Concurrency & Parallelism

Ioan Raicu
Computer Science Department
Illinois Institute of Technology

CS 553: Cloud Computing
October 6th, 2025

Logistics

- HW2 grades have been posted
- HW3 grading is underway
- HW4 will be posted today (parallelism, I/O, and sort)

Seminars

- Tuesday, October 7, 2025
  - Who: Simone Silvestri (https://silvestri.engr.uky.edu/)
  - From: University of Kentucky
  - Research area: Internet of Things + Network Management
- Monday, October 20, 2025
  - Who: Alexandru Orhean (https://www.cdm.depaul.edu/Faculty-and-Staff/Pages/faculty-info.aspx?fid=1648)
  - From: DePaul
  - Research area: Distributed Systems

Intel x86 Processors

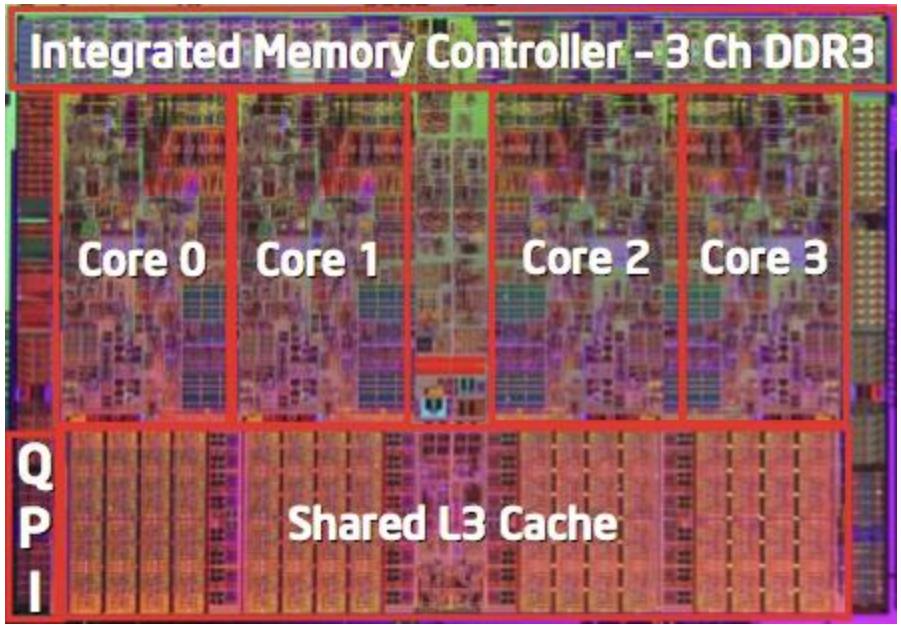
- Dominate laptop/desktop/server market
- Evolutionary design
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on
- x86 is a Complex Instruction Set Computer (CISC)
  - Many different instructions with many different formats
    - But, only small subset encountered with Linux programs
- Compare: Reduced Instruction Set Computer (RISC)
  - RISC: *very few* instructions, with *very few* modes for each
  - RISC can be quite fast (but Intel still wins on speed!)
  - Current RISC renaissance (e.g., ARM, RISCV), especially for low-power

Intel x86 Evolution: Milestones

Name	*Date*	*Transistors*	*MHz*
• 8086	1978	29K	5-10
– First 16-bit Intel processor. Basis for IBM PC & DOS			
– 1MB address space			
• 386	1985	275K	16-33
– First 32 bit Intel processor , referred to as IA32			
– Added “flat addressing”, capable of running Unix			
• Pentium 4E	2004	125M	2800-3800
– First 64-bit Intel x86 processor, referred to as x86-64			
• Core 2	2006	291M	1060-3333
– First multi-core Intel processor			
• Core i7	2008	731M	1600-4400
– Four cores (our shark machines)			

Intel x86 Processors, cont.

- Machine Evolution
  - 386 1985 0.3M
  - Pentium 1993 3.1M
  - Pentium/MMX 1997 4.5M
  - PentiumPro 1995 6.5M
  - Pentium III 1999 8.2M
  - Pentium 4 2000 42M
  - Core 2 Duo 2006 291M
  - Core i7 2008 731M
  - Core i7 Skylake 2015 1.9B



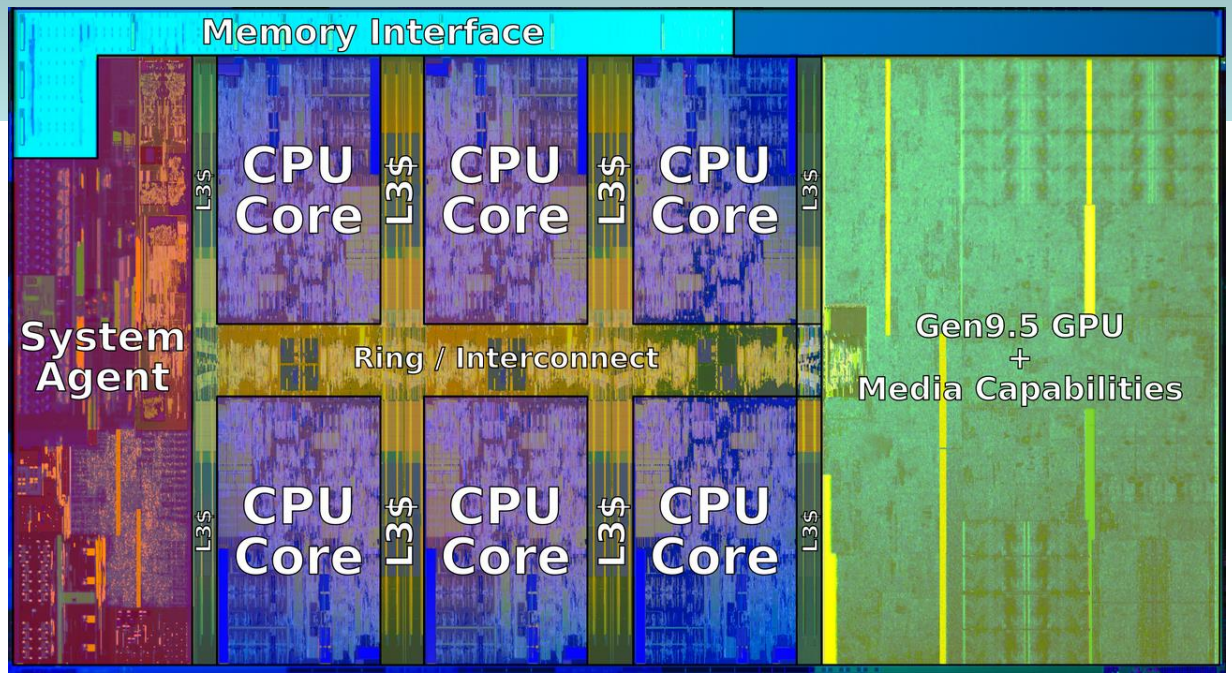
- Added Features
  - Instructions to support multimedia operations
  - Instructions to enable more efficient conditional operations
  - Transition from 32 bits to 64 bits, and more cores

Intel x86 Processors, cont.

- Past Generations
  - 1st Pentium Pro 1995 600 nm
  - 1st Pentium III 1999 250 nm
  - 1st Pentium 4 2000 180 nm
  - 1st Core 2 Duo 2006 65 nm
- Recent & Upcoming Generations
  1. Nehalem 2008 45 nm
  2. Sandy Bridge 2011 32 nm
  3. Ivy Bridge 2012 22 nm
  4. Haswell 2013 22 nm
  5. Broadwell 2014 14 nm
  6. Skylake 2015 14 nm
  7. Kaby Lake 2016 14 nm
  8. Coffee Lake 2017 14 nm

Process technology dimension
= width of narrowest wires
(10 nm ≈ 100 atoms wide)

2018 State of the Art: Coffee Lake



- Mobile Model: Core i7
  - –2.2-3.2 GHz
  - –45 W

- **Desktop Model: Core i7**
  - Integrated graphics
  - 2.4-4.0 GHz
  - 35-95 W

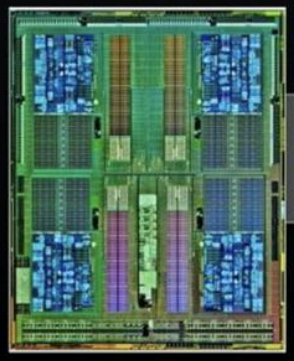
- **Server Model: Xeon E**
  - Integrated graphics
  - Multi-socket enabled
  - 3.3-3.8 GHz
  - 80-95 W

x86 Clones: Advanced Micro Devices (AMD)

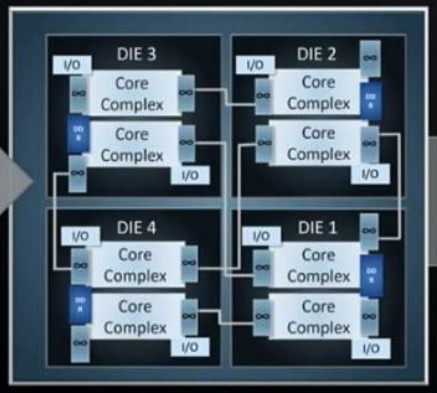
- Historically
  - AMD has followed just behind Intel; slower but cheaper
- Then
  - Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
  - Built Opteron: tough competitor to Pentium 4
  - Developed x86-64, their own extension to 64 bits
- Recent Years
  - Intel got its act together
    - 1995-2011: Lead semiconductor “fab” in world
    - 2019: #2 largest by $$ (#1 is Samsung)
  - AMD fell behind
    - Relies on external semiconductor manufacturer GlobalFoundaries
    - Ca. 2019 CPUs (e.g., Ryzen) are competitive again

AMD Epyc CPU

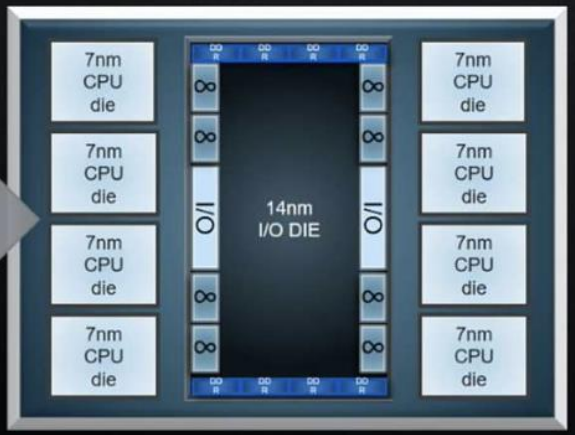
MULTICHIP ARCHITECTURE REVOLUTION



Monolithic die



Multi-die MCM



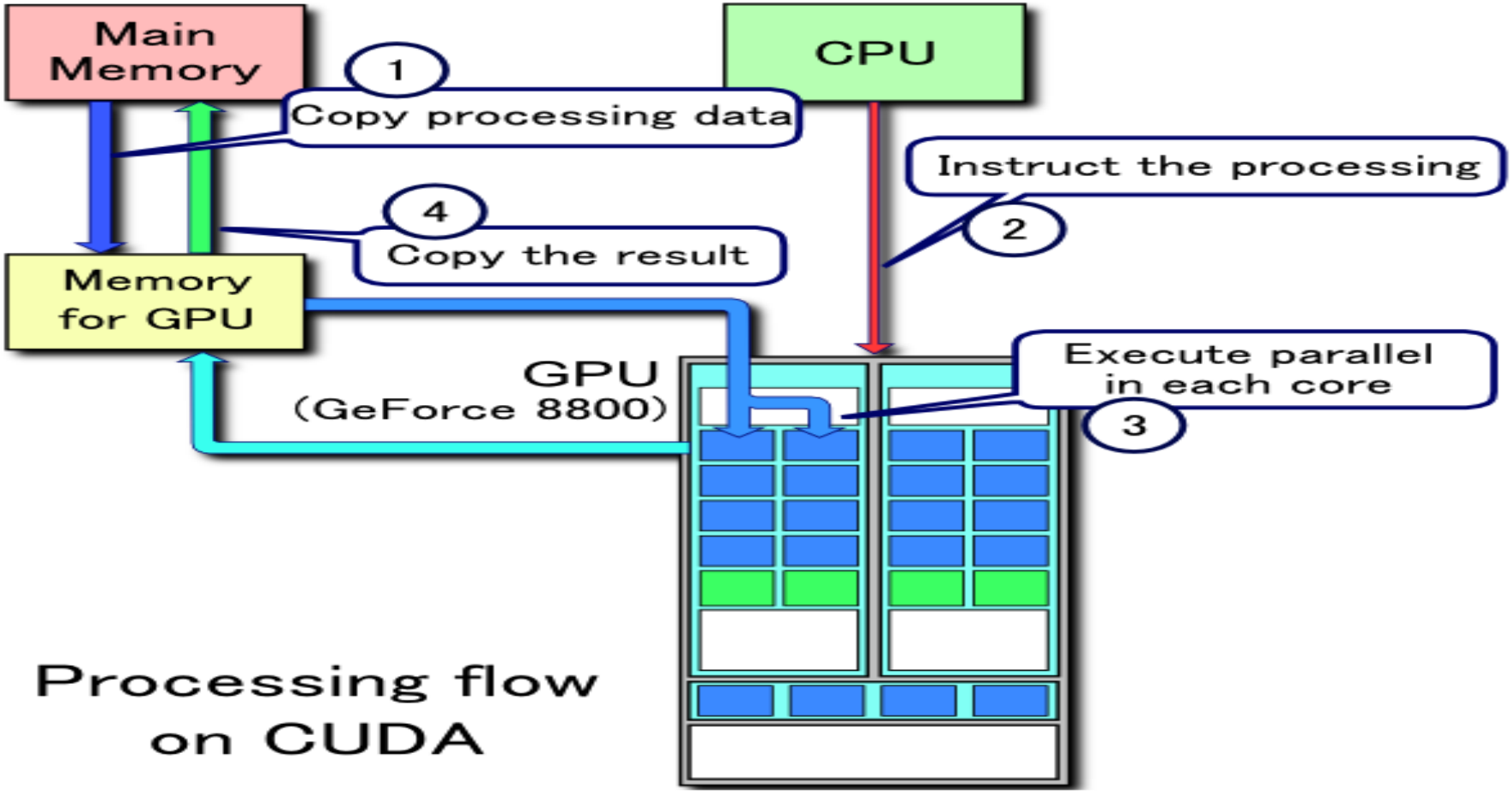
Chiplet

Intel's 64-Bit History

- 2001: Intel Attempts Radical Shift from IA32 to IA64
  - Totally different architecture (Itanium, AKA "Itanic")
  - Executes IA32 code only as legacy
  - Performance disappointing
- 2003: AMD Steps in with Evolutionary Solution
  - x86-64 (now called "AMD64")
- Intel Felt Obligated to Focus on IA64
  - Hard to admit mistake or that AMD is better
- 2004: Intel Announces EM64T extension to IA32
  - Extended Memory 64-bit Technology
  - Almost identical to x86-64!
- Virtually all x86 processors support x86-64
  - But, lots of code still runs in 32-bit mode

NVIDIA V100 GPU

GPU Computing

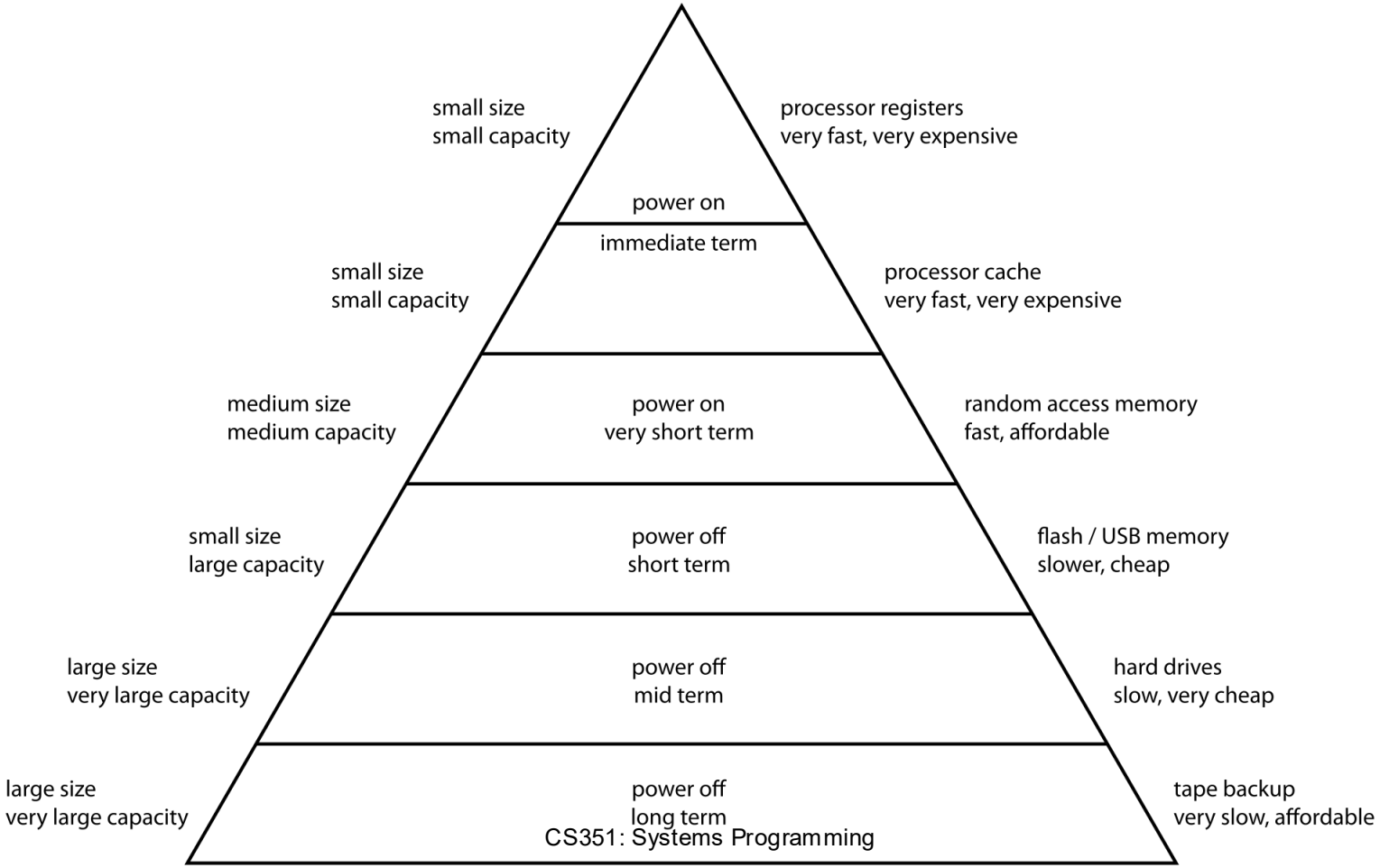


SIMD vs MIMD

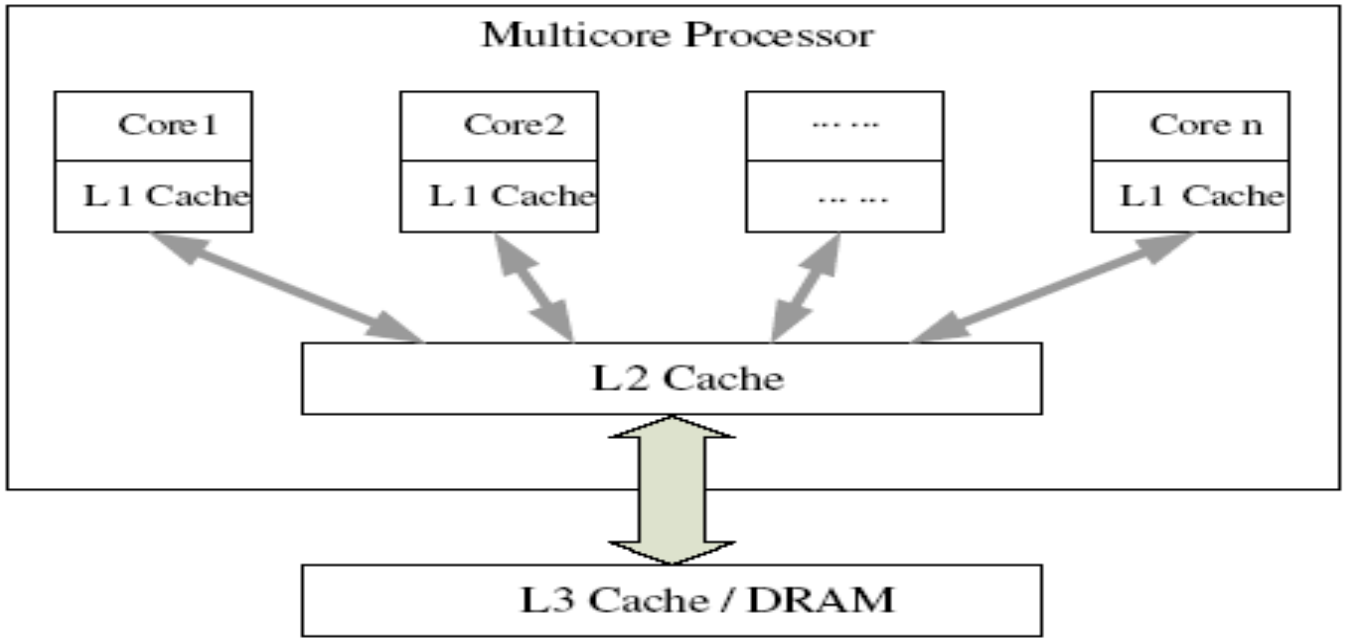
- Multiple Instruction Multiple Data
  - Typical for CPUs
- Single Instruction Multiple Data
  - Typical for GPUs

Memory Hierarchy

Computer Memory Hierarchy

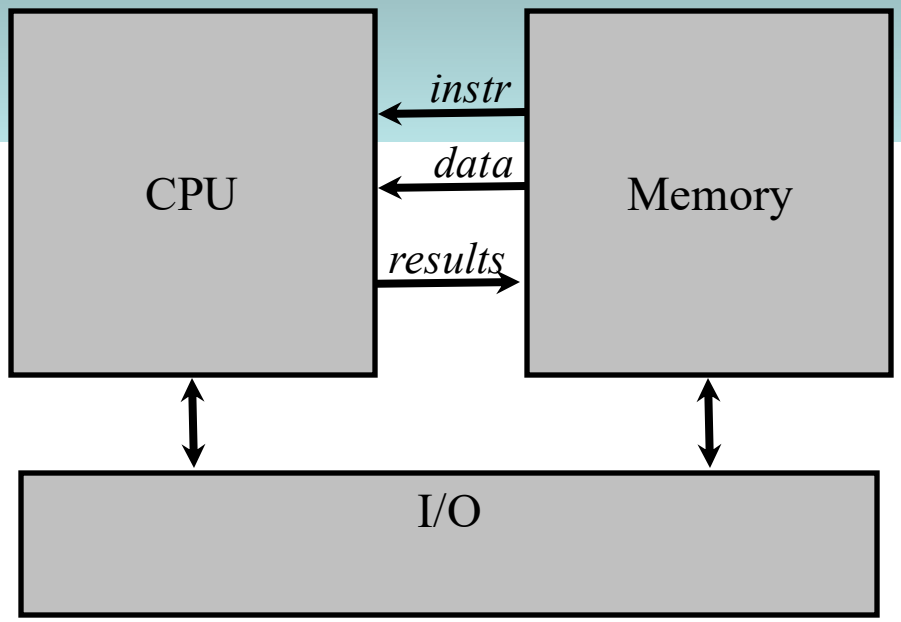


Multicore and Many-Core Architectures



- The schematic of a modern multicore CPU chip using a hierarchy of caches, where the L1 cache is private to each core. The L2 cache is also on chip and shared by all cores. The L3 cache or DRAM is off the multicore chip.

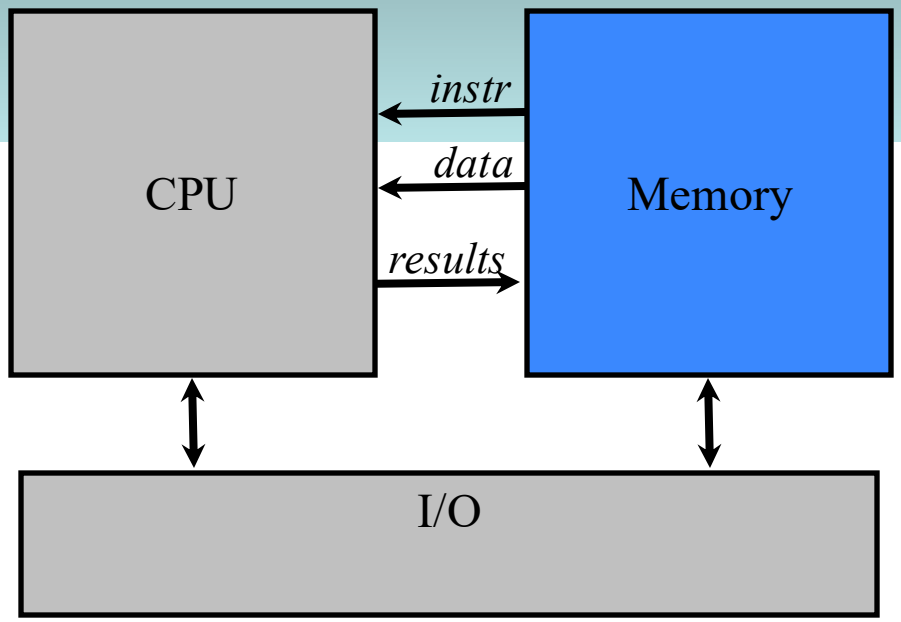
Memory Hierarchy



again, recall the *Von Neumann architecture*

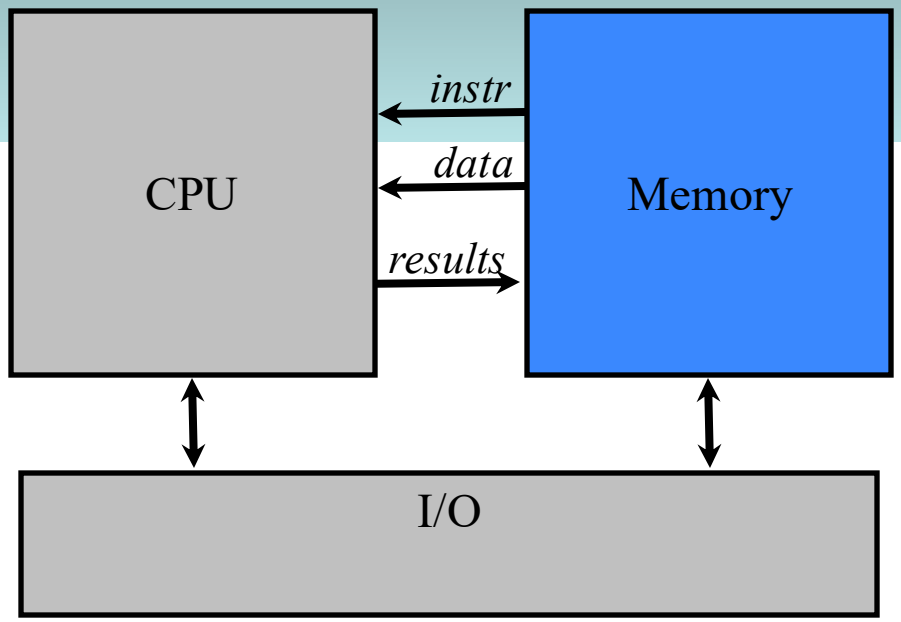
— a *stored-program computer* with programs and data stored in the same memory

Memory Hierarchy



"memory" is an *idealized* storage device that holds our programs (instructions) and data (operands)

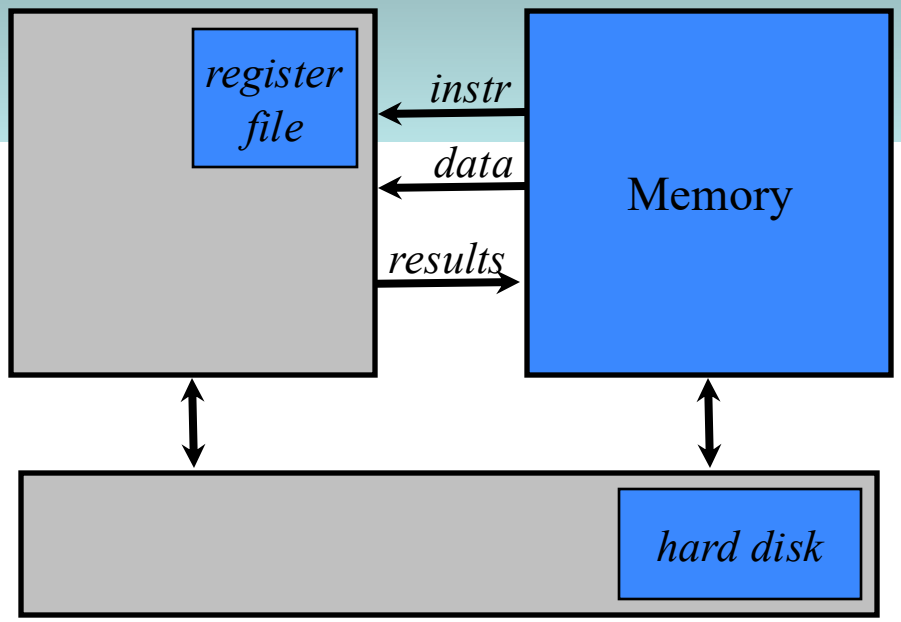
Memory Hierarchy



colloquially: “RAM”, *random access memory*

~ big array of byte-accessible data

Memory Hierarchy



in reality, “memory” is a combination of storage systems with very different access characteristics

Memory Hierarchy

common types of “memory”:

SRAM, DRAM, NVRAM, HDD

Memory Hierarchy

SRAM

- **S**tatic **R**andom **A**ccess **M**emory
- Data stable as long as power applied
- 6+ transistors (e.g. D-flip-flop) per bit
  - Complex & expensive, but fast!

Memory Hierarchy

DRAM

- **D**ynamic **R**andom **A**ccess **M**emory
- 1 capacitor + 1 transistor per bit
  - Requires period “refresh” @ 64ms
  - Much denser & cheaper than SRAM

Memory Hierarchy

NVRAM, e.g., Flash

- **N**on-**V**olatile **R**andom **A**ccess **M**emory
  - Data persists without power
- 1+ bits/transistor (low read/write granularity)
- Updates may require block erasure
- Flash has limited writes per block (100K+)

Memory Hierarchy

HDD

- **H**ard **D**isk **D**rive
- Spinning magnetic platters with multiple read/write “heads”
  - Data access requires *mechanical seek*

Memory Hierarchy

On *Distance*

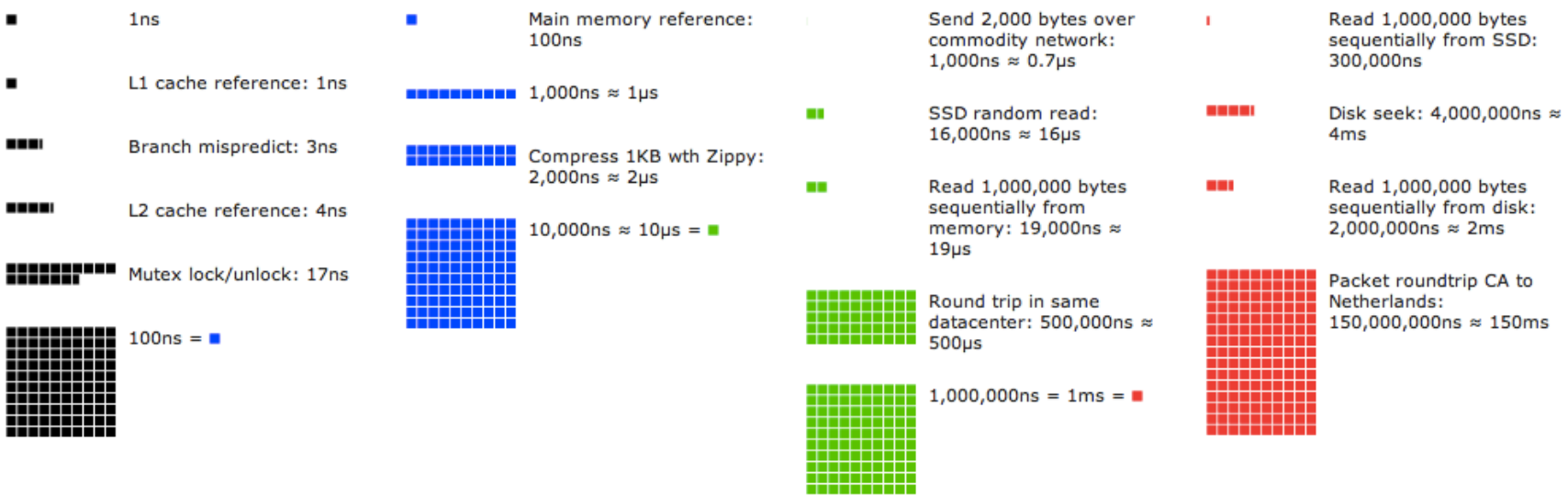
- Speed of light ≈ 1×10^9 ft/s ≈ 1ft/ns
  - i.e., in 3GHz CPU, 4in / cycle
    - max access dist (round trip) = 2 in!
- Pays to keep things we need often *close* to the CPU!

Memory Hierarchy

Relative Speeds

Type	Size	Access latency	Unit
Registers	8 - 32 words	0 - 1 cycles	(ns)
On-board SRAM	32 - 256 KB	1 - 3 cycles	(ns)
Off-board SRAM	256 KB - 16 MB	~10 cycles	(ns)
DRAM	128 MB - 64 GB	~100 cycles	(ns)
SSD	≤ 1 TB	~10,000 cycles	(µs)
HDD	≤ 4 TB	~10,000,000 cycles	(ms)

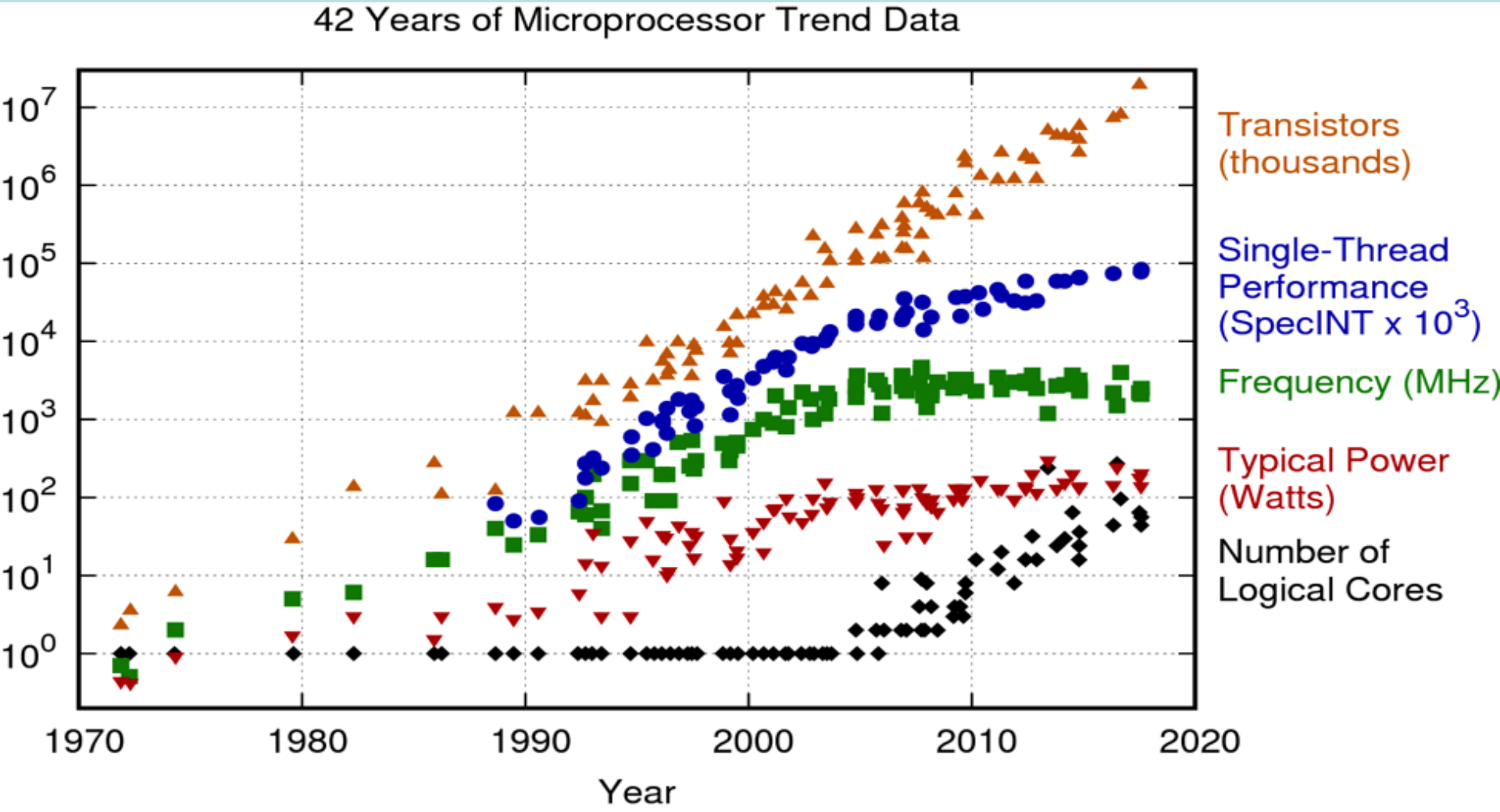
Memory Hierarchy



"Numbers Every Programmer Should Know"

http://www.eecs.berkeley.edu/~rcs/research/interactive_latency.html

Evolution of Multi-Core Processors



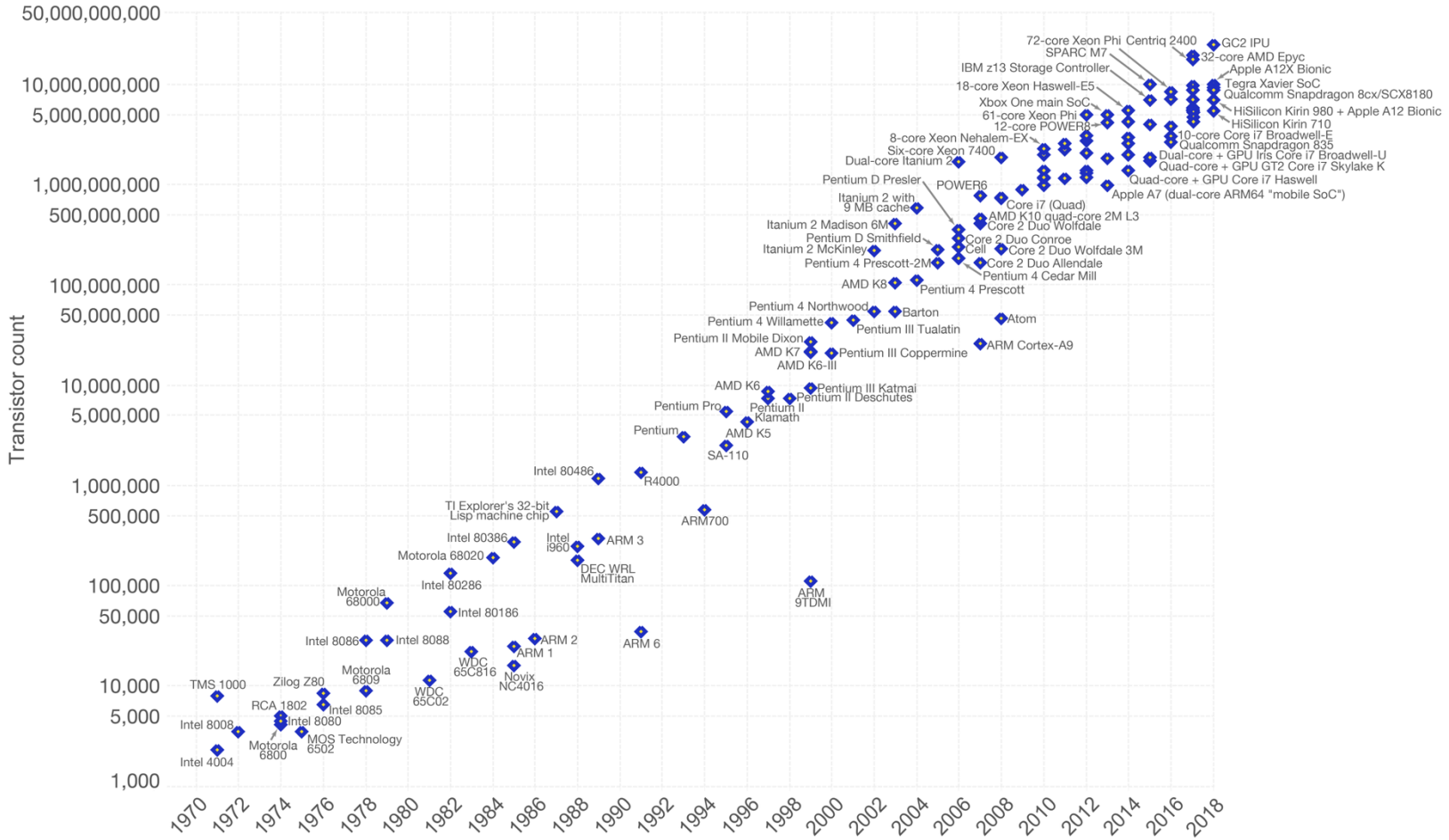
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Moore's Law

Moore's Law – The number of transistors on integrated circuit chips (1971-2018)

Our World in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are linked to Moore's law.

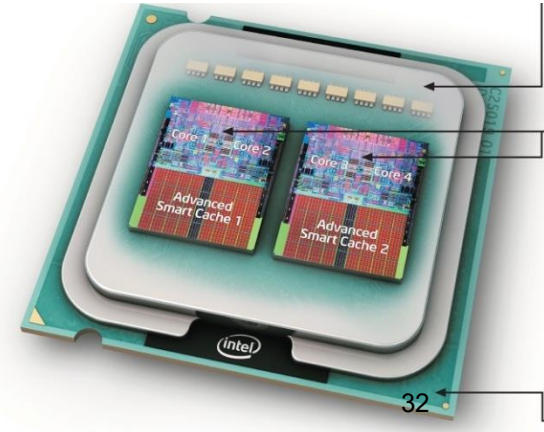


Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)

The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

Review: Multicore everywhere!

- Multicore processors are taking over, *many*core is coming
- The processor is the “new transistor”
- This is a “sea change” for HW designers and especially for programmers



How can we harness (many | multi)cores?

- Is it good enough to just have multiple programs running simultaneously?
- We want per-program performance gains!

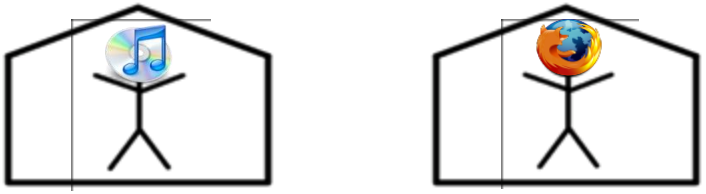
Crysis, Crytek 2007

Multiprogramming/Timesharing Systems

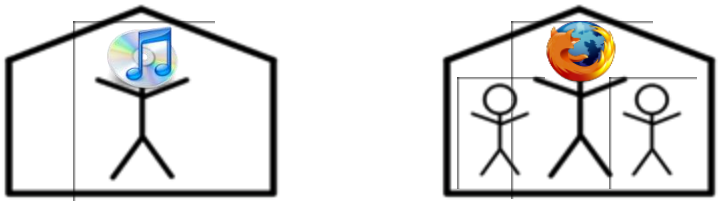
- Goal: to provide interleaved execution of several processes to give an illusion of many simultaneously executing processes.
- Computer can be a single-processor or multi-processor machine.
- The OS must keep track of the state for each active process and make sure that the correct information is properly installed when a process is given control of the CPU.
- Many resource allocation issues to consider:
  - How to give each process a chance to run?
  - How is main memory allocated to processes?
  - How are I/O devices scheduled among processes?

Definitions: threads v.s. processes

- A *process* is a "program" with its own address space.
  - A process has at least one thread!



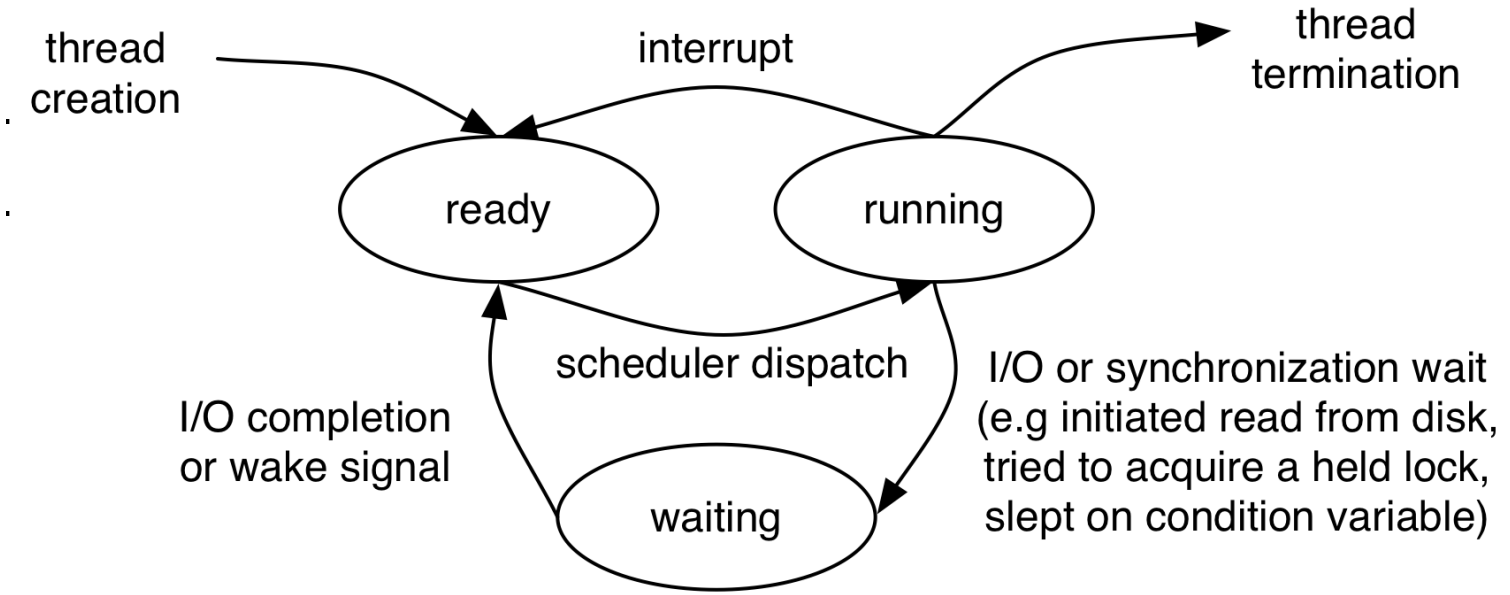
- A *thread of execution* is an independent sequential computational task with its own control flow, stack, registers, etc.
  - There can be many threads in the same process sharing the same address space



  - There are several APIs for threads in several languages. We will cover the PThread API in C.

How are threads *scheduled?*

- Threads/processes are run sequentially on one core or simultaneously on multiple cores
  - The operating system schedules threads and



Based on diagram from Silberschatz, Galvin, and Gagne

What's in a Process?

- Dynamic execution context of an executing program
- Several processes may run the same program, but each is a distinct process with its own state
- Process state includes:
  - The code for the running program;
  - The static data;
  - Space for dynamic data (heap)& the heap pointer (HP);
  - The Program Counter (PC) indicating the next instruction;
  - An execution stack and the stack pointer (SP);
  - Values of CPU registers;
  - A set of OS resources;
  - Process execution state (ready, running, etc.)

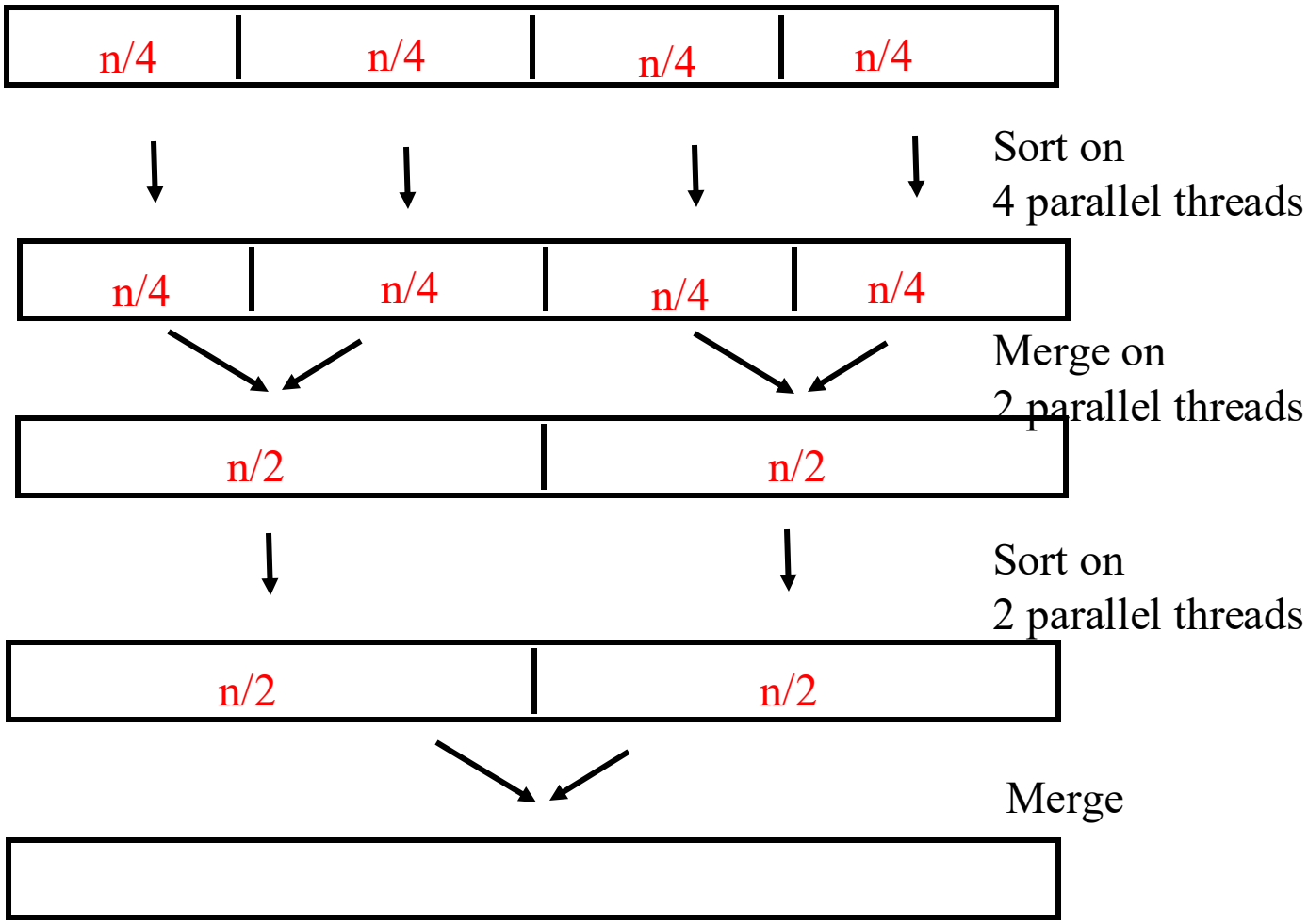
Introduction to Threads

- Multitasking OS can do more than one thing concurrently by running more than a single process
- A process can do several things concurrently by running more than a single thread
- Each thread is a different stream of control that can execute its instructions independently.
- Ex: A program (e.g. Browser) may consist of the following threads:
  - GUI thread
  - I/O thread
  - Computation thread

Defining Threads

- A thread defines a single sequential execution stream within a process
- Threads are bound to a single process
- Does each thread have its own stack, PC and registers?
- Each process may have multiple threads of control within it:
  - The address space of a process is shared or not?
  - No system calls are required to cooperate among threads
  - Simpler than message passing and shared-memory

Parallel Algorithms: Eg. mergesort



Is there a speed-up ?

Benefits of Threads: Summary

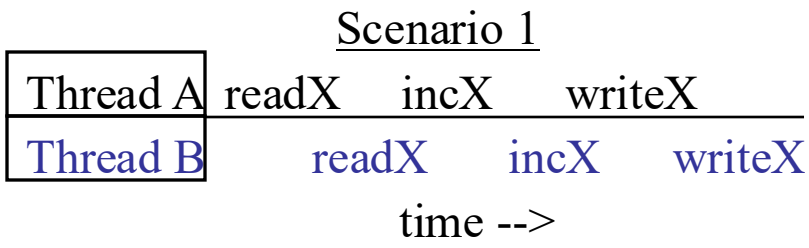
1. Superior programming model of parallel sequential activities with a shared store
2. Easier to create and destroy threads than processes.
3. Better CPU utilization (e.g. dispatcher thread continues to process requests while worker threads wait for I/O to finish)
4. Guidelines for allocation in multi-processor systems

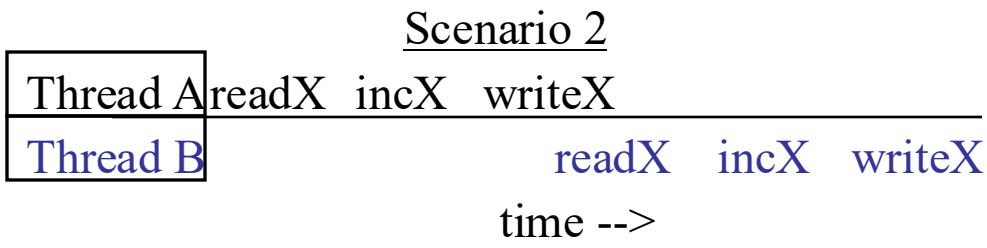
Processes and Threads

- A UNIX Process is
  - a running program with
  - a bundle of resources (file descriptor table, address space)
- A thread has its own
  - stack
  - program counter (PC)
  - All the other resources are shared by **all** threads of that process. These include:
    - open files
    - virtual address space
    - child processes

How can we make threads cooperate?

- If task can be completely decoupled into independent sub-tasks, cooperation required is minimal
  - Starting and stopping communication
- Trouble when they need to share data!
- Race conditions:





- We need to force some serialization
  - Synchronization constructs do that!

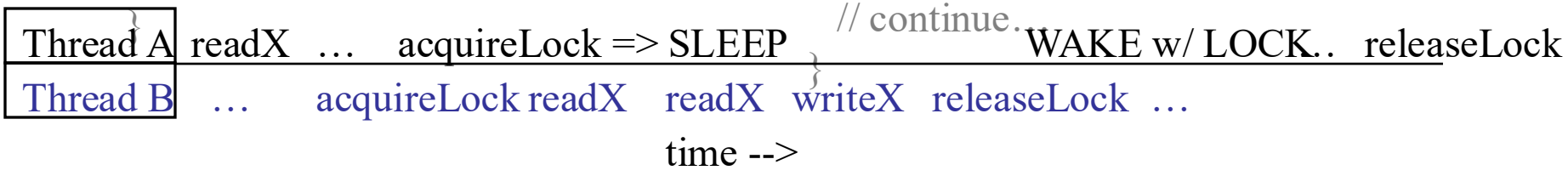
Lock / mutex semantics

- A *lock* (mutual exclusion, mutex) guards a *critical section* in code so that only one thread at a time runs its corresponding section
  - *acquire* a lock before entering crit. section
  - *releases* the lock when exiting crit. section
  - Threads share locks, one per section to synchronize
- If a thread tries to acquire an in-use lock, that thread is put to sleep
  - When the lock is released, the thread wakes up *with the lock*! (blocking call)

Lock / mutex syntax example in PThreads

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
int x;
    threadA() {
      int temp = foo(x);
      pthread_mutex_lock(&lock);
      x = bar(x) + temp;
      pthread_mutex_unlock(&lock);
      // continue…
    }
```

```
threadB() {
  int temp = foo(9000);
  pthread_mutex_lock(&lock);
  baz(x) + bar(x);
  x *= temp;
  pthread_mutex_unlock(&lock);
  // continue…
}
```



- But locks don't solve everything…
  - Problem: potential deadlock!

```
threadA() {
  pthread_mutex_lock(&lock1);
  pthread_mutex_lock(&lock2);
}
```

```
threadB() {
  pthread_mutex_lock(&lock2);
  pthread_mutex_lock(&lock1);
}
```

Condition variable semantics

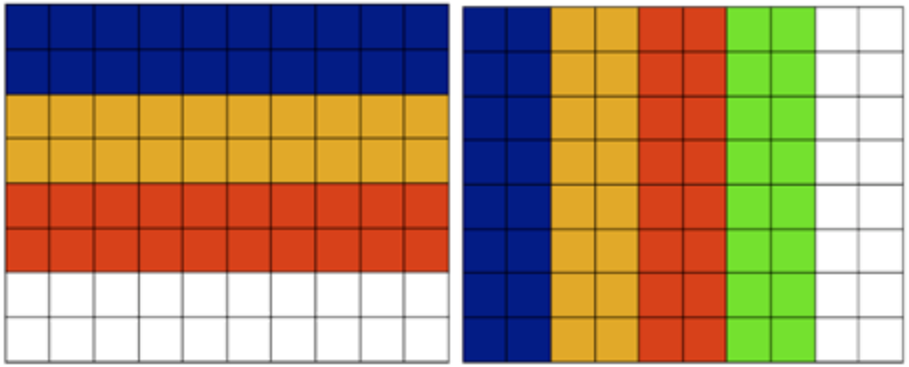
- *A condition variable* (CV) is an object that threads can sleep on and be woken from
  - *Wait* or *sleep* on a CV
  - *Signal* a thread sleeping on a CV to wake
  - *Broadcast* all threads sleeping on a CV to wake
  - I like to think of them as thread pillows…
- *Always* associated with a lock!
  - Acquire a lock before touching a CV
  - Sleeping on a CV releases the lock in the thread's sleep
  - If a thread wakes from a CV it will have the lock
- Multiple CVs often share the same lock

Speedup issues: overhead

- More threads does not always mean better!
  - I only have two cores…
  - Threads can spend too much time *synchronizing* (e.g. waiting on locks and condition variables)
- Synchronization is a form of overhead
  - Also communication and creation/deletion overhead

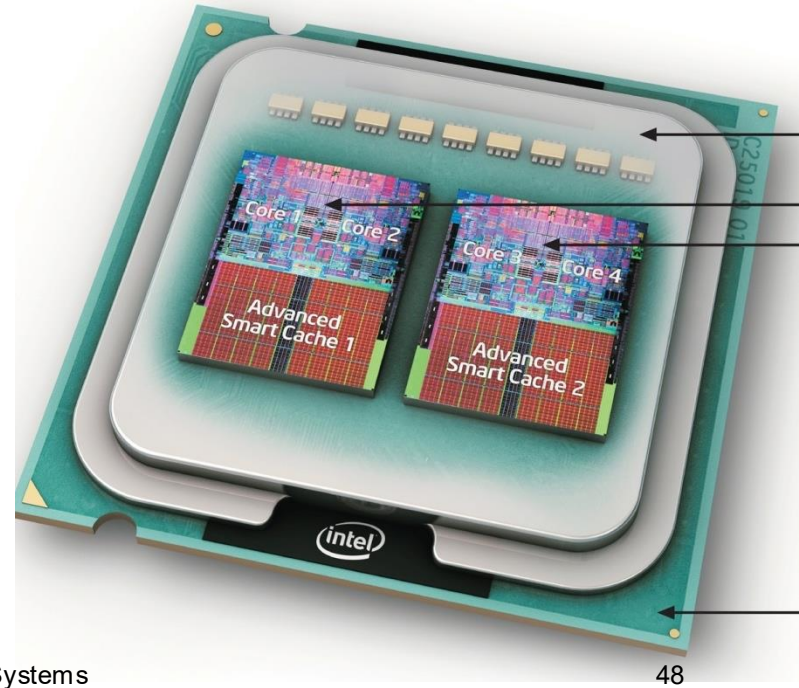
Speedup issues: caches

- Caches are often one of the largest considerations in performance
- For multicore, common to have independent L1 caches and shared L2 caches
- Can drive domain decomposition design



(a) "Horizontal" Decomposition

(b) "Vertical" Decomposition

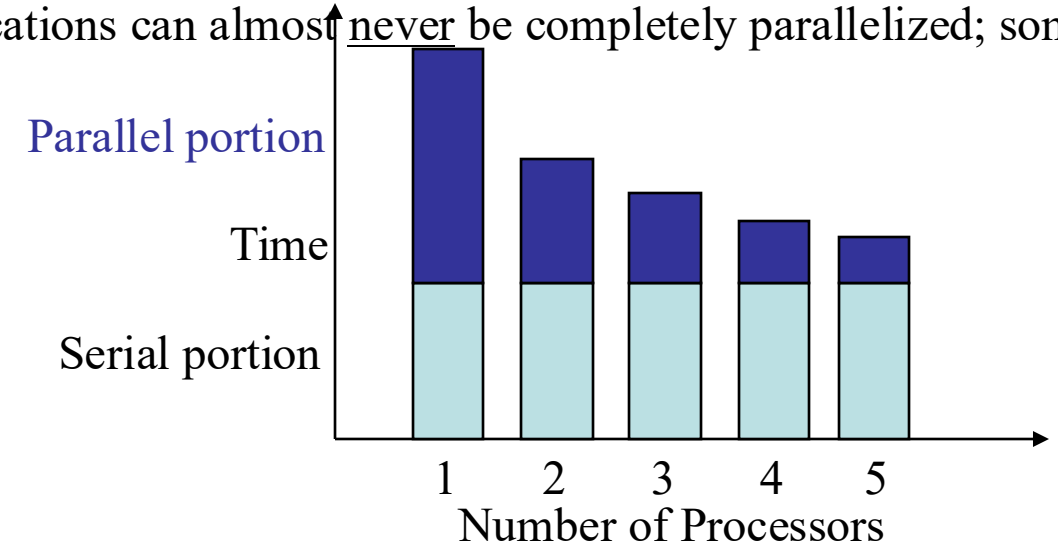


System Scalability

- ***Size Scalability:***
  - This refers to achieve higher performance or performing more functionality by increasing the *machine size*. The word "size" refers to adding the number of processors; more cache, memory, storage or I/O channels. The most obvious way to simple counting the number of processors installed. Not all parallel computer or distributed architectures are equally size-scalable. For example, IBM S2 was scaled up to 512 processors in 1997. But in 2008, the IBM BlueGene/L system scaled up to 65,000 processors.
- ***Software Scalability:***
  - This refers to upgrades in OS or compilers, adding mathematical and engineering libraries, porting new application software, and install more user-friendly programming environment. Some software upgrade may not work with large system configurations. Testing and fine-tuning of new software on larger system is a non-trivial job.
- ***Application scalability:***
  - This refers to the match of *problem size* scalability with the *machine size* scalability. Problem size affects the size of the data set or the workload increase. Instead of increasing machine size, users can enlarge the problem size to enhance the system efficiency or cost-effectiveness.
- ***Technology Scalability:***
  - This refers to a system that can adapt to changes in building technologies, such as those component and networking technologies discussed in Section 3.1. Scaling a system design with new technology must consider three aspects: *time, space,* and *heterogeneity.* Time refers to generation scalability. Changing to new-generation processors, one must consider the impact to motherboard, power supply, packaging and cooling, etc. Based on the past experience, most system upgrade their commodity processors every 3 to 5 years. Space is more related to packaging and energy concerns. Technology scalability demands harmony and portability among suppliers.

Amdahl's Law

- Applications can almost never be completely parallelized; some serial code remains



- s is serial fraction of program, P is # of processors
- Amdahl's law:

Speedup(P) = Time(1) / Time(P)

≤ 1 / (s + ((1-s) / P)), and as P □ ∞

≤ 1/s

- Even if the parallel portion of your application speeds up perfectly, your performance may be limited by the sequential portion

Amdahl's Law

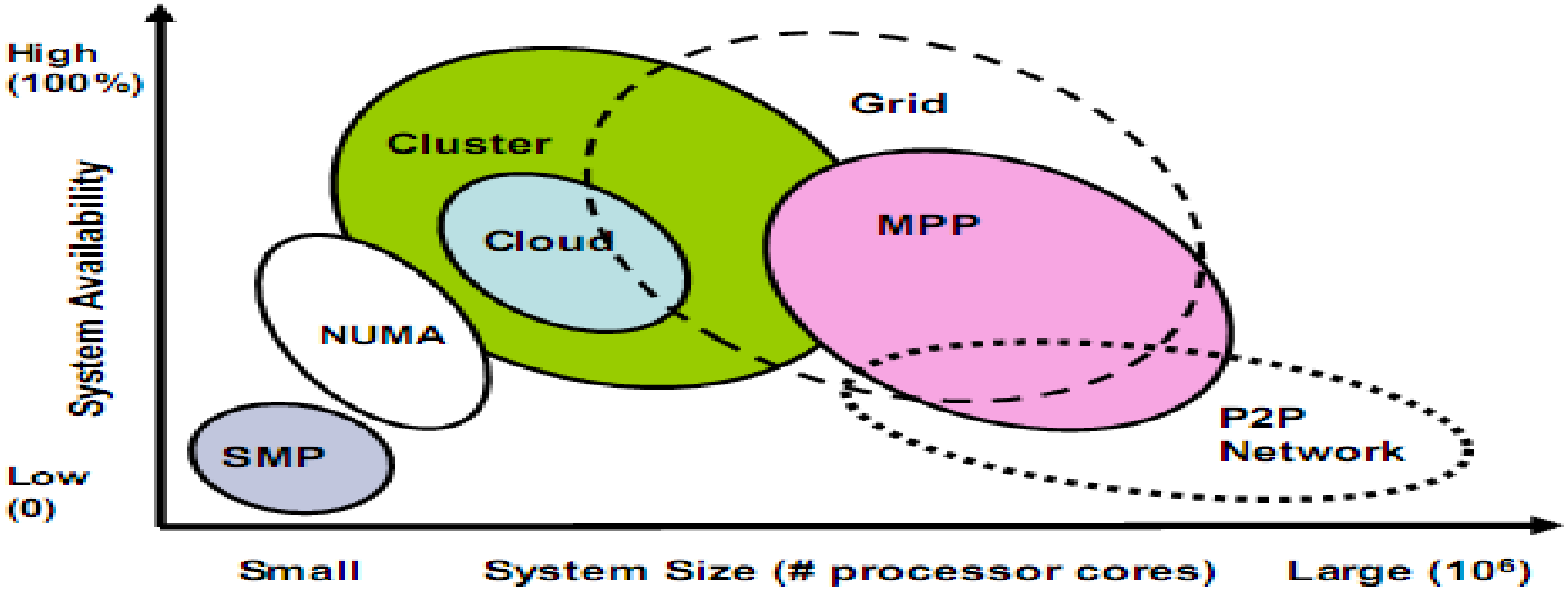
- **Speedup = $S = T / [\alpha T + (1-\alpha) T/n] = 1 / [\alpha + (1-\alpha)/n]$**
- The maximum speedup of n is achieved, only if the sequential bottleneck α is reduced to zero or the code is fully parallelizable with α = 0. As the cluster becomes sufficiently large, i.e. $n \rightarrow \infty$, S approaches $1/\alpha$, an upper bound on the speedup S.
- Amdahl's law teaches us that we should make the sequential bottleneck as small as possible.
- Increasing the cluster size alone may not give a good speedup in this case.

Amdahl's Law

- $E = S / n = 1 / [\ \alpha\ n + 1 - \alpha\ ]$
- Fixed workload size poses efficiency problems
- **Gustafson's Law**
  - Allows variable size workloads to increase efficiency
  - For fixed workload, apply Amdahl's law
  - For scaled problems, apply Gustafson's Law

Fault-Tolerance and System Availability

- *System Availability = MTTF / (MTTF + MTTR)*



Parallel and Distributed Programming Models

- Parallel and Distributed Programming Models and Toolsets

Model	Objectives and Web Link	Attractive Features Implemented
MPI	Message-Passing Interface is a library of subprograms that can be called from C or Fortran to write parallel programs running on distributed computer systems [3, 28, 42]	Specify synchronous or asynchronous point-to-point and collective communication commands and I/O operations in user programs for message-passing execution
MapReduce	A web programming model for scalable data processing on large cluster over large datasets, or in web search operations [17]	A *Map* function generates a set of intermediate key/value pairs. A *Reduce* function to merge all intermediate values with the same key
Hadoop	A software library to write and run large user applications on vas datasets in business applications. http://hadoop.apache.org/core/	Hadoop is scalable, economical, efficient and reliable in providing users with easy access of commercial clusters

§ Cache-Friendly Code

Matrix Multiplication

In general, cache friendly code:

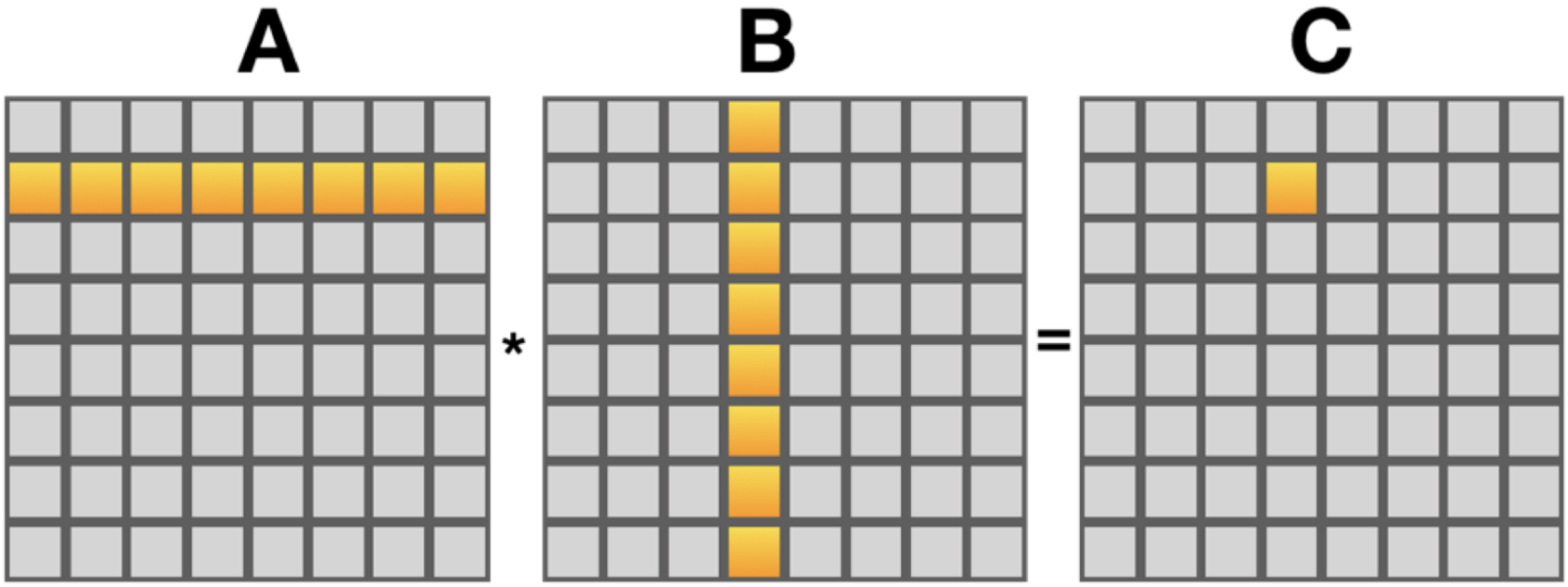
- exhibits *high locality* (temporal & spatial)
- maximizes cache *utilization*
- keeps *working set* size small
- avoids random memory access patterns

Matrix Multiplication

case study in software/cache interaction:

matrix multiplication

Matrix Multiplication

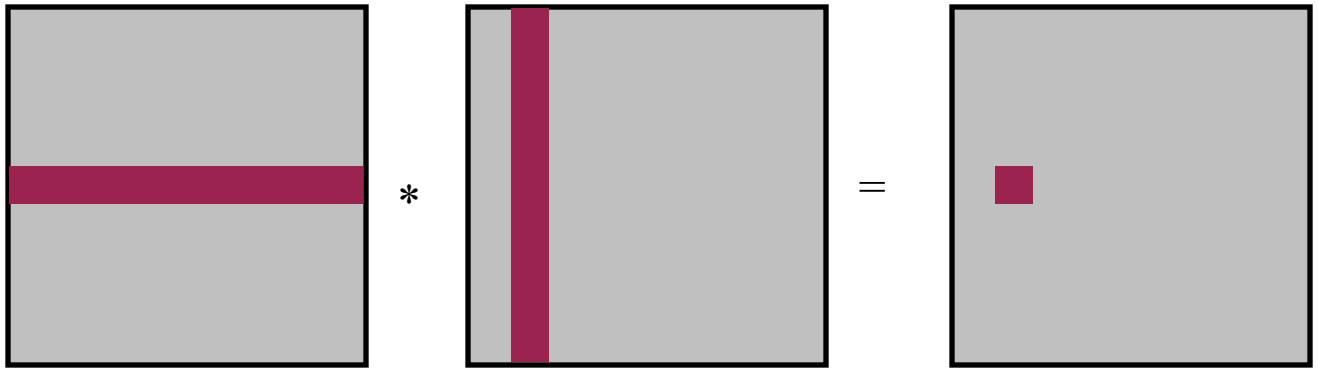


- More info at:
  - https://gist.github.com/nadavrot/5b35d44e8ba3dd718e595e40184d03f0
  - https://www.cs.utexas.edu/users/pingali/CS378/2008sp/papers/gotoPaper.pdf

Matrix Multiplication

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix}$$

$$\begin{aligned} c_{ij} &= \begin{pmatrix} a_{i1} & a_{i2} & a_{i3} \end{pmatrix} \cdot \begin{pmatrix} b_{1j} & b_{2j} & b_{3j} \end{pmatrix} \\ &= a_{i1}b_{1j} + a_{i2}b_{2j} + a_{i3}b_{3j} \end{aligned}$$



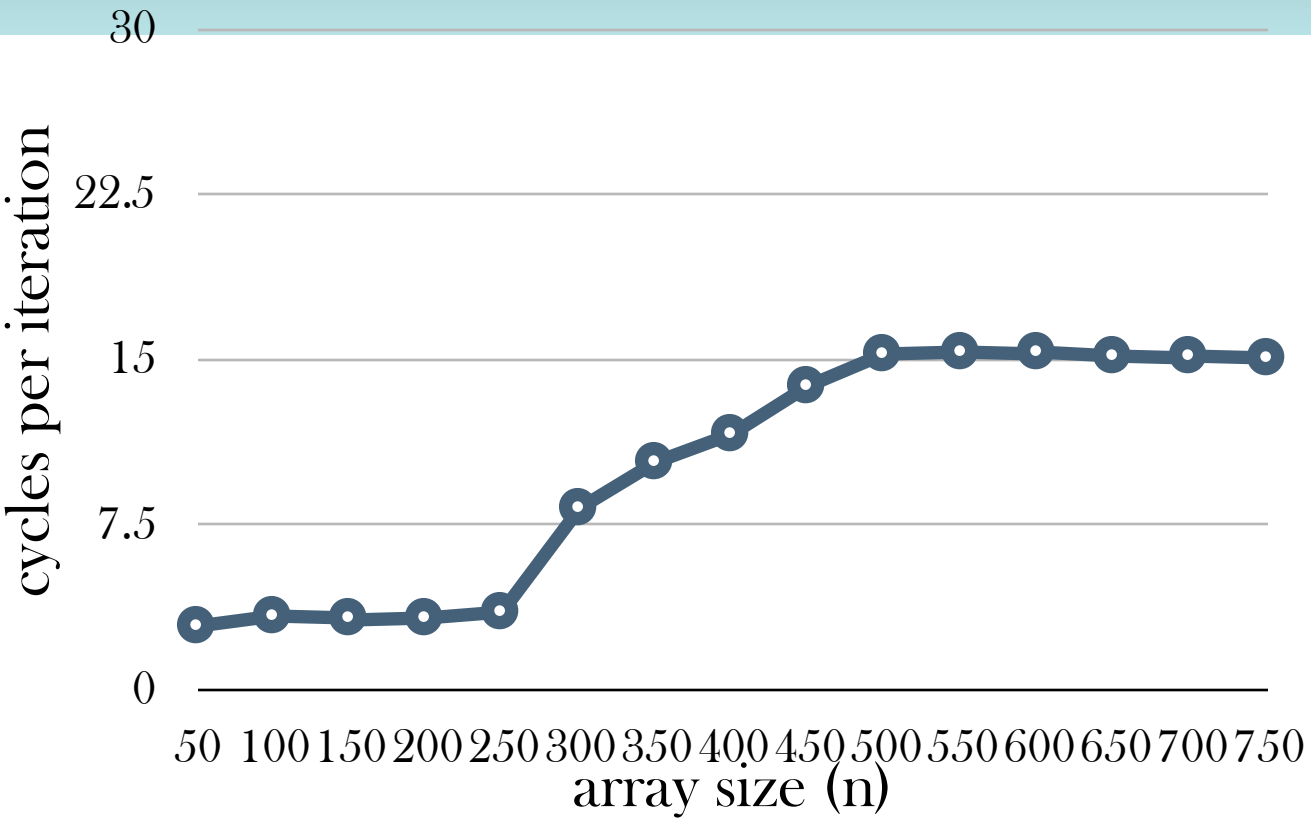
Matrix Multiplication

canonical implementation:

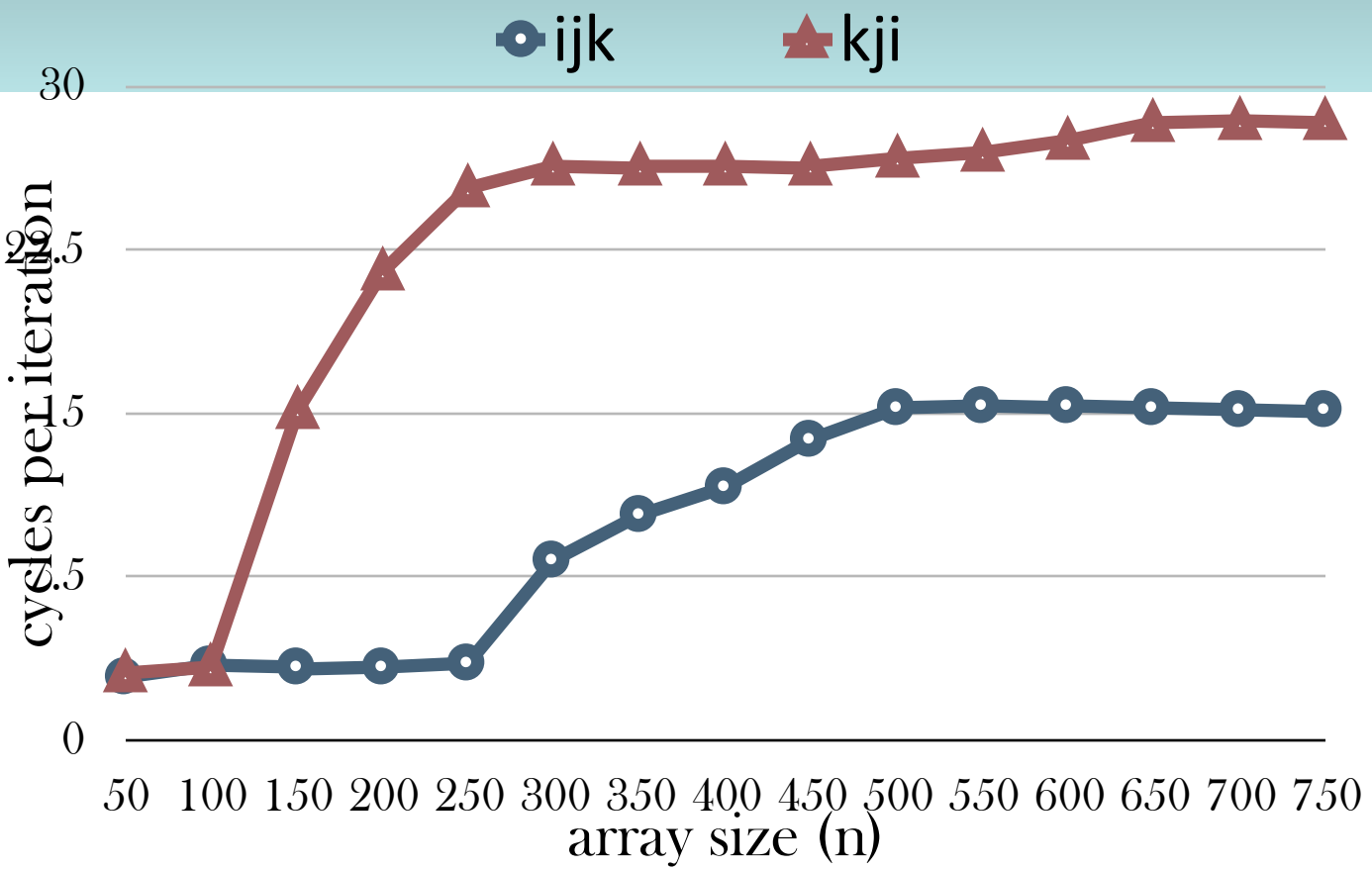
```
#define MAXN 1000
typedef double array[MAXN][MAXN];

/* multiply (compute the inner product of) two square matrices
 * A and B with dimensions n x n, placing the result in C  */
void matrix_mult(array A, array B, array C, int n) {
    int i, j, k;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            C[i][j] = 0.0;
            for (k = 0; k < n; k++)
                C[i][j] += A[i][k]*B[k][j];
        }
    }
}
```

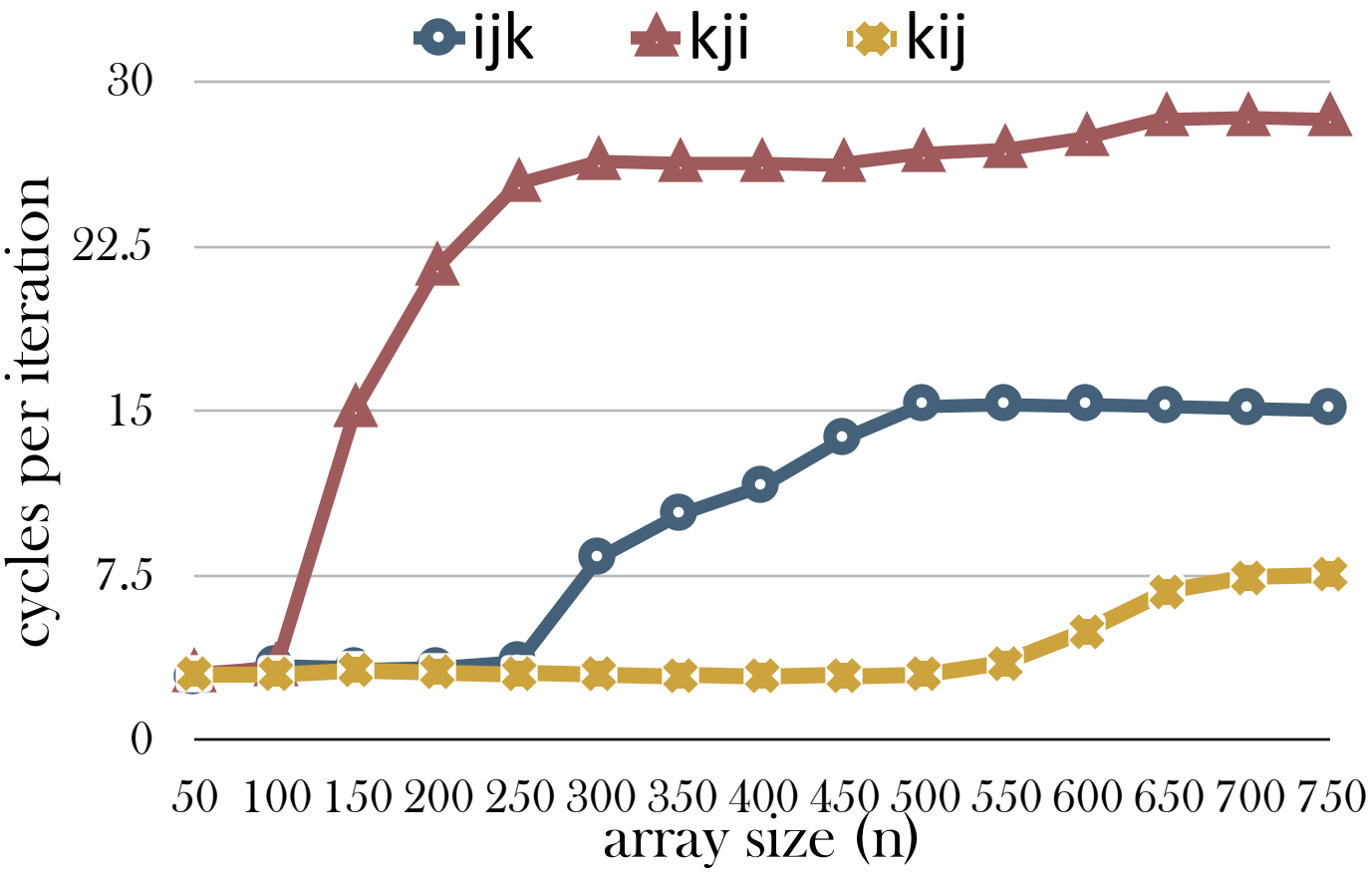
Matrix Multiplication



Matrix Multiplication



Matrix Multiplication



Instruction Level Parallelism

```
1    e = a + b
2    f = c + d
3    m = e * f
```

- Op 3 depends on the results of op 1 and 2
- Op 1 and 2 must be complete before op 3
- Op 1 and 2 can be calculated simultaneously
- Assuming each op can be completed in one unit of time ==> these 3 instructions need 2 units of time

Fused Multiply-Add

- $a \leftarrow a + (b \times c)$
  - This operation can be performed in 1 fmadd instruction
- More at: https://en.wikipedia.org/wiki/Multiply–accumulate_operation#Fused_multiply–add

Multi-threading

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>  //Header file for sleep(). man 3 sleep for details.
#include <pthread.h>

// A normal C function that is executed as a thread
// when its name is specified in pthread_create()
void *myThreadFun(void *vargp)
{
    sleep(1);
    printf("Printing GeeksQuiz from Thread \n");
    return NULL;
}

int main()
{
    pthread_t thread_id;
    printf("Before Thread\n");
    pthread_create(&thread_id, NULL, myThreadFun, NULL);
    pthread_join(thread_id, NULL);
    printf("After Thread\n");
    exit(0);
}
```

- gcc multithread.c -lpthread

Multi-threading

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

// Let us create a global variable to change it in threads
int g = 0;

// The function to be executed by all threads
void *myThreadFun(void *vargp)
{
    // Store the value argument passed to this thread
    int *myid = (int *)vargp;

    // Let us create a static variable to observe its changes
    static int s = 0;

    // Change static and global variables
    ++s; ++g;

    // Print the argument, static and global variables
    printf("Thread ID: %d, Static: %d, Global: %d\n", *myid, ++s, ++g);
}

int main()
{
    int i;
    pthread_t tid;

    // Let us create three threads
    for (i = 0; i < 3; i++)
        pthread_create(&tid, NULL, myThreadFun, (void *)&tid);

    pthread_exit(NULL);
    return 0;
}
```

Other hints

- GCC compiler optimizations (e.g. -O3)
- Matrix transpose?
- Find out the size of cache line (64 bytes?)
- Find out the size of L1 cache size?
- Multi-threading without any locks on matrices using static partitioning of work
- Give compiler hints to assign certain variables to registers

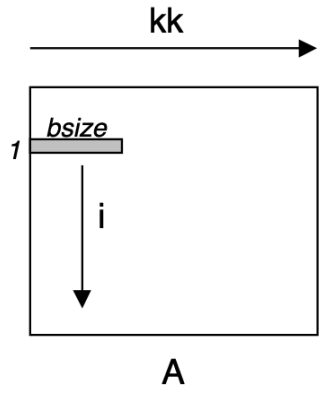
Other hints

remaining problem: *working set size* grows beyond capacity of cache

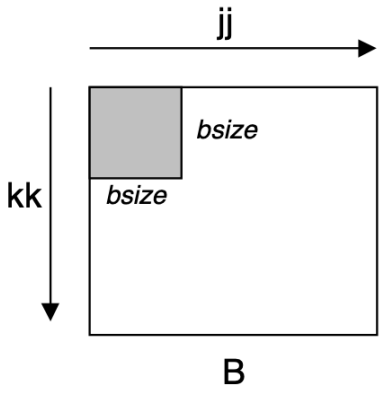
smaller strides can help, to an extent (by leveraging spatial locality)

Matrix Multiplication

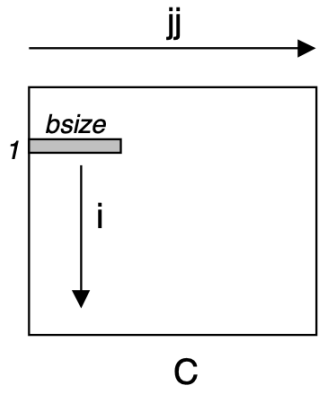
idea for optimization: deal with matrices in smaller chunks at a time that will fit in the cache — “blocking”



Use *1 x bsize* row sliver *bsize* times



Use *bsize x bsize* block *n* times in succession



Update successive elements of *1 x bsize* row sliver

Questions