b>	100's	10K's	M's	G's	T's
<b>Cost/bit:</b>	highest				lowest

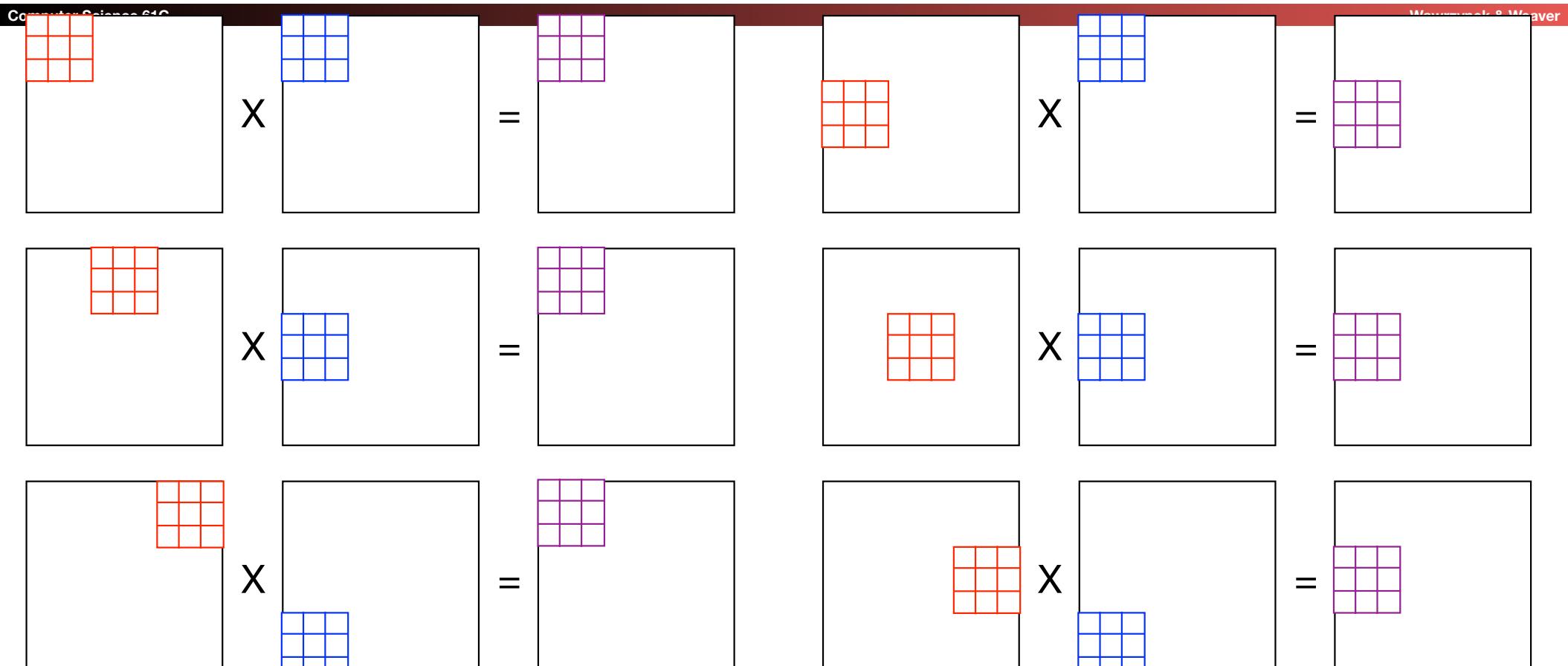
- Where are the operands (A, B, C) stored?
- What happens as N increases?
- **Idea:** arrange that most accesses are to fast cache!

# Blocking

- Idea:
  - Rearrange code to use values loaded in cache many times
  - Only “few” accesses to slow main memory (DRAM) per floating point operation
    - -> throughput limited by FP hardware and cache, not slow DRAM
  - P&H, RISC-V edition p. 465

# 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

N	Gflops			
	scalar	avx	unroll	blocking
32	1.30	4.56	12.95	13.80
160	1.30	5.47	19.70	21.79
480	1.32	5.27	14.50	20.17
960	0.91	3.64	6.91	15.82

# Big Idea Reminder: 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



$$\text{Execution Time w/ E} = \text{Execution Time w/o E} \times [(1-F) + F/S]$$

$$\text{Speedup w/ E} = 1 / [(1-F) + F/S]$$

# Big Idea: Amdahl's Law

Speedup =

$$\frac{1}{(1 - F) + \frac{F}{S}}$$

Non-speed-up part  $\rightarrow$   $F < 1$

Speed-up part  $\rightarrow$   $S > 1 = \frac{\text{Original Time}}{\text{New time}}$

$\frac{F}{S} > 1$

Example: the execution time of half of the program can be accelerated by a factor of 2.  
What is the program speed-up overall?

$$\frac{1}{\frac{0.5 + 0.5}{2}} = \frac{1}{0.5 + 0.25} = 1.33$$

# Example #1: Amdahl's Law

$$\text{Speedup w/ E} = 1 / [ (1-F) + F/S ]$$

- Consider an enhancement which runs 20 times faster but which is only usable 25% of the time

$$\text{Speedup w/ E} = 1/(.75 + .25/20) = 1.31$$

- What if its usable only 15% of the time?

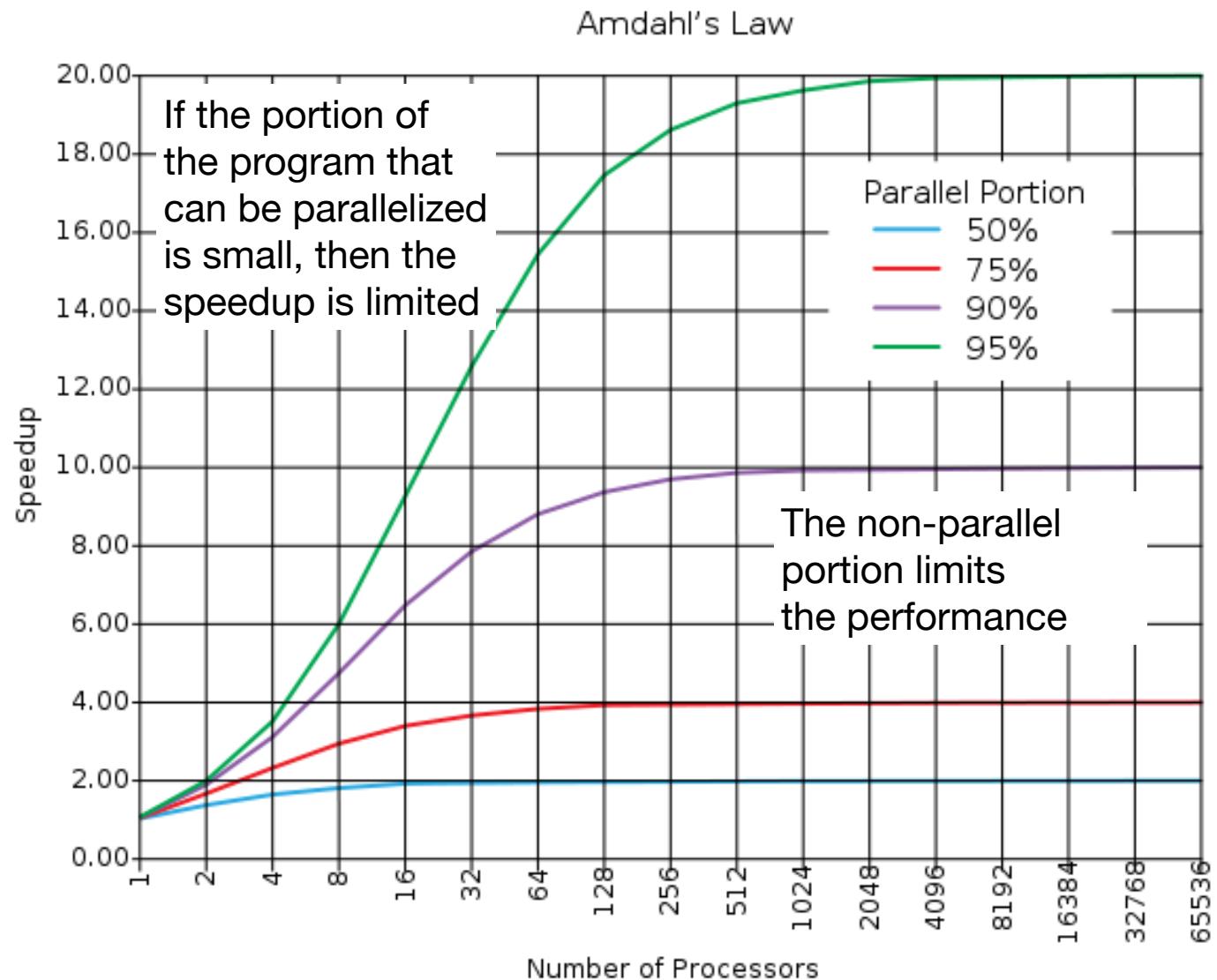
$$\text{Speedup w/ E} = 1/(.85 + .15/20) = 1.17$$

- Amdahl's Law tells us that to achieve linear speedup with 100 processors, none of the original computation can be scalar!
- To get a speedup of 90 from 100 processors, the percentage of the original program that could be scalar would have to be 0.1% or less

$$\text{Speedup w/ E} = 1/(.001 + .999/100) = 90.99$$

# Amdahl's Law

Computer Science 61C



# Strong and Weak Scaling

- To get good speedup on a parallel processor while keeping the problem size fixed is harder than getting good speedup by increasing the size of the problem.
  - **Strong scaling:** when speedup can be achieved on a parallel processor without increasing the size of the problem
  - **Weak scaling:** when speedup is achieved on a parallel processor by increasing the size of the problem proportionally to the increase in the number of processors
- **Load balancing** is another important factor: every processor doing same amount of work
  - Just one unit with twice the load of others cuts speedup almost in half

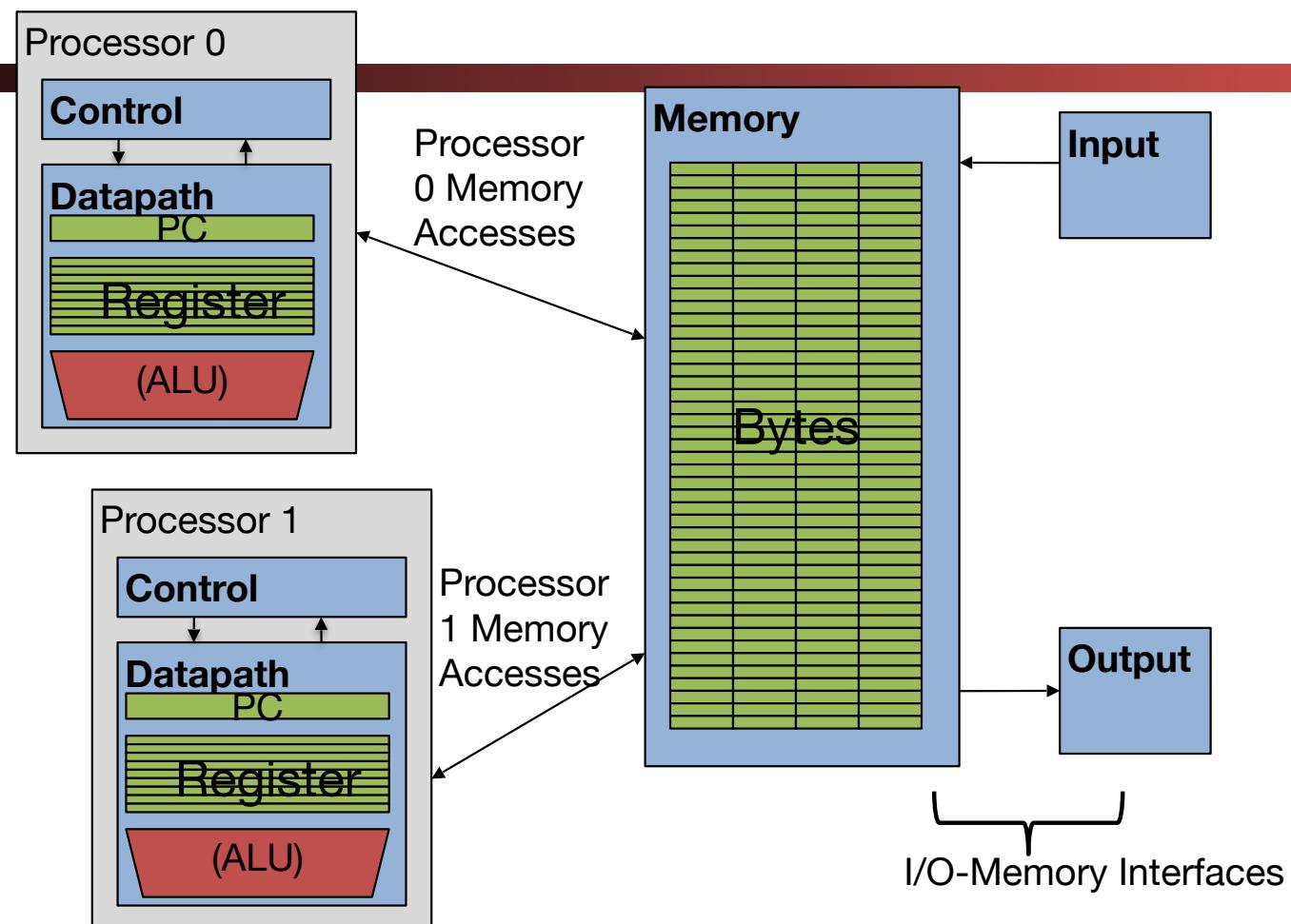
# Amdahl's Law In The Real World...

- Lets look at the federal budget:
  - Price of a ***single*** F35: ~\$100M
    - Air Force ***alone*** wants to buy ~ 50 next year
  - Line item: “Purchase F35 fighter jets for the Air Force:” ~\$5B
    - This doesn't include the Navy's Air Force's purchases...  
Or the Navy's Army's Air Force's purchases...
  - Line item: “Fund Corporation for Public Broadcasting:” ~\$500M
- If you want to reduce the cost of the federal government...
  - Which line item is more significant?

# Amdahl's Law and Premature Optimization...

- The runtime of a new program is really...
  - The runtime of the program on all the inputs you ever run it on
  - The time it takes you to **write** the program in the first place!
- So don't **prematurely optimize**
  - Worry about **getting things right first**, you may never have to optimize it at all
  - Likewise, **worry about making your code readable and well documented:**
    - Since the **runtime of a modified version of the program is the runtime on all inputs plus the time it takes you to relearn what you did in order to modify it!**

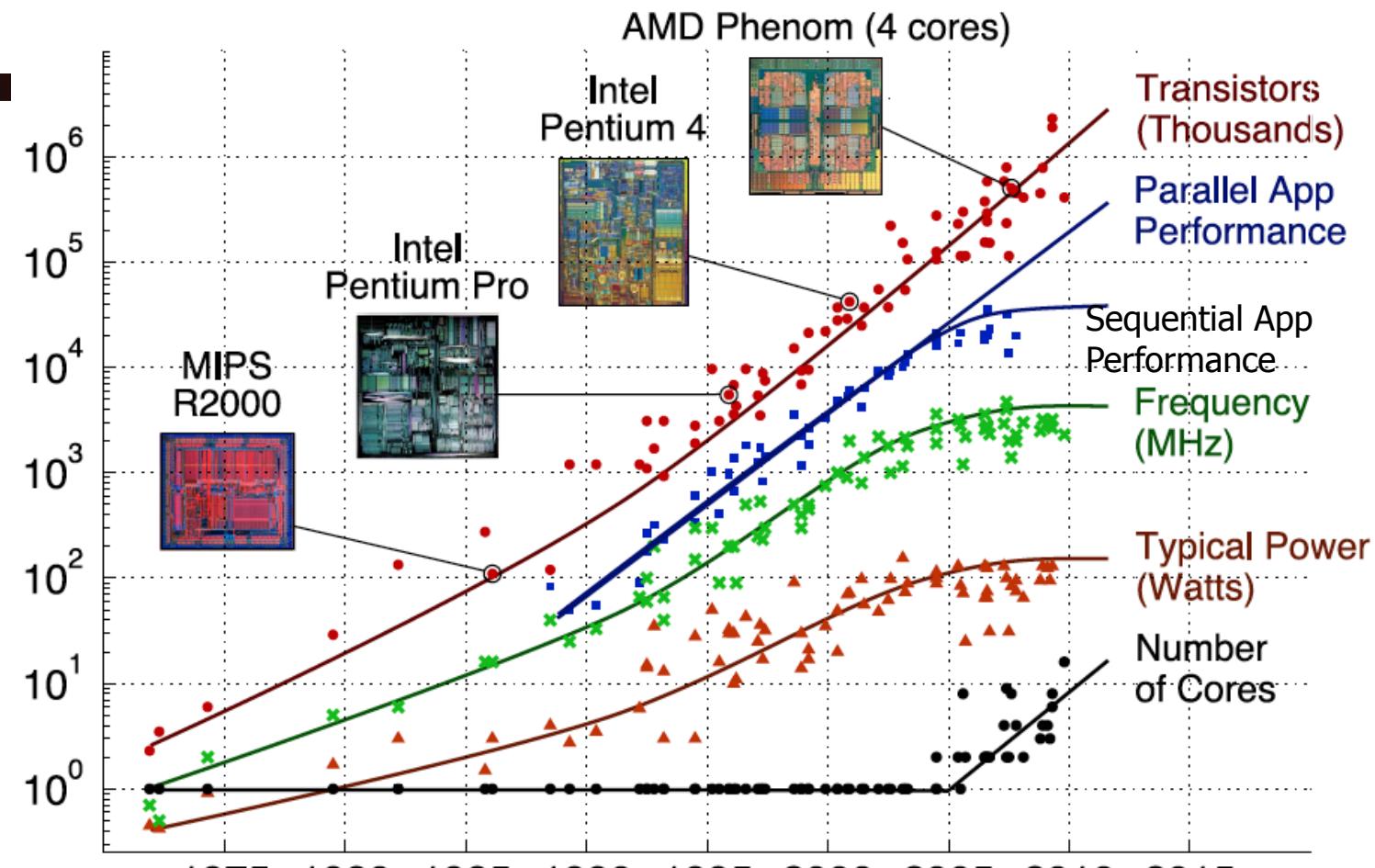
# Simple Multiprocessor



# 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:
  - Deliver high throughput for independent jobs via job-level parallelism
    - E.g. your operating system & different programs
  - 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



Data partially collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond

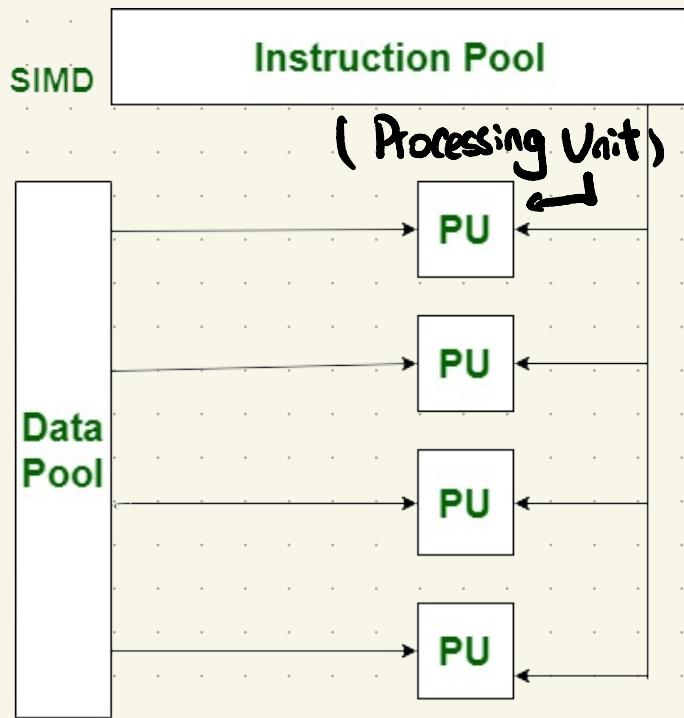
# Parallelism the Only Path to Higher Performance

- Sequential processor performance not expected to increase much:
  - We pretty much hit a brick wall a few years back in our ability to improve single-thread performance: Apple got a one-time boost with transition to ARM
- If want apps with more capability we have to embrace parallel processing (SIMD and MIMD)
- In mobile systems, use multiple cores and GPUs
  - All iPhones starting with the 4s are multicore
  - iPhone 12 CPU is 6 cores!
    - Two cores very fast: Burn lots of power but very good sequential performance
    - Four cores power efficient: Lower sequential performance but better ops/joule
    - Plus a 4 core GPU
    - Plus a 16 core processor for machine learning (optimized for 16b floating point!)
- In warehouse-scale computers, use multiple nodes, and all the MIMD/SIMD capability of each node

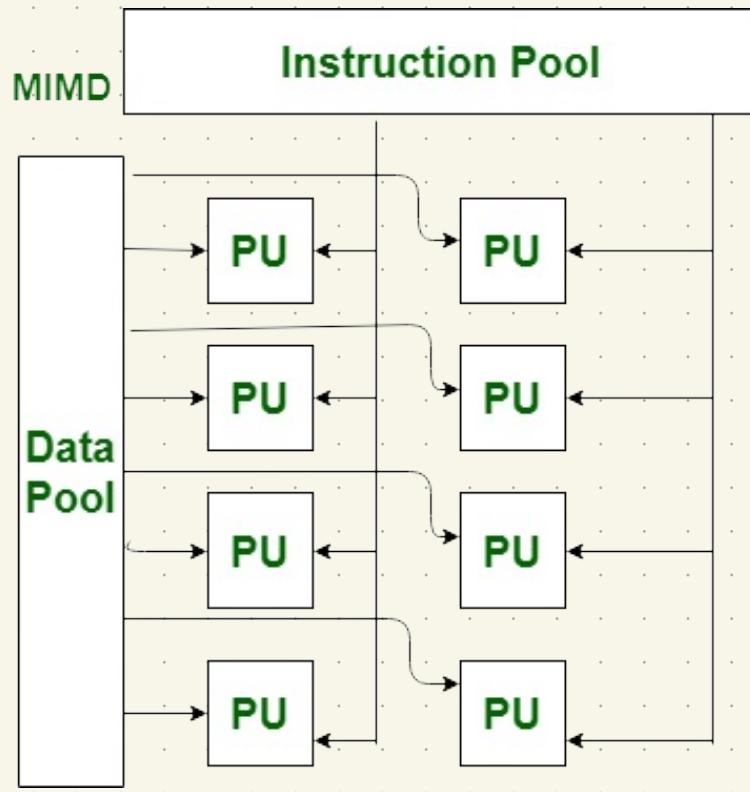
# Comparing Types of Parallelism...

- 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: so 500x-1000x performance!
  - Common approach is "vector" like we see with Intel AVX:
    - EG, 512b vector of double-precision floating point: 8 elements at a time
- MIMD-type parallelism (data-dependent Branches!)
  - A MIMD-favorable problem **will not map easily** to a SIMD-type fabric

# SIMD (Single Instruction Multiple Data)



# MIMD (Multiple Instruction Multiple Data)



Mem ↓ (Inst ↓)

Cost ↓

1 Decoder

∅ Efficiency ↓

Latent / Tact  
Synchronization  
Complexity ↓

Mem ↑

Cost ↑

2+ Decoder Complexity ↑

∅ Efficiency ↑

Accurate or  
explicit sync.

# Multiprocessors and You

- Only path to performance is parallelism
  - Clock rates flat or declining
  - CPI generally flat
  - SIMD now ~4-16 words wide on the CPU
  - SIMD accelerators even more
    - Nvidia GP100 GPU: 5 TFLOPs of 64b Floating Point, 10 for 32b FP  
1792 CUDA cores for 64b Floating Point (3584 for 32b)
  - 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
  - If you can scale up you can ***then scale down***

# Threads

- **Thread:** a sequential flow of instructions that performs some task
- Each thread has a PC + processor registers and accesses the ***shared memory of the process***
- Each core provides one or more ***hardware threads*** that actively execute instructions
  - Common Intel chips support 2 threads/core
    - So a 4 core Intel processor can support 8 hardware threads
    - The RPi4 has only 1 thread per core -> 4 cores -> 4 hardware threads
  - Operating system multiplexes multiple ***software threads*** onto the available ***hardware threads***

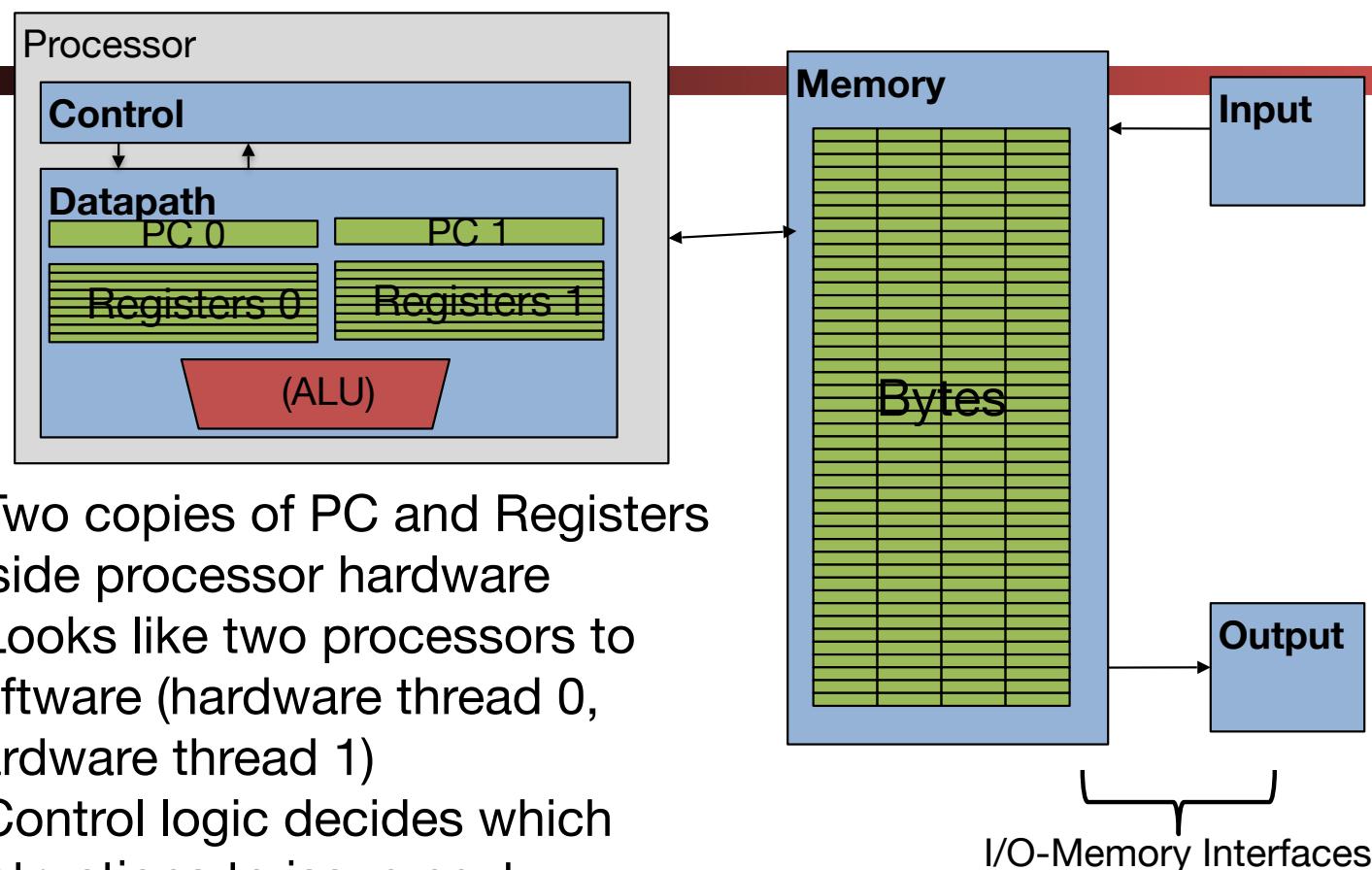
# Operating System Threads

- 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 switch to another thread
  - 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 is the biggest one
- 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 (Thread Level Parallelism)
  - Commercial multi-user workloads
- Intel calls this HyperThreading
  - Will actually issue from two threads at the same time!

# Hardware Multithreading



- 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 even mix from dif threads

# Multithreading vs. Multicore

- Multithreading => Better Utilization
  - $\approx 1\%$  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** ( $L2\$$  or just  $L3\$$ ), Memory Controller
- Modern machines do both
  - Multiple cores with multiple threads per core

# Nick's MacBook Pro

## MacBookPro 13" (2020)

- /usr/sbin/sysctl -a | grep hw\.

...

hw.physicalcpu: 4

hw.logicalcpu: 8

...

hw.cpufrequency =  
2,000,000,000

hw.memsize = 34,359,738,368

hw.cachelinesize = 64  
hw.l1icachesize: 32,768  
hw.l1dcachesize: 49,152  
hw.l2cachesize: 524,288  
hw.l3cachesize:  
6,291,456

# Nick's Zoom-Cave Beast

- AMD Ryzen 9 3900X 12 core CPU
  - 2 threads/core
- Nvidia 2080 GPU
  - 2944 CUDA SIMD processor cores
- **Gratuitous BlinkenLights...**
  - Hey, those are the factory lights on the CPU and GPU...
  - But I did get a transparent case...



# Nick's \$45 Raspberry Pi 4...

Quad-Core processor

1 thread/core

3-issue out-of-order superscalar,  
16 stage pipeline

128b wide SIMD/vector instructions  
(4x single precision floating point)

512 KB shared L2 cache

L1 I\$ is 48 KB

L1 D\$ is 32 KB

4 GB RAM

Gb Ethernet, 802.11, Bluetooth

- Even the smallest and cheapest systems are now heavily parallel
  - OK full kit cost \$75... With HDMI cable, power supply, case, SD-card



# Lastest modern processors: Big/Little design

- You need "big" processors for both single threaded and multi-threaded performance
  - After all, you don't want to wait around...
- But such processors are **very** inefficient
  - Lots of power, lots of silicon
  - And a lot of time you don't need a big processor, because you don't need the performance
- Modern big/little design
  - Intel Alder Lake: i9 version: 8 performance cores (with 2 threads/core) + 4 efficiency cores
    - Efficiency cores only support one thread/core, and are designed in a block of 4 with a shared L2 cache
  - Apple M1 Pro: 8 performance cores, 2 efficiency cores
    - All cores are single thread/core

# 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
  - But beyond the C language itself
- Easy to compile: cc –fopenmp name.c

# Shared Memory Model with Explicit Thread-based Parallelism

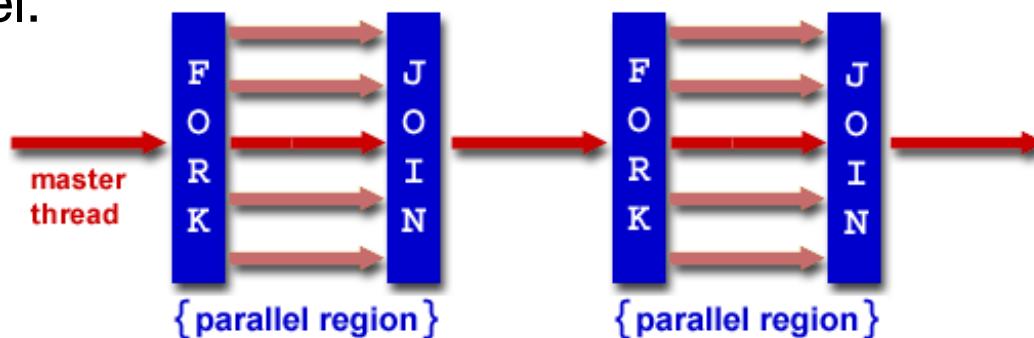
- Multiple threads in a shared memory environment, explicit programming model with full programmer control over parallelization
- **Pros:**
  - Takes advantage of shared memory, programmer need not worry (that much) about data placement
  - Compiler directives are simple and easy to use
  - Legacy serial 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)
  - Amdahl's law is gonna get you after not too many cores...

# OpenMP in CS61C

- 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

预处理指令

- **Pragmas** are a preprocessor mechanism C provides for language extensions
- Commonly implemented pragmas: structure packing, symbol aliasing, floating point exception modes (not covered in 61C)
- 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
  - EG, RPi 4 has 4 threads by default

# 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
  - Be careful when timing results!

# 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);
        /* Only master thread does this */
        if (tid == 0) {
            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 different threads attempts to access the 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 Beer Milk In the After Times...

- 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
    addi t1,x0,1      # t1 = Locked value
Loop:   lw t0,0(s0)    # load lock
        bne t0,x0,Loop  # loop if locked
Lock:  sw t1,0(s0)    # Unlocked, so lock
```

- Unlock

```
Unlock:
    sw x0,0(s0)
```

- Any problems with this?

# Possible Lock Problem

- Thread 1

```
addi t1,x0,1
```

```
Loop: lw t0,0($0)
```

```
bne t0,x0,Loop
```

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

- Thread 2

```
addi t1,x0,1
```

```
Loop: lw t0,0($0)
```

```
bne t0,x0,Loop
```

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

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
  - Needed even on uniprocessor systems
    - After all, Interrupts happen, and can trigger thread context switches...

# Synchronization in RISC-V option one: 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:  
 $s4 \leftrightarrow \text{Mem}(s1)$

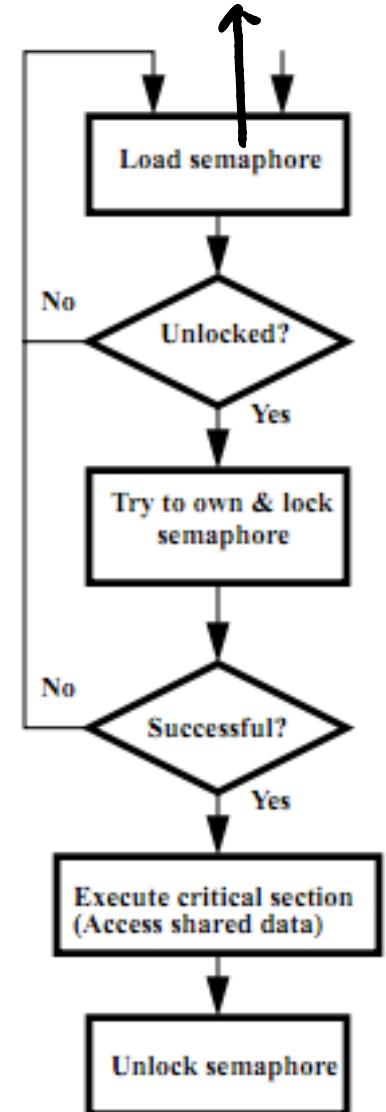
```
try:  
    lr  t1, s1      #load reserved  
    sc  t0, s1, s4  #store conditional  
    bne t0,x0,try   #loop if sc fails  
    add s4,x0,t1    #load value in s4
```

sc would fail if another threads executes sc here

# Test-and-Set

Computer Science 61C

- 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 RISC-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
```

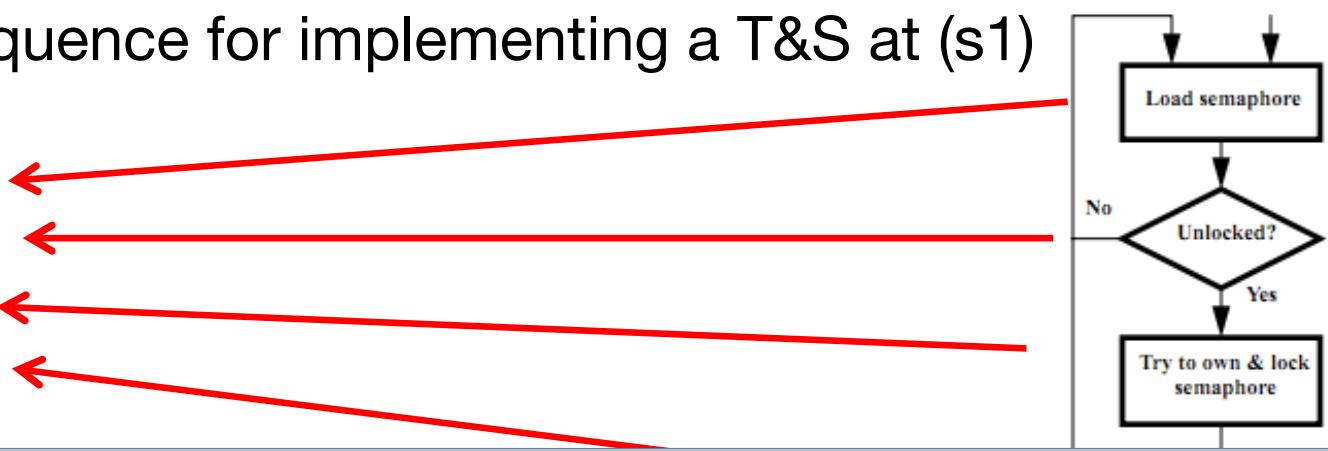
Locked:

```
# critical sec
```

Unlock:

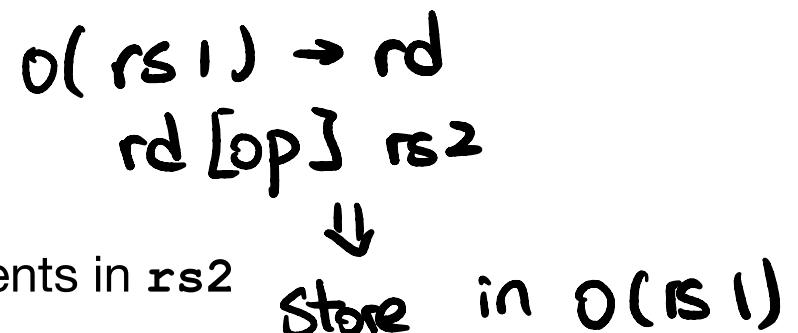
```
sw x0,0(s1)
```

Idea is that not for programmers to use this directly, but as a tool for enabling implementation of parallel libraries



# RISC-V Alternative: Atomic Memory Operations

- Three instruction rtype instructions
  - Swap, and, add, 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 allow atomic swap as a primitive
    - It also allows "reduction operations" that are common to be efficiently implemented



# 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