TRANSFORMERS: AGE OF PARALLEL MACHINES

(a biased introduction to Computer architecture)

Ana Lucia Varbanescu, University of Twente, NL

A.L.Varbanescu@utwente.nl

What's in a name?

- @AI enthusiasts: this is not about the AI transformers models [1]
- @Movie enthusiasts: this is a word-play on the transformer movies [2]
- @All (the others): this is about how computer architecture and computing systems have been transformed in the past 15 years

[1] Vaswani et al. "Attention Is All You Need" - https://arxiv.org/abs/1706.03762
[2] https://www.imdb.com/list/ls069544665/

Assumptions

- We need computing systems for **high-performance computing**
- … thus we focus on how machines are build to provide high-performance
- … and we talk about that in the context of applications

- **Main goal:** best possible performance for our applications in computational science & engineering

- What else is out there (but we won't cover)?
  - Real-time systems – guarantees are everything
  - Embedded systems – efficiency and scale is everything
  - Shared (large) systems (e.g., cloud computing) – sharing is ~~caring~~ everything
  - Computing continuum – a mix of everything from IoT through Edge/Fog to Cloud

Agenda (ambitious)

- Part 1 : Introduction to computer systems
  - CPUs, Memory, Caching, Accelerators
- Part 2 : Parallelism and parallel machines
  - Flynn's taxonomy, SIMD/vectorization, multi-core/many-core
  - Alternative architectures (FPGAs, AI-based, ...)
- Part 3 : Performance and tools
  - Basic metrics, models, counters …
- Part 4 : Where to ?
  - Famous last words …

"Larry, do you remember where we buried our hidden agenda?"

PART1: COMPUTING MACHINES

Computer Systems

Simplistic definition

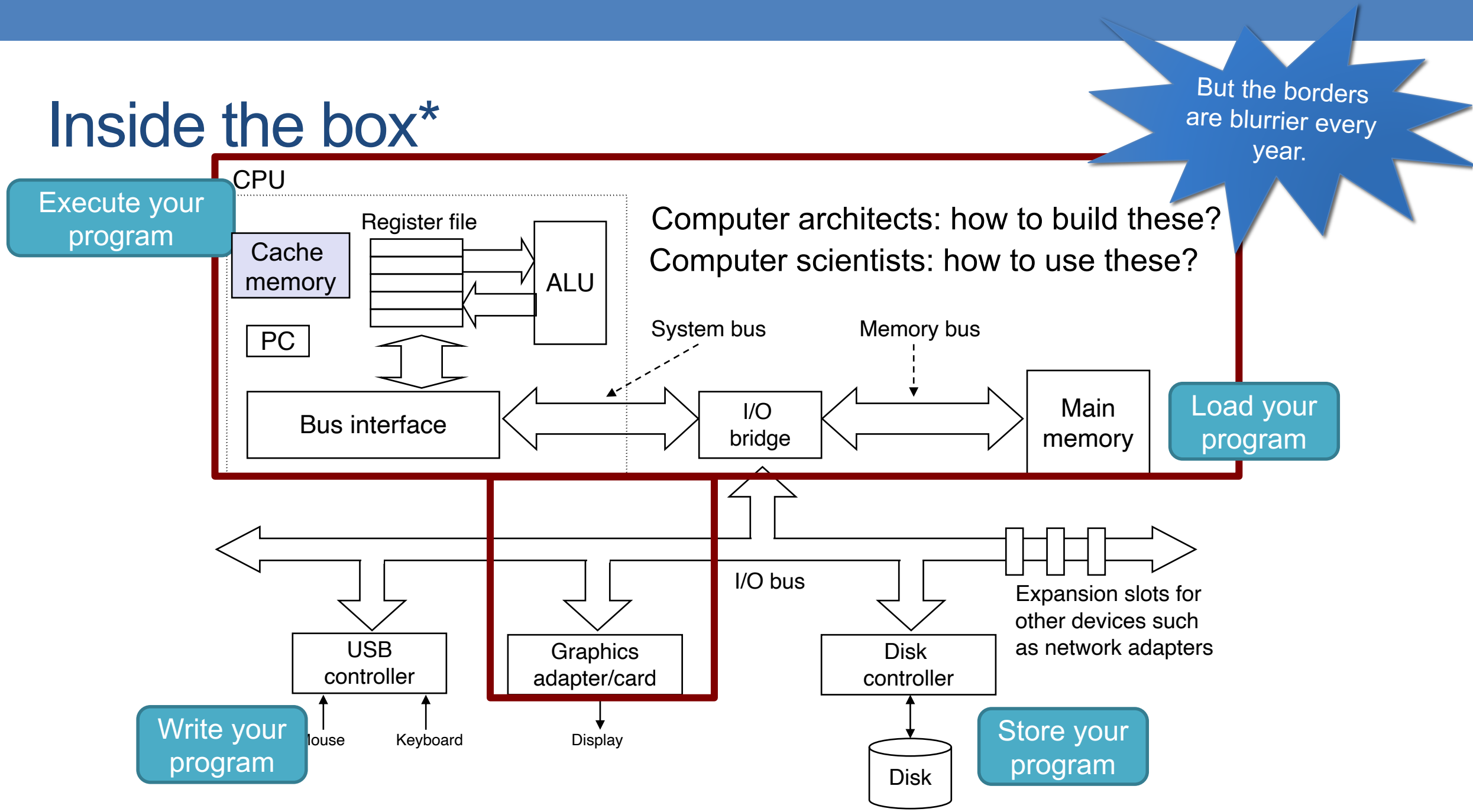
A mix of hardware and software (systems) used to execute applications.

Traditional goals:

- High(er)-performance systems
- Low(er)-power systems
- More efficient systems
- Higher availability systems
- Reliable systems
- Programmable systems
- ….

Out of the box

Inside the box*



*Adapted from "Computer Systems: A Programmer's View" (Ch1) by Bryant and O'Hallaron

So … why should you care?

- High-performance computing (HPC)* makes extensive use of computer systems
  - … in fact, novel developments in computer systems make modern HPC possible!

- You are a responsible user
  - Write *correct* applications.
  - Write *efficient* applications.

- You are an innovator
  - Build the next big HPC* application or machine

*Replace “HPC” with “society” for a broader, still valid statement.

Data representation & computer arithmetic

Bits & bit vectors

- All information in computer systems is represented by **bits**
  - Numbers, text, images, code, applications …
  - Why bits?
    - Easy to store with bistable elements
    - Reliably transmitted on noisy and inaccurate wires

- All data is represented as bit vectors
  - What does this mean: 01101001101001010110100101101001

Interpretation is key!

Data types guide interpretation

```
unsigned long a = 011010011010… => 1772448105
double d = 011010011010… => 2.49963182032e+25
char x[20] = 011010011010… => “i¥ii”
```

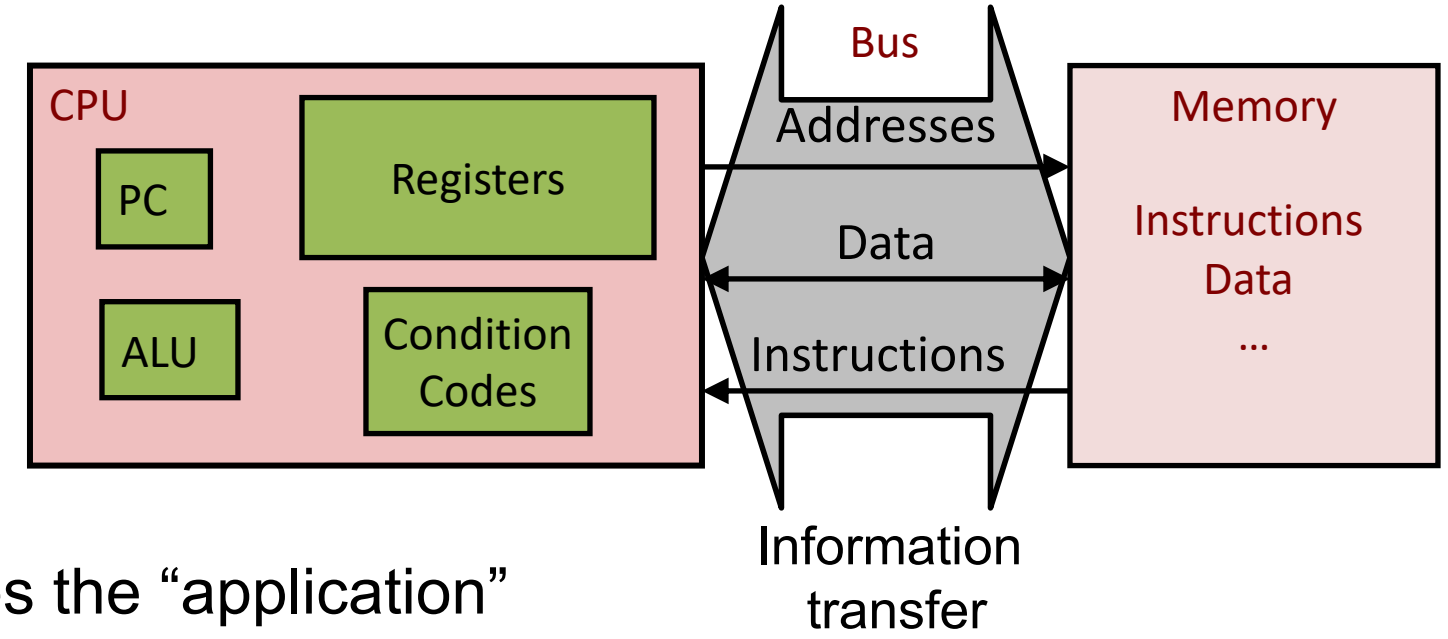
Why should you care?

- **Correctness**
  - Some numbers cannot be represented (range, precision …)
  - Some operations are “inaccurate” (range, precision …)
  - Different data types can/cannot support certain operations
- **Performance** & **efficiency**
  - Data takes memory/storage space
  - Larger data takes more time/energy to process
  - Some operations are faster than others

High **performance** and **efficiency** are always **affected** by your **choices** on data representation and operations.

Processor basic operation

A processor's inner workings



- CPU = executes the "application"
  - Manages the execution progress (PC)
  - Fetches needed instructions and data (addresses)
  - Executes (ALU) operations and manages results
- Memory = stores the executable code of the application and the data
  - Receives request + address, replies with data (a bit vector)
- Bus = facilitates information (=bits) movement

Why should you care?

Your program is compiled* and stored in memory and executed on the CPU**

Correctness

- Compilers do not fix bugs
- Debuggers help fixing bugs (and you run the debugging process…)

Performance & **efficiency**

- Code structure “guides” compilers to generate better executables
- Information movements takes time
- Processor resources are not infinite

High **performance** and **efficiency** are always **affected** by your **choices** when coding and compiling.

* For most programming models, including C/C++/Fortran

** Accelerators may also be used, with similar principles (TBD)

The memory hierarchy

Memory

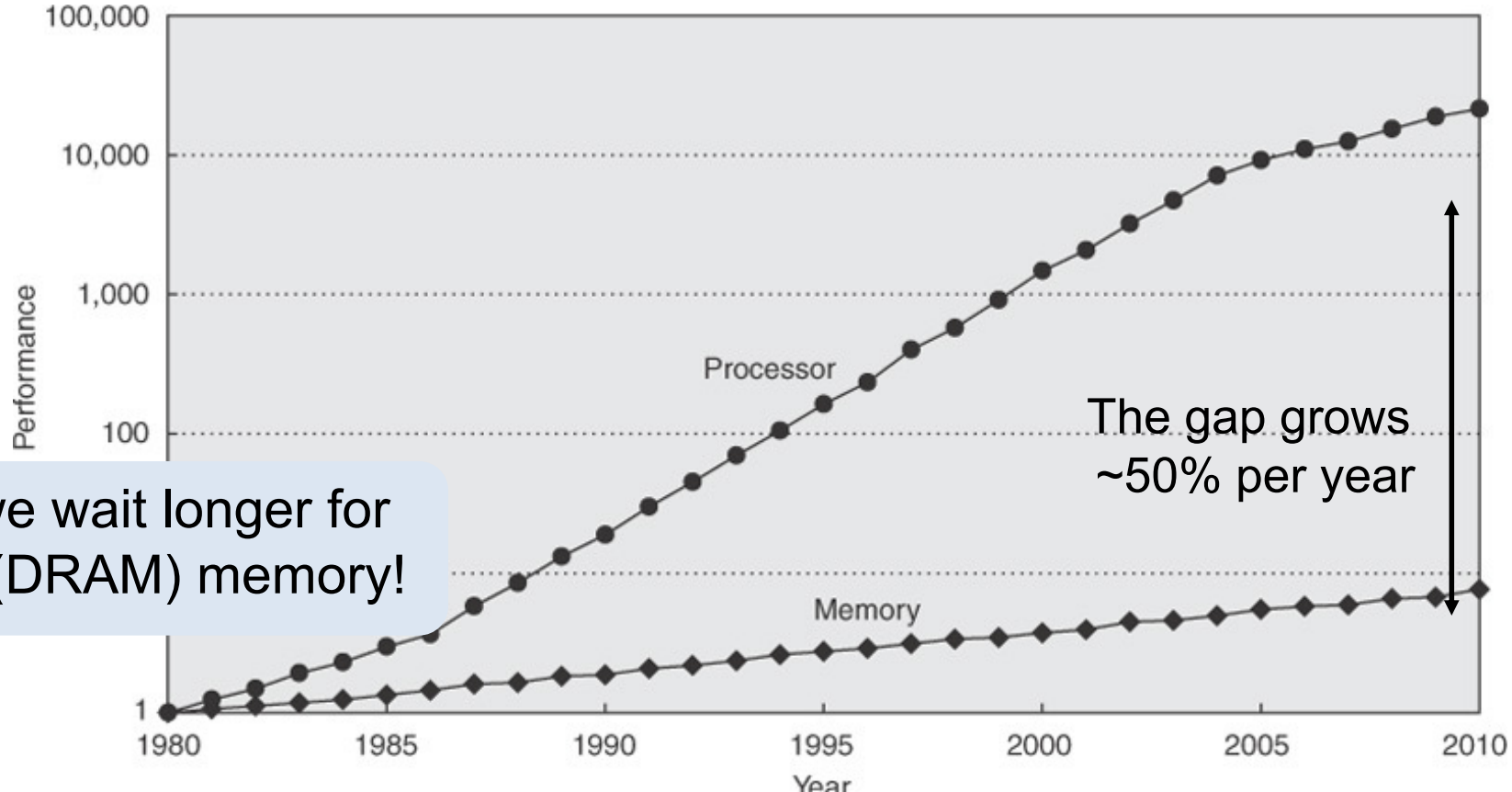
- Typically organized as linear spaces, with some word-size granularity

- Code and data are stored in memory
- Everything that lives in memory has an “address”
  - Visible at assembly level
  - Accessible via pointers/variable names/… from the program itself

- Memory operations are slow!
  - Off-chip
  - Request read/write
  - Search and find

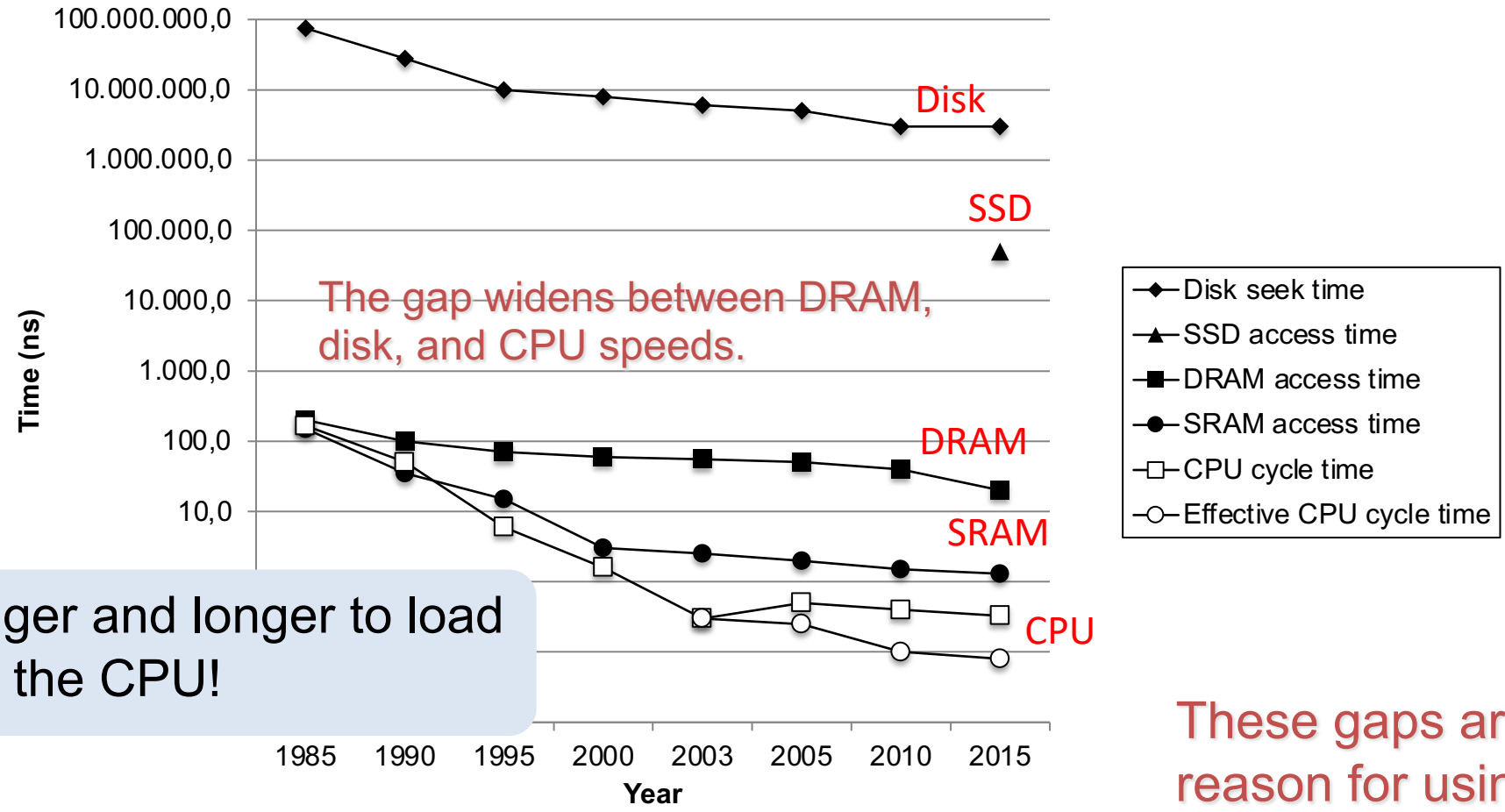
The CPU-Memory Gap

- Flat memory model
  - All accesses = same latency
  - Memory latency slower to improve than processor speed



… which means we wait longer for any access to the (DRAM) memory!

The CPU-Memories Gap

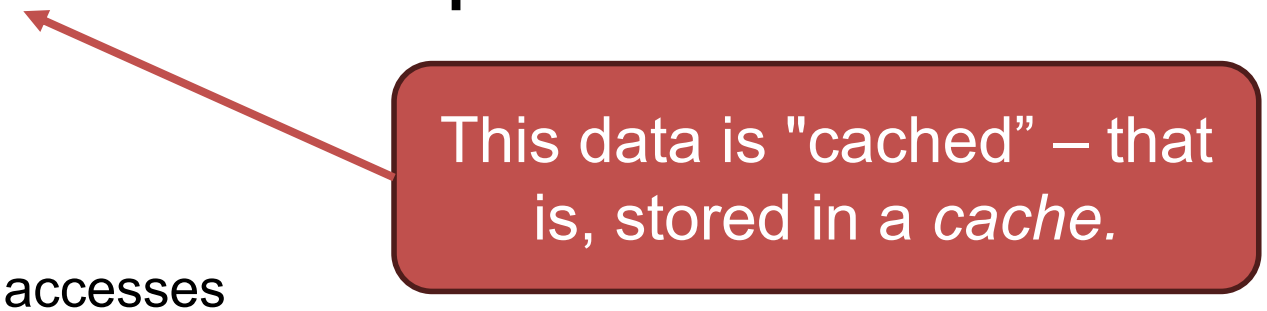


Data takes longer and longer to load to the CPU!

These gaps are the main reason for using a memory hierarchy.

Memory hierarchy

- A single memory for the entire system is not efficient!
- Several memory spaces
  - Large size, low cost, high latency – main memory
  - Small size, high cost, low latency – caches / registers
- Main idea: Bring **some of the data closer to the processor**
  - Smaller latency => faster access
  - Smaller capacity => not all data fits!
- Who can benefit?
  - Applications with **locality** in their data accesses
    - Spatial locality
    - Temporal locality



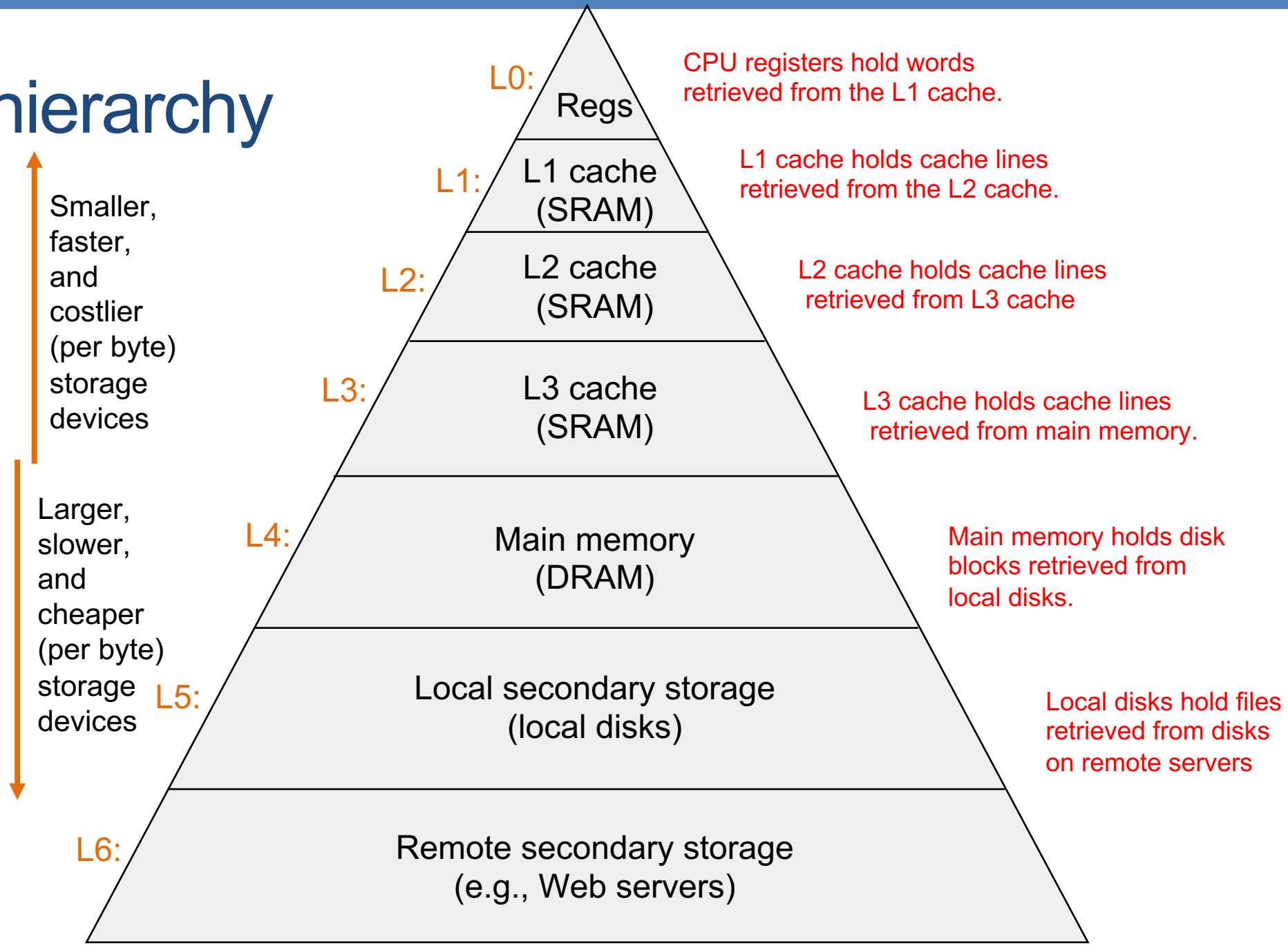
Memory hierarchy and caches

- **Cache:** A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.

- Memory hierarchy
  - Multiple layers of memory, from **small & fast** (lower levels) to **large & slow** (higher levels)
  - *For each k, the faster, smaller device at level k is a **cache** for the larger, slower device at level k+1.*

- How/why do memory hierarchies work?
  - **Locality** => data at level k is used more often than data at level k+1.
    - Level k+1 can be slower, and thus larger and cheaper.

Memory hierarchy



Caching in the Memory Hierarchy

Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-8 bytes words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware MMU
L1 cache	64-byte blocks	On-Chip L1	4	Hardware
L2 cache	64-byte blocks	On-Chip L2	10	Hardware
Virtual Memory	4-KB pages	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Disk firmware
Network buffer cache	Parts of files	Local disk	10,000,000	NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

Memory hierarchy

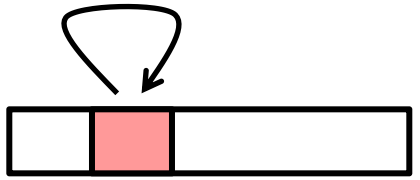
- Challenges
  - Size: no space for every memory address
  - Organization: what gets loaded & where ?
  - Policies: who's in, who's out, when, why?

- Performance
  - Hit = access found data in fast memory => low latency
  - Miss = data not in fast memory => high latency + penalty
  - **Metric**: hit ratio (H) = the fraction of accesses that hit => the higher the ratio, the better the performance!

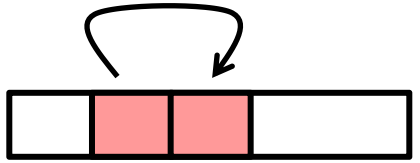
Locality

- Principle of Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently

- Temporal locality:
  - Recently referenced items are likely to be referenced again in the near future



- Spatial locality:
  - Items with nearby addresses tend to be referenced close together in time



Locality Example

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- Data references
  - Reference array elements in succession (stride-1 reference pattern). — Spatial locality
  - Reference variable `sum` each iteration. — Temporal locality
- Instruction references
  - Reference instructions in sequence. — Spatial locality
  - Cycle through loop repeatedly. — Temporal locality

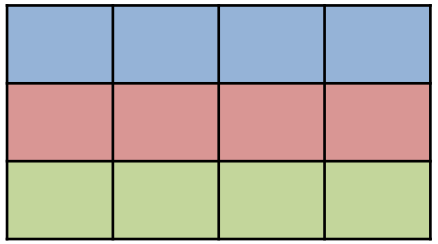
Qualitative estimates of locality

- Question: Does this function have good locality with respect to array `a`?

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Matrix



Row-major in memory

Every read from the matrix fetches a cache line => assume 4 elements
Assume row-major order and N,M very large => reading a[0][0] will bring in blue elements, while reading a[1][0] will need red elements.
This is **poor locality** – not reusing the same or close-by elements.

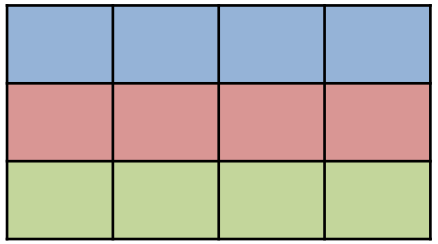
Qualitative estimates of locality

- Question: Does this function have good locality with respect to array `a`?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Matrix



Row-major in memory

Every read from the matrix fetches a cache line => assume 4 elements
Assume row-major order and N,M very large => reading a[0][0] will bring in blue elements and reading a[0][1]..a[0][3] will need blue elements.
This is **great locality** – reusing the same or close-by elements.

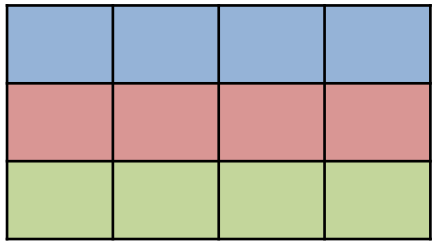
Qualitative estimates of locality

- Question: Does this function have good locality with respect to array `a`?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M/2; i++)
        for (j = 0; j < N/4; j++)
            sum += a[i*2][j*4];
    return sum;
}
```

Matrix

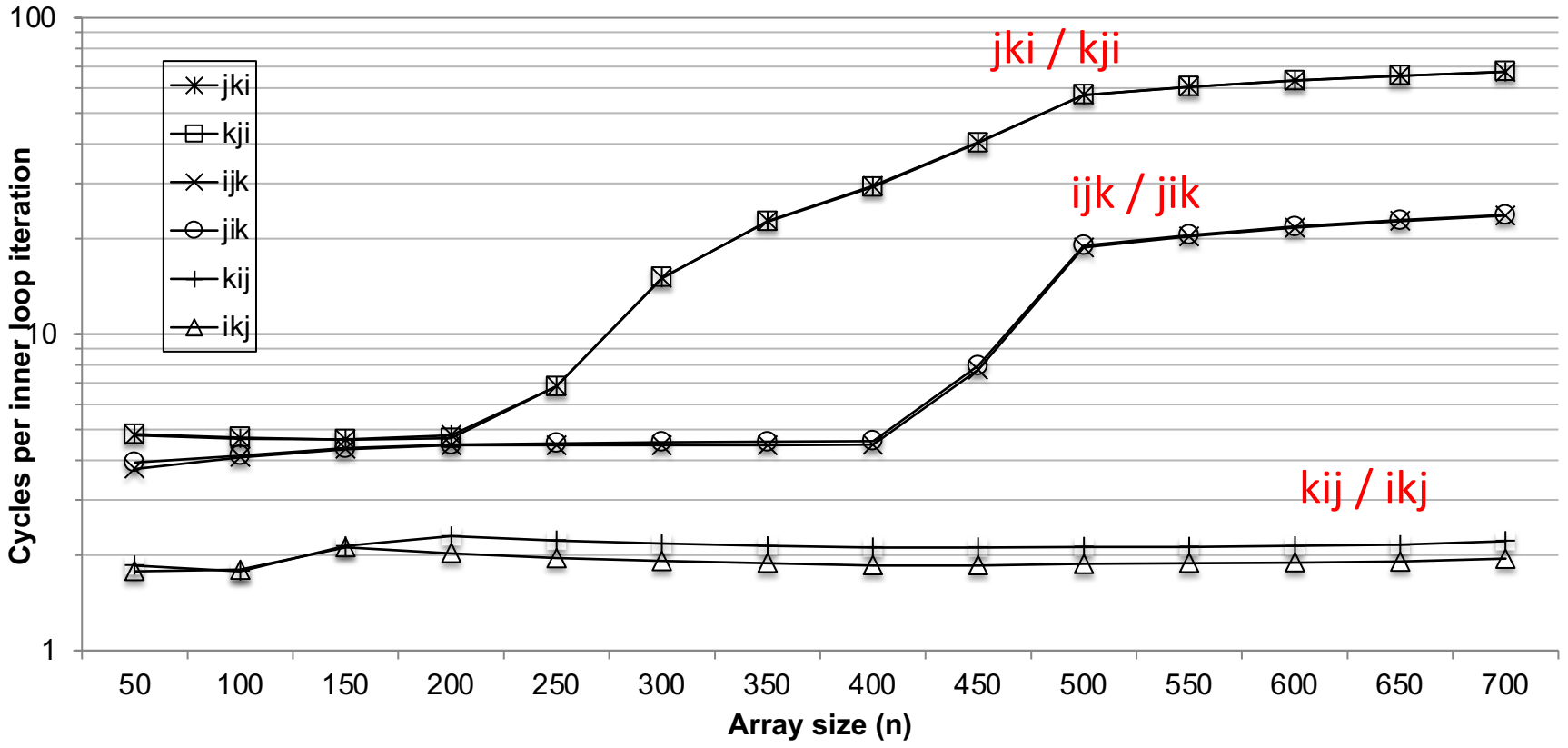


Row-major in memory

Every read from the matrix fetches a cache line => assume 4 elements
Assume row-major order and N,M very large => reading a[0][0] will bring in blue elements, while reading such scattered data from a further will need different colors. This is **non-perfect locality** – depends on sizes …

Matrix Multiplication

Good vs bad locality / caching …




```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
   sum = 0.0;
   for (k=0; k<n; k++)
     sum += a[i][k] * b[k][j];
   c[i][j] = sum;
 }
}
```
ijk / jik

```
for (k=0; k<n; k++) {
 for (i=0; i<n; i++) {
  r = a[i][k];
  for (j=0; j<n; j++)
   c[i][j] += r * b[k][j];
 }
}
```
kij / ikj

```
for (j=0; j<n; j++) {
 for (k=0; k<n; k++) {
   r = b[k][j];
   for (i=0; i<n; i++)
    c[i][j] += a[i][k] * r;
 }
}
```
jki / kji

Why should you care?

Correctness

- Memory allocation and out-of-bounds errors are very common, yet difficult to spot

Performance & **efficiency**

- Caching is the most used method to improve memory latency
- Memory access patterns impact performance
- Data structures and in-memory layouts are essential
- Processor & memory architecture variants may “prefer” different access patterns

Memory operations are the main bottleneck in most HPC today!
Check your **data memory layout** and **access patterns** to improve **locality!!**

In summary: Computing systems basics …

- … are essential for the building HPC systems
- … and for programming them

- Be literate in these topics ☺
  - Caching
  - Processing
  - Data representation
  - Instructions

- … else you will have trouble programming these machine efficiently.