CMOR 421/521
High performance computing

Jesse Chan, 1/14/25

What are we supposed to learn?

- High performance computing (HPC): basics of single (serial) CPUs, multi-core CPUs, distributed computing, basic GPUs.
- Some aspects of hardware, basic parallel computing.
- In all cases, the effective use of parallelism tends to reduce to effectively accessing data (e.g., efficiently accessing **memory**, reducing **communication**).
  - We will focus less on *how* to write efficient programs and more on *why* a program is or is not efficient.

Why memory and communication are important

"A supercomputer is a device for turning compute-bound problems into I/O-bound problems."

Ken Batcher

- This class will be focused on understanding this quote - first for single (serial) CPUs, next for multi-core CPUs, next distributed machines, and finally GPUs.
- In all cases, the focus will be on utilizing memory efficiently: when performing operations in parallel is easy, something else becomes the bottleneck.

Administrative stuff

Assumptions / prerequisites

- You should be familiar with linear algebra, mainly matrix-vector and matrix-matrix multiplication.
- You should have a good amount of prior experience with Linux, SSH, LaTeX, as well as C and C++.
  - Expectations: should be familiar with CMOR 420/520 level C/C++ scientific programming.
  - This course focuses more on “C-style C++”; we won’t need much OOP, classes, or program design.

Administrative stuff:

- See Canvas page for notes and assignments (no website). Lectures will also be hybrid/recorded on Canvas.
- Grade based on exercises and assignments only.
  - 20% weekly exercises (semi-completion; intended to keep everyone up-to-date with the course materials)
  - 80% assignments (roughly 1 per month). 520 assignments will include an extra problem.
- Late policy: automatic 10% deduction each day.

Computing resources

- You will need access to a programmable computer (an iPad or Chromebook will **not** work). *Please let me know if you do not have access to one.*
  - For some parts of the course, you can SSH into clear.rice.edu or the NOTS cluster (should be available for this class early February)
- You are encouraged to bring a laptop to class to try out codes and experiment alongside the lectures.

What do we need to download/buy?

- Nothing, lecture materials, lecture recordings, and demo codes will be posted to Canvas.
- Information can also be found in online documentation or searching (forums, StackOverflow, etc).
  - ChatGPT and other LLMs can also be great resources, but make sure to test for correctness.

Grading

- It is more time-consuming to grade coding assignments than other homework. You will lose points if it is not easy for us to grade your code!
- You are responsible for ensuring that the graders and I can easily compile, run, and understand your code. This is especially tricky for HPC codes, which are often “uglier” than other scientific programs.
- Clean up and organize your code before turning it in:
  - Make your variable names clear and descriptive.
  - Add comments explaining complex or un-intuitive chunks of code.
  - Remove unused or unnecessary code, separate concerns, make it modular.

Grading cont., extensions

- If there are issues with your code, tell us!
  - Code that doesn’t run or produces incorrect results will have a significant number of points deducted.
  - If you tell us that it doesn’t work, we will be more lenient (the more precise the better - this can help us figure out what went wrong).
- I’m happy to provide extensions, but if you need one, please notify me *at least* 2 days in advance of the deadline.
  - Of course, the exceptions are medical or family emergencies.

Office hours

- Office hours are tentatively after class on Thursdays, 2:15-3:15pm
  - Location: Duncan Hall 2007, or Duncan Hall 2011 if it’s too crowded.
- You can always schedule an appointment; just email or find me after class.
- I try to answer emails quickly, but I may not be able to respond as promptly over the weekends and later in the evenings.

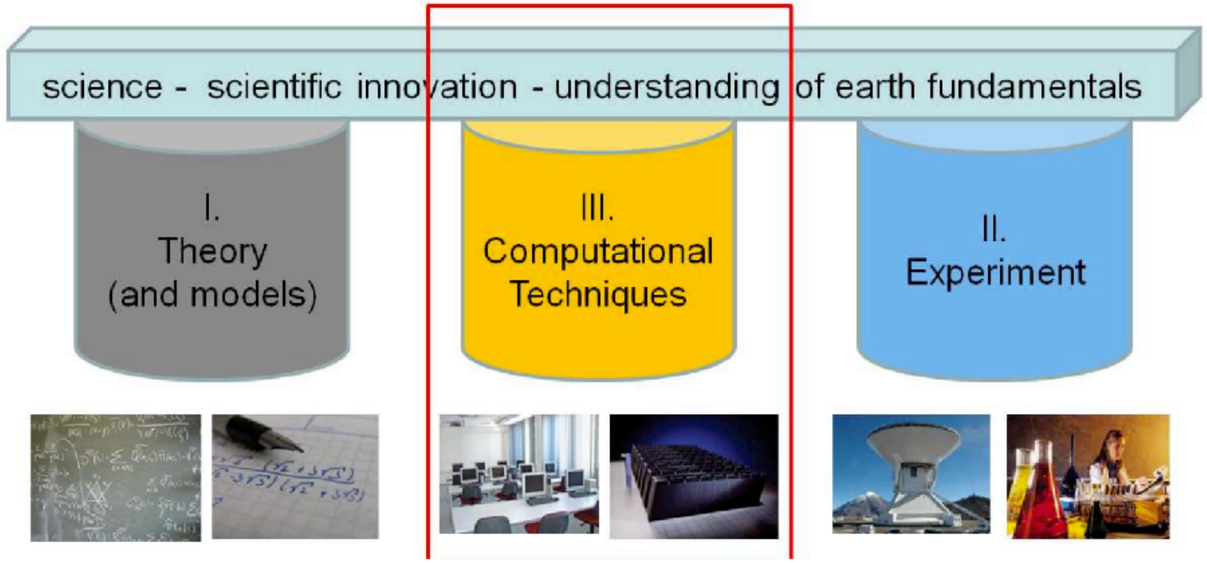
Generative AI policies

- Generative AI tools (ChatGPT, Gemini, Copilot, etc) are generally better at programming than many other scientific tasks (math, logic).
- You may use generative AI tools for assignments, but please note in your writeup which tool you used and provide your prompt history.
    - You are still responsible for making sure that the code generated compiles and runs properly, is readable, and correct.

Some motivation for HPC and parallel computing

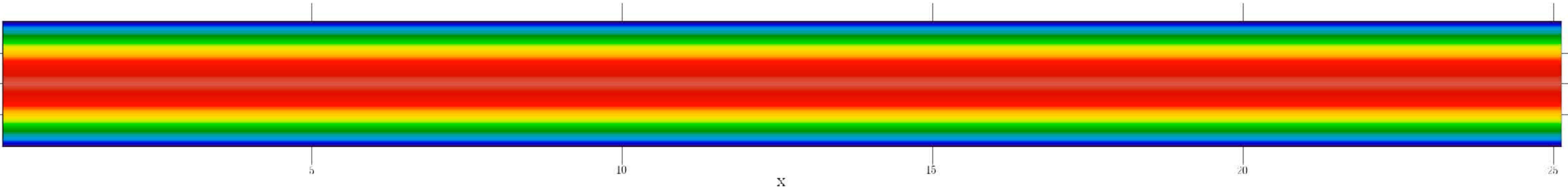
Modern science and computing

- All areas of science heavily utilize computing nowadays: data analysis and processing, simulation, AI/ML, visualization.
- HPC/parallelization typically used to either **speed up** or **scale up** a task.
  - Some tasks are infeasible without massive parallelization
- HPC is just as important for smaller codes; together, several small speed-ups make a huge difference.



The “third pillar” of science

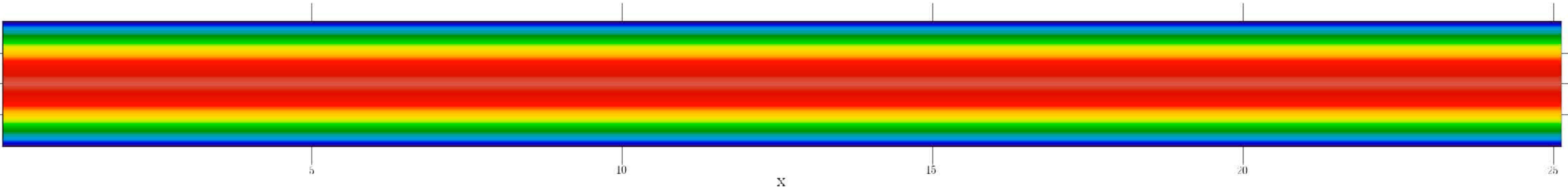
Numerical solution of differential equations



Movie from Myoungkyu Lee and Robert D. Moser

- I work in numerical simulations involving complex time-dependent partial differential equations (PDEs), specifically computational fluid dynamics.
- Turbulent length scales require 1000s of grid points in each direction; in 3D, this easily leads to billions of unknowns.

Numerical solution of differential equations

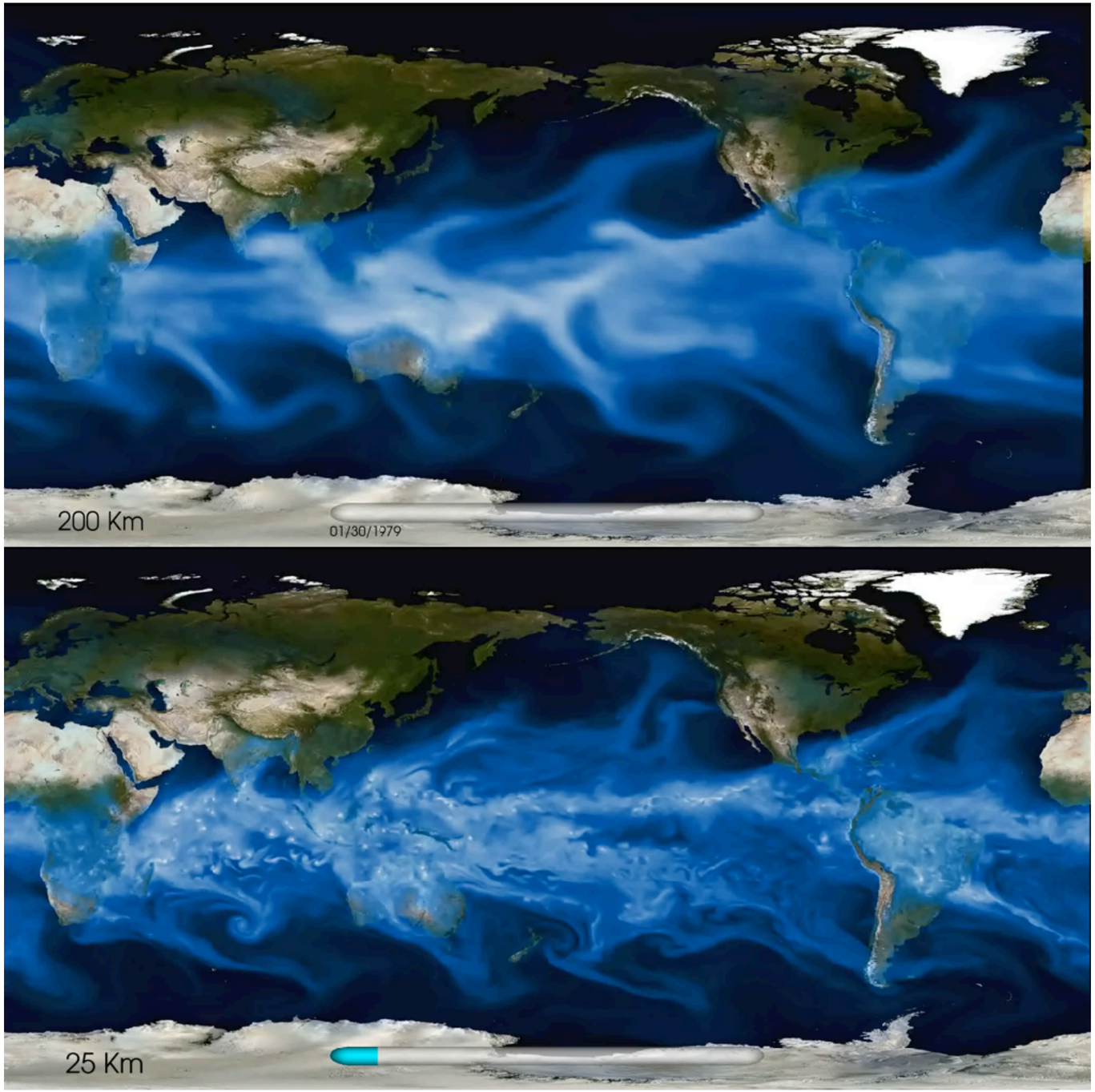


Movie from Myoungkyu Lee and Robert D. Moser

- I work in numerical simulations involving complex time-dependent partial differential equations (PDEs), specifically computational fluid dynamics.
- Turbulent length scales require 1000s of grid points in each direction; in 3D, this easily leads to billions of unknowns.

High resolution simulations

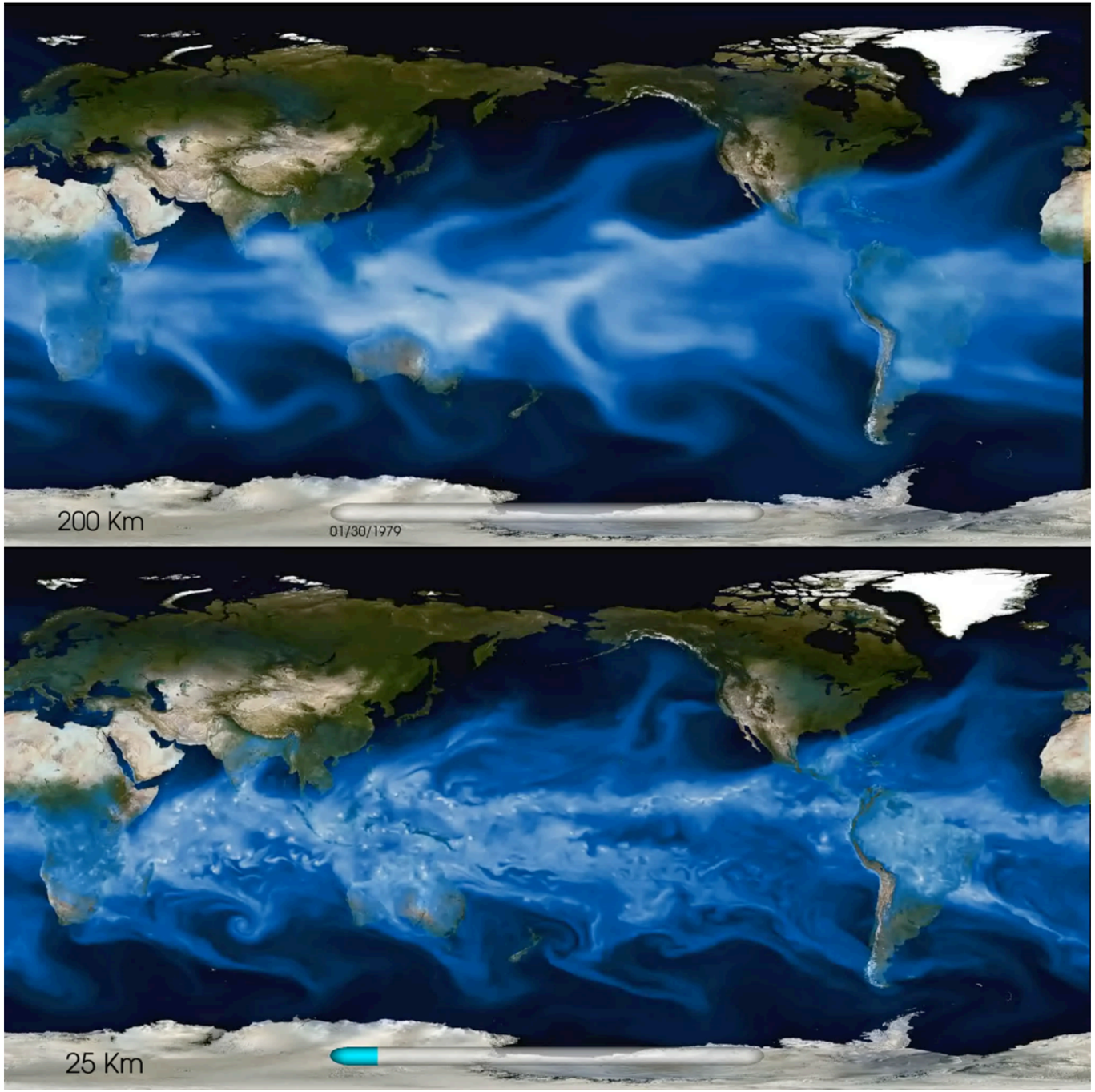
- Climate science simulations of global-scale fluid dynamics are very computationally expensive.
- Simulations are done on different generations of supercomputers; newer computers enable higher resolution simulations.
- Not just pretty movies; the higher resolution simulation discovers hurricanes that are not visible at the lower resolution simulation.



Michael Wehner, Prabhat, Chris Algieri, Fuyu Li, Bill Collins, Lawrence Berkeley National Laboratory; Kevin Reed, University of Michigan; Andrew Gettelman, Julio Bacmeister, Richard Neale, National Center for Atmospheric Research

High resolution simulations

- Climate science simulations of global-scale fluid dynamics are very computationally expensive.
- Simulations are done on different generations of supercomputers; newer computers enable higher resolution simulations.
- Not just pretty movies; the higher resolution simulation discovers hurricanes that are not visible at the lower resolution simulation.



Michael Wehner, Prabhat, Chris Algieri, Fuyu Li, Bill Collins, Lawrence Berkeley National Laboratory; Kevin Reed, University of Michigan; Andrew Gettelman, Julio Bacmeister, Richard Neale, National Center for Atmospheric Research

Enabling huge ensembles of simulations

High Throughput HPC for Materials Design

Design of Materials for Batteries, Solar Panels and More

Software

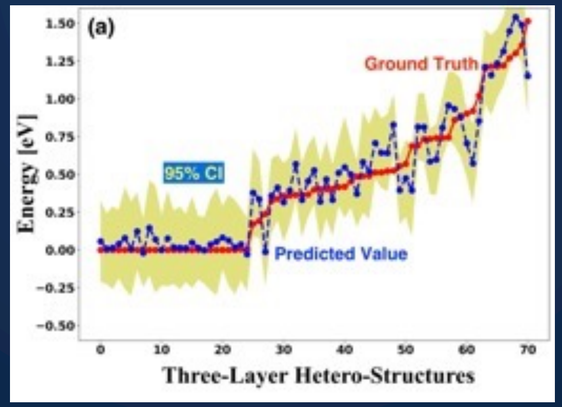
Supercomputers

Screening

> 40,000 Users

NANOPOROUS MATERIALS	530,243
INORGANIC COMPOUNDS	131,613
BAND STRUCTURES	76,194
MOLECULES	49,705

Data



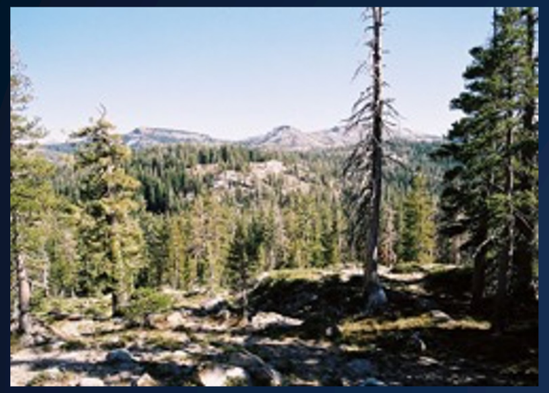
- Use of Bayesian optimization for layered materials
- [Bassman et al, npj Computational Materials 2018]

Kristin Persson, Gerd Ceder, MSE UCB and LBNL, Materials Project

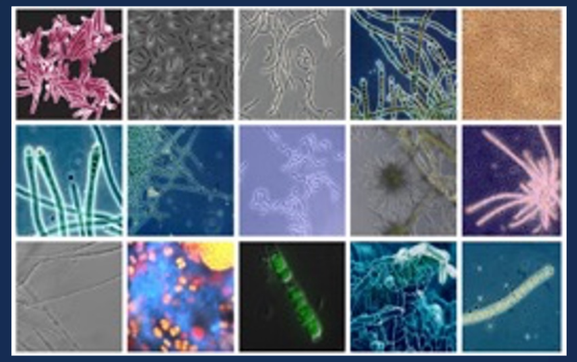
Slide from Aydin Buluc

Data science and analysis

High Performance Data Analytics (HPDA) for Genomics



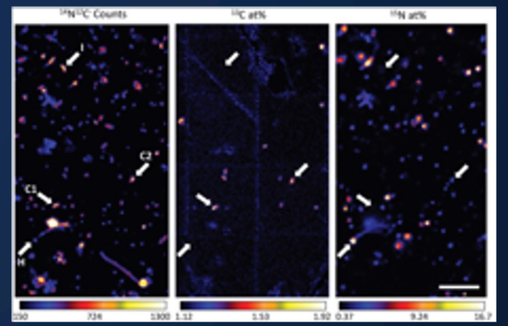
What happens to microbes after a wildfire? (1.5TB)



What are the microbial dynamics of soil carbon cycling? (3.3 TB)

How do microbes affect disease and growth of switchgrass for biofuels (4TB)

What at the seasonal fluctuations in a wetland mangrove? (1.6 TB)



Combine genomics with isotope tracing methods for improved functional understanding (8TB)

JGI-NERSC-KBase FICUS projects, MetaHipMer assembler ExaBiome project

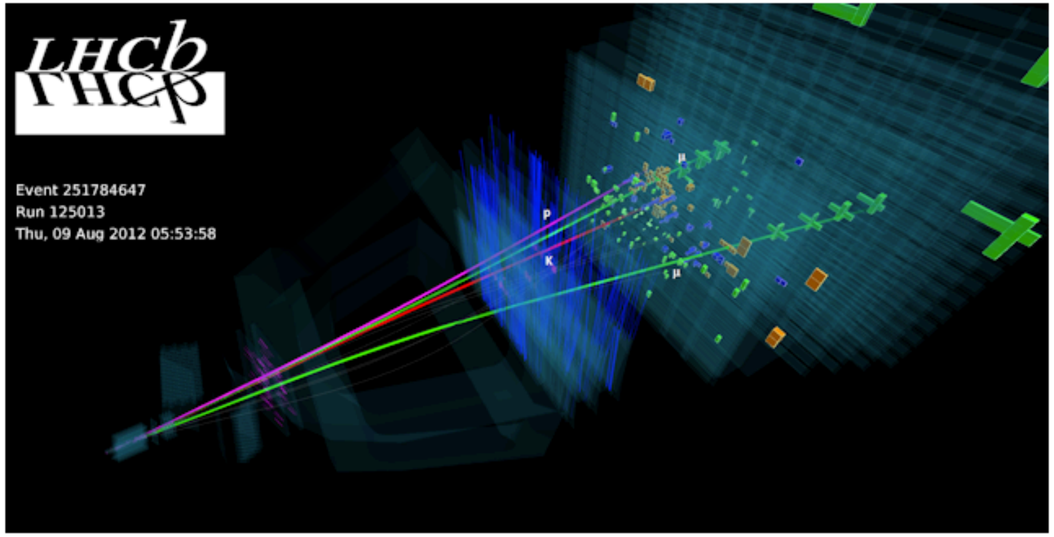
Slide from Aydin Buluc

LHCb releases the entire Run I dataset

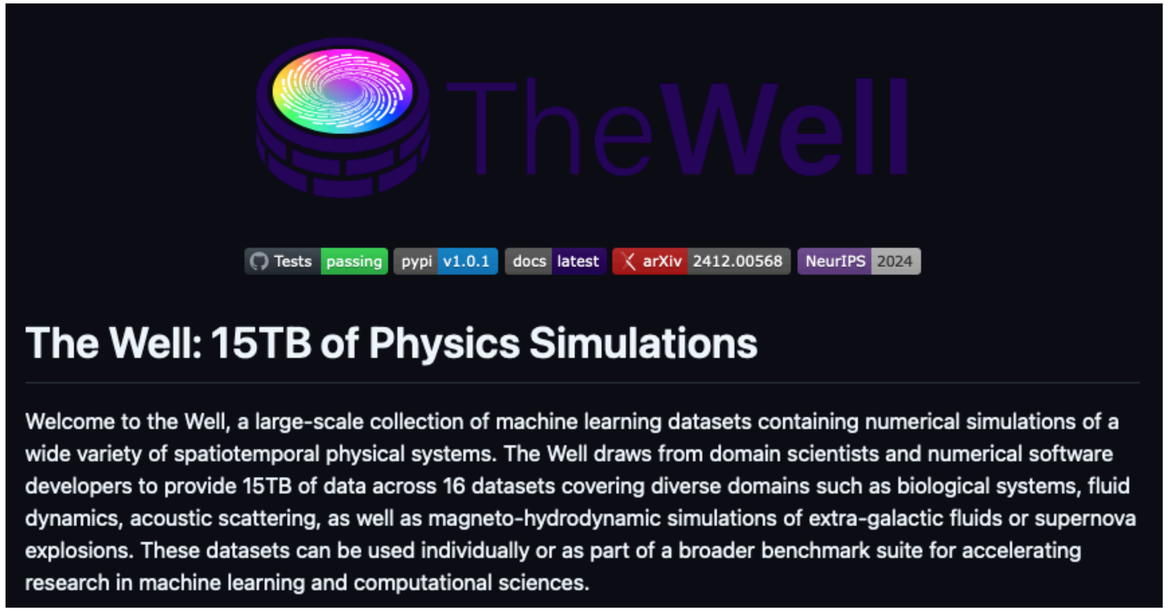
By **pietrzyk**

DEC 20, 2023

Today the LHCb collaboration complete the release of data collected throughout the Run I of the Large Hadron Collider at CERN. The sample made available amounts to approximately 800 terabytes (TB) of data. These data, collected by the LHCb experiment in 2011 and 2012, contain information obtained from proton-proton collisions[1] . The format made available provides pre-filtered data, suitable for a wide range of physics studies. The image below displays an event recorded during 2012.

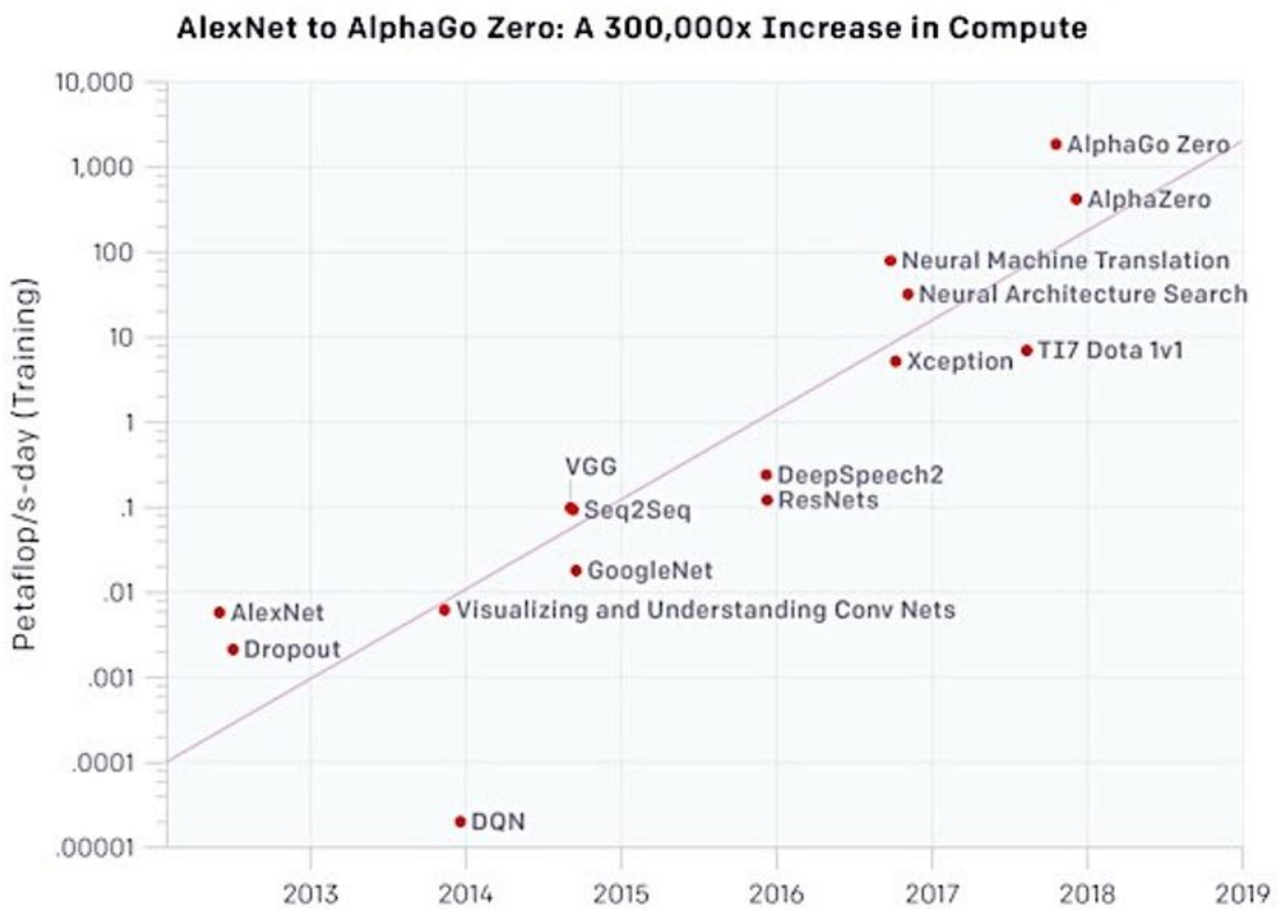


The size of data sets has exploded in multiple scientific fields



ML and AI are very compute hungry

- HPC, parallel, and GPU computing all existed before modern AI/ML
- However, AI/ML training turned out to be an almost ideal fit for modern GPU-based HPC.
- AI/ML now driving GPU hardware development.



https://blog.openai.com/ai-and-compute/

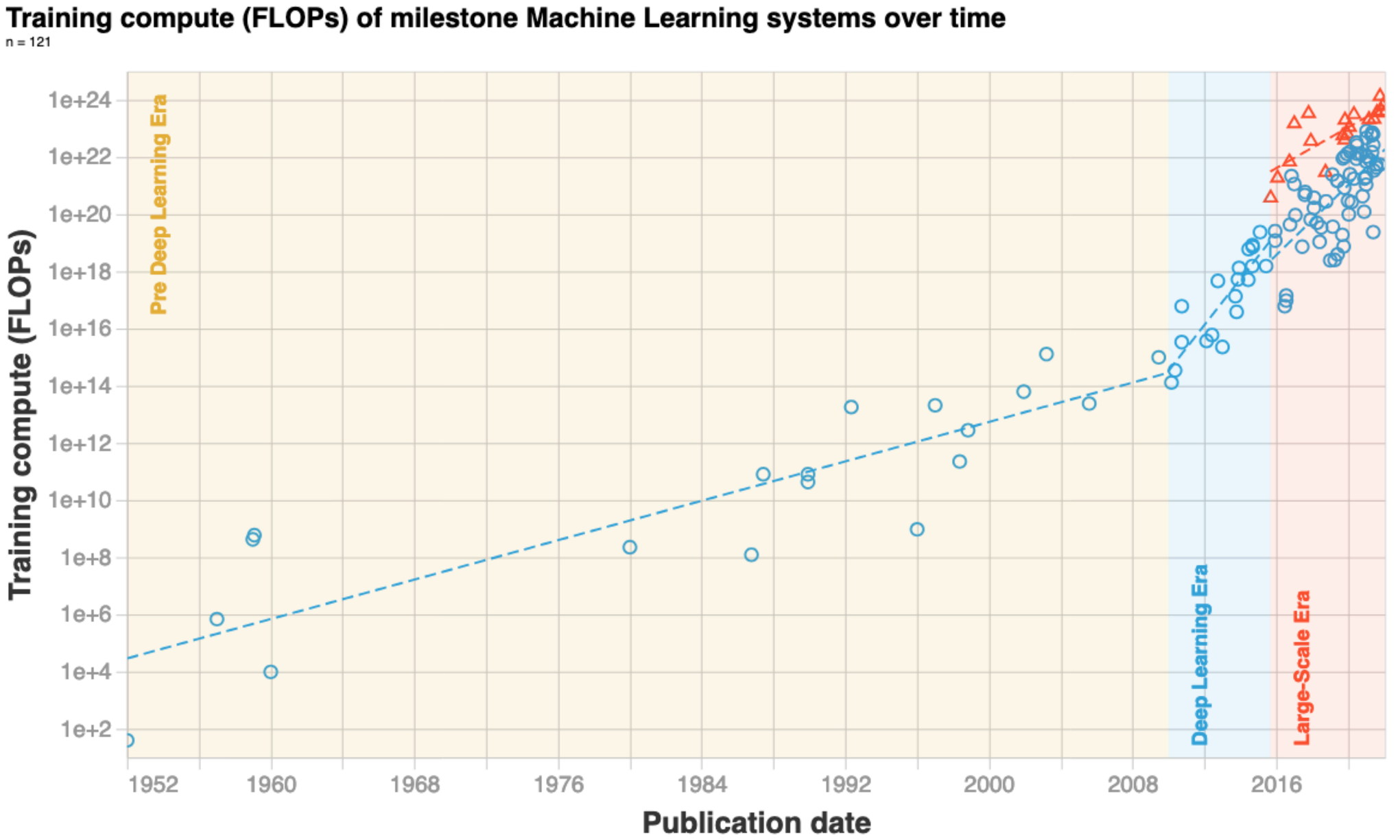


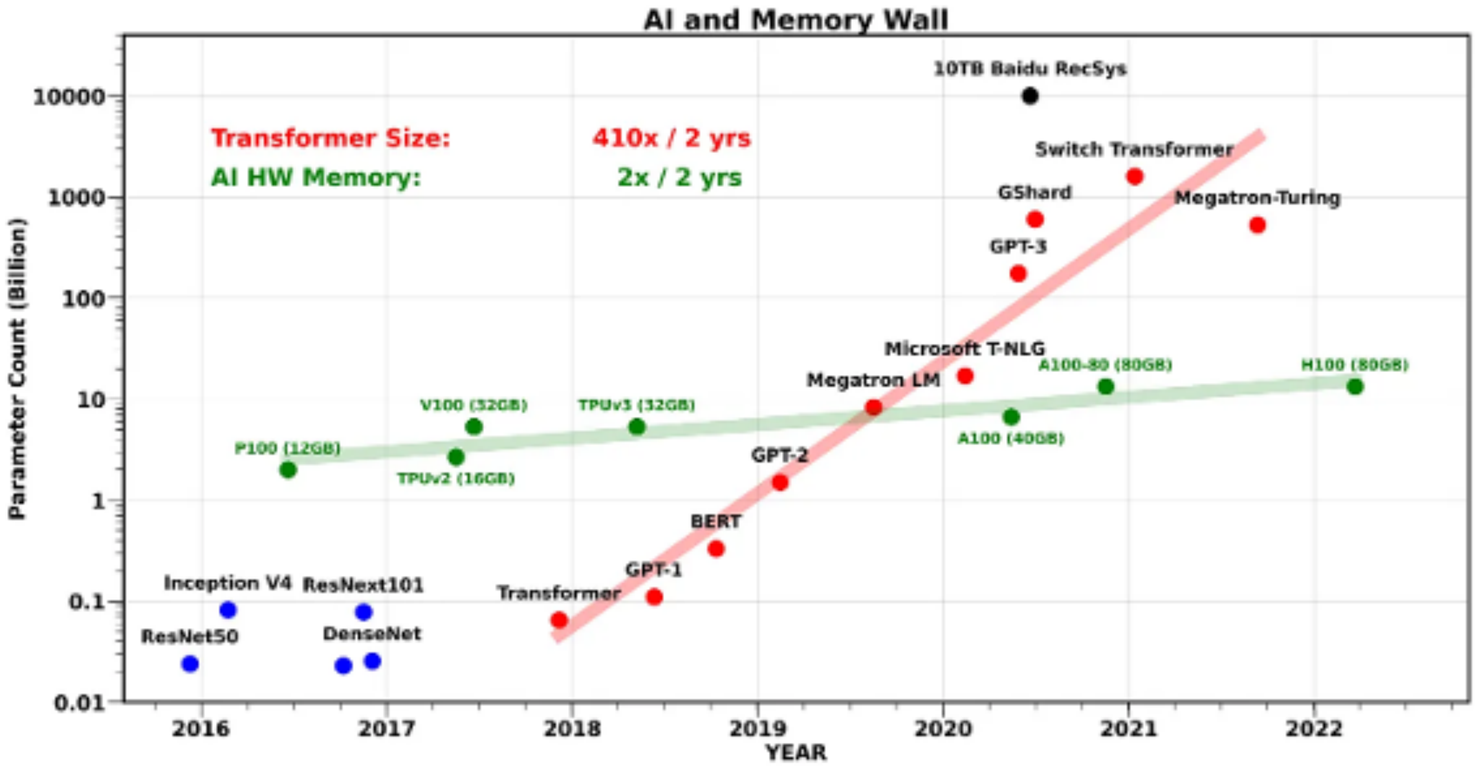
Figure 1: Trends in $n = 121$ milestone ML models between 1952 and 2022. We distinguish three eras. Notice the change of slope circa 2010, matching the advent of Deep Learning; and the emergence of a new large-scale trend in late 2015.

"Compute trends across three eras of machine learning", Sevilla et al., 2022

https://blog.openai.com/ai-and-compute/

AI, ML, deep learning are memory hungry too

- The size of AI models has also grown incredibly quickly.
- Parallelism is necessary just to store the model parameters (~40 TB of memory for the 10TB Baidu model)



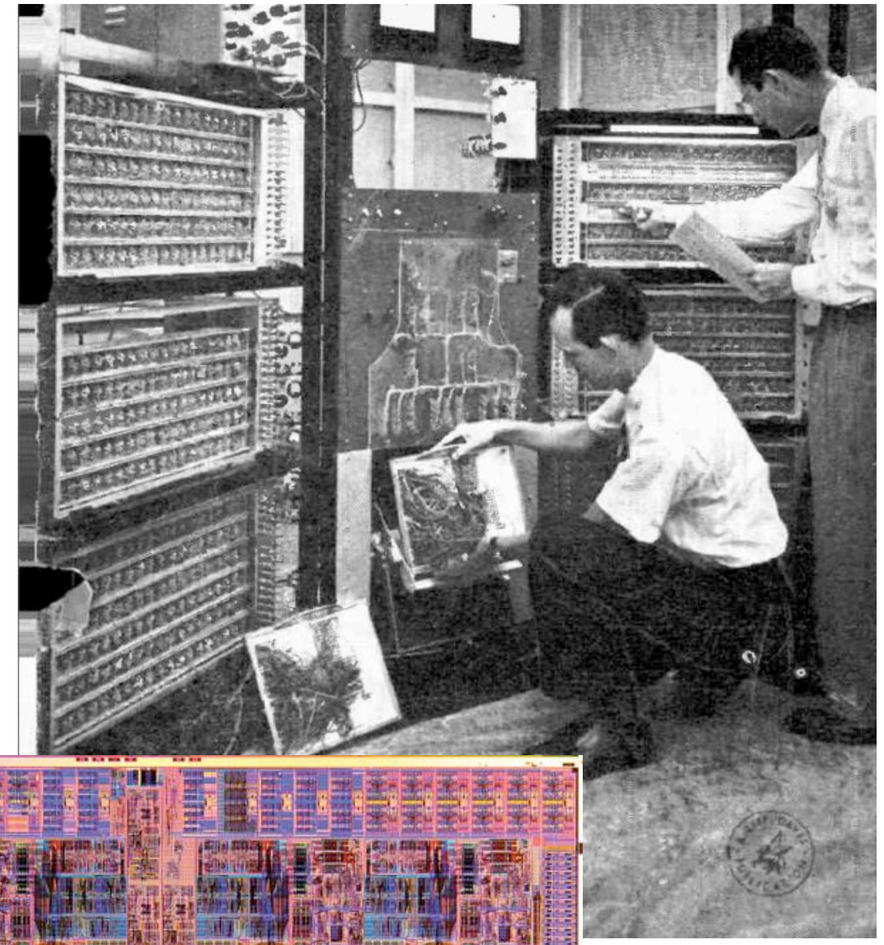
https://medium.com/riselab/ai-and-memory-wall-2cb4265cb0b8

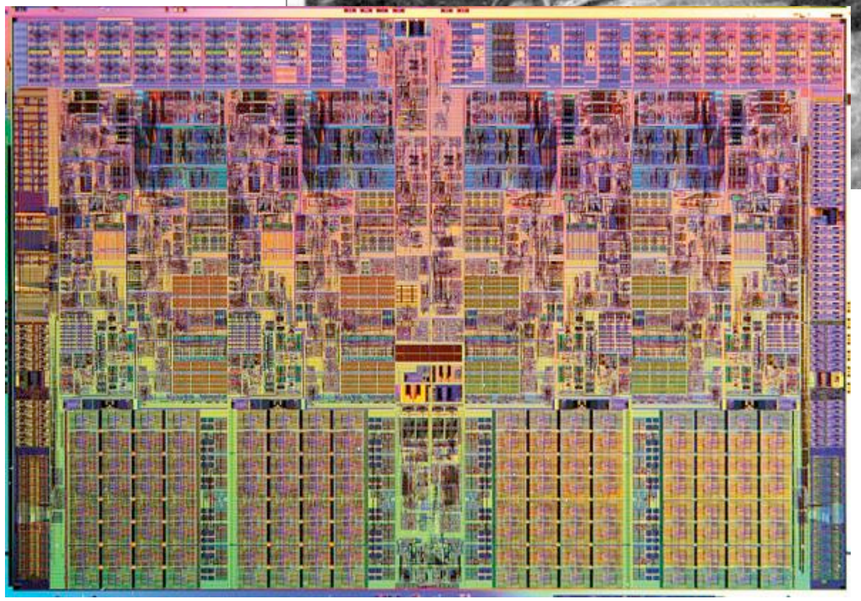
Some cartoons and stories about computer architectures

Note: I am *not* an expert on computer hardware.

Central processing unit (CPU)

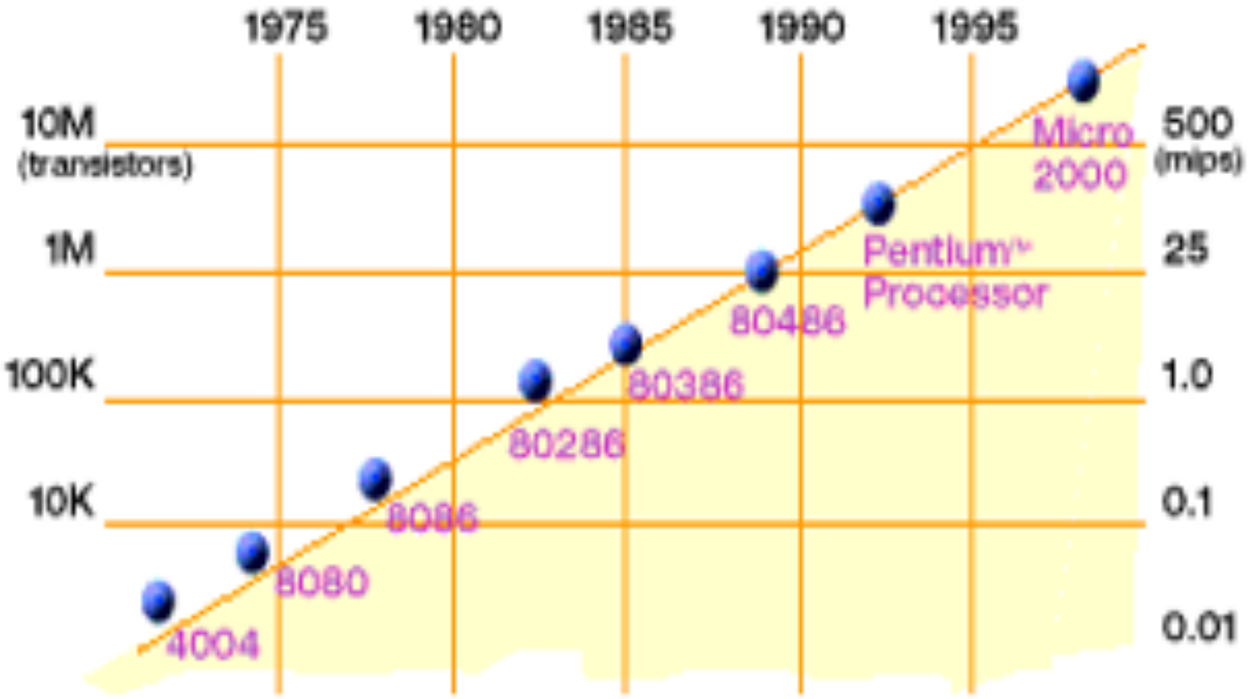
- For our purposes, CPUs perform arithmetic operations.
  - The clock speed (GHz) is how many operations per second.
- Transistors are used to build binary switches (logic gates)
  - Until recently, the more transistors are on a CPU, the higher the clock speed.



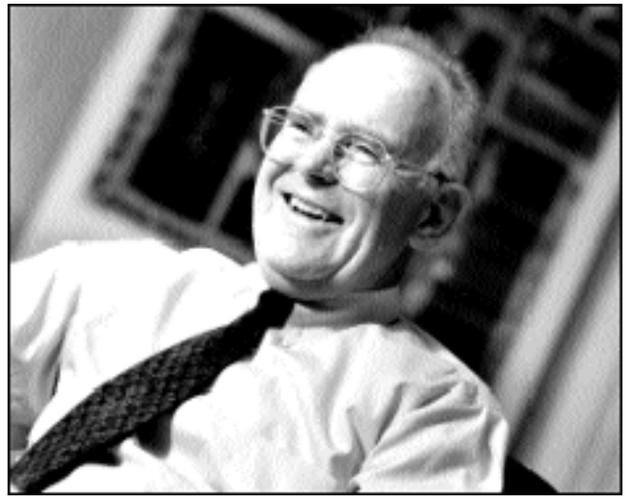


Moore’s Law

- Gordon Moore (one of the founders of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months.
- Until ~2006, you could just wait for a new computer to be released and your code would magically become 2x faster.

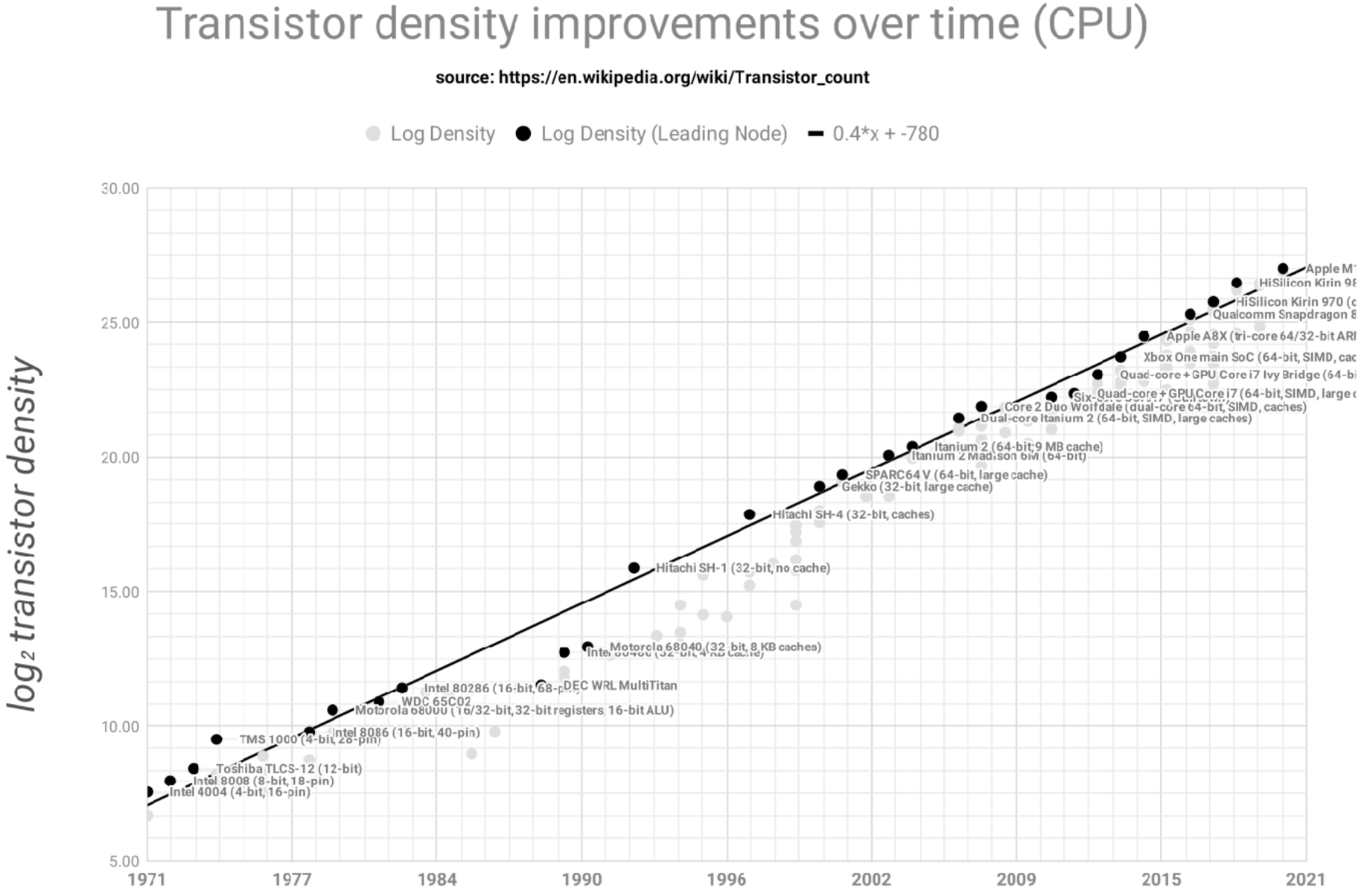


Notice the log scales!



Moore’s law continues to this day, but…

- The number of transistors is still doubling about every 1.5 years. However, a lot has changed behind the scenes.
- Old code doesn’t automatically become faster on newer computers.
- Roughly speaking, this is the motivation for parallel computer architectures.



Why did code just get faster until ~2006?

- Moore's law: the numbers of transistors doubles every 1.5 years.
- Clock speed (the number of operations executed per second) also grew exponentially.
- Moreover, the power consumption *didn't* grow at the same rate (Dennard scaling)

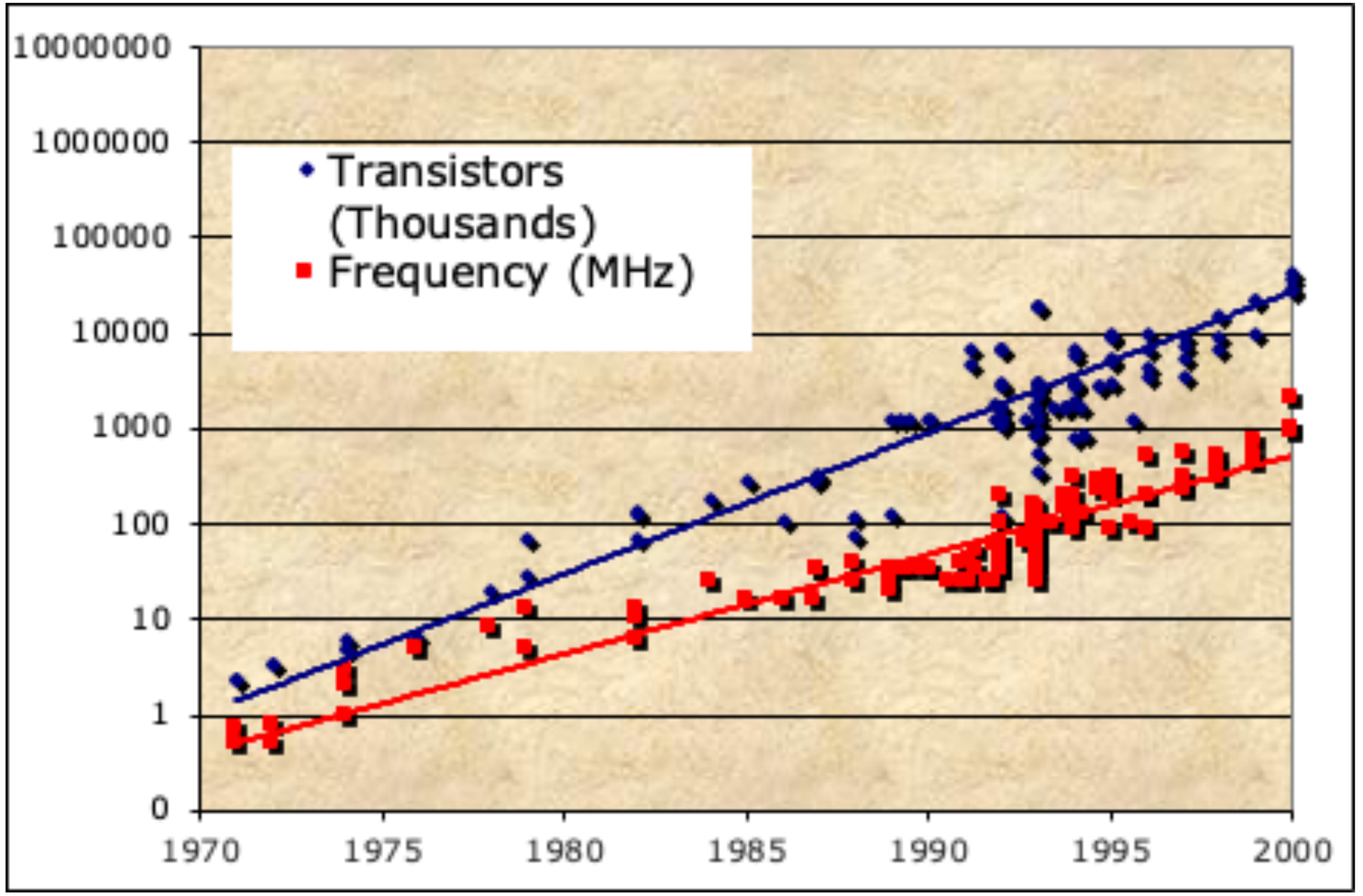
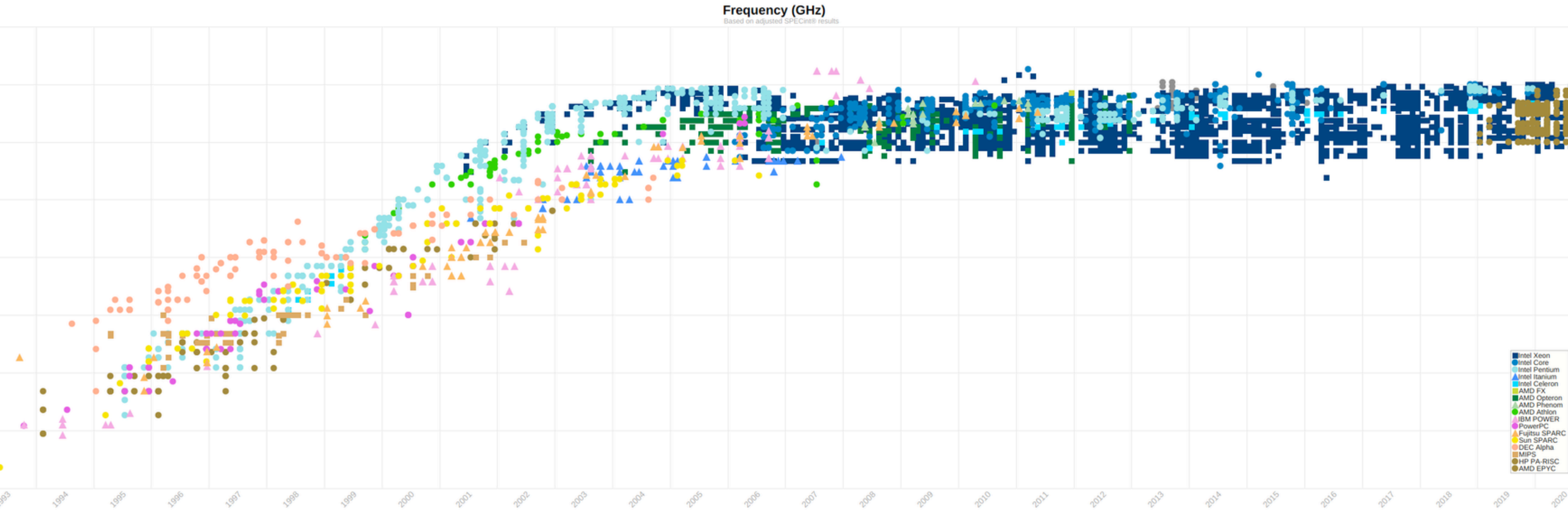


Figure from Aydin Buluc

What happened in ~2006?

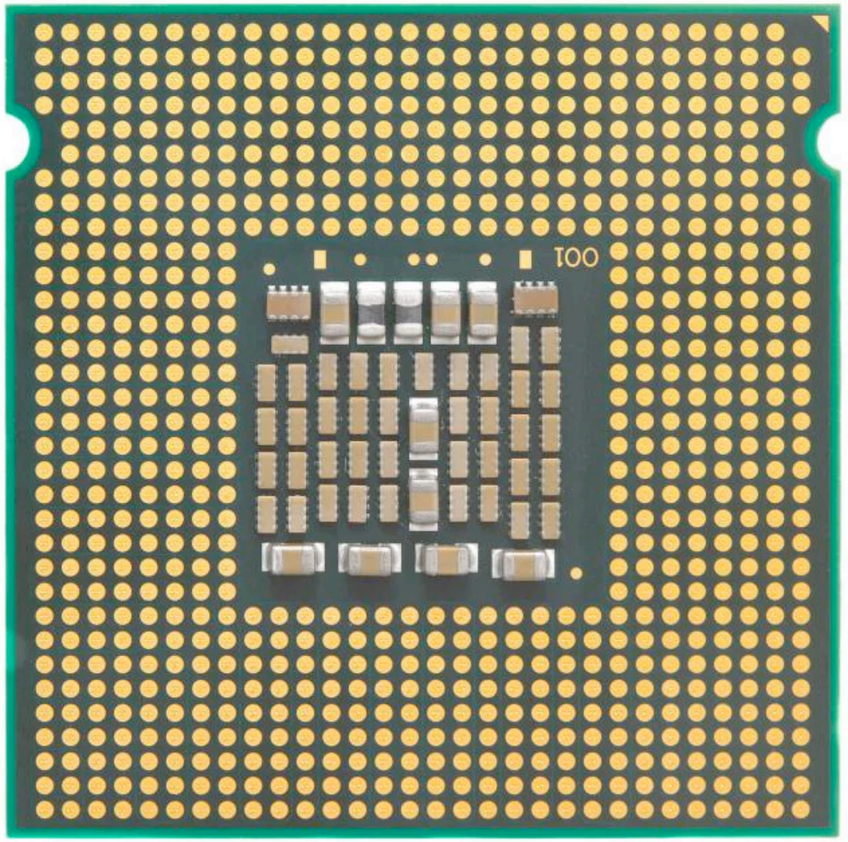
CPU clock speed suddenly leveled off



https://www.lesswrong.com/posts/aNAFrGbzXddQBMDqh/moore-s-law-ai-and-the-pace-of-progress

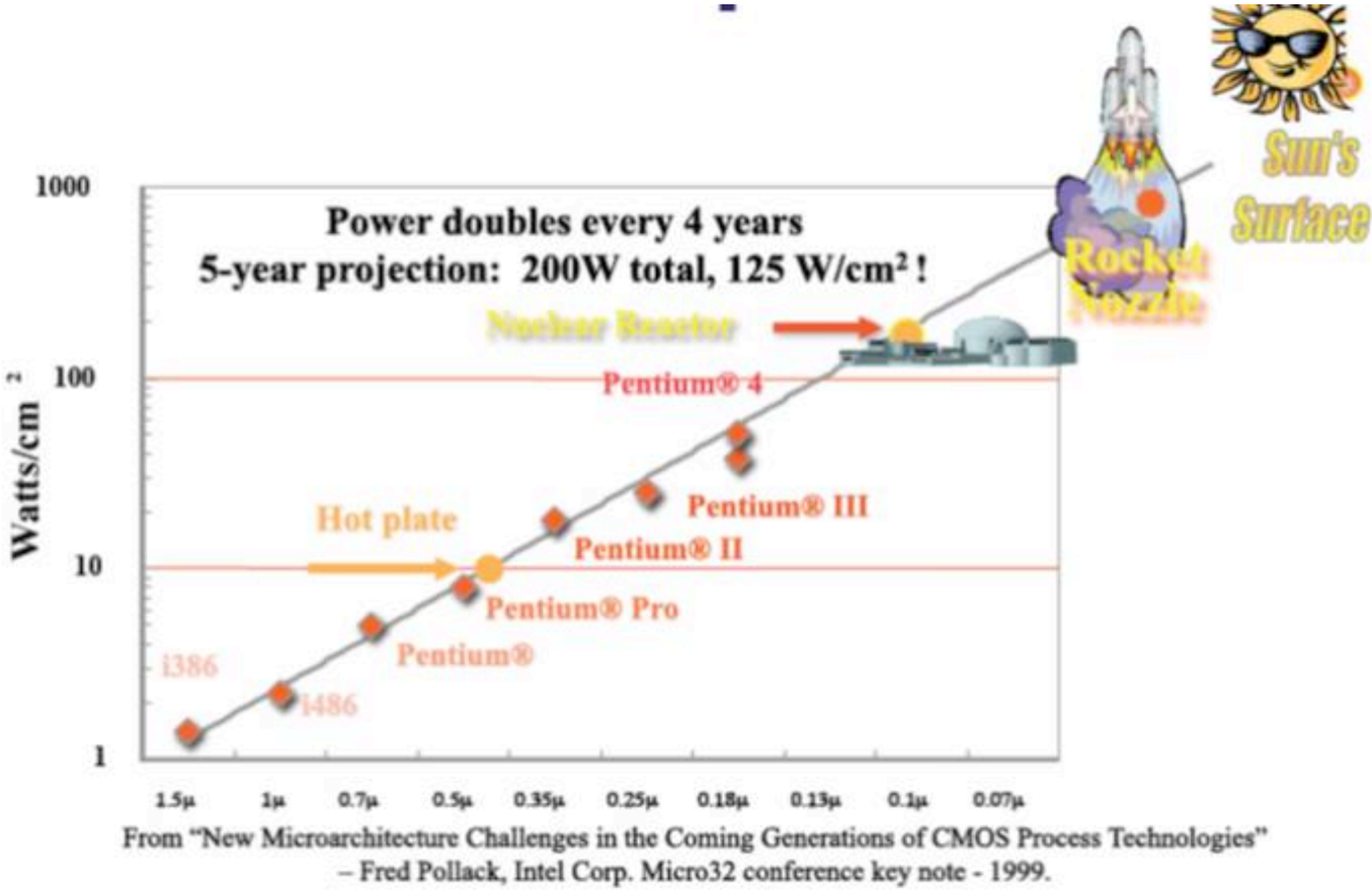
Transistors tended to shrink

- If you shrink a transistor by a factor “x”, then
  - You get x^2 more transistors per unit of area increases
  - Transistors are closer together, wires are shorter, so it takes less time to transfer data; can increase the clock rate by ~x.
  - Can also make the chip/die larger to fit more transistors, increase clock rate.
- However, there are physical limits to how small transistors can be, as well as how efficient they can be as they shrink.



Physical constraints and concerns

- Dennard scaling: shrinking a transistor makes it more power efficient…up to a point.
- As transistors get smaller, more power is lost due to leakage.
  - Results in a lot of heat!
- CPU's also tend to waste a lot of energy; modern architectures (ARM, GPU) also address this.

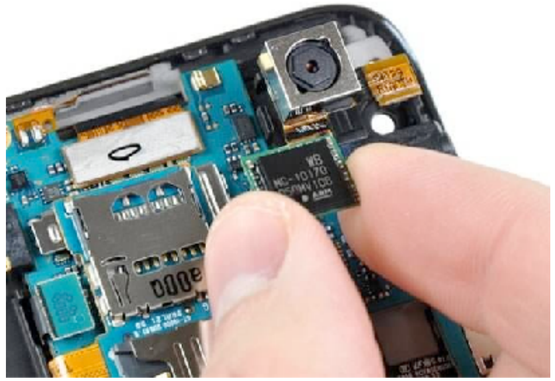


From "New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies" – Fred Pollack, Intel Corp. Micro32 conference key note - 1999.

So where does that leave computing?

- Dennard scaling is dead; the clock speed of an individual CPU has stalled.
- The only way to continue increasing computing power is to use multiple cores per CPU and rely on parallelism.
  - Moore’s Law still applies - but with the caveat that the doubling of transistors is now going towards multi-core processors.
  - Manufacturing defects easier to handle with multi-core processors.
- A related trend: a focus on energy efficiency for parallel architectures (e.g., in mobile devices, GPUs, specialized hardware like FPGAs).

What does that mean for this class?



CPU GPU TPU

- All computing architectures (from phones to laptops to GPUs to supercomputing clusters) are relying on parallelism.
- However, in contrast to pre-2006 trends, serial codes cannot automatically take advantage of multiple cores and parallelism. Must account for:
  - Different types (shared memory, distributed) of parallelism.
  - Different scales:12 cores on a laptop CPU vs 1000s of GPU cores.

HPC vs parallel computing

- Why is this class called High Performance Computing (HPC) and how is it different from Parallel Computing?
  - Not just parallel computing; serial optimizations are important too.
  - We don’t focus much on parallel *algorithms* and algorithmic analysis, more on *performance* analysis and some representative model problems.
  - See Prof. Mellor-Crummey for a great parallel computing class
- There is still lots of room for non-parallel HPC: even on a single CPU, most programs *significantly* under-utilize computational resources!

Review of some programming and numerical computing concepts

Scientific computing in C++

- Programming languages abstract away lower-level hardware and assembly-type details.
- In HPC, it helps to be aware of lower-level and hardware details:
  - How variables are represented
  - Where variables are in memory (e.g., the pointer address and memory layout)

```
int main(){
    double num1 = 10;
    double num2 = 5;

    // Some arithmetic operations
    double sum = num1 + num2;
    double difference = num1 - num2;
    double product = num1 * num2;
    double quotient = num1 / num2;
    double remainder = num1 % num2;

    return 0;
}
```

Data types in scientific computing

- For numerical computing, it can help to know the difference between different data types (e.g., integer, float, double, long int, etc) and what distinguishes them from each other
  - Integers and decimal (float, double) behave very differently!
- To start, numbers are represented in binary in a computer.
  - If we represent an integer in binary, 0110 corresponds to

$$5 = 0 * (2^0) + 1 * (2^1) + 1 * (2^2) + 0 * (2^3)$$

Binary representations of integers

- 5 = 0110 = 0 * (2^0) + 1 * (2^1) + 1 * (2^2) + 0 * (2^3)
  - Binary 0/1 digits are coefficients in a binary expansion
  - We use 4 “bits” here; 64-bit integers are most common, but 8-bit, 16-bit, and 32-bit integers are all used.
- What does this expansion imply about the numbers that can be represented in binary?

Binary representations of integers

- 5 = 0110 = 0 * (2^0) + 1 * (2^1) + 1 * (2^2) + 0 * (2^3)
  - Binary 0/1 digits are coefficients in a binary expansion
  - We use 4 “bits” here; 64-bit integers are most common, but 8-bit, 16-bit, and 32-bit integers are all used.
- What does this expansion imply about the numbers that can be represented in binary?
    - There is a min/max; for example, $2^N - 1$ is the largest “unsigned int” you can represent with N bits.

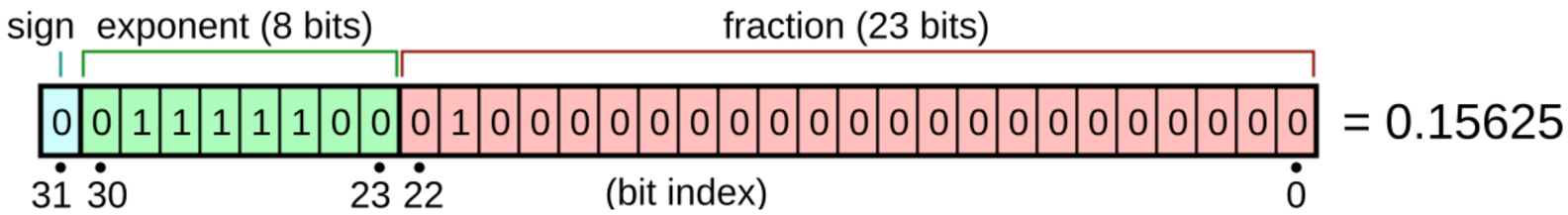
Unsigned vs signed integers

- You can have both “unsigned integer” or “signed integer” types (e.g., “unsigned int” vs “int”)
  - Both have the same bit representation, but can yield different results because they are *interpreted* differently
  - For signed integers: one bit determines the sign.
- Advantages of unsigned integers: represent larger values, provides additional information about what the variable is for.

What about decimal numbers?

- Easy to represent integers as binary; what about decimal?
  - Fixed point representation: represent digits before and after a decimal point (fixed number of representable digits)
  - Floating point representation: represent a number as a coefficient and exponent. Higher *relative* accuracy.

$$12.345 = \underbrace{12345}_{\text{significand}} \times \underbrace{10^{\overbrace{-3}^{\text{exponent}}}}_{\text{base}}$$



https://en.wikipedia.org/wiki/Floating-point_arithmetic

Consequences of floating point representations

- Rounding: the sum of two floating point numbers may not be representable as another floating point number, so the result is *rounded* to the nearest representable number.
- Checking equality of floating point numbers is unreliable!

```
julia> sin(1.0 * pi) == 0.0
false
```

- Catastrophic cancellation: subtracting large numbers from each other can result in large relative errors due to rounding.
- Overflow: NaNs/Infs for undefined operations or very large numbers.

Some other things to know

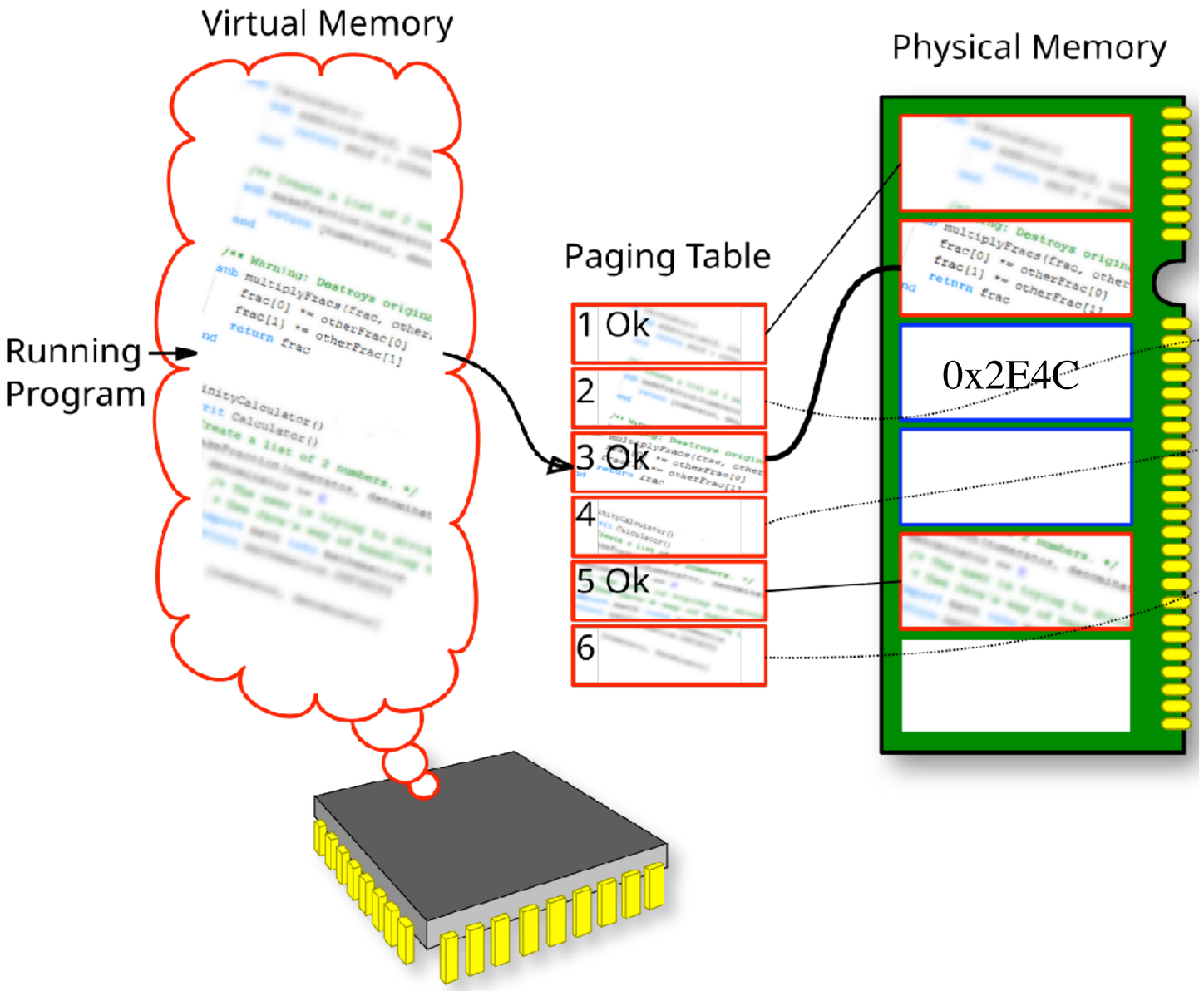
- Changing the order of operations can change results
- We will use integer division “ / ” and the modulus operation “%” fairly often.
  - Integer division rounds down (truncates), e.g., “int a = 5 / 3” yields “a = 1”.
  - Modulus returns the remainder of integer division
- Integer division and modulus can be used to recover nested loop indices from a single loop index.

Arrays in C++

- This class uses a lot of basic arrays in C++. You should be comfortable with:
  - Dynamic memory allocation and deallocation
  - Pointers, pointer arithmetic
  - Memory layout of arrays

```cpp
#include <iostream>
using namespace std;

int main(){
    int size = 1e9;

    // Allocate memory for an integer array
    int* myArray = new int[size];
    for (int i = 0; i < size; i++) {
        myArray[i] = i * i;
    }

    // Calculate the sum of the array elements
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += myArray[i];
    }

    cout << "Sum of array elements: " << sum << endl;

    // Deallocate memory
    delete[] myArray;

    return 0;
}
```

Pointers



- A pointer address is a **memory address** of a variable in C.
- Provides the variable's *location* but not what's contained inside.
- The address is *not* typed, but pointers in C are typed. Why? C uses the pointer type for array indexing…

https://en.wikipedia.org/wiki/Memory_address

Pointers in C (and C++)

- Pointers store a *memory address* as a hexadecimal number
- Syntax: specify the variable type and add *
  - e.g., “int * x”, “float * y”, “int ** z”, …
- You *reference* a variable’s address via “&”: “int* x_ptr = &x;”
  - Referencing converts a variable to its address.
- You *de-reference* a pointer address via * : “int x = *x_ptr;”
  - De-referencing converts an address to a value.

Why Hex?

Hexadecimal numbers can encode more information in fewer digits than smaller bases

Decimal (10)	Binary (2)	Hexadecimal (16)
1	0000 0001	1
2	0000 0010	2
3	0000 0011	3
8	0000 1000	8
15	0000 1111	F
16	0001 0000	10
255	1111 1111	FF

Allocating memory: stack vs heap memory

- Computer memory (e.g., RAM) is split into “stack” and “heap” memory by the operating system.
  - The “stack” is faster, but has a smaller fixed size (~8 MB).
    - If each double is 8 bytes, what’s the largest vector that this can store? What about the largest matrix?
  - The “heap” is larger, but is (slightly) slower and has to be manually managed (“allocated” and “free’d” by the user)
- So far, every variable we have defined used “stack” memory, but exceeding stack memory limits leads to a “stack overflow”. We will use “heap” memory for large arrays.

Static vs dynamic memory allocation

- Computers have both static (stack) and dynamic (heap) memory.
  - This class focuses on heap memory, but you should use stack memory when possible (mostly small fixed size arrays).
- Allocation “binds” heap / RAM memory to a pointer.
  - Should “free” a pointer when you’re done with it to avoid memory leaks.

```c
#include <stdio.h>

int main(void) {
  int x[3];
}
```

Uses *stack* memory

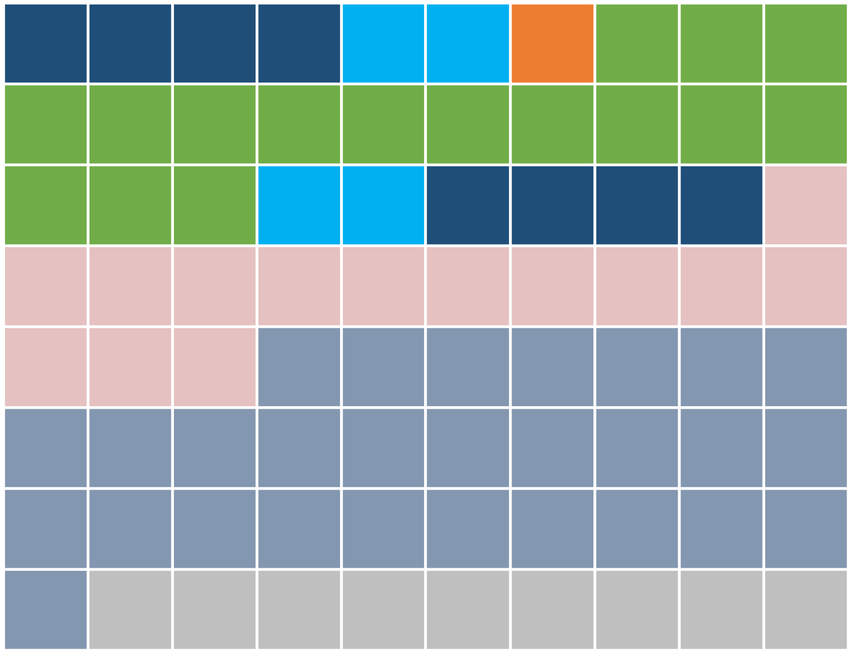
```c
#include <stdio.h>

int main(void) {
  int * x = (int *) malloc(3 * sizeof(int));
  free(x);
}
```

Uses *heap* memory

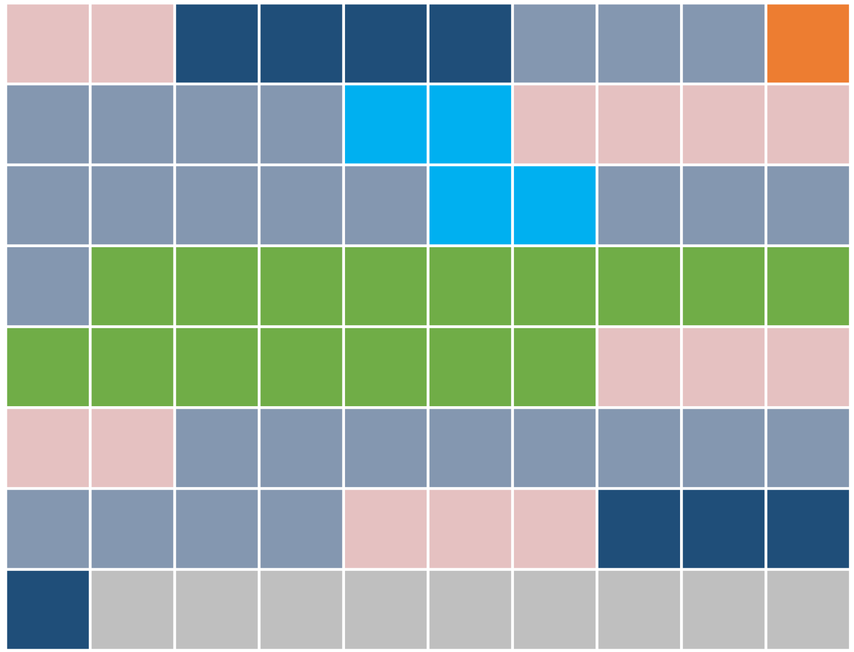
Stack and Heap Management

- **Stack:** Programs/subprograms (functions) are given memory when they are invoked and this memory is freed when they complete
- **Heap:** Programs can allocate and free memory whenever they want throughout their life

Stack



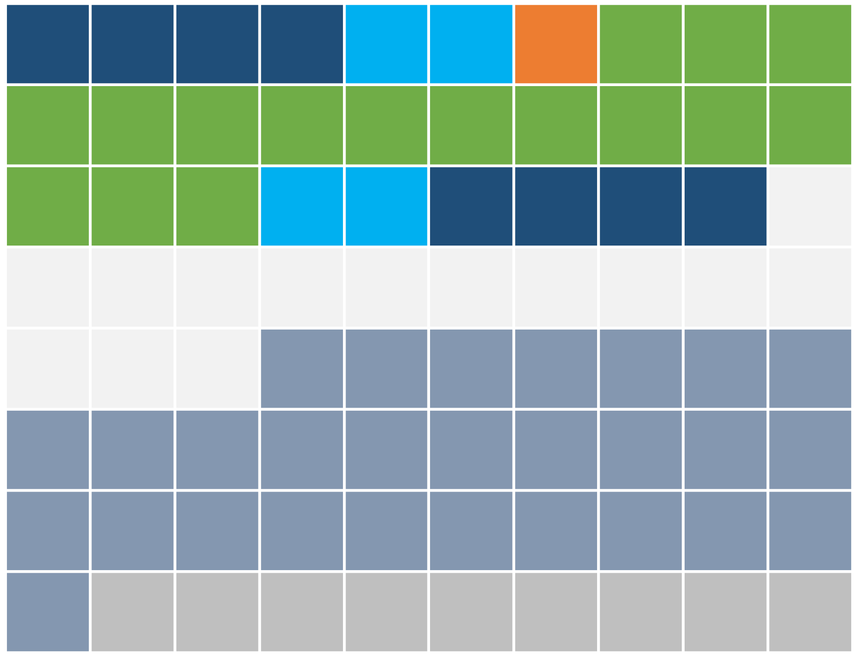
Heap



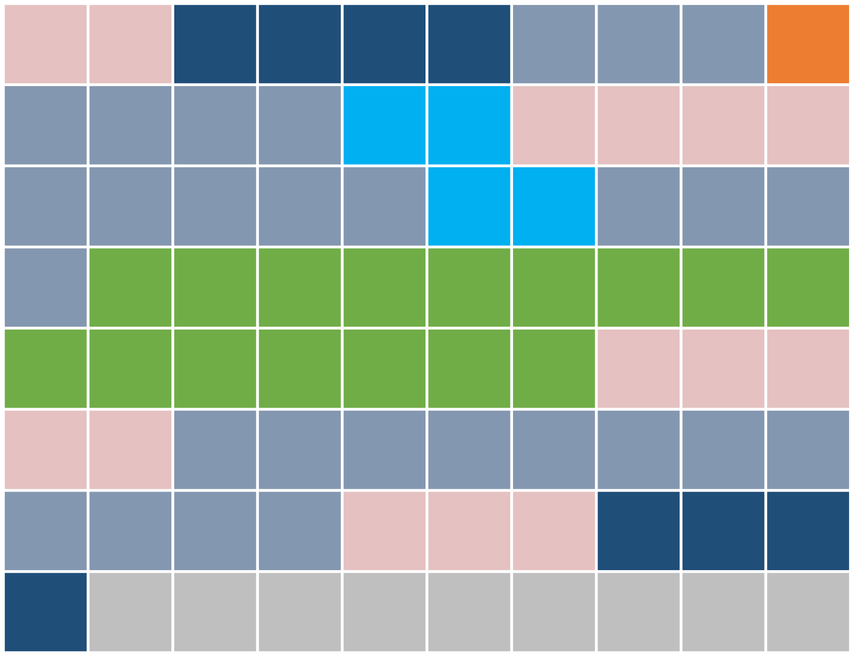
Stack and Heap Management

Stack: When a program finishes, its stack memory is returned to the computer; the computer may rearrange to keep the stack contiguous

Stack



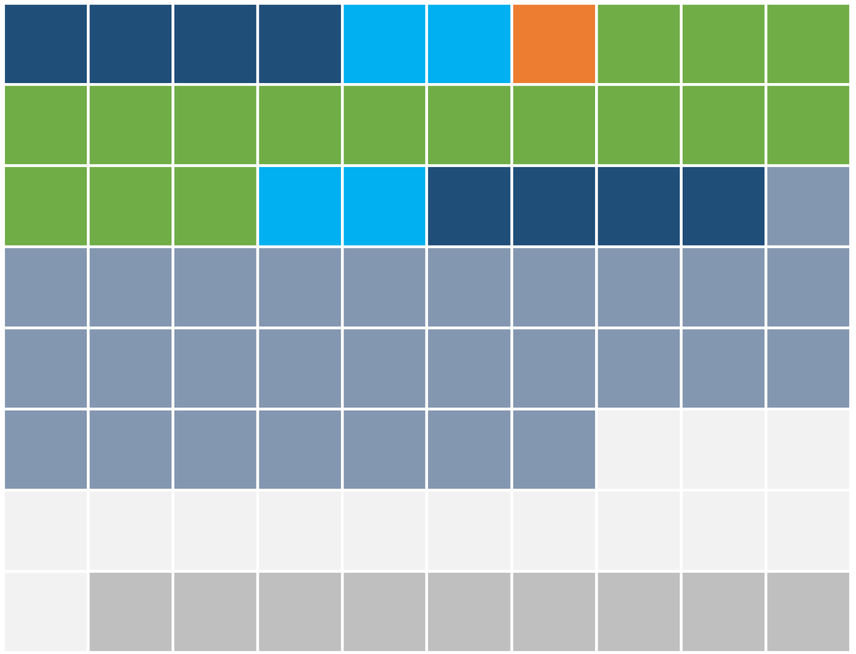
Heap



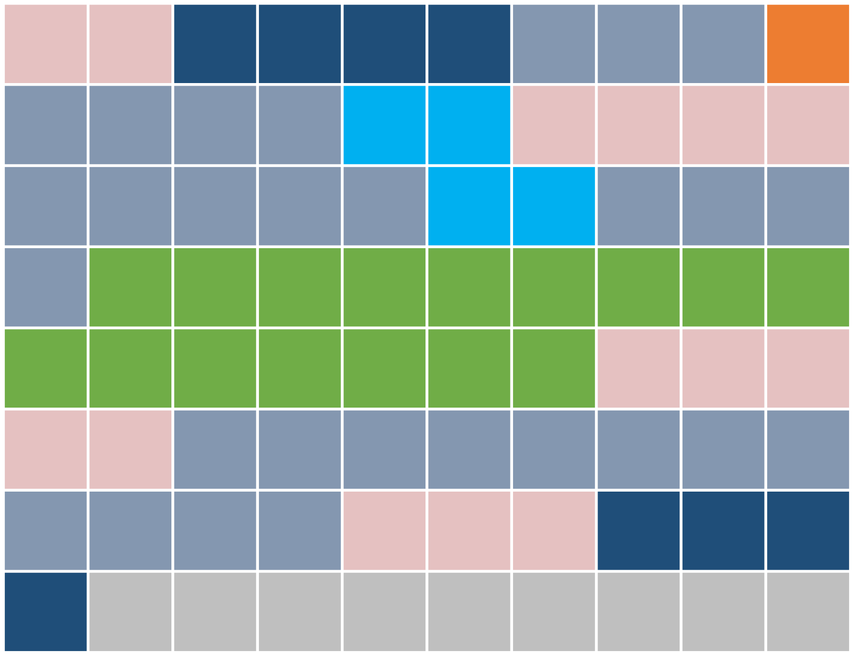
Stack and Heap Management

- **Stack: Pointers to stack memory are dangerous** because they can become undefined as programs close

Stack



Heap

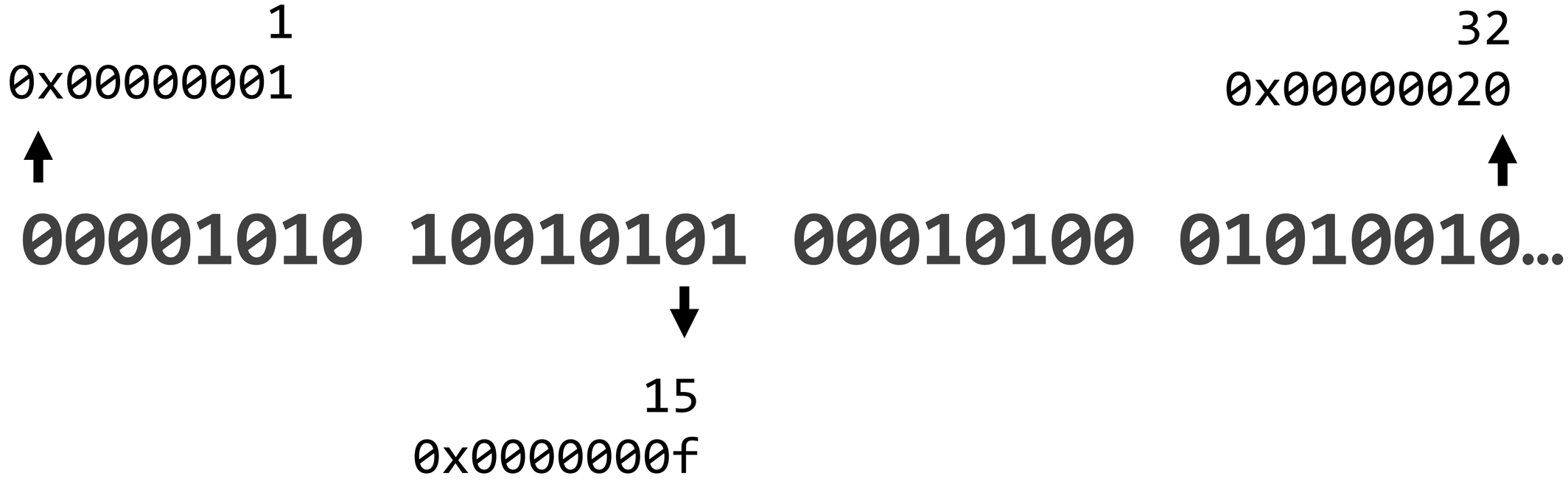


Memory: Bits -> Bytes

00001010 10010101 00010100 01010010…

1 byte = 8 bits (8 binary digits)

Memory: bits and addresses



Memory: Back to Bits

Computers have a lot of memory, and the memory is “aligned” with power of 2

00001010100101010001010010…

01001010010100010100000010101000…

01010101001010101010000001001000…

00010100101111001000101001000001…

Arrays, variable types, and pointers organize these bits into interpretable “chunks”.

Variable Size in Memory

The amount of memory each variable has can vary with the system and compiler

Always the same:

- **1 bit = a 0 or a 1;** the fundamental unit of memory
- **1 byte = 8 bits**

The system decides the size of a “word” of memory:

- **16bit (really old):** 2 bytes per word = 16 bits
- **32bit (older but still around):** 4 bytes per word = 32 bits
- **64bit:** 8 bytes per word = 64 bits

Variable Size in Memory

All you really need to know about variable sizes:

char <= short int <= int <= long int <= long long int

float < double <= long double

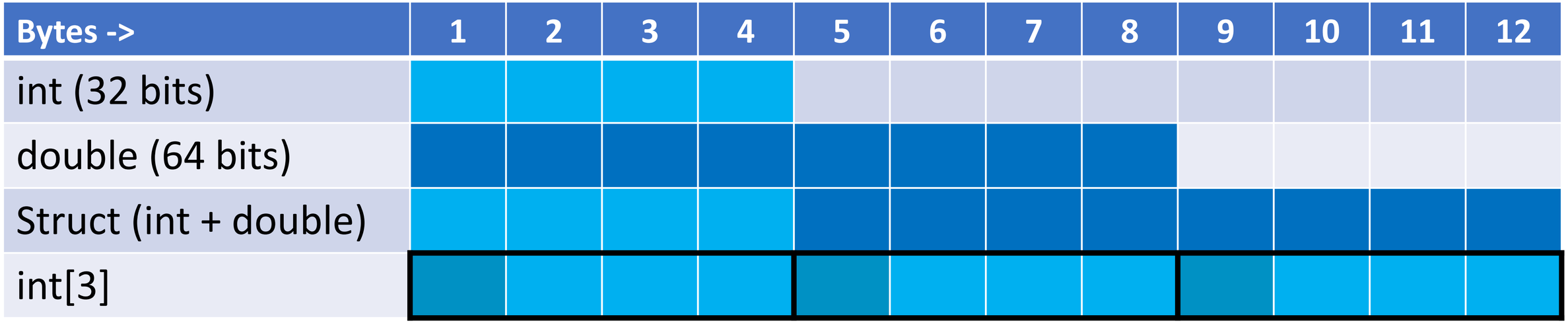
Typically:

- **char:** 8bit
- **int:** 32bit
- **long long int:** 64bit
- **float:** 32bit (7-8 digits of precision)
- **double:** 64bit (15-16 digits of precision)

DO NOT ASSUME memory size, use sizeof()

Why do pointers have types?

- **Recall: there is no type “pointer”.**
- We define “**int** * ptr;” or “**double** * ptr;”
- If we want to access a variable at the address given, the computer needs to know how much to grab



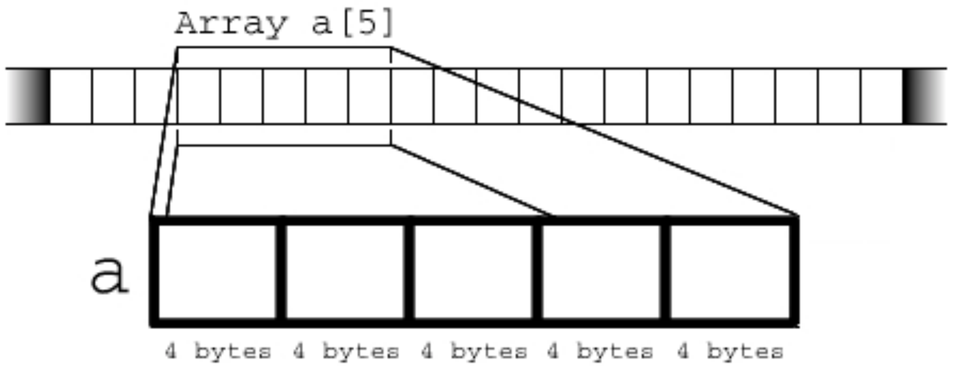
What happens when you access an array?

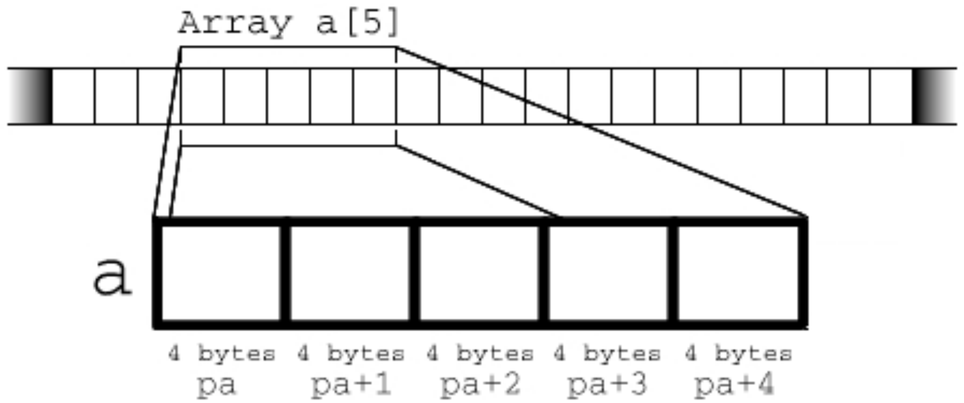
- “int * x = new int [3]”
  - allocates memory for 3 ints
  - binds the pointer “x” to the address of “x[0]”
- What does “x[i]” actually do?
  - accesses the address of “x[0]”
  - steps forward “i * sizeof(int)” bits in memory (e.g., performs pointer arithmetic)
  - returns the value at that memory address.

Bytes ->	1	2	3	4	5	6	7	8	9	10	11	12
int[3]	x[0]				x[1]				x[2]			

Pointers arithmetic and indexing, cont.

- C knows that integers are 4 bytes each. “(a + 1)” returns the memory address *one integer over* from the first element of “a”.





- Note also that arrays defined this way in C are *contiguous in memory*, e.g., **elements that are next to each other in the array are next to each other in memory**.

- Why is memory layout important? Efficiency!

http://www.rc.usf.edu/tutorials/classes/tutorial/c_intro/chapter7.html

Exercises: pointers and arithmetic

- You can create an array in C via “int x[4] = {1, 2, 3, 4};”. Note that “x[0], …, x[3]” accesses the entries of the array.
  - What will accessing “x[4]” return?
  - What about “*x”?
  - What about “*(x+2)”?
  - What about “*(&x[3]-2)”?

Matrix representations and layout in memory

From matrices to arrays

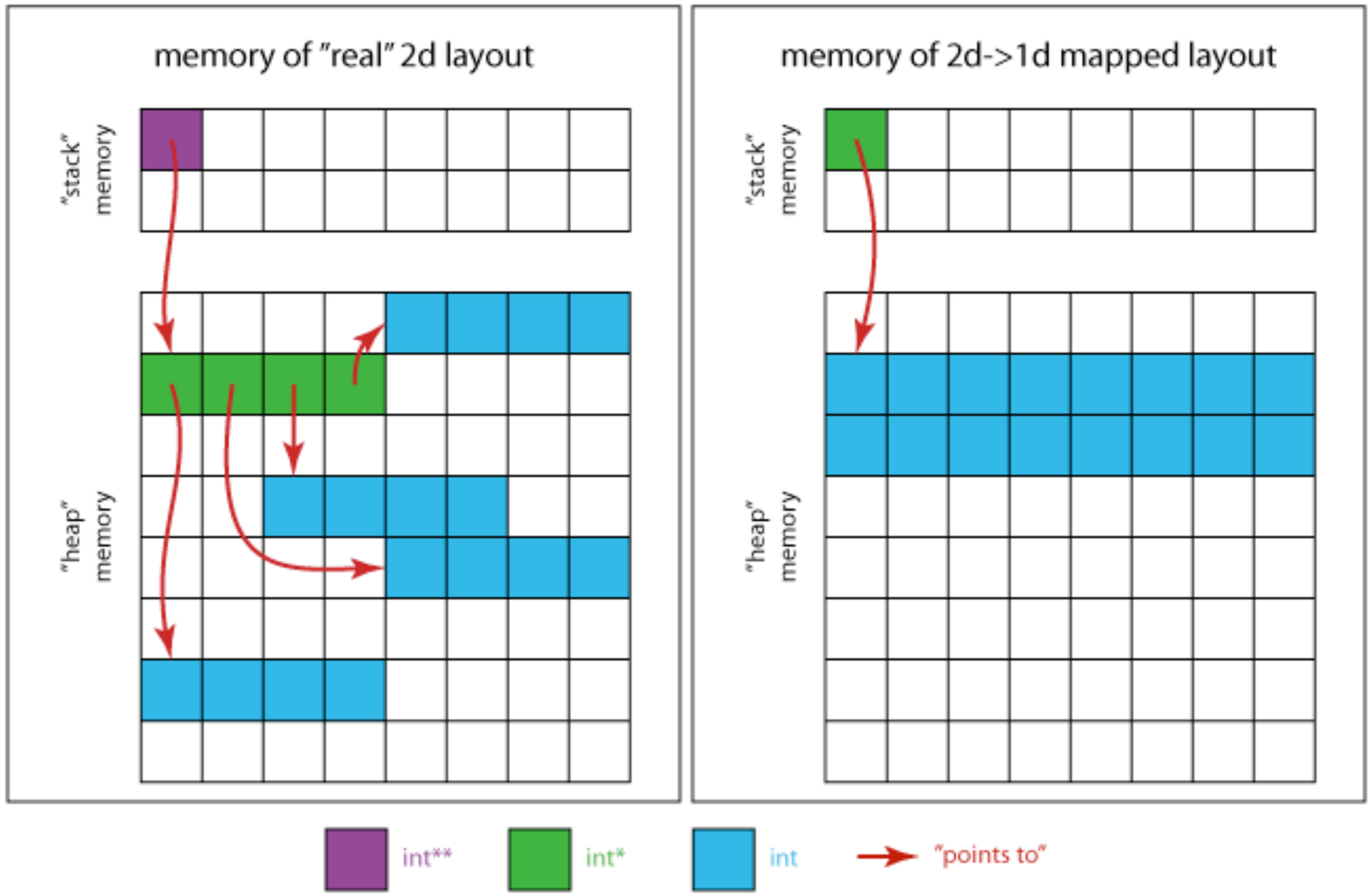
- Arrays include vectors, matrices, tensors, …
- Typically represents a list of numbers of one type (e.g., an array typically doesn't have both 32-bit and 64-bit numbers)
- What's the difference between arrays and other containers like Python Dictionaries or C++ stl containers? **Speed.**
  - Arrays are very *low-overhead* containers, which make them important for efficient programs.

Low-overhead programming

- In general, we will work with raw pointer arrays since they have the lowest overhead.
- Reducing overhead makes it easier to observe in practice what we expect to happen in theory.
- **Demo**: summing up elements of a matrix represented as:
  - A raw pointer array
  - A vectors of vectors from the C++ standard library
- Which is faster? Do compiler optimization flags change this?

Why the difference (even after -O3 opt)?

- The two types of arrays have **different layouts in memory**.
- The vector of vectors layout is more general, but results in fragmented memory accesses.
- The contiguous array results in *structured* memory accesses.

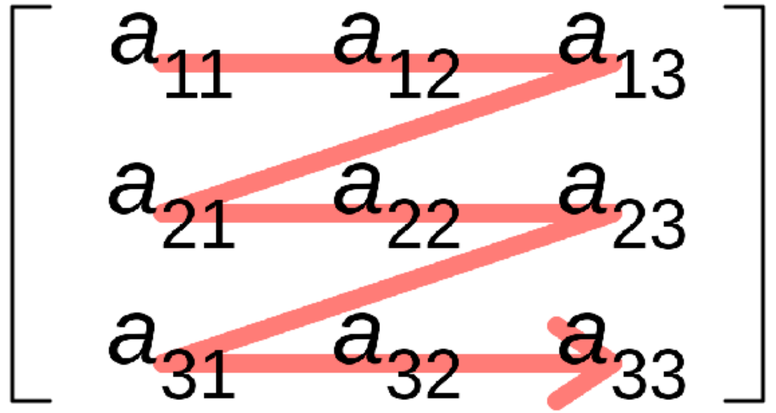


https://stackoverflow.com/questions/17259877/1d-or-2d-array-whats-faster/17260533#17260533

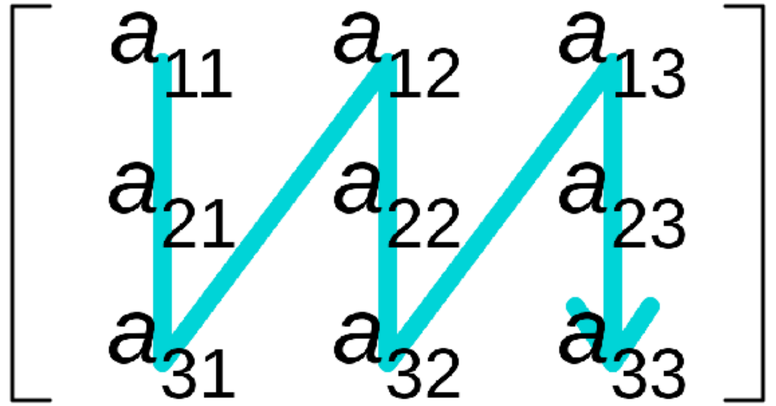
Matrix storage formats

- Contiguous storage formats are row or column major ordering
- Non-contiguous storage formats: elements are not guaranteed to be next to each other.
- Why is this important? It is faster to access an elements *neighbor* than a *random* entry.

Row-major order



Column-major order



https://en.wikipedia.org/wiki/Row-_and_column-major_order

What a matrix looks like in memory

```
int A[3][4];
```

i\j	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11

A[1][2] = 6

In memory, mD arrays are typically flat arrays

A	0	1	2	3	4	5	6	7	8	9	10	11
A[i][0]	0				1				2			
A[i_const][j]	0	1	2	3	0	1	2	3	0	1	2	3
A[i_flat]	0	1	2	3	4	5	6	7	8	9	10	11

```
1D: A[i]       = *(A + i)
2D: A[i][j]    = *(A + n2*i + j)
    A[0][i_flat] = *(A + i_flat)
```

How to represent a matrix: format 1

Format 1: store a matrix as a single “double *” vector (row or column major) and manually index (contiguous storage)

- Advantages: simple, contiguous storage, easy to change between row and column major
- Disadvantages: annoying to manually index.

```
// Approach 1: flat array, column major
double * A;
A = malloc(sizeof(double) * m * n);
for (int j = 0; j < n; ++j){
  for (int i = 0; i < m; ++i){
    A[i + j * n] = (double) (i+j);
  }
}
```

How to represent a matrix: format 2

Format 2: store a matrix as a “double **” array, dynamically allocate memory for each row.

- Advantages: convenient, can use multi-dimensional indexing
- Disadvantages: non-contiguous memory

```
// Approach 2: array of arrays (row major)
double ** B;
B = malloc(sizeof(double*) * m);
for (int i = 0; i < m; ++i){
  B[i] = (double*) malloc(sizeof(double) * n);
  for (int j = 0; j < n; ++j){
    B[i][j] = (double) (i+j);
  }
}
```

How to represent a matrix: format 3

- Format 3: store a matrix as a “double **” array, but allocate one single contiguous chunk of memory for the entire array.
- Use pointer arithmetic to assign the pointer location for C[i]

```
// Approach 3: array of arrays (row major)
double ** C = malloc(sizeof(double*) * m);

// allocate a vector of contiguous memory
C[0] = malloc(sizeof(double) * m * n);
for (int i = 1; i < m; ++i){
  C[i] = C[0] + i * n;
}
```

Matrices and dynamic memory allocation

- The important part is the underlying matrix representation, the memory layout, and how you access different entries.
  - Whether it’s represented as a 1D or 2D array is mostly syntactic sugar (for HPC at least).
- For this course, we will mostly work with matrices as flat arrays (vectors) and deal with indexing manually (Format 1)

Exercise: matrix sums

- Write two functions to sum entries of a square matrix, looping over rows then columns & columns then rows
  - For this demo, you can download and run matsum.cpp on Canvas
- Time your implementation (use -O3) for matrix sizes n = 32, 64, …, 8192.
- Which version is faster (and by how much)? What other trends do you observe?

Example for a row major storage format

```
double val = 0.0;
for (int i = 0; i < n; ++i)
{
  for (int j = 0; j < n; ++j)
  {
    val += A[j + i * n];
  }
}
return val;
```

Loop over rows, then columns

```
double val = 0.0;
for (int j = 0; j < n; ++j)
{
  for (int i = 0; i < n; ++i)
  {
    val += A[j + i * n];
  }
}
return val;
```

Loop over columns, then rows

What trends do we notice?

- Inner loop over rows is faster (contiguous memory access)
- The -O3 makes a big difference!
- No difference in runtime for small matrices.
  - What constitutes a “small” matrix?
- As the matrix gets larger, the ratio of runtime seems to approach a constant. What is the constant?

Understanding these results

- To understand these results, we have to learn a little bit about CPU hardware - specifically, the CPU memory model.
- Roughly speaking, CPU computing power has increased so quickly (due to Moore's Law, Dennard scaling, etc) that it is *much* faster to perform an operation than to retrieve data from memory (e.g., RAM or hard disk).
- CPU designers introduced a *memory hierarchy* to bridge this gap; taking advantage of this results in faster CPU codes.

CPU memory hierarchy

Why focus on a single CPU?

Isn’t parallelism more important?

- Parallelism adds more resources; serial (single CPU) optimizations make more efficient use of existing resources.
  - Inefficient single-core usage makes code look artificially scalable…
- Most serial codes leave a lot on the table! Typical codes achieve between 10-20% of the peak performance.
  - Peak performance tends to be “gamed” and very hard to achieve; IMO about 50% of peak performance is already pretty good.
- Introduces **theoretical models** which can help us understand multi-core CPU and GPU performance as well.

Memory bandwidth vs latency

Bandwidth

≈ data throughput (bits/second)

Low Bandwidth

High Bandwidth

Latency

≈ delay due data travel time (ms)

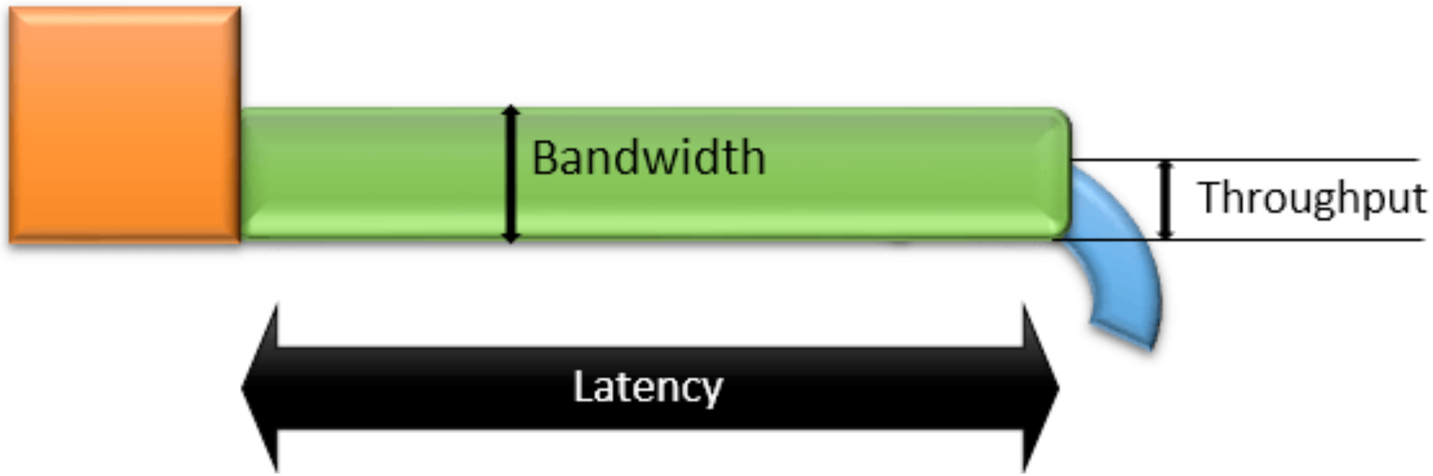
Low Latency

High Latency

Image: Katie Hempenius

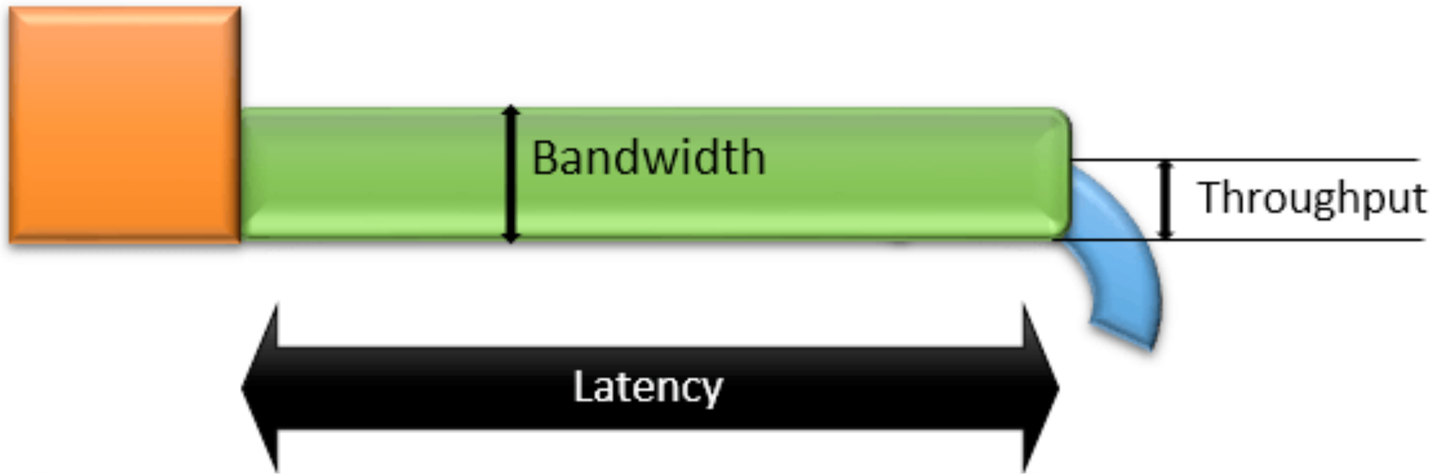
What does efficient memory use look like?

- Latency is a constant cost associated with transferring data from RAM to a CPU
- Bandwidth is the maximum rate at which data can be transferred from RAM to CPU.



What does efficient memory use look like?

- Latency is a constant cost associated with transferring data from RAM to a CPU
- Bandwidth is the maximum rate at which data can be transferred from RAM to CPU.

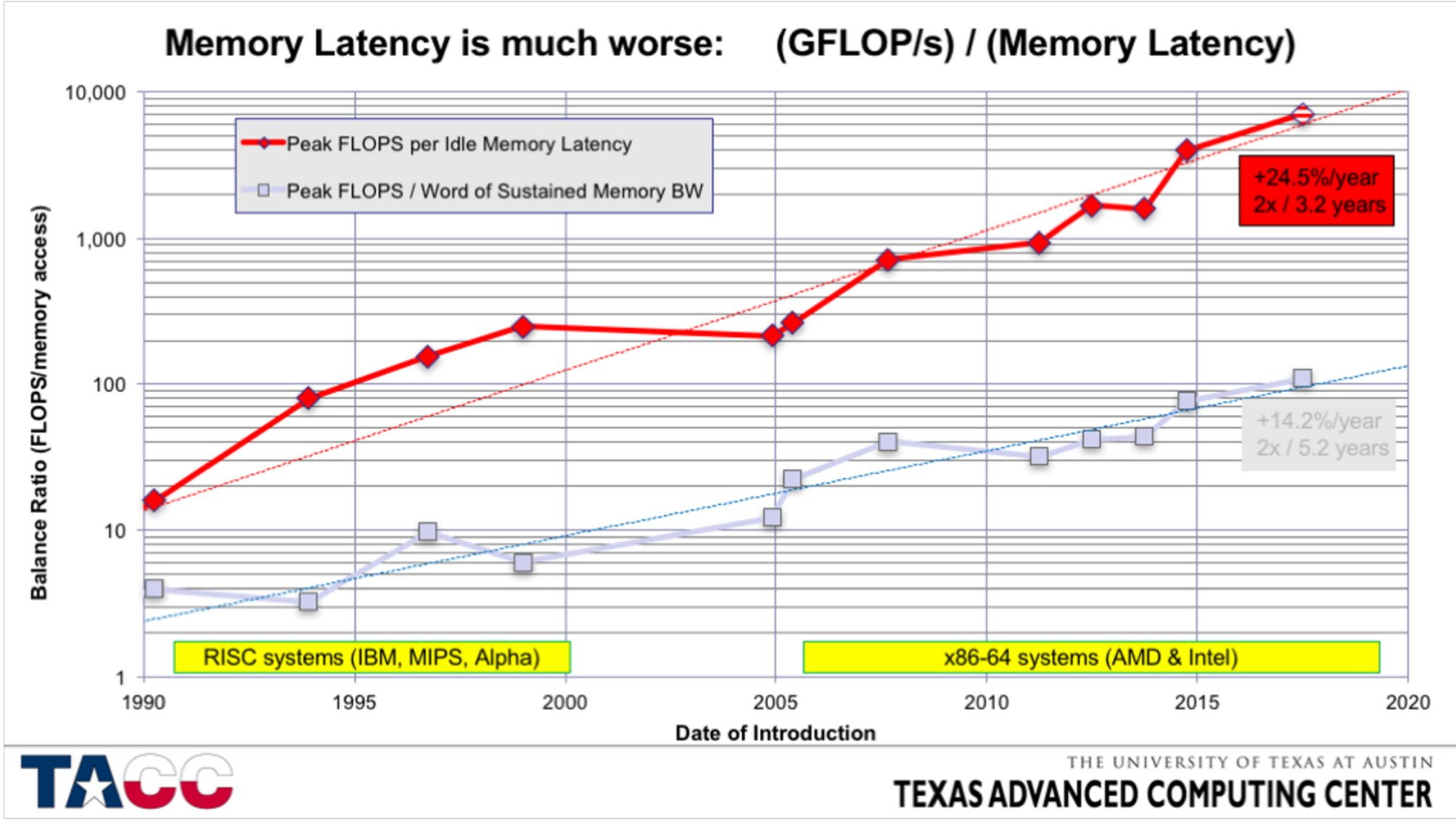


- Ideally you want to transfer lots of memory to amortize latency costs.
- Worst case: intermittently access small amounts of memory.

Memory is slow relative to FLOPS

FLOPS = floating point operations

Y-axis = cost of memory bandwidth and memory latency relative to arithmetic operations.



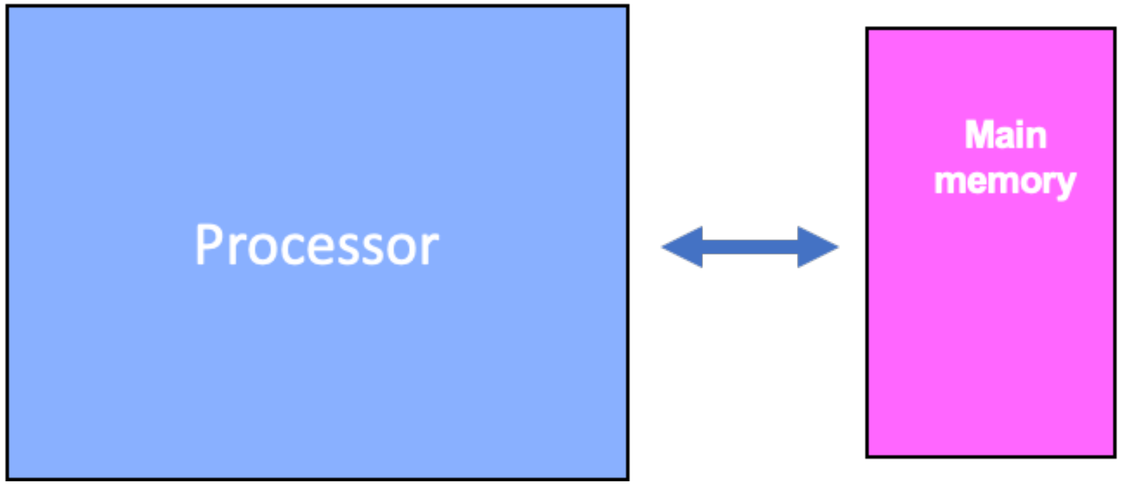
TACC

THE UNIVERSITY OF TEXAS AT AUSTIN
TEXAS ADVANCED COMPUTING CENTER

Focusing on memory

Cost of accessing memory is often overlooked

- Simple cost model: memory accesses (reading/writing to RAM) has two costs:
  - Latency: cost α to load/store a single "word" of memory
  - Bandwidth: average rate (bytes/sec) to load/store a large chunk of memory. We will use β to denote inverse time/byte.
- Cost = $\alpha + \beta * n$, where n is the number of "words" of memory transferred.



How to address memory bottlenecks?

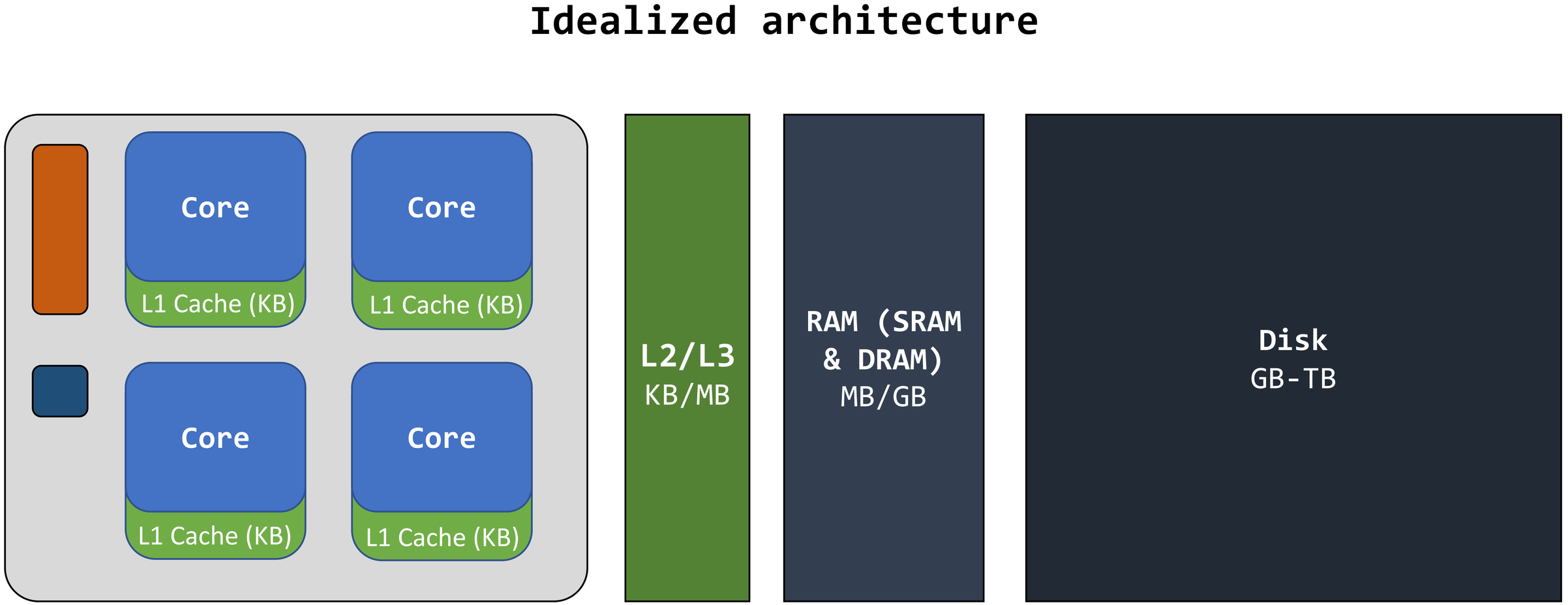
Accessing memory can dominate a program's runtime

- Can't we just make the memory faster?
  - Makes the computer more expensive, challenging to do in practice due to physical limitations and tighter manufacturing tolerances.
- Alternative: design memory systems around "common access patterns"
  - Prioritize commonly accessed memory to balance performance and cost.
  - Introduce intermediate levels of "staging" memory to offset high cost of accessing RAM.

What are common memory access patterns?

- Some common algorithmic operations: looping over array entries, performing some iteration over a single variable (e.g., accumulation).
- **Spatial locality:** after accessing an element of an array, often you'll want to access a *nearby* element next.
- **Temporal locality:** after accessing an element of an array, often you'll want to use it again for another operation.
- Idea of modern memory hierarchies: design a system which is faster for highly spatially and temporally local operations.

Computer architecture: caches

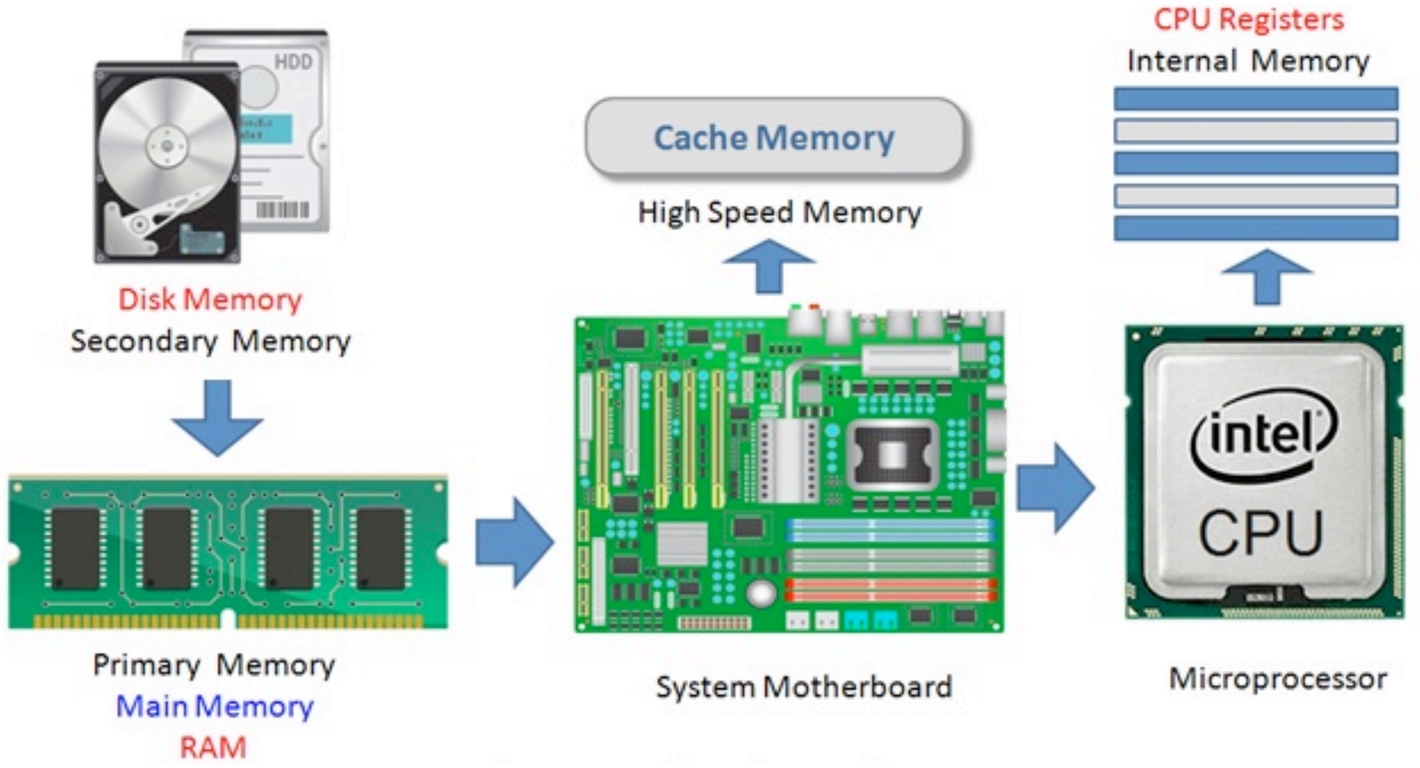


What is a memory hierarchy?

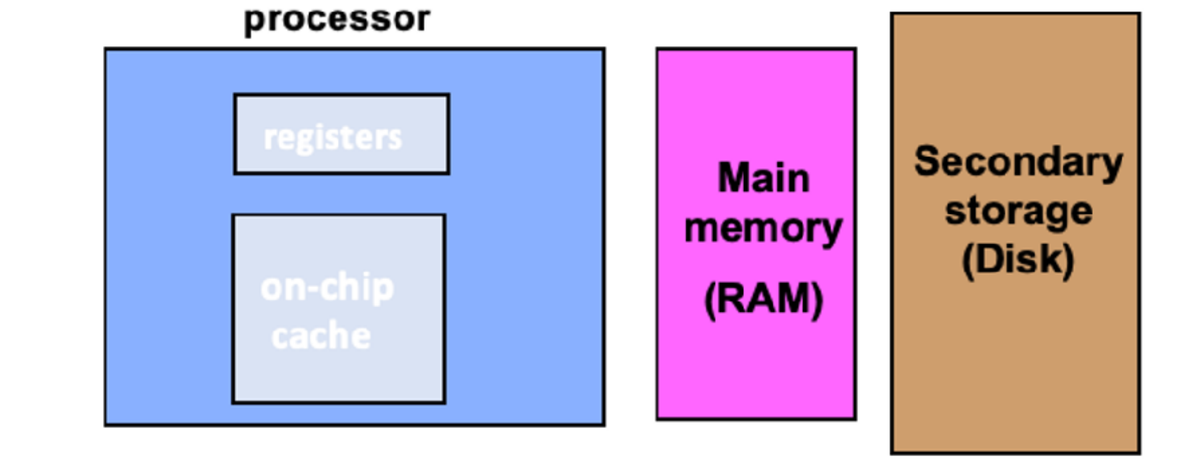
- Modern computers use a memory hierarchy to balance performance and cost
- They typically have access to multiple levels of memory:
  - From fastest to slowest: register memory, L1, L2, maybe L3 caches, RAM, hard disk, networked or cloud storage, …
- The closer to the arithmetic logic units (ALUs, the part of the CPU that is responsible for performing arithmetic operations), the faster the memory is.
  - Tradeoff: the faster the memory is, typically the smaller it is as well.

What different types of memory are there?

- Register memory: on each core, specialized functionality, fastest memory, very small (~64 bytes).
- Cache hierarchy: level 1 cache (L1), level 2 cache (L2), L3, L4.
  - L1 and L2 tend to be fastest
- RAM: where program memory lives (stack and heap); O(10) to O(100) GBs
- Disk: long term memory; 100s of GBs to a few TBs



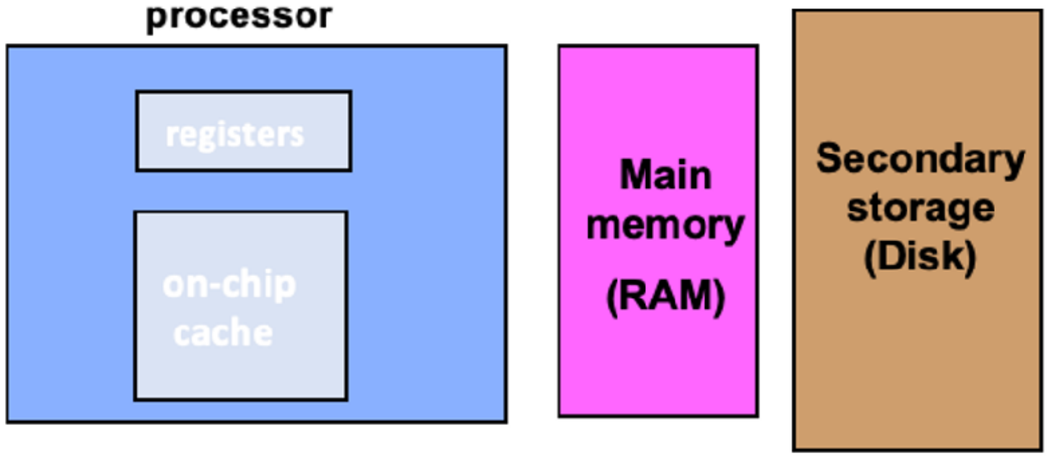
Memory hierarchy: speed and size



Speed	1ns	10-100ns	10ms
Size	KB	MB-GB	TB

Image adapted from Jim Demmel

Memory hierarchy: speed and size

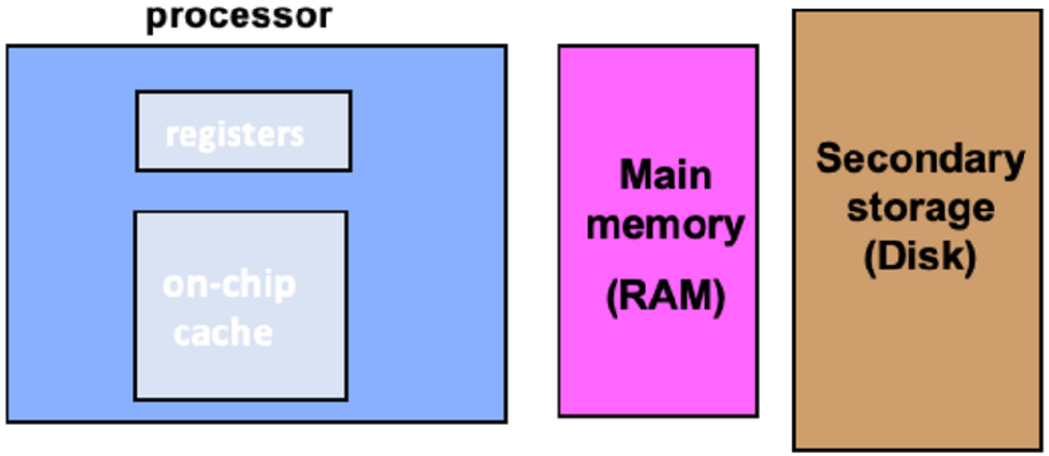


Speed	1ns	10-100ns	10ms
Size	KB	MB-GB	TB

Pick up a pencil from your desk

Image adapted from Jim Demmel

Memory hierarchy: speed and size



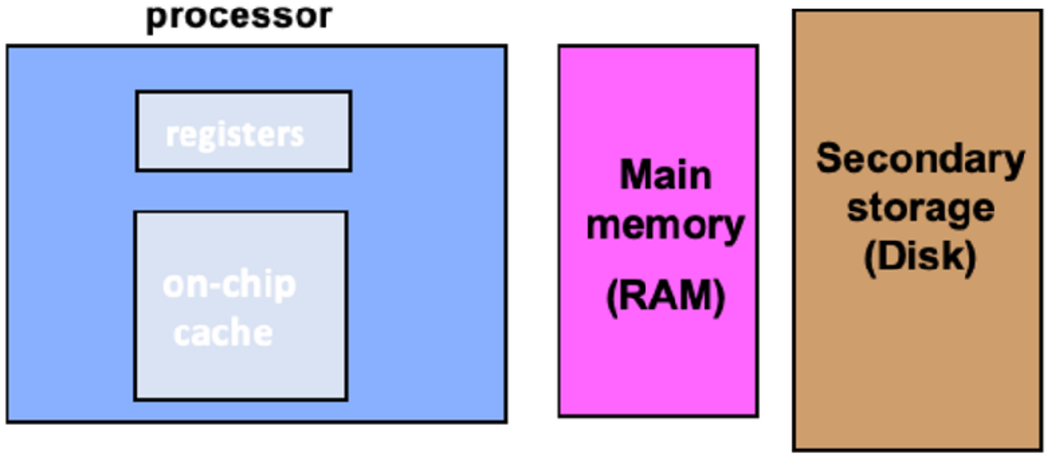
Speed	1ns	10-100ns	10ms
Size	KB	MB-GB	TB

Pick up a pencil from your desk

Grab a box of pencils from the drawer

Image adapted from Jim Demmel

Memory hierarchy: speed and size



Speed	1ns	10-100ns	10ms
Size	KB	MB-GB	TB

Pick up a pencil from your desk

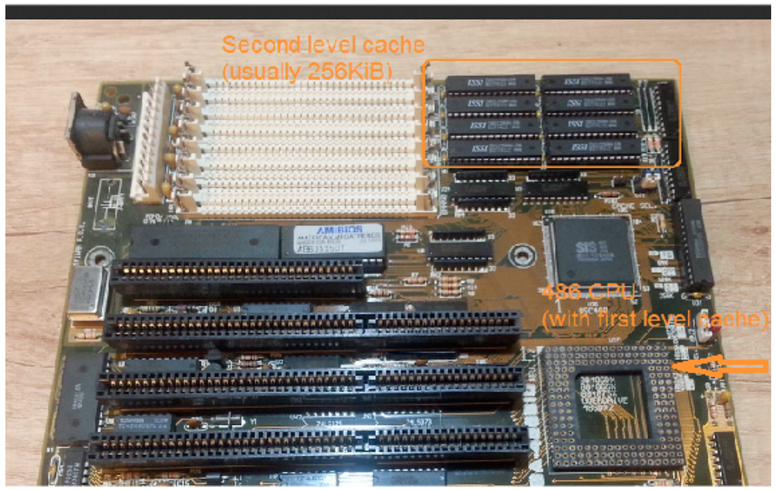
Grab a box of pencils from the drawer

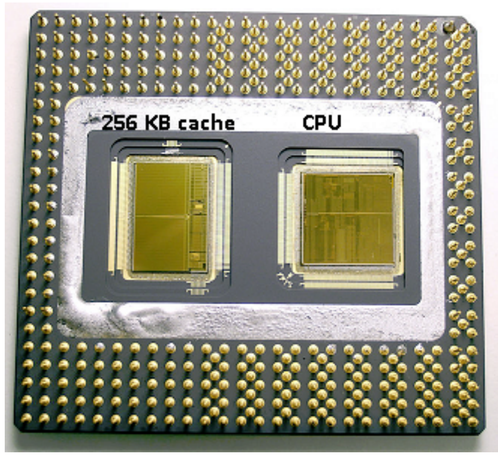
Buy a case of pencils from Costco

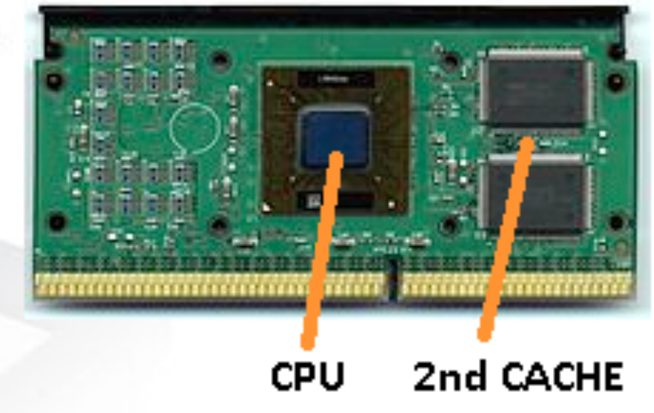
Image adapted from Jim Demmel

The cache hierarchy

- Cache hierarchy:
  - L1 cache: built into each core; couple hundred KB; ~100x faster than RAM
  - L2 cache: per CPU or per core; 100s of KB to MBs; ~25x faster than RAM
  - L3/L4 caches: shared between 2 or more cores. Hundreds of KB to dozens of MB. Often used as staging areas for L1 and L2 caches.
- How does a hierarchy of caches (potentially) improve performance?

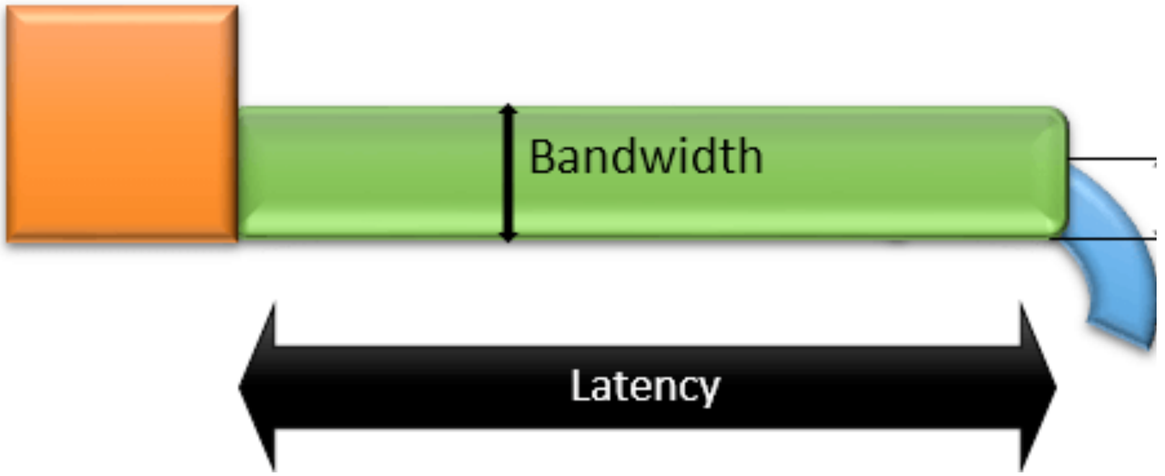






The cache hierarchy

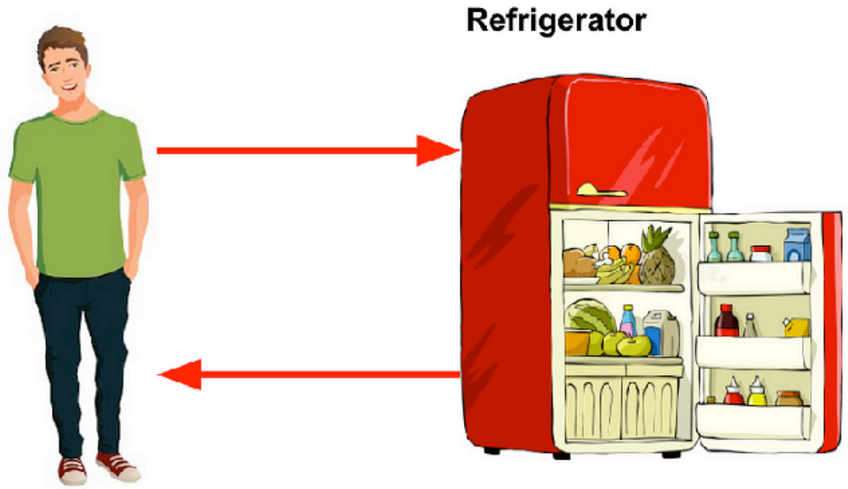
- Recall that, due to memory latency, you ideally want
  - A small number of memory accesses
  - A lot of data transferred during each memory access



- The cache system takes advantage of this: every time you access a memory address, the CPU actually pulls several additional entries *near* that memory address into cache (this is a “cache line”).
- Now, if you try to access a memory address that was in the cache line, you can access that from faster L1/L2 cache instead of RAM!

Cache hits and misses

A cache miss is similar to when you have to go back to the store because you forgot to buy something you needed.



- Is the variable we need next in cache or not?
  - If a variable is in cache, it's a "cache hit" and is fast to access.
  - If a variable is not in cache, it's called a "cache miss", and we have to read from a slower cache or RAM memory.
- Cache misses are expensive: reading from a lower level of cache or RAM takes 10-100x longer than reading from L1 cache.

Back of the envelope calculations

- Some numbers:
  - Register/L1 cache are the fastest, ~4 CPU cycles (~1 ns to access).
  - L2 cache is slower, ~10 CPU cycles (~2.5 ns to access).
  - L3 cache is slower still, ~40 CPU cycles (~10 ns to access).
- Suppose we access data from memory 100 times.
  - If all 100 accesses are L1 cache hits, it takes about ~100 ns
  - If there's *a single* L1/L2 cache miss, then there's 99 + 10 = ~109 ns, or ~10% slowdown due to a single entry.

Why contiguous memory access is fast

Bytes ->	1	2	3	4	5	6	7	8	9	10	11	12
x	x[0]				x[1]				x[2]			

Accessing x[0] should pull x[1], x[2] into cache as well.

- Caching grabs a contiguous chunk of memory near an array entry.
- Walking along the fastest dimension first will yield more cache hits.
- Cache-aware programming is the study of algorithms designed to exploit the cache hierarchy
  - Example: binary trees vs B-trees

Contiguous memory access for a vector

Store a matrix as a single “double *” vector (row or column major) and manually index (contiguous storage)

- Simple, easy to see if memory accesses are contiguous.
- Can extend to higher dimensional arrays too

```
// Approach 1: flat array, column major
double * A;
A = malloc(sizeof(double) * m * n);
for (int j = 0; j < n; ++j){
  for (int i = 0; i < m; ++i){
    A[i + j * n] = (double) (i+j);
  }
}
```

```
A[i + j * n + k * n^2]
```

Contiguous memory access for matrices

- You can store a matrix as a “double **” array, but allocate one single contiguous chunk of memory for the entire array.
- Use pointer arithmetic to assign the pointer location for C[I]

```
// Approach 3: array of arrays (row major)
double ** C = malloc(sizeof(double*) * m);

// allocate a vector of contiguous memory
C[0] = malloc(sizeof(double) * m * n);
for (int i = 1; i < m; ++i){
  C[i] = C[0] + i * n;
}
```

Cache behavior can be pretty complicated

- Example: eventually a cache fills up, then the cache has to *evict* some line of memory to make room for new data.
  - There are rules for choosing memory to evict from a cache.
  - There are “victim caches” (often one higher level of cache) which pick up evicted cache lines in case they’re needed again.
- Caching *can* drastically improve computer/program performance, but:
  - it depends on how the program is structured
  - It can be hard to predict (often performance can be better than you expect).

Do caches explain the behavior we observe?

- Inner loop over is faster (contiguous memory access)
- We expect more cache hits and thus more fast memory accesses with the first example.
- The second example would generate more cache misses and result in more slow memory accesses.

```
for (int i = 0; i < n; ++i)
{
  for (int j = 0; j < n; ++j)
  {
    val += A[j + i * n];
  }
}
```

```
for (int j = 0; j < n; ++j)
{
  for (int i = 0; i < n; ++i)
  {
    val += A[j + i * n];
  }
}
```

Do caches explain the behavior we observe?

- L1 cache: **100x** faster than RAM. Typically 16KB - 128 KB per core
- L2 cache: **25x** faster than RAM. Typically 128KB – 1MB per core
- 4x difference in L1 and L2 cache, about ~4x difference in runtime?
- If we have an n-by-n matrix of doubles, this is 8n^2 bytes. What size matrix fits into L1 cache?

```
g++ -std=c++11 -O3 matsum.cpp
```

```
(base) XXH62CK9GR:code jchan985$ ./a.out 512
Matrix size n = 512
ij elapsed time = 0.000272084
ji elapsed time = 0.000291209
(base) XXH62CK9GR:code jchan985$ ./a.out 1024
Matrix size n = 1024
ij elapsed time = 0.000903875
ji elapsed time = 0.00126129
(base) XXH62CK9GR:code jchan985$ ./a.out 2048
Matrix size n = 2048
ij elapsed time = 0.00426775
ji elapsed time = 0.0187515
```

Linear algebra applications

Taking advantage of cache in practice

- Matrix accesses should respect memory layout.
  - If it's column major, loop over columns, then rows.
  - If it's row major, loop over rows, then columns.
- What about matrix-vector and matrix-matrix multiplication?
- Are additional improvements possible?

```
for (int j = 0; j < n; ++j){
    for (int i = 0; i < n; ++i){
        b[i] += A[i + j * n] * x[j]
    }
}
```

Matrix-vector multiplication b += A * x

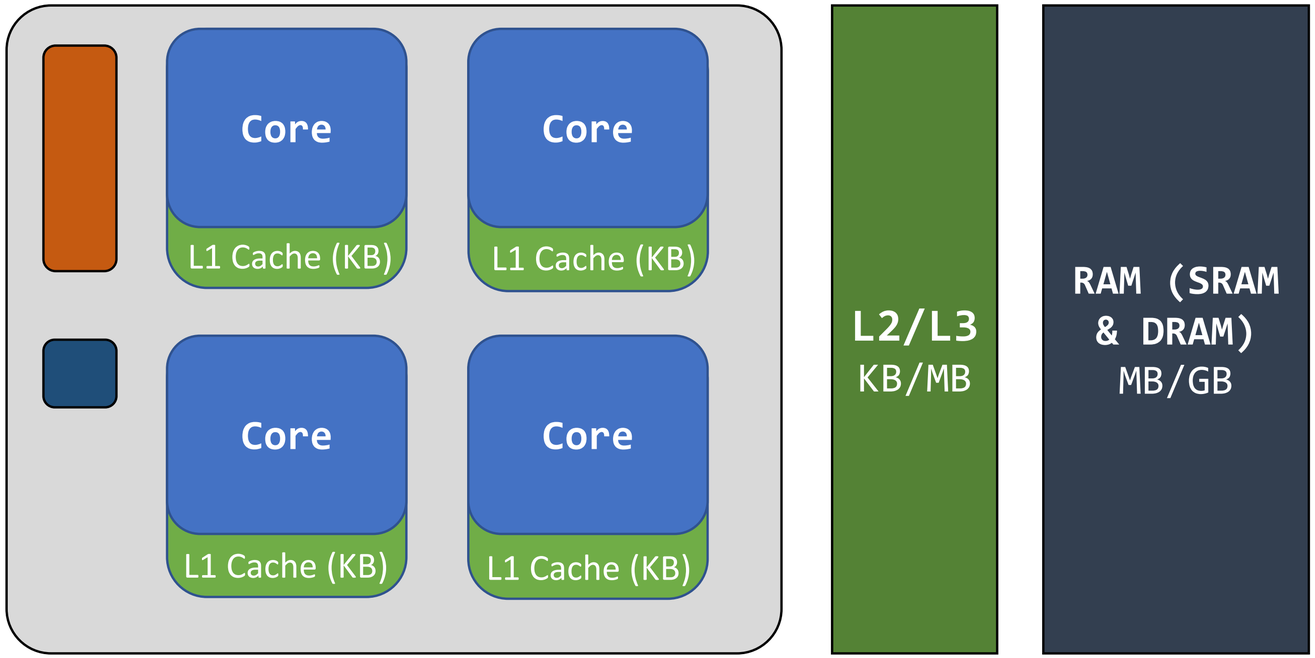
```
for (int i = 0; i < n; ++i){
    for (int j = 0; j < n; ++j){
        b[i] += A[j + i * n] * x[j]
    }
}
```

A basic theoretical framework

- We’ve established that contiguous memory layouts matter due to caches and the idea of a memory hierarchy.
- Can we use this to build a simple theoretical framework to analyze the efficiency of different functions?
  - Some codes will be much more cache-efficient than others. Are there factors that predict performance?

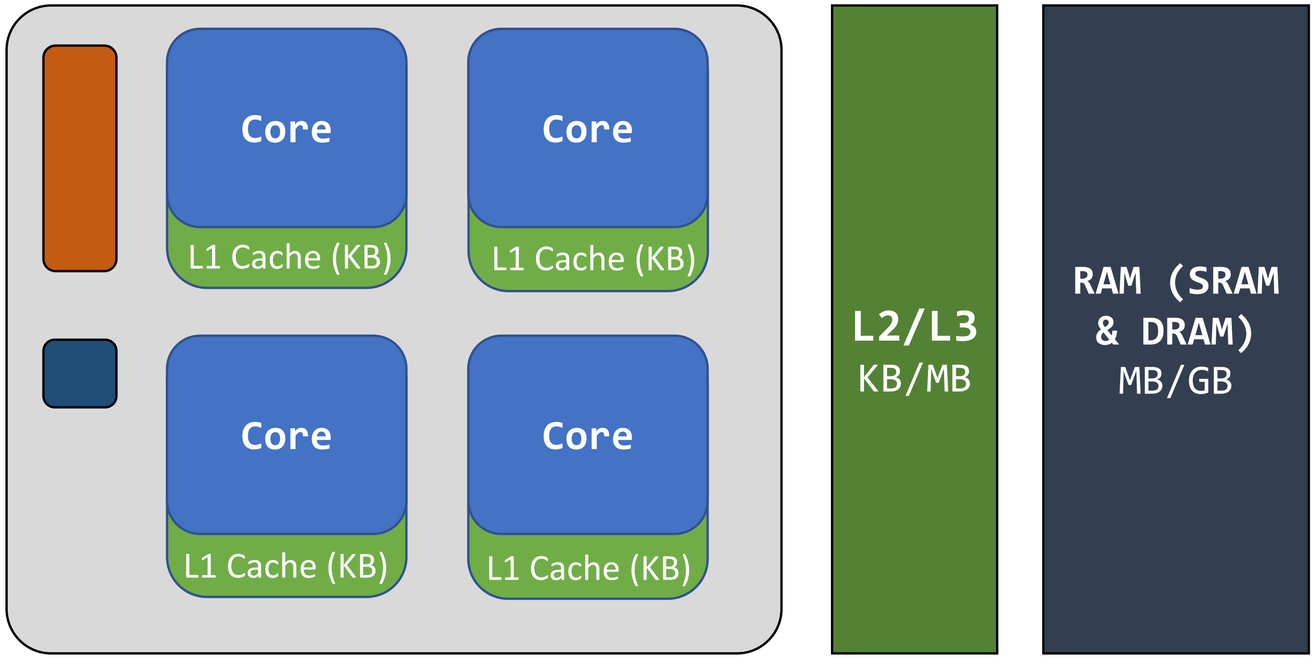
Building a simplified model of memory

- Assume only two levels of memory: fast memory and slow memory.
  - Ignore intermediate levels of cache and hard disk.
- Assume all data starts in slow memory (e.g., RAM).
- Account for arithmetic operations and slow memory movement



Building a simplified model of memory

- Assume only two levels of memory: fast memory and slow memory.
  - Ignore intermediate levels of cache and hard disk.
- Assume all data starts in slow memory (e.g., RAM).
- Account for arithmetic operations and slow memory movement



m = number of “words” of memory moved between fast and slow memory

t_m = time per slow memory operation (inverse bandwidth in best case)

f = number of arithmetic operations

t_f = time per arithmetic operation (which is << t_m)

Building a simplified model of memory, cont.

m = number of “words” of memory moved between fast and slow memory
t_m = time per slow memory operation (inverse bandwidth in best case)
f = number of arithmetic operations
t_f = time per arithmetic operation (which is << t_m)

- Assume fast memory accesses are *free* (not true, but simplifies things).
- Total runtime: computational + memory costs $(f \times t_f) + (m \times t_m)$
- Factor out computational cost: $(f \times t_f)\left(1 + \frac{m}{f} \times \frac{t_m}{t_f}\right)$, note that $\frac{t_m}{t_f} \gg 1$.

Building a simplified model of memory, cont.

m = number of “words” of memory moved between fast and slow memory
t_m = time per slow memory operation (inverse bandwidth in best case)
f = number of arithmetic operations
t_f = time per arithmetic operation (which is << t_m)

- Factor out computational cost: $(f \times t_f)\left(1 + \frac{m}{f} \times \frac{t_m}{t_f}\right)$, note that $\frac{t_m}{t_f} \gg 1$.
- Define the ***computational intensity (CI)*** as $CI = \frac{f}{m}$.
- If CI is large, $\frac{m}{f} = \frac{1}{CI}$ is small, and we’re closer to the minimum time $(f \times t_f)$.

A reminder on simplifying assumptions

- Constant “peak” computation rate. In reality, the computation rate can vary depending on queued tasks, background computations
- Cost of fast memory access is zero (probably only true for register memory, not L1 cache or above)
- Simplified model of how data is read: assumes there are no extra cache evictions when reading data into fast memory
- No accounting for memory *latency* at all!
  - Less of an issue when we have contiguous memory accesses.

Examples of CI: matrix sum

- Consider the matrix sum demo.
  - Only arithmetic operation is n^2 additions
  - Requires n^2 memory reads of A.
    - “val” isn’t counted since it’s small enough to be in fast memory.
- The computational intensity is just CI = 1, which is pretty small

```
double val = 0.0;
for (int i = 0; i < n; ++i)
{
  for (int j = 0; j < n; ++j)
  {
    val += A[j + i * n];
  }
}
return val;
```

Examples of CI: matrix-vector product

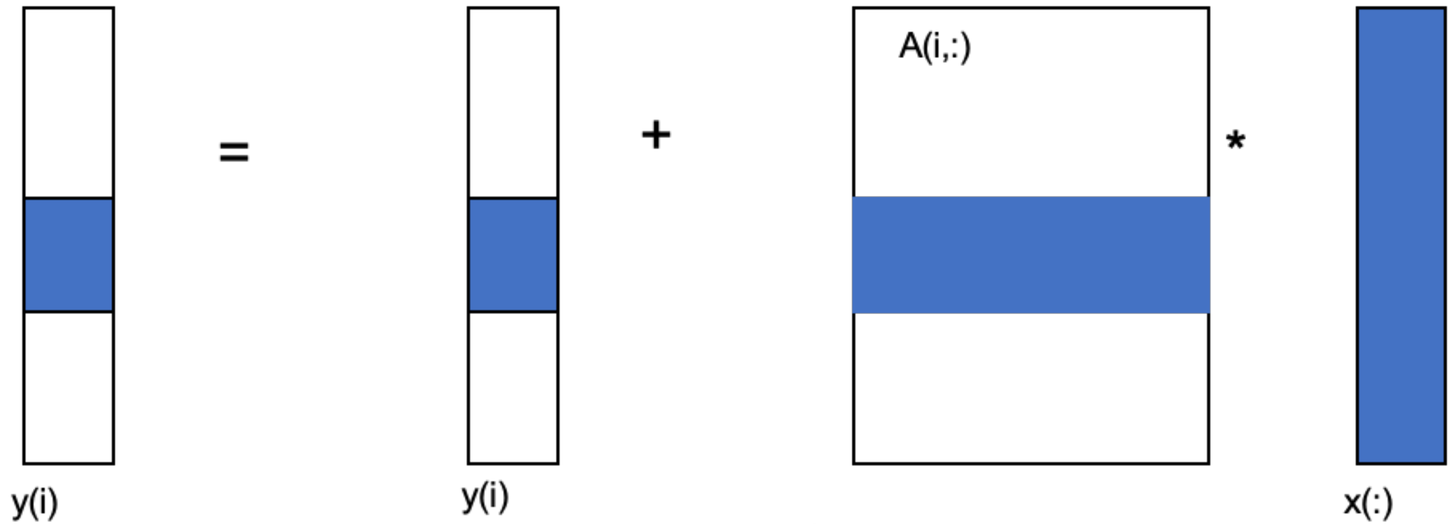


Image from Demmel

- Consider a matrix-vector product $y = Ax$.
- Assume row major storage, and that matrices are large enough so that the entire matrix A does not fit into cache.

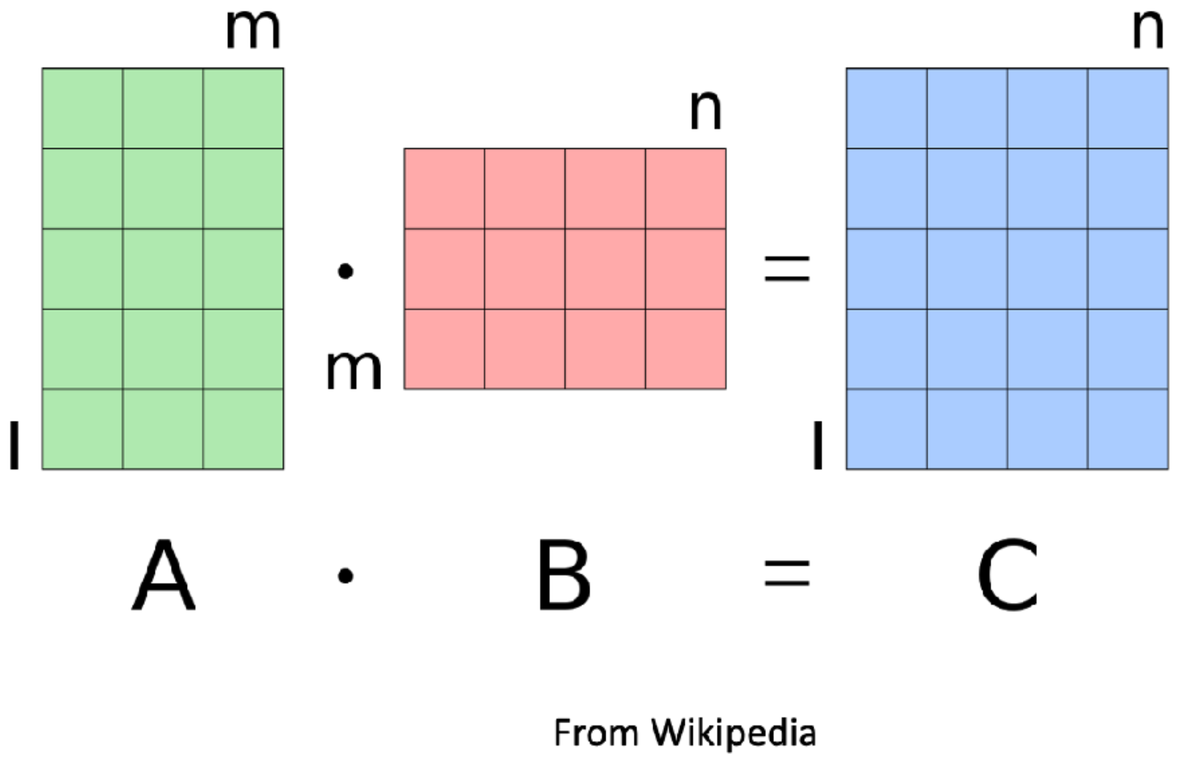
Examples of CI: matrix-vector product, cont.

```
for i = 1:n
    for j = 1:n
        y(i) = y(i) + A(i,j)*x(j)
```

- Since A has n times more entries than x, y, memory costs are dominated by accesses of A_{ij} as opposed to vector entries.
- 2 operations per loop iteration = $2n^2$ arithmetic operations total
- Read A, x, y and *write* y to slow memory. Total $3n + n^2$ memory reads
- CI is only ~2 as n increases!

What about matrix-matrix multiplication?

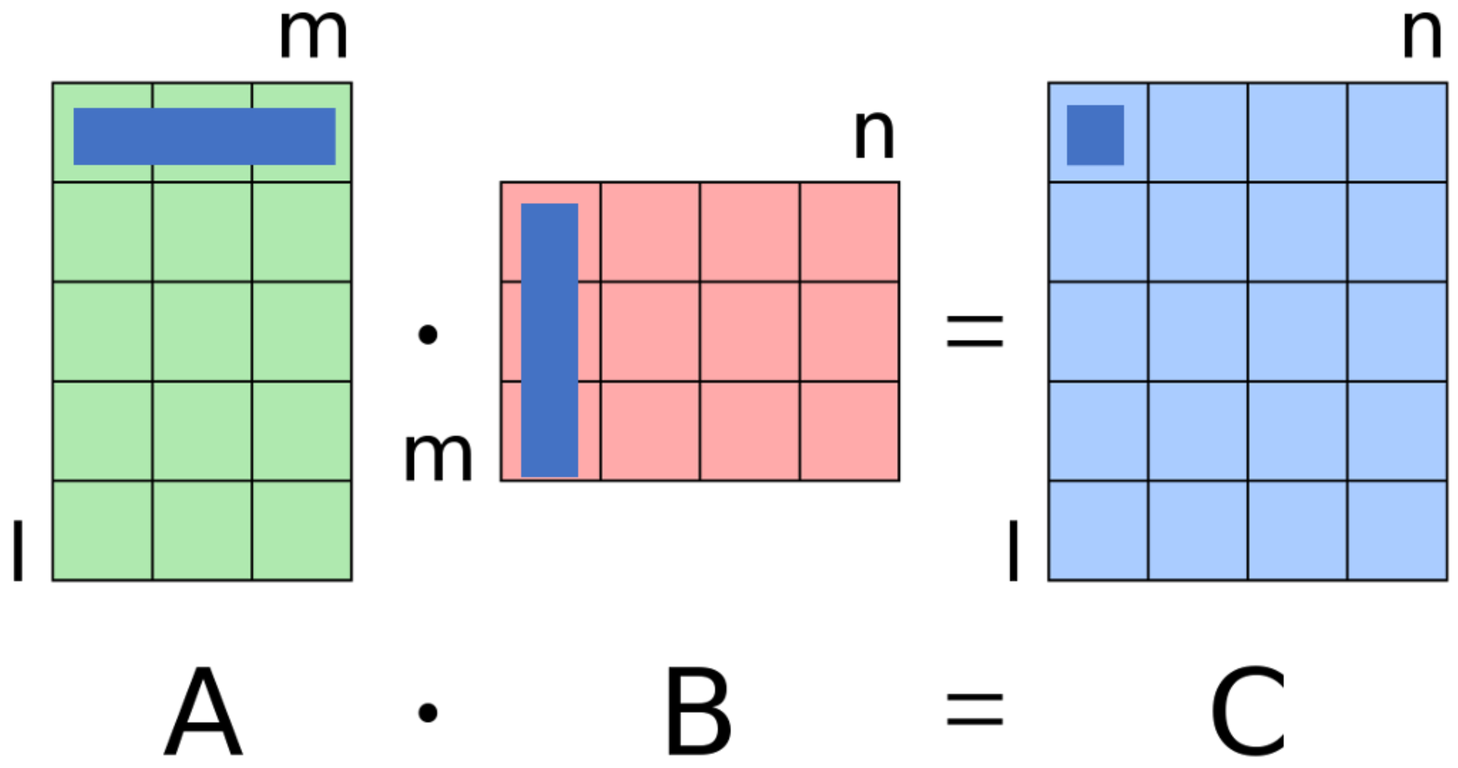
- Back of the envelope calculations (assuming a square n x n matrix)
  - Total $3n^2$ memory accesses
  - Total $2n^3$ arithmetic operations
  - CI could potentially be $O(n)$!
- In practice, we will achieve much lower CI, but the ceiling is higher than for matrix-vector multiplication.



From Wikipedia

The “usual” implementation

$$C_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}$$



Libraries provide a general version of matrix multiply
$C = aC + b(AB)$ where a, b are scalars

Demo

- Lets compare timings for a few examples (assume all matrices row major).
  - Matrix sum
  - Matrix-vector multiplication
  - Matrix-matrix multiplication
- What do we observe in terms of their timings? What should we expect?
- How can we make matrix multiplication more efficient?

Demo results

- Timings for n = 512, 1024
  - Matrix sum: 240, 906 microseconds
  - Matrix-vector multiplication: 279, 1328 microseconds
  - Matrix-matrix multiplication: 178520, 1.53499e+06
    - If we divide by n, these timings become ~348, ~1499
    - Matmul is actually less efficient than matvec, relatively speaking!
- Can improve matmul performance by storing B in row major format; timings scaled b n are ~203, ~941. A little better than matvec now.

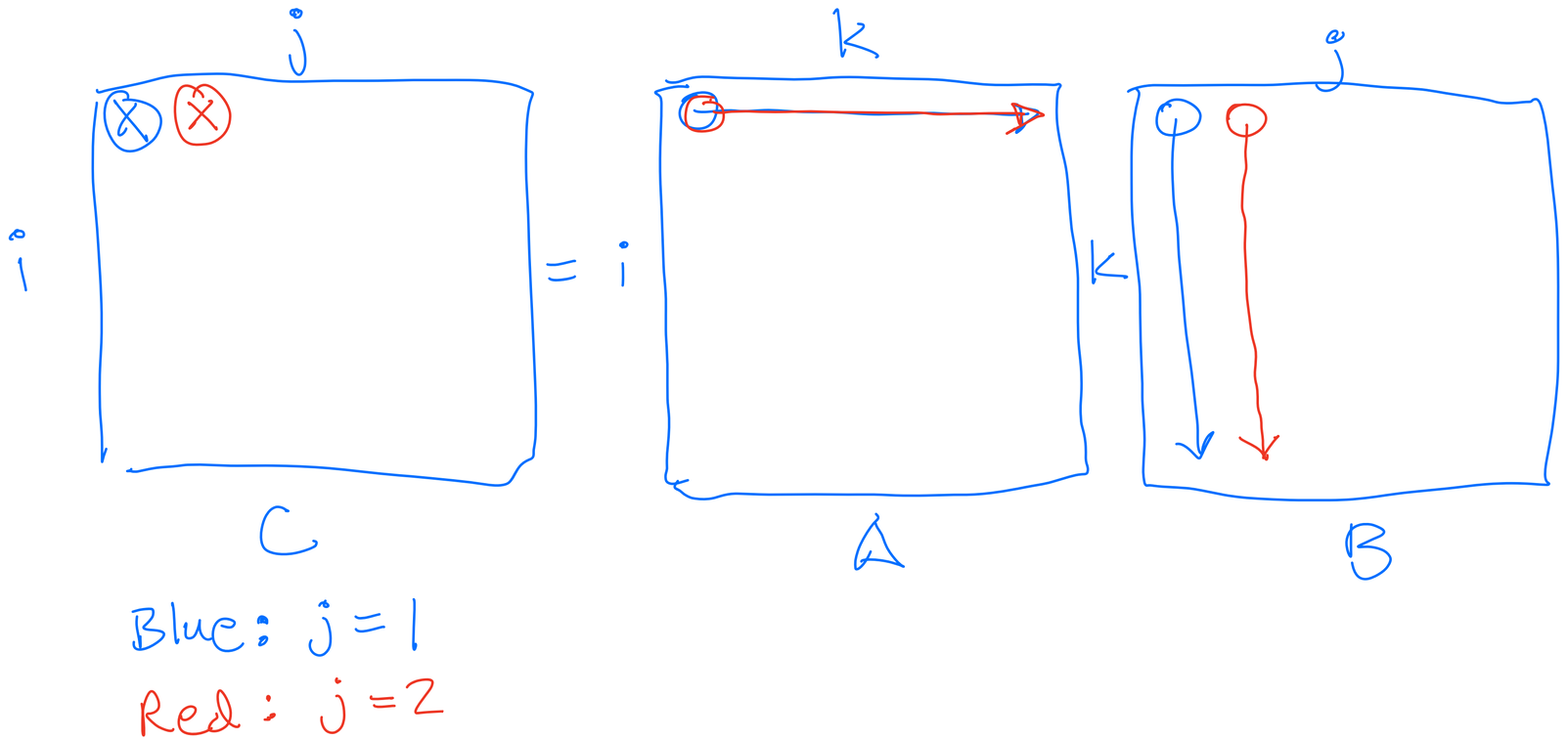
Analyzing matrix-matrix multiplication

```
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

- Assume matrices are square for now, assume B is column major so that we have contiguous memory accesses.
- What is the CI of the “usual” implementation?

The CI of the “usual” implementation

```
for i = 1 to n
     for j = 1 to n
          for k = 1 to n
C(i,j) + A(i,k) * B(k,j)
```



- Due to order of traversal, rows of C, A remain in fast memory as j changes.
- However, columns of B are moved into fast memory and evicted as j changes.
- Assume at most 3 matrix rows fit into cache; each column of B is read into cache and then evicted for each value of j (n extra reads of B).

The CI of the “usual” implementation

```
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

- $2n^3$ arithmetic operations, but $3n^2 + n^3$ memory reads
- CI is asymptotically still ~2 - the exact same cost as matrix-vector multiplication!

The CI of the “usual” implementation

```
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

2 operations each inner loop

- $2n^3$ arithmetic operations, but $3n^2 + n^3$ memory reads
- CI is asymptotically still ~2 - the exact same cost as matrix-vector multiplication!

The CI of the “usual” implementation

```
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

Read each entry of C twice, $2n^2$ memory reads

2 operations each inner loop

- $2n^3$ arithmetic operations, but $3n^2 + n^3$ memory reads
- CI is asymptotically still ~2 - the exact same cost as matrix-vector multiplication!

The CI of the “usual” implementation

```
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

Read each entry of A once, n^2 memory reads

Read each entry of C twice, $2n^2$ memory reads

2 operations each inner loop

- $2n^3$ arithmetic operations, but $3n^2 + n^3$ memory reads
- CI is asymptotically still ~2 - the exact same cost as matrix-vector multiplication!

The CI of the “usual” implementation

```
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

Read each column of B n times - n^3 memory reads.

Read each entry of A once, n^2 memory reads

Read each entry of C twice, $2n^2$ memory reads

2 operations each inner loop

- $2n^3$ arithmetic operations, but $3n^2 + n^3$ memory reads
- CI is asymptotically still ~2 - the exact same cost as matrix-vector multiplication!

Memory bottleneck for matrix multiplication

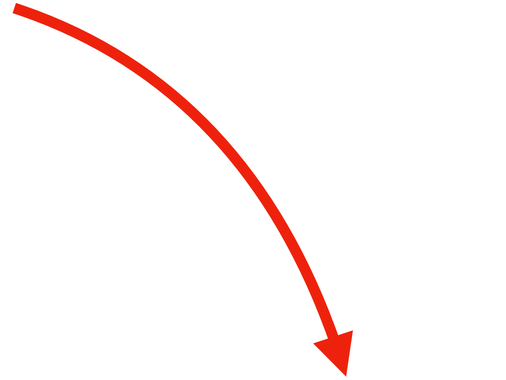
```
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

- The CI ~ 2 is because we have n^3 operations but also n^3 memory reads
- Repeated reading of columns of B over and over wastes time.
- Can we fix this by reducing repeated reads of B?

Memory bottleneck for matrix multiplication

```
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

Read each column of B n times - n^3 memory reads!!



- The CI ~ 2 is because we have n^3 operations but also n^3 memory reads
- Repeated reading of columns of B over and over wastes time.
- Can we fix this by reducing repeated reads of B?

Cache blocking

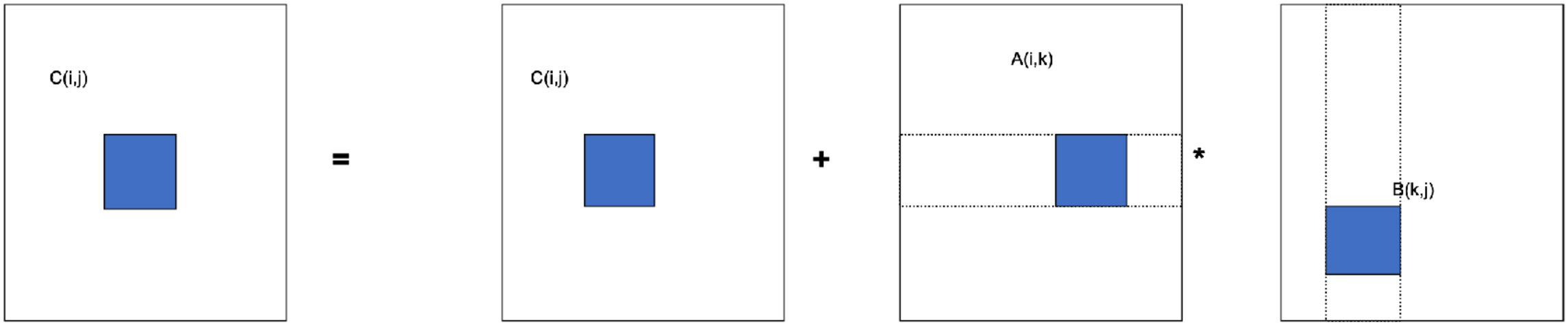
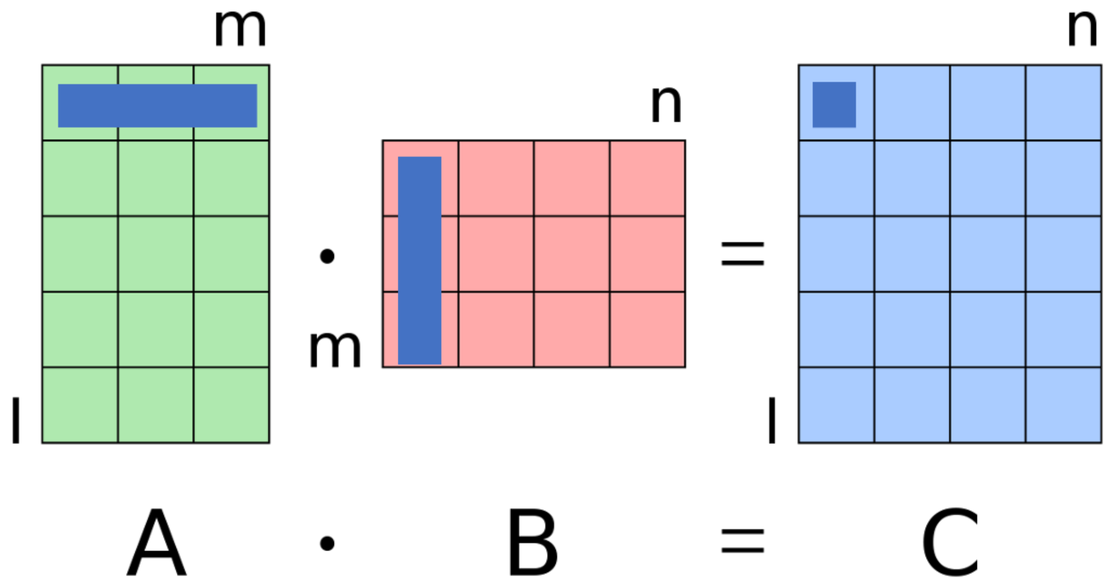


Image from Demmel

- Divide each $n \times n$ matrix into b non-overlapping blocks
- Assume for simplicity that $n = Nb$ for some integer N
- Core idea: b should be determined so that the block fits into fast memory

Why does cache blocking help?



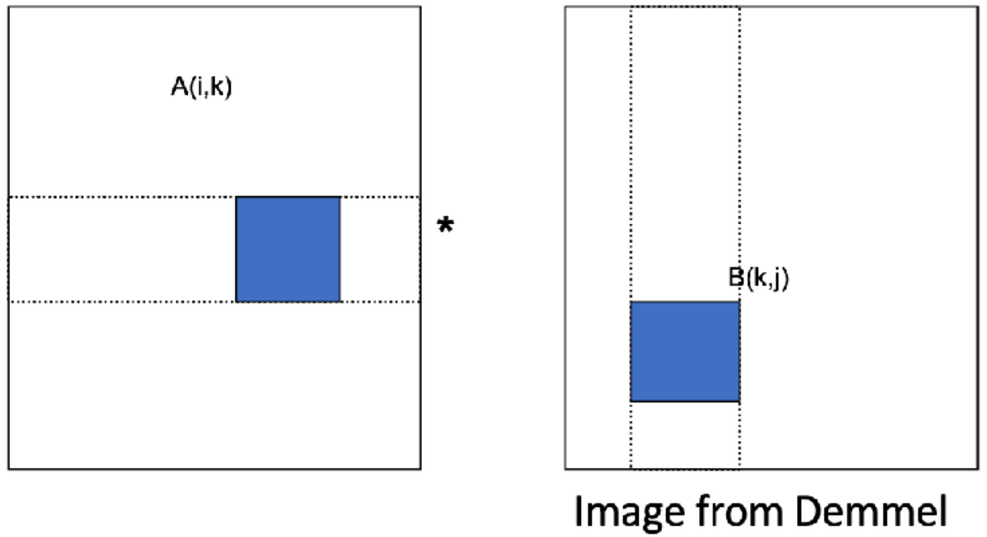
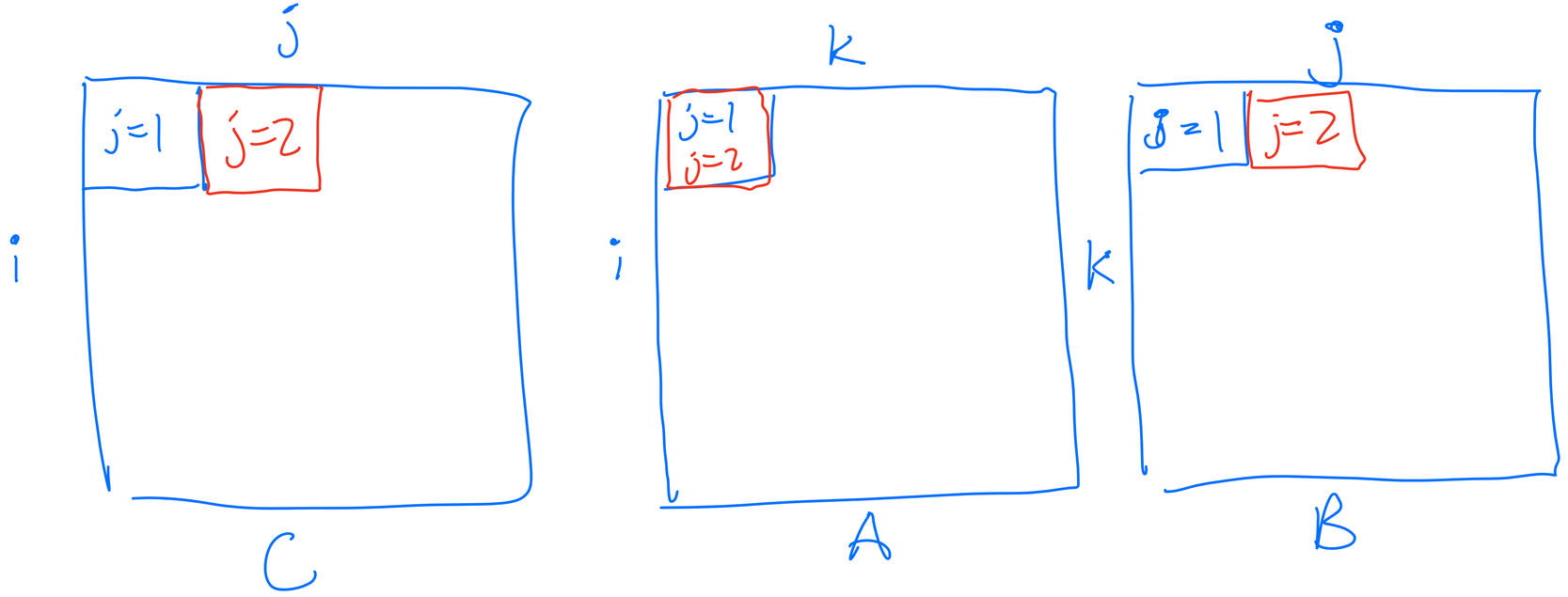


Image from Demmel

- We still do the exact same number of operations and memory reads overall.
- Previously, we read a column of B, then immediately discarded it. Here, we read b columns of a block of B *into fast memory;* each column stays in cache longer before being evicted.

Cache blocking memory access pattern

```
for i = 1 to N
      for j = 1 to N
            for k = 1 to N
            // block matmul
            // 3 nested loops
```



- Suppose $3b^2$ is the cache size. Then, 3 blocks fit into fast memory.
- As j changes, $b \times b$ blocks of B still get evicted from cache and re-read, but only N times instead of $n = Nb$ for naive matrix multiplication.

What is the CI of cache blocked matmul?

- $3N^2$ memory reads of b^2 entries of A and C, $3(Nb)^2 = 3n^2$
- N^3 memory reads of b^2 entries of B, $N(Nb)^2 = Nn^2$ reads
- $2n^3$ operations implies that the CI is $2n^3/(3n^2 + Nn^2)$.

```
for i = 1 to N
        for j = 1 to N
                {read block C(i,j) into fast memory}
                for k = 1 to N
                        {read block A(i,k) into fast memory}
                        {read block B(k,j) into fast memory}
                        // do a matrix multiply on blocks
                        // this is really 3 nested loops!
                        C(i,j) = C(i,j) + A(i,k) * B(k,j)

{write block C(i,j) back to slow memory}
```

Implementation is really 6 nested loops

What is the CI of cache blocked matmul?

- $3N^2$ memory reads of b^2 entries of A and C, $3(Nb)^2 = 3n^2$
- N^3 memory reads of b^2 entries of B, $N(Nb)^2 = Nn^2$ reads
- $2n^3$ operations implies that the CI is $2n^3/(3n^2 + Nn^2)$.

```
for i = 1 to N
        for j = 1 to N
                {read block C(i,j) into fast memory}
                for k = 1 to N
                        {read block A(i,k) into fast memory}
                        {read block B(k,j) into fast memory}
                        // do a matrix multiply on blocks
                        // this is really 3 nested loops!
                        C(i,j) = C(i,j) + A(i,k) * B(k,j)

{write block C(i,j) back to slow memory}
```

Implementation is really 6 nested loops

For large n, the CI is $\approx 2n/N = 2b$!

Demo

- Lets compare naive matrix multiply with cache blocked matrix multiply
  - What are the performance differences?
  - How does performance depend on the block size b?

Cache blocked matmul in practice

- Key assumption: $b \times b$ blocks of A, B, C *all* fit into fast memory!
  - Implies that $b = \sqrt{\text{fast memory size}/3}$
  - You can also just benchmark and tune the block size once for each computer architecture
- Code gets uglier if:
  - Dimensions of A, B, C are not perfectly divisible by block size b
  - You want to optimize further by exploiting multiple levels of cache

Performance model analysis of cache blocking

- Recall our performance model runtime estimate $(f \times t_f)\left(1 + \frac{1}{CI} \times \frac{t_m}{t_f}\right)$
  - Need $CI \geq t_m/t_f$ to get half of peak performance
  - Matmul has $CI = O(b) = O(\sqrt{\text{fast mem. size}/3})$
  - Implies fast memory size for matmul should be $3(t_m/t_f)^2$ to get 50% of peak, which is close (e.g., $3(100)^2 \approx 30000$, implies ~240KB L1 cache)
- However, in practice, blocking/tiling doesn't typically achieve $O(b)$ CI without additional optimizations due to model assumptions.

Alternatives to cache blocking

- What are the downsides of cache blocking/loop tiling?
  - Blocking/tiling is known as a “cache-aware” algorithm (you have to know what size your cache is to implement this).
- There are “cache-oblivious” implementations for matmul
  - Recursive matmul, Strassen’s algorithm.
  - Recall that recursion is when a function calls itself - e.g., factorial(n) = factorial(n-1) * n, plus the base case that factorial(1) = 1.

Recursive matmul

$$C = \begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = A \cdot B = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix} = \begin{pmatrix} A_{00}\cdot B_{00} + A_{01}\cdot B_{10} & A_{00}\cdot B_{01} + A_{01}\cdot B_{11} \\ A_{10}\cdot B_{00} + A_{11}\cdot B_{10} & A_{10}\cdot B_{01} + A_{11}\cdot B_{11} \end{pmatrix}$$

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \bullet \begin{bmatrix} B_{00} & B_{10} \\ B_{10} & B_{11} \end{bmatrix} = \begin{bmatrix} A_{00}*B_{00} + A_{01}*B_{10} & A_{00}*B_{01} + A_{01}*B_{11} \\ A_{10}*B_{00} + A_{11}*B_{10} & A_{10}*B_{01} + A_{11}*B_{11} \end{bmatrix}$$

Image from Demmel

- Idea: decompose all matrices into 2×2 block matrices
- Each individual block is the sum of the product of two other blocks
- Use recursion when computing the product of two other blocks

Recursive matmul, cont.

```
function RMM(C, A, B, n)
if (n == 1){
    C = A * B;
} else {
    C00 = RMM(A00, B00, n / 2) + RMM(A01, B10, n / 2);
    C01 = RMM(A00, B01, n / 2) + RMM(A01, B11, n / 2);
    C10 = RMM(A10, B00, n / 2) + RMM(A11, B10, n / 2);
    C11 = RMM(A10, B01, n / 2) + RMM(A11, B11, n / 2);
}
```

- Why does recursive matrix multiplication work?
  - You recurse down to the base case (size 1 matrices), then back up.
  - Memory only gets accessed in the upward sweep, so matrices start small but increase in size each level. They fill up cache gradually until it’s full.

Analyzing recursive matmul: operations

- We can count the number of floating point operations for level $i > 1$
  - 4 sums of $(n_i/2 \times n_i/2)$ matrices per level = n_i^2 operations.
  - 8 matrix multiplications of size $(n_i/2 \times n_i/2)$.
  - Can show this gives you $2n^3 - n^2 + \ldots$ operations over $\log_2(n)$ levels; asymptotically the same number of operations as standard matrix multiplication.

Analyzing recursive matmul: memory

- Memory movement: since cache is not explicitly managed, we assume arrays move into fast memory until the cache (fast memory) is full.
  - 8 * (memory movement for n/2 matmul) + 4 * 3 * (n/2)^2 (subblocks) per level if A, B, C don't fit in fast memory
  - Can show that the slow memory cost over all recursive levels is $O(n^3/\sqrt{\text{size of fast memory} + \dots})$, similar to blocking/tiling.
- In practice, implementing recursive matmul requires matrix "views". For example, the following line of pseudocode

```
C00 = RMM(A00, B00, n / 2) + RMM(A01, B10, n / 2);
```

shouldn't actually allocate a new matrix C00, but instead do in-place updates.

Variations on recursive matmul

- Most cache oblivious algorithms still use parameters like block size.
  - Cache-oblivious algorithms usually stop recursion early (when matrices are small enough) then execute a hand-optimized “microkernel”.
- Alternative decomposition: alternate between vertical and horizontal splitting
  - Advantage: easier to implement for arbitrary sized matrices.

Variations on recursive matmul

- Most cache oblivious algorithms still use parameters like block size.
  - Cache-oblivious algorithms usually stop recursion early (when matrices are small enough) then execute a hand-optimized “microkernel”.
- Alternative decomposition: alternate between vertical and horizontal splitting
  - Advantage: easier to implement for arbitrary sized matrices.

If the number of rows in A is the largest dimension

$$C = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix}$$

Variations on recursive matmul

- Most cache oblivious algorithms still use parameters like block size.
  - Cache-oblivious algorithms usually stop recursion early (when matrices are small enough) then execute a hand-optimized “microkernel”.
- Alternative decomposition: alternate between vertical and horizontal splitting
  - Advantage: easier to implement for arbitrary sized matrices.

If the number of rows in A is the largest dimension

$$C = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix}$$

If the number of columns in B is the largest dimension

$$C = A\begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} AB_1 & AB_2 \end{pmatrix}$$

Variations on recursive matmul

- Most cache oblivious algorithms still use parameters like block size.
  - Cache-oblivious algorithms usually stop recursion early (when matrices are small enough) then execute a hand-optimized “microkernel”.
- Alternative decomposition: alternate between vertical and horizontal splitting
  - Advantage: easier to implement for arbitrary sized matrices.

If the number of rows in A is the largest dimension

$$C = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix}$$

If the number of columns in B is the largest dimension

$$C = A\begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} AB_1 & AB_2 \end{pmatrix}$$

If the number of columns in A (rows in B) is the largest dimension

$$C = \begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2$$

Variations on recursive matmul, cont.

- Additional optimizations: copy data into a different format prior to operating on it.
  - Block versions of row/column major ordering: improve cache efficiency when reading blocks
  - Recursive block ordering of matrix entries (e.g., Z-ordering, used to efficiently build quad/octrees)

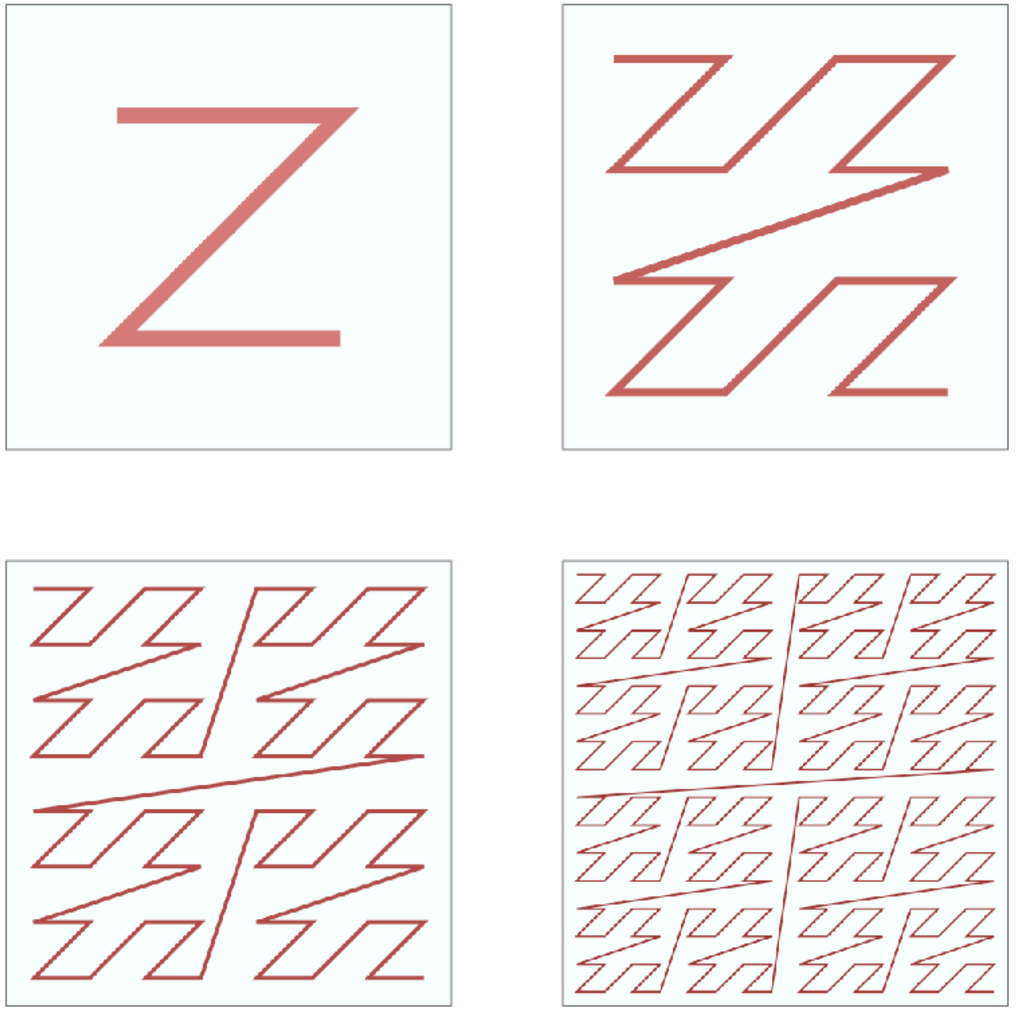


Image from Wikipedia

Strassen's algorithm

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{bmatrix}$$

$$\begin{aligned}
M_1 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}); \\
M_2 &= (A_{21} + A_{22}) \times B_{11}; \\
M_3 &= A_{11} \times (B_{12} - B_{22}); \\
M_4 &= A_{22} \times (B_{21} - B_{11}); \\
M_5 &= (A_{11} + A_{12}) \times B_{22}; \\
M_6 &= (A_{21} - A_{11}) \times (B_{11} + B_{12}); \\
M_7 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}),
\end{aligned}$$

Recursive algorithm, but a more complex 2x2 decomposition

Image from Wikipedia

Strassen: recursive matmul with a twist

- Exploits the fact that matrix-matrix multiplication dominates the memory access costs.
- Strassen requires only 7 matrix multiplications per level!
- The first matrix multiplication algorithm to beat $O(n^3)$: has asymptotic cost of $O(n^{2.8074})$
- Note additional memory required for intermediate M_i terms

$$
\begin{aligned}
M_1 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}); \\
M_2 &= (A_{21} + A_{22}) \times B_{11}; \\
M_3 &= A_{11} \times (B_{12} - B_{22}); \\
M_4 &= A_{22} \times (B_{21} - B_{11}); \\
M_5 &= (A_{11} + A_{12}) \times B_{22}; \\
M_6 &= (A_{21} - A_{11}) \times (B_{11} + B_{12}); \\
M_7 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}),
\end{aligned}
$$

Image from Wikipedia

Software libraries

- Linear algebra libraries are probably the most successful example of HPC software on CPU architectures.
  - LINPACK benchmarks used to create the Top500 list of supercomputers
- BLAS (Basic Linear Algebra Subprograms) specifies interfaces and operations. A vendor then provides optimized implementations of BLAS for their specific architecture.
  - Level 1 BLAS: vector operations (dot, scalar*x + y). $O(n)$ operations, low CI
  - Level 2 BLAS: matrix-vector operations. $O(n^2)$ operations, low-ish CI
  - Level 3 BLAS: matrix-matrix operations. $O(n^3)$ operations, high CI
- Example of recursive algorithms in practice: LibFLAME

BLAS 1 vs BLAS 2 vs BLAS 3 performance

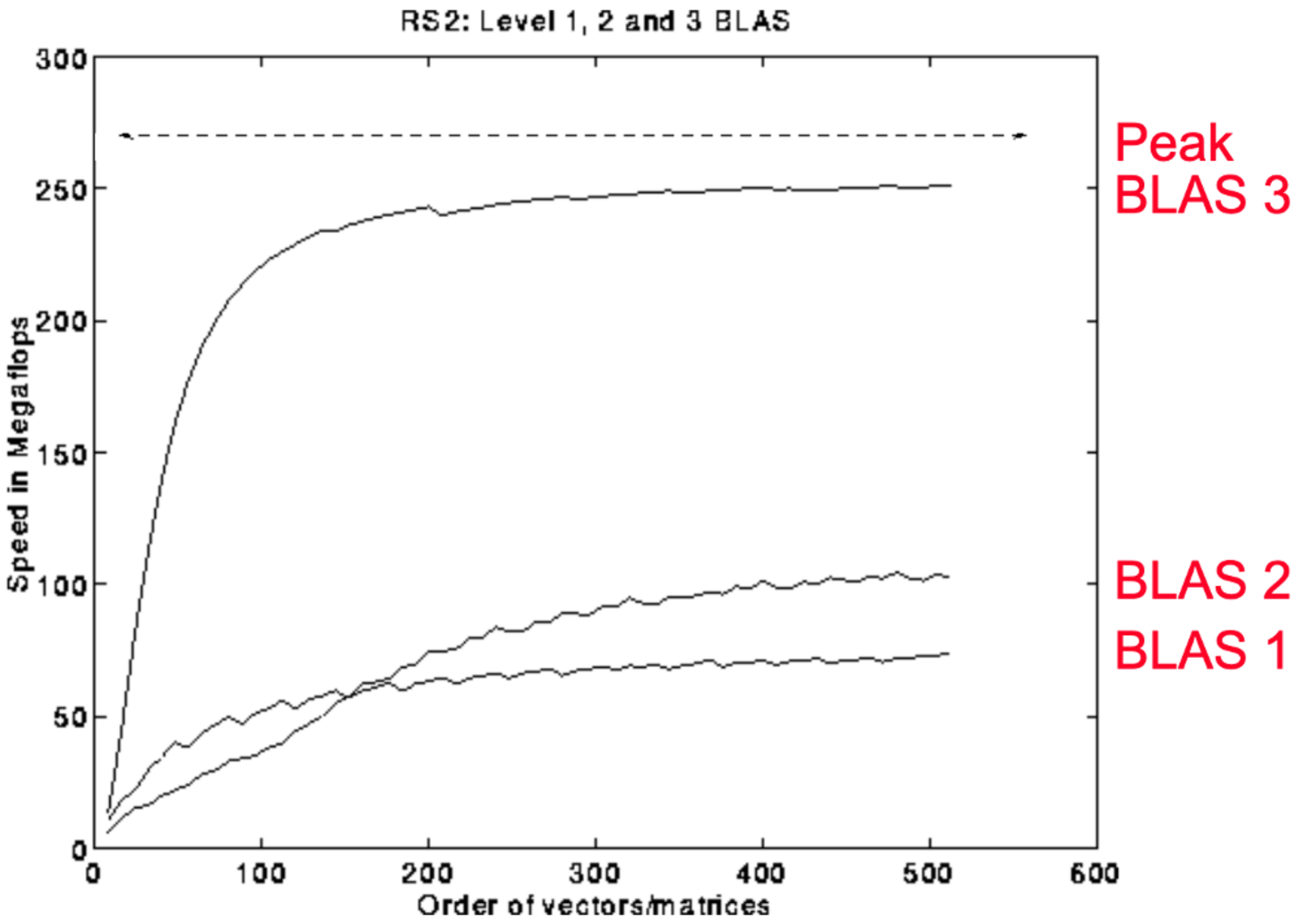


Image from Tao Yang