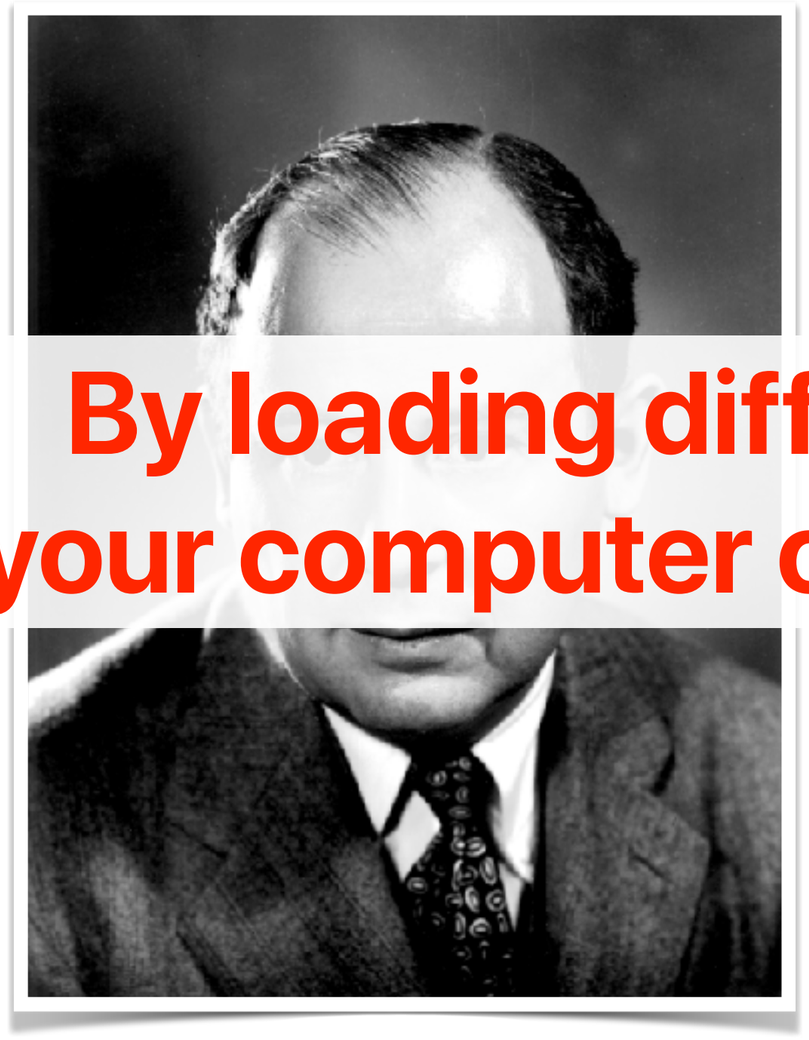
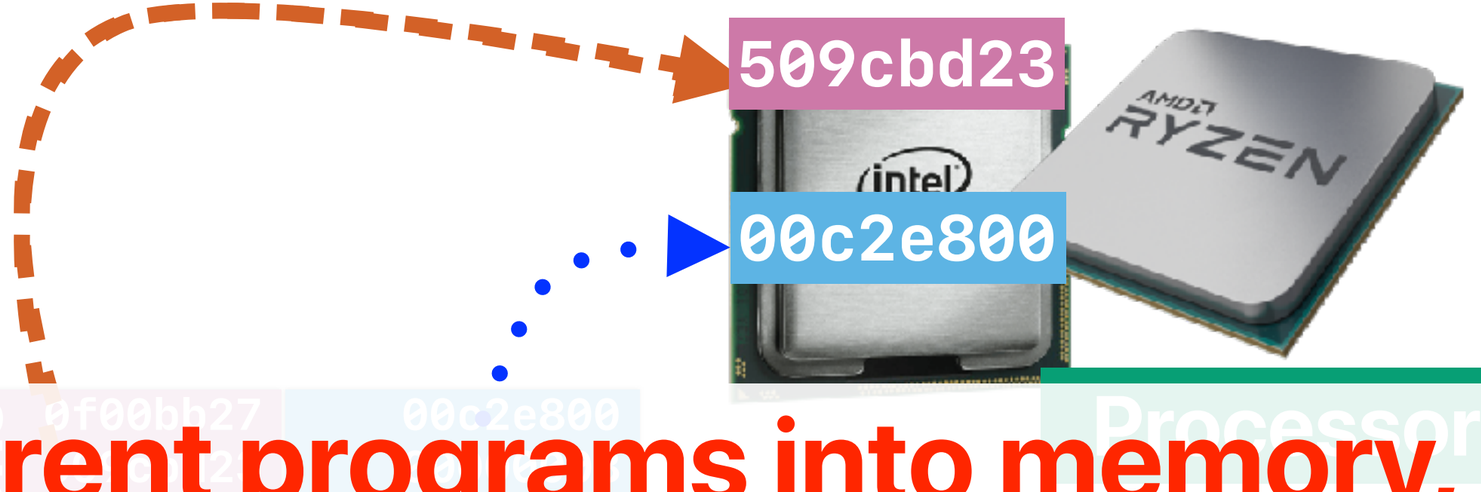
Performance

Hung-Wei Tseng

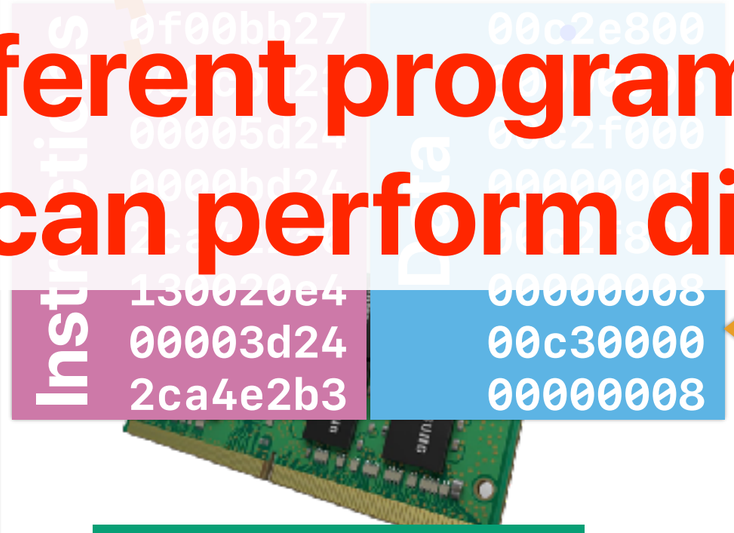
Recap: von Neumman Architecture





Processor

By loading different programs into memory, your computer can perform different functions



Memory

Storage

Recap: Start with this simple program in C

```
int A[] =
{1,2,3,4,5,6,7,8,9,10,1,2,3,4
,5,6,7,8,9,10};
```

Compiler

```
Contents of section .data:
 0000 01000000 02000000 03000000 04000000
 0010 05000000 06000000 07000000 08000000
 0020 09000000 0a000000 01000000 02000000
 0030 03000000 04000000 05000000 06000000
 0040 07000000 08000000 09000000 0a000000
```

Data

```
int main()
{
    int i=0, sum=0;
    for(i = 0; i < 20; i++)
    {
        sum += A[i];
    }
    return 0;
}
```

control flow operations

logical operations

memory access

arithmetic operations

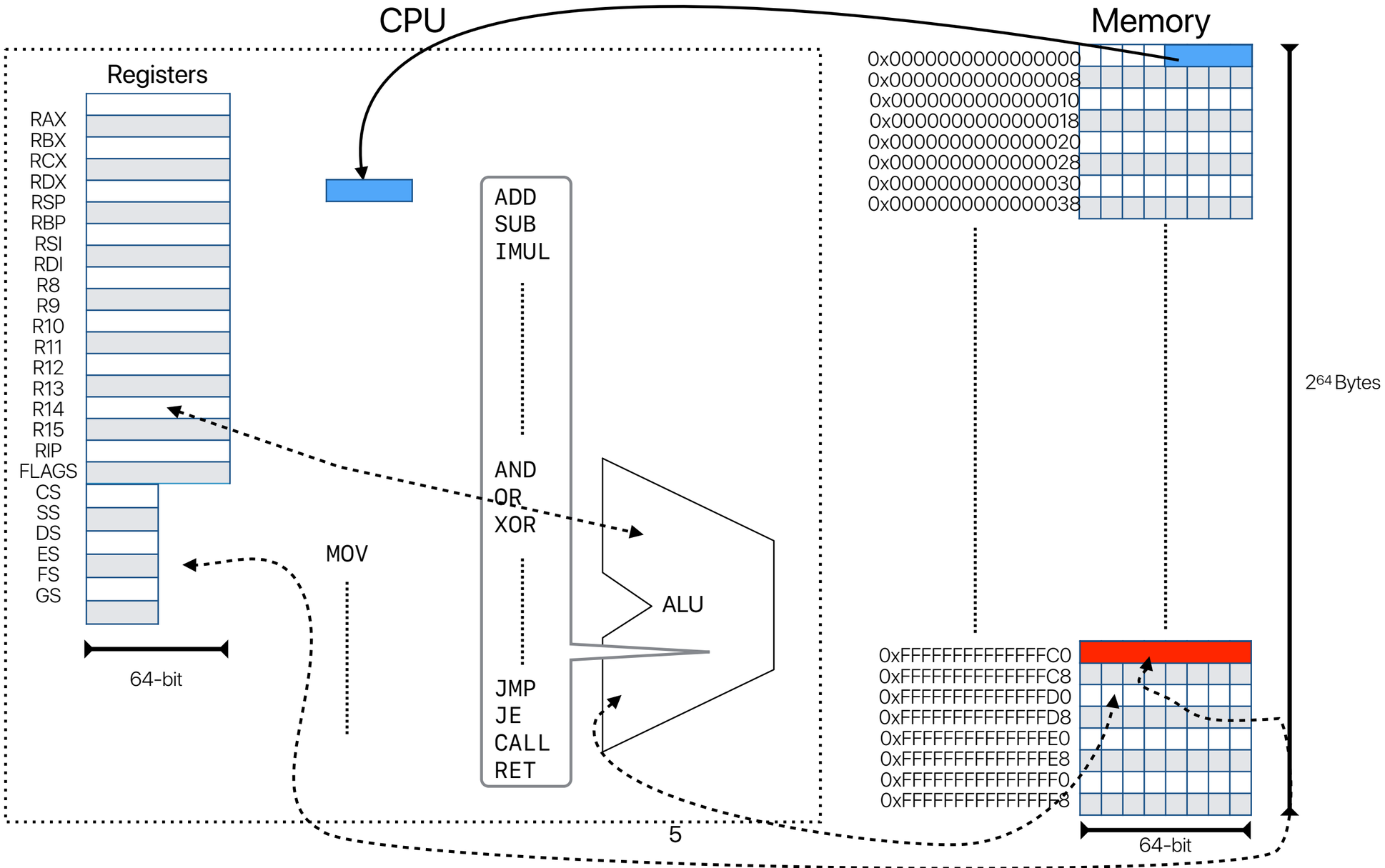
```
main:
.LFB0:
    endbr64
    pushq   %rbp
    movq    %rsp, %rbp
    movl    $0, -8(%rbp)
    movl    $0, -4(%rbp)
    movl    $0, -8(%rbp)
    jmp .L2
.L3:
    movl    -8(%rbp), %eax
    cltq
    leaq    0(,%rax,4), %rdx
    leaq    A(%rip), %rax
    movl    (%rdx,%rax),
%eax
    addl    %eax, -4(%rbp)
    addl    $1, -8(%rbp)
.L2:
    cmpl    $19, -8(%rbp)
    jle .L3
    movl    $0, %eax
    popq    %rbp
    ret
```

Compiler

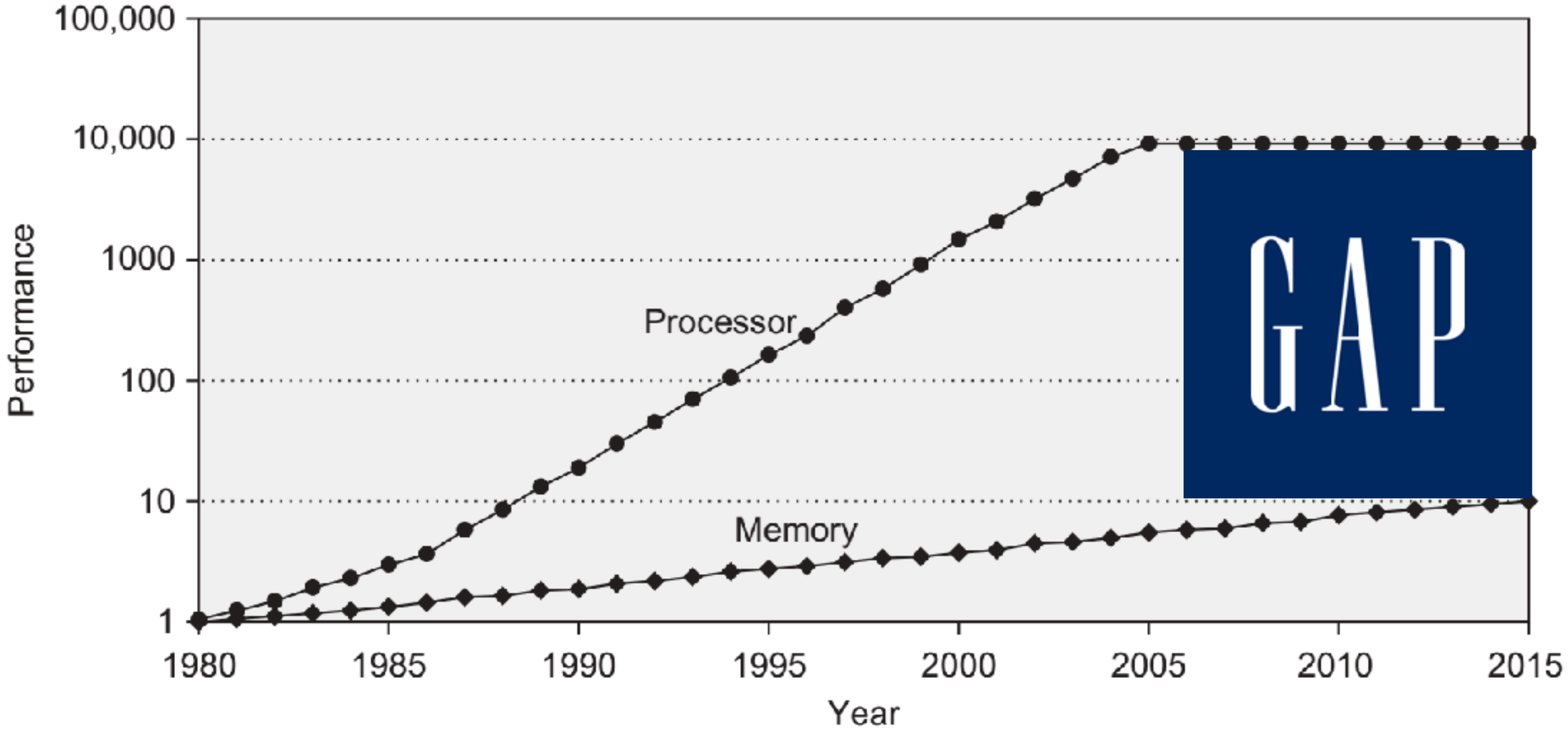
```
Contents of section .text:
 0000 f30f1efa 554889e5 c745f800 000000c7
 0010 45fc0000 0000c745 f8000000 00eb1e8b
 0020 45f84898 488d1425 00000000 488d0500
 0030 0000008b 04020145 fc8345f8 01837df8
 0040 137edcb8 00000000 5dc3
```

Instructions

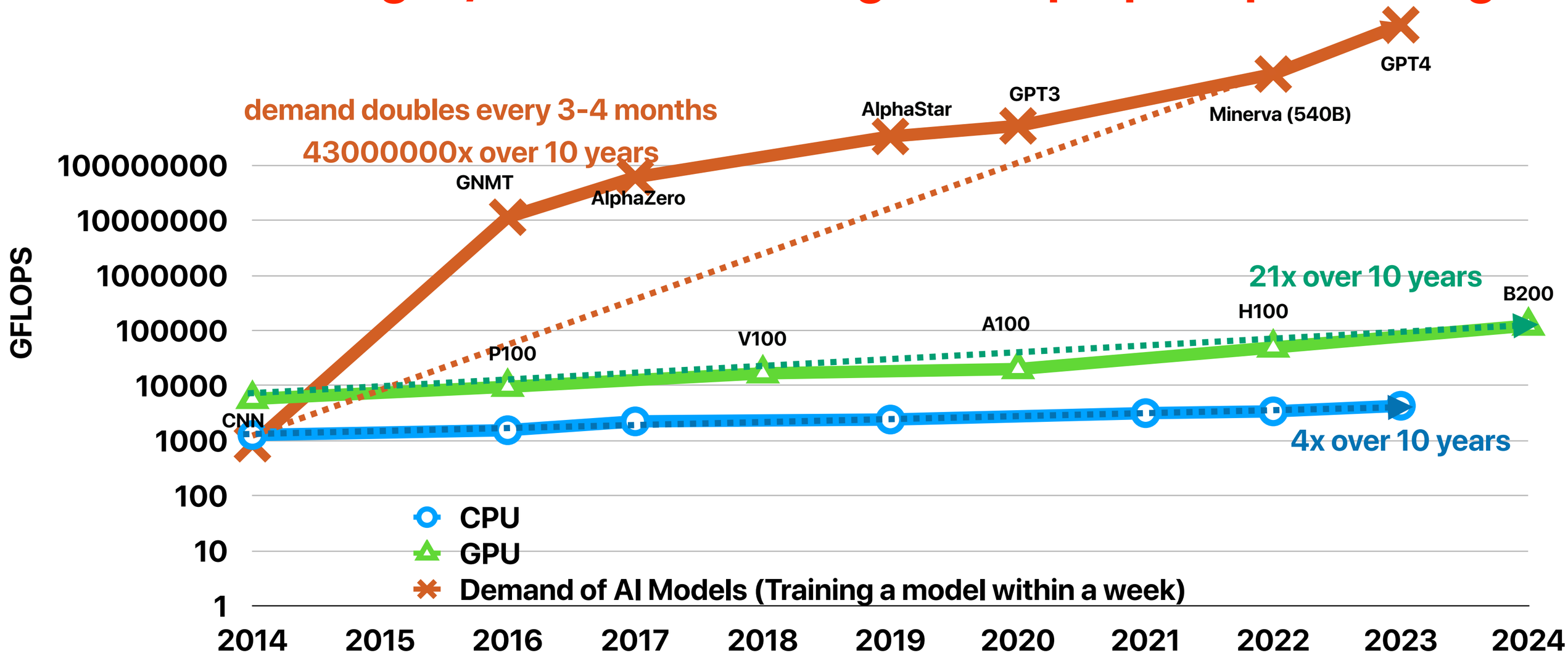
x86 ISA: the abstracted machine



Recap: Performance gap between Processor/Memory



Mis-matching AI/ML demand and general-purpose processing



https://ourworldindata.org/grapher/artificial-intelligence-training-computation

Outline

- Definition of “Performance”
- The performance equation
- The metric matters
- What affects each factor in “Performance Equation”

Best Nation

Schools in the National Univers
a full range of undergraduate r
producing groundbreaking res

To unlock full rankings, SAT/A

SUMMARY

443

Scho

Loca

Rank

Nat

Best Compute

Best Computer Science Schools

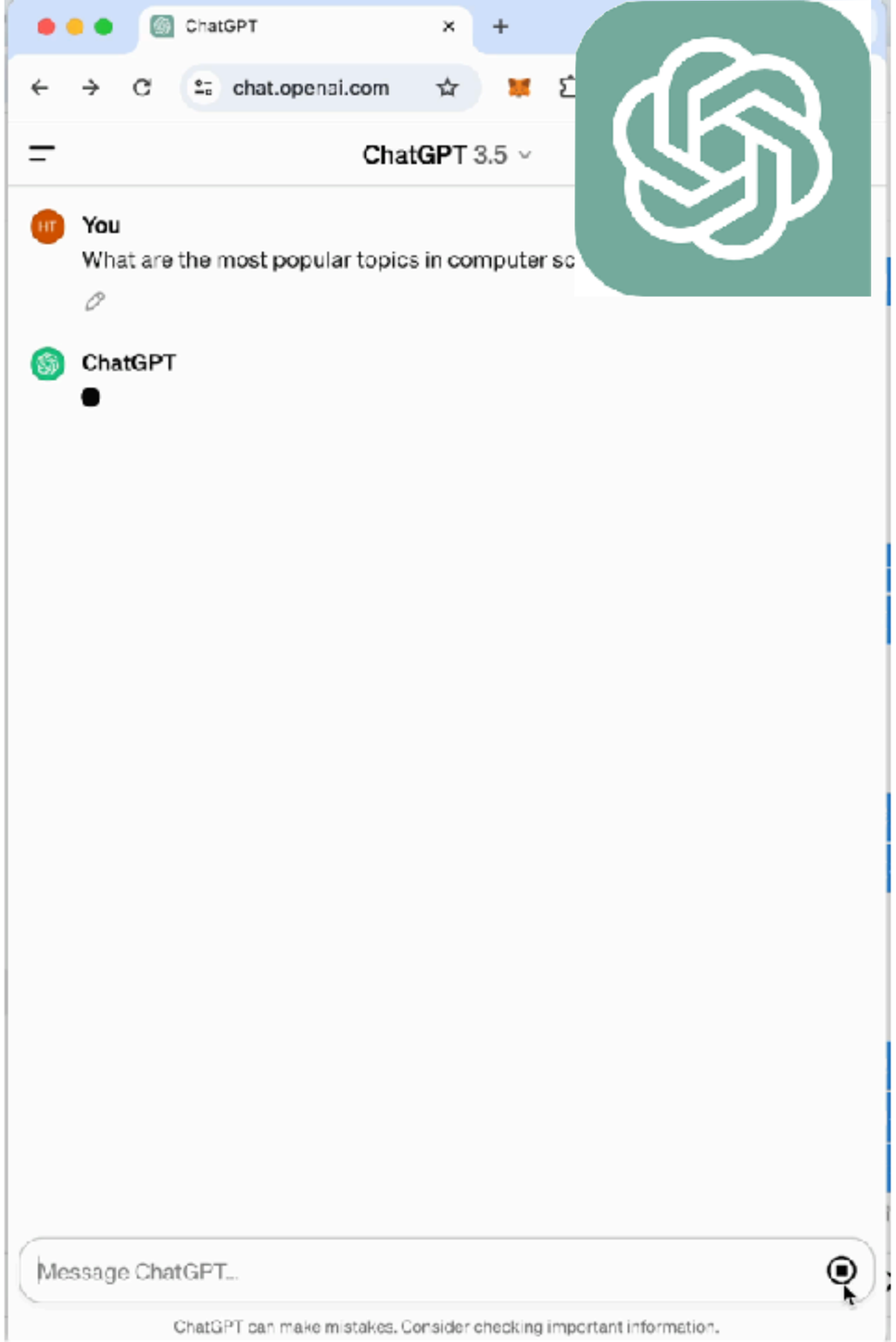
Ranked in 2022, part of Best Science S

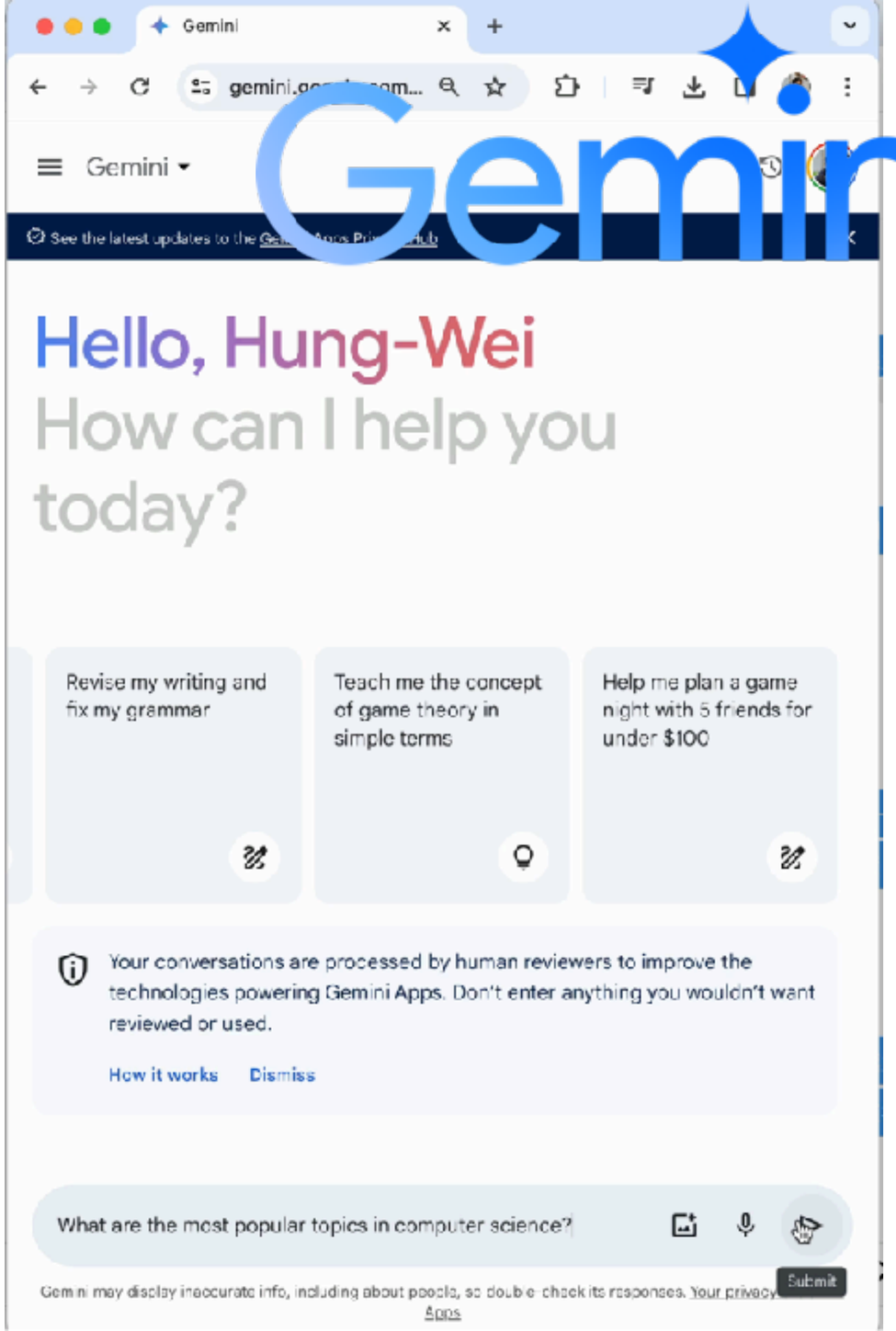
Earning a graduate degree in computer
technology companies and colleges ar
reflects its average rating on a scale fr
institutions. Read the methodology »

UC San Diego Ranked No. 1 Public University by Washington Monthly

Campus celebrated as a leader in social mobility, research and public

What does it really mean by “better” performance


ChatGPT
chat.openai.com
ChatGPT 3.5
You
What are the most popular topics in computer sc
ChatGPT
Message ChatGPT...
ChatGPT can make mistakes. Consider checking important information.


Gemini
Gemini
Hello, Hung-Wei
How can I help you today?
Revise my writing and fix my grammar
Teach me the concept of game theory in simple terms
Help me plan a game night with 5 friends for under $100
Your conversations are processed by human reviewers to improve the technologies powering Gemini Apps. Don't enter anything you wouldn't want reviewed or used.
How it works
Dismiss
What are the most popular topics in computer science?
Submit

Peer instruction

- Before the lecture — You need to complete the required **reading**
- During the lecture — I'll bring in activities to ENGAGE you in exploring your understanding of the material
  - Popup questions
  - Individual **thinking** — use polls in Zoom to express your opinion
  - Group **discussion**
    - Breakout rooms based on your residential colleges!
    - Use polls in Zoom to express your group's opinion
  - Whole-classroom **discussion** — we would like to hear from you

Read **Think** **Discuss**

Important performance metrics

- End-to-end latency — how much time the program/operation takes from the beginning to the end
- Response time — how much time the user starts to feel the program is running/finishing
- Throughput/bandwidth — the average amount of work/data can the program/system deliver within the execution time
- Energy consumption — the aggregated power during the execution time
- Cost of operation — the amount of money necessary for finishing an operation
- Quality of results — the human perception of the execution result
- Power consumption — the heat generation produced by the circuit

Important performance metrics

- End-to-end latency — how much **time** the program/operation takes from the beginning to the end
- Response time — how much **time** the user starts to feel the program is running/finishing
- Throughput/bandwidth — the average amount of work/data can the program/system deliver within the execution **time**
- Energy consumption — the aggregated power during the execution **time**
- Cost of operation — the amount of money necessary for finishing an operation (related to **time**)
- Quality of results — the human perception of the execution result
- Power consumption — the heat generation produced by the circuit

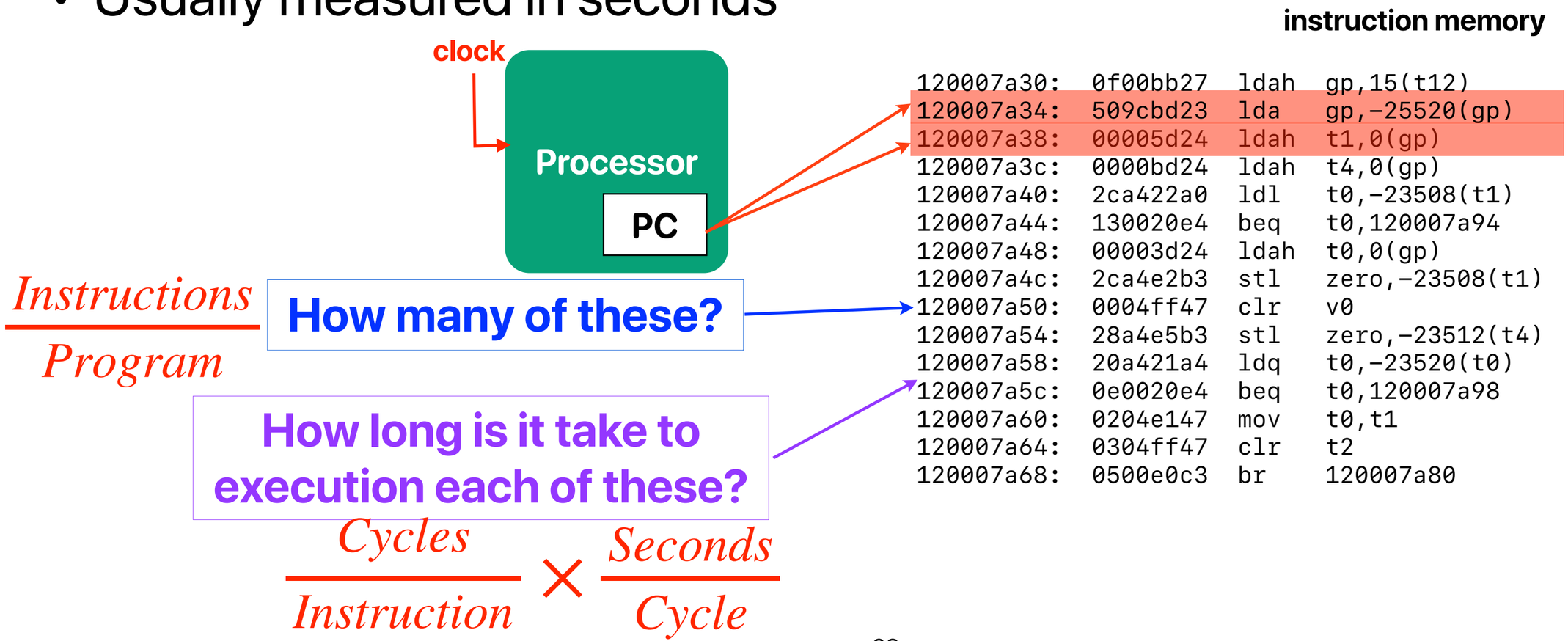
Takeaways: What does “perfect” mean?

- Latency is the most fundamental performance metric

Let’s start with “end-to-end latency” as the default metric — how long it takes to execute a program?

Execution Time

- The simplest kind of performance
- Shorter execution time means better performance
- Usually measured in seconds



CPU Performance Equation

$$Performance = \frac{1}{Execution\ Time}$$

$$Execution\ Time = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Seconds}{Cycle}$$

$$ET = IC \times CPI \times CT$$

$$1GHz = 10^9 Hz = \frac{1}{10^9} sec\ per\ cycle = 1\ ns\ per\ cycle$$

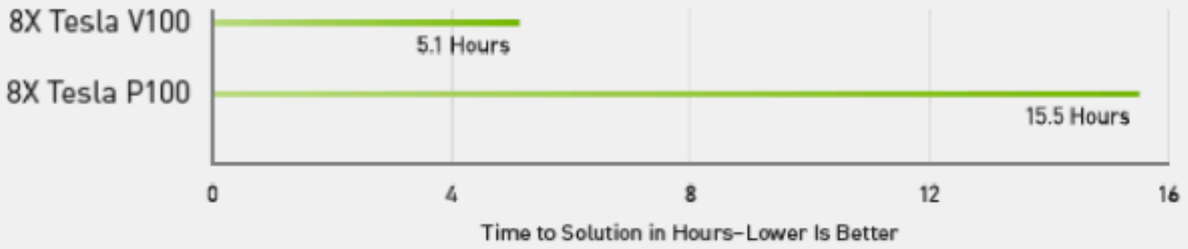
$$\frac{1}{Frequency(i.e., clock\ rate)}$$

Takeaways: What does “perfect” mean?

- Latency is the most fundamental performance metric
- Instruction count, cycles per instruction, cycle time define the latency of execution on CPUs

Choose the right metric — Latency v.s. Throughput/Bandwidth

Deep Learning Training in Less Than a Workday



Server Config: Dual Xeon E5-2699 v4 2.6 GHz | 8X NVIDIA® Tesla® P100 or V100 | ResNet-50 Training on MXNet for 90 Epochs with 1.28M ImageNet Dataset.

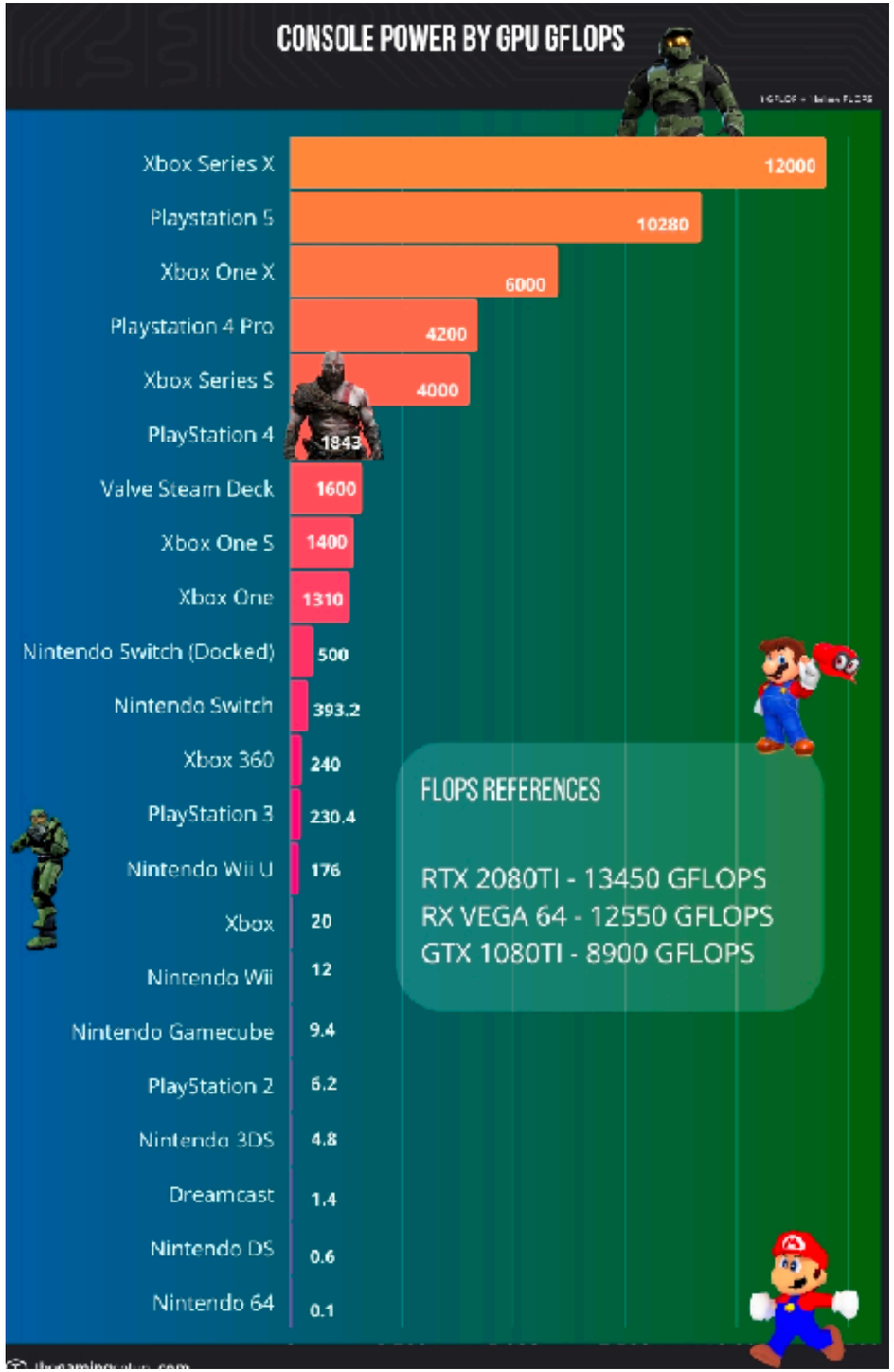
AI TRAINING

From recognizing speech to training virtual personal assistants and teaching autonomous cars to drive, data scientists are taking on increasingly complex challenges with AI. Solving these kinds of problems requires training deep learning models that are exponentially growing in complexity, in a practical amount of time.

With 640 Tensor Cores, Tesla V100 is the world's first GPU to break the 100 teraFLOPS (TFLOPS) barrier of deep learning performance. The next generation of NVIDIA NVLink™ connects multiple V100 GPUs at up to 300 GB/s to create the world's most powerful computing servers. AI models that would consume weeks of computing resources on previous systems can now be trained in a few days. With this dramatic reduction in training time, a whole new world of problems will now be solvable with AI.

TFLOPS (Tera FLoating-point Operations Per Second)

	TFLOPS	clock rate
Switch	1	921 MHz
PS5	10.28	2.23 GHz
XBox Series X	12	1.825 GHz
GeForce RTX 3090	40	1.395 GHz



Let's measure the FLOPS of matrix multiplications

```
for(i = 0; i < ARRAY_SIZE; i++) {
  for(j = 0; j < ARRAY_SIZE; j++) {
    for(k = 0; k < ARRAY_SIZE; k++) {
      c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

Floating point operations per second (FLOP"S"):

$$FLOPS = \frac{2^{34}}{ET_{seconds}}$$

Floating point operations (FLOP"s"):

$$i \times j \times k \times 2$$

Given $i = j = k = 2048$

$2^{3\times11} \times 2 = 2^{34}$ **FLOPs in total**

TFLOPS (Tera FLoating-point Operations Per Second)

$$TFLOPS = \frac{\#\ of\ floating\ point\ instructions \times 10^{-12}}{Exection\ Time}$$

Demo: matmul on GPU

Size	Latency	Relative Latency	Throughput (Output Numbers Per Second)	Relative Throughput
16x16x16	~ 0.09ms	1	0.09ms/256	1
32x32x32	~ 0.09ms	1	0.09ms/1024	4
64x64x64	~ 0.09ms	1	0.09ms/4096	16

Larger throughput doesn’t mean shorter latency!

Is TFLOPS (Tera FLoating-point Operations Per Second) a good metric?

$$TFLOPS = \frac{\#\ of\ floating\ point\ instructions \times 10^{-12}}{Exection\ Time}$$

$$= \frac{IC \times \%\ of\ floating\ point\ instructions \times 10^{-12}}{IC \times CPI \times CT}$$

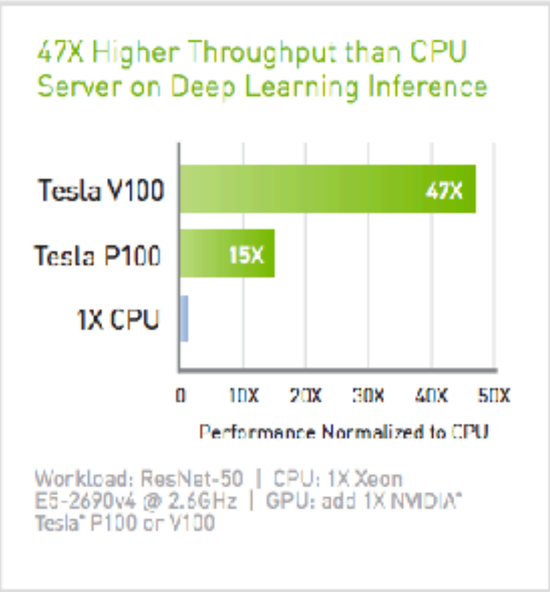
$$= \frac{\%\ of\ floating\ point\ instructions \times 10^{-12}}{CPI \times CT}$$

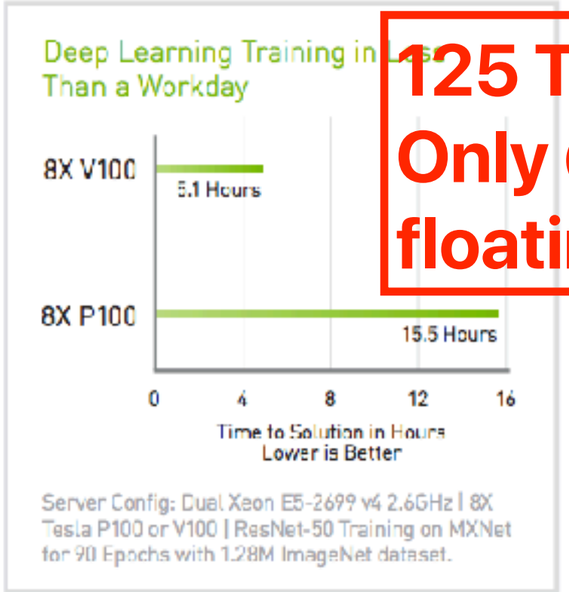
IC is gone!

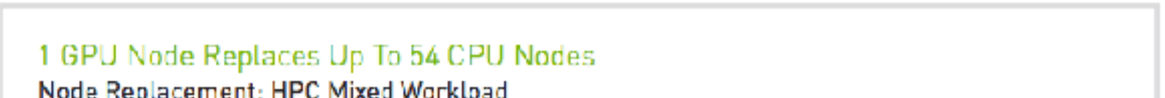
- Cannot compare different ISA/compiler
  - What if the compiler can generate code with fewer instructions?
  - What if new architecture has more IC but also lower CPI?
- Does not make sense if the application is not floating point intensive

The Most Advanced Data Center GPU Ever Built.

NVIDIA® Tesla® V100 is the world's most advanced data center GPU ever built to accelerate AI, HPC, and graphics. Powered by NVIDIA Volta, the latest GPU architecture, Tesla V100 offers the performance of up to 100 CPUs in a single GPU—enabling data scientists, researchers, and engineers to tackle challenges that were once thought impossible.

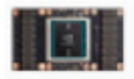






125 TFLOPS Only @ 16-bit floating point

SPECIFICATIONS



	Tesla V100 PCIe	Tesla V100 SXM2
GPU Architecture	NVIDIA Volta	
NVIDIA Tensor Cores	640	
NVIDIA CUDA® Cores	5,120	
Double-Precision Performance	7 TFLOPS	7.8 TFLOPS
Single-Precision Performance	14 TFLOPS	15.7 TFLOPS
Tensor Performance	112 TFLOPS	125 TFLOPS
GPU Memory	32GB /16GB HBM2	
Memory Bandwidth	900GB/sec	
ECC	Yes	
Interconnect Bandwidth	32GB/sec	300GB/sec
System Interface	PCIe Gen3	NVIDIA NVLink
Form Factor	PCIe Full Height/Length	SXM2
Max Power		

Latency v.s. Bandwidth/Throughput

- Latency — the amount of time to finish an operation
  - End-to-end execution time of “something”
  - Access time
  - Response time
- Throughput — the amount of work can be done within a given period of time (typically “something” per “timeframe” or the other way around)
  - Bandwidth (MB/Sec, GB/Sec, Mbps, Gbps)
  - IOPs (I/O operations per second)
  - FLOPs (Floating-point operations per second)
  - IPS (Inferences per second)

Takeaways: What does “perfect” mean?

- Latency is the most fundamental performance metric
- Classic CPU performance equation —Instruction count (IC), cycles per instruction (CPI), cycle time (CT) define the latency of execution on CPUs
- Performance metrics without considering all three factors in the classic performance equation can mislead — anything throughput typically miss one of them

What Affects Each Factor in Performance Equation

Demo — programmer & performance

A

```
for(i = 0; i < ARRAY_SIZE; i++)
{
  for(j = 0; j < ARRAY_SIZE; j++)
  {
    c[i][j] = a[i][j]+b[i][j];
  }
}
```

B

```
for(j = 0; j < ARRAY_SIZE; j++)
{
  for(i = 0; i < ARRAY_SIZE; i++)
  {
    c[i][j] = a[i][j]+b[i][j];
  }
}
```

$O(n^2)$ **Complexity** $O(n^2)$

Instruction Count?

Clock Rate

CPI

Demo — programmer & performance

A

```
for(i = 0; i < ARRAY_SIZE; i++)
{
  for(j = 0; j < ARRAY_SIZE; j++)
  {
    c[i][j] = a[i][j]+b[i][j];
  }
}
```

B

```
for(j = 0; j < ARRAY_SIZE; j++)
{
  for(i = 0; i < ARRAY_SIZE; i++)
  {
    c[i][j] = a[i][j]+b[i][j];
  }
}
```

$O(n^2)$	Complexity	$O(n^2)$
Same	Instruction Count?	Same
Same	Clock Rate	Same
???	CPI	???

Use “performance counters” to figure out!

- Modern processors provides performance counters
  - instruction counts
  - cache accesses/misses
  - branch instructions/mis-predictions
- How to get their values?
  - You may use “perf stat” in linux
  - You may use Instruments —> Time Profiler on a Mac
  - Intel’s vtune — only works on Windows w/ intel processors
  - You can also create your own functions to obtain counter values

Programmers can also set the cycle time

https://software.intel.com/sites/default/files/comment/1716807/how-to-change-frequency-on-linux-pub.txt

```
==========================================
Subject: setting CPU speed on running linux system


If the OS is Linux, you can manually control the CPU speed by reading and writing some virtual files in the "/proc"

1.) Is the system capable of software CPU speed control?
If the "directory" /sys/devices/system/cpu/cpu0/cpufreq exists, speed is controllable.
-- If it does not exist, you may need to go to the BIOS and turn on EIST and any other C and P state control and vi



2.) What speed is the box set to now?
Do the following:
$ cd /sys/devices/system/cpu
$ cat ./cpu0/cpufreq/cpuinfo_max_freq
3193000
$ cat ./cpu0/cpufreq/cpuinfo_min_freq
1596000

3.) What speeds can I set to?
Do
$ cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_available_frequencies
It will list highest settable to lowest; example from my NHM "Smackover" DX58SO HEDT board, I see:
3193000 3192000 3059000 2926000 2793000 2660000 2527000 2394000 2261000 2128000 1995000 1862000 1729000 159600
You can choose from among those numbers to set the "high water" mark and "low water" mark for speed.  If you set "h


4.) Show me how to set all to highest settable speed!
Use the following little sh/ksh/bash script:
$ cd /sys/devices/system/cpu  # a virtual directory made visible by device drivers
$ newSpeedTop=`awk '{print $1}' ./cpu0/cpufreq/scaling_available_frequencies`
$ newSpeedLow=$newSpeedTop  # make them the same in this example
$ for c in ./cpu[0-9]* ; do
>   echo $newSpeedTop >${c}/cpufreq/scaling_max_freq
>   echo $newSpeedLow >${c}/cpufreq/scaling_min_freq
> done
$

5.) How do I return to the default - i.e. allow machine to vary from highest to lowest?
Edit line # 3 of the script above, and re-run it.  Change the line:
$ newSpeedLow=$newSpeedTop  # make them the same in this example
```

How programmer affects performance?

- Performance equation consists of the following three factors
  1. IC
  2. CPI
  3. CT

  How many can a **programmer** affect?

  A. 0
  B. 1
  C. 2
  D. 3

Takeaways: What matters?

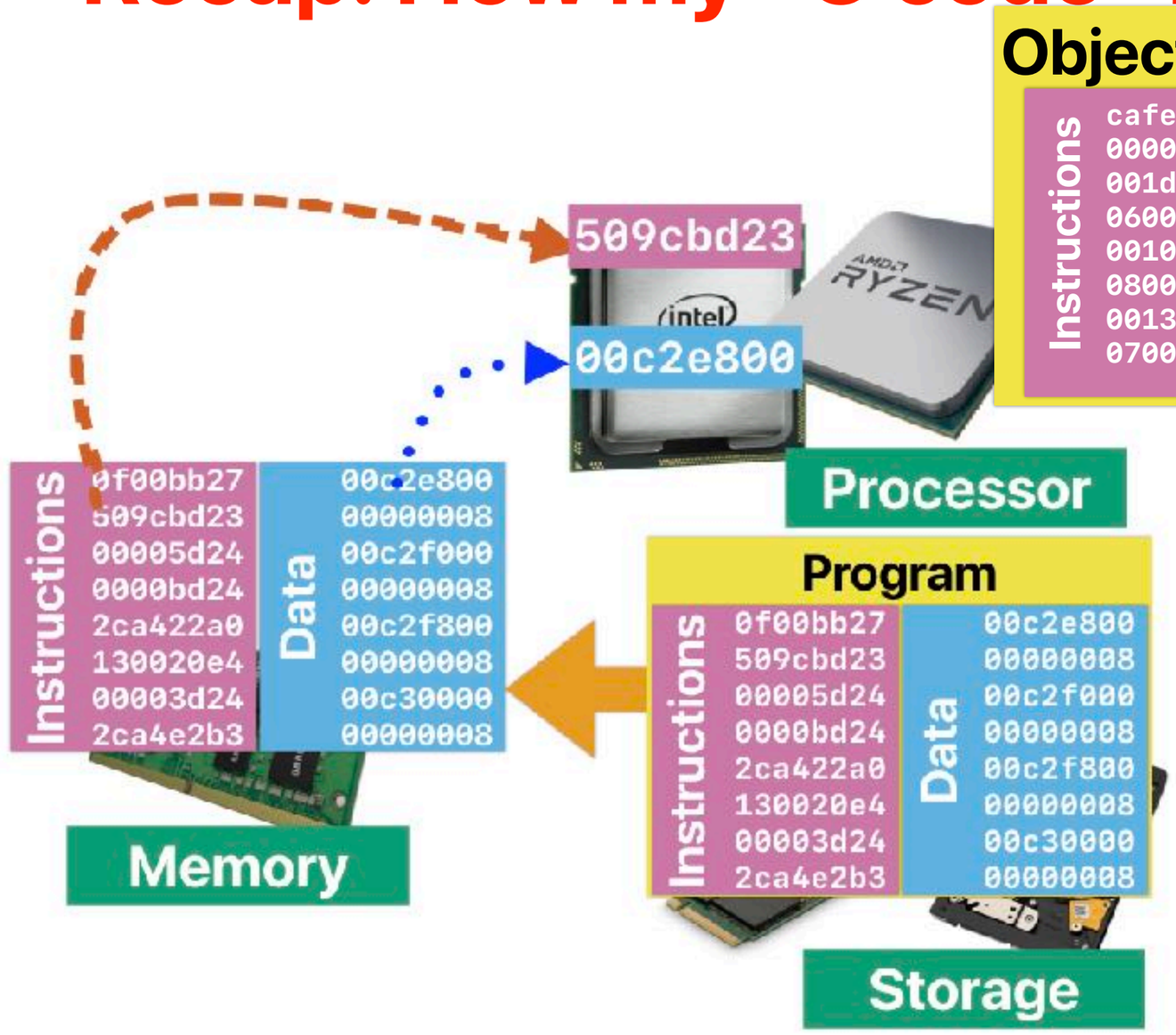
- Programmers can control all three factors in the classic performance equation

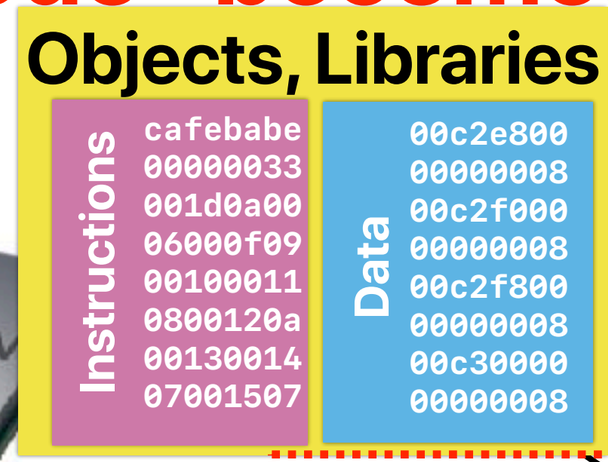
Programming languages

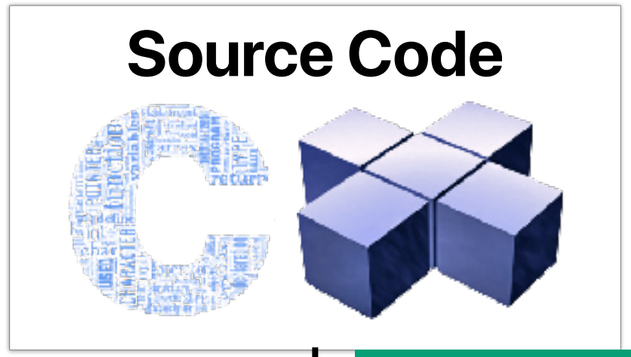
- How many instructions are there in “Hello, world!”

	Instruction count	LOC	Ranking
C	600k	6	1
C++	3M	6	2
Java	~145M	8	5
Perl	~12M	4	3
Python	~33M	1	4
GO (Interpreter)	~1200M	1	6
GO (Compiled)	~1.7M	1	
Rust	~1.4M	1	

Recap: How my "C code" becomes a "program"

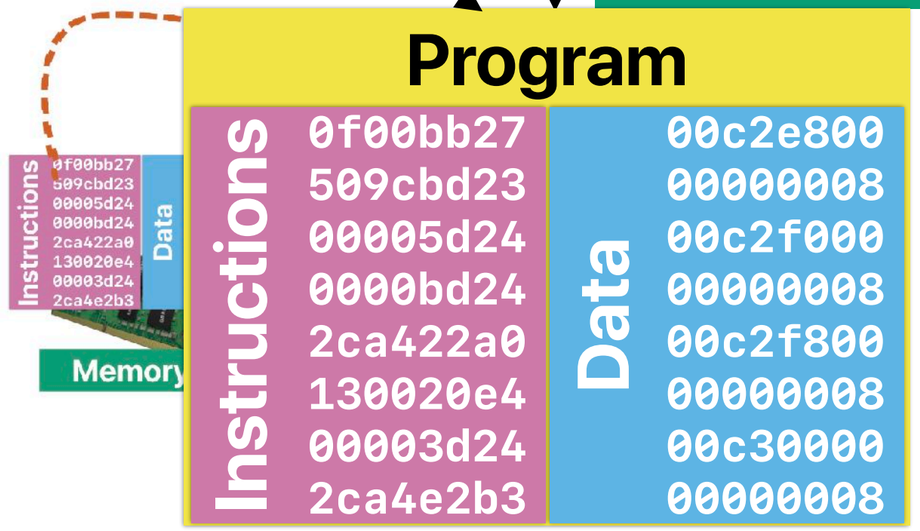






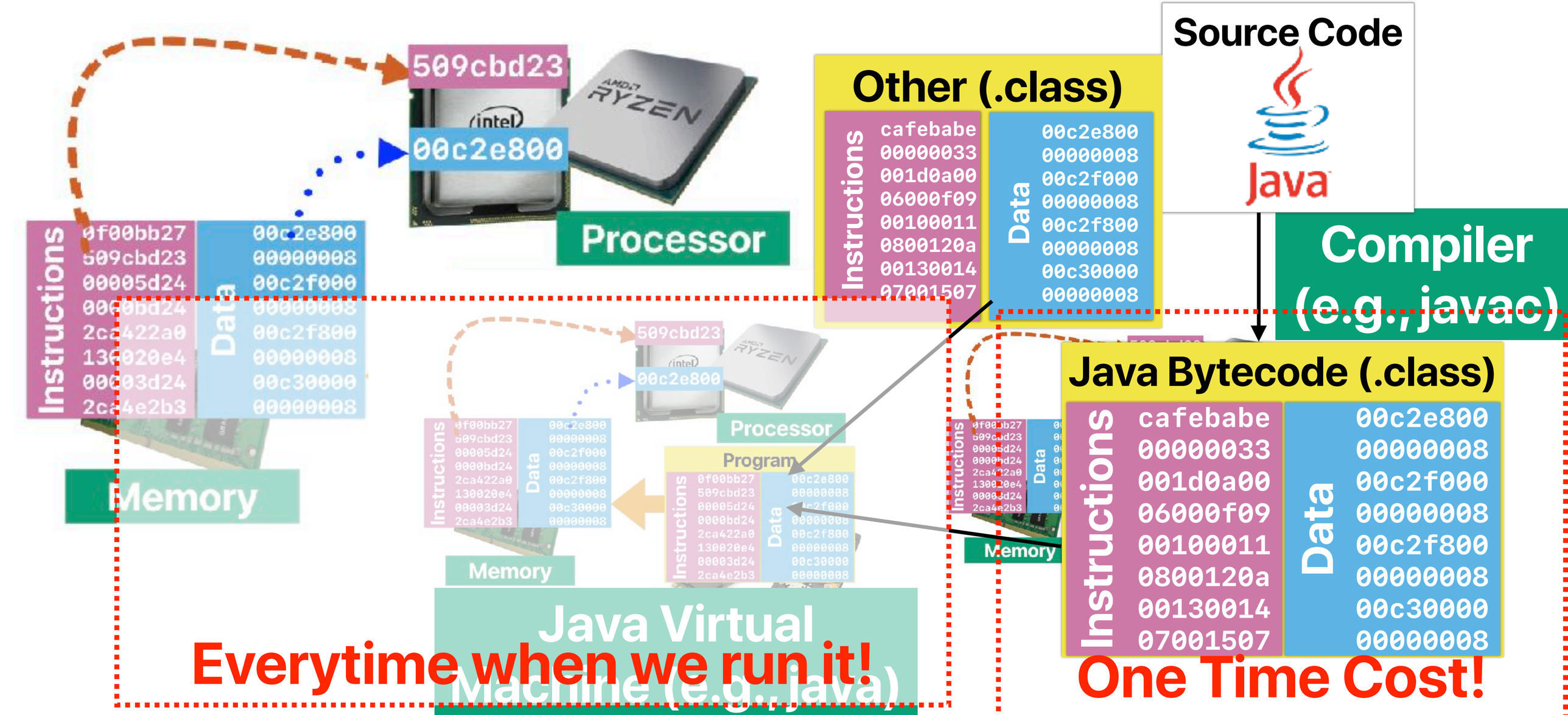
Linker

Compiler (e.g., gcc)

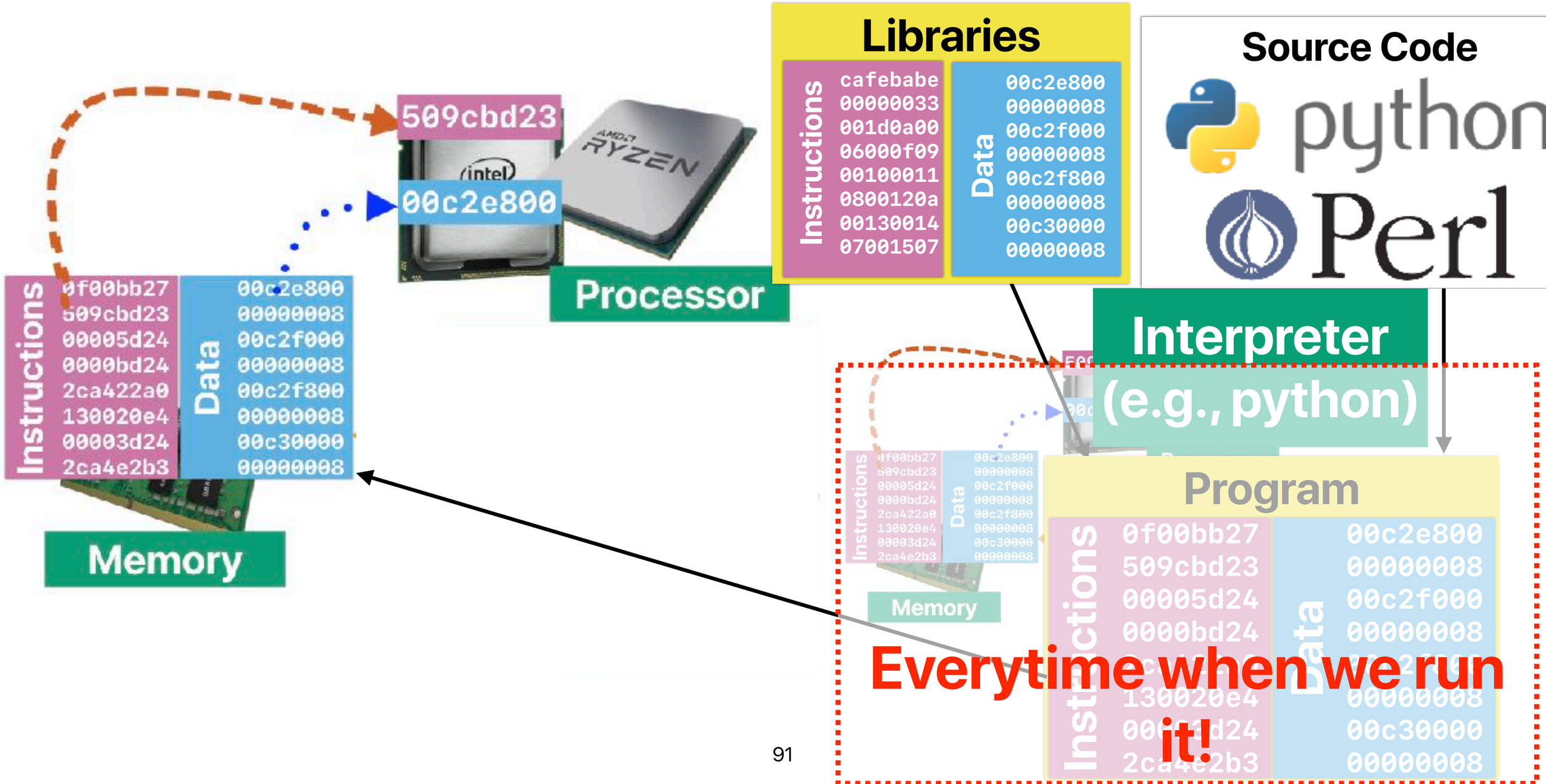


One Time Cost!

Recap: How my "Java code" becomes a "program"



Recap: How my "Python code" becomes a "program"



Takeaways: What matters?

- Programmers can control all three factors in the classic performance equation
- Different programming languages can generate machine operations with different orders of magnitude performance — programmers need to make wise choice of that!

Revisited the demo with compiler optimizations!

- gcc has different optimization levels.
  - -O0 — no optimizations
  - -O3 — typically the best-performing optimization

A

```
for(i = 0; i < ARRAY_SIZE; i++)
{
  for(j = 0; j < ARRAY_SIZE; j++)
  {
    c[i][j] = a[i][j]+b[i][j];
  }
}
```

B

```
for(j = 0; j < ARRAY_SIZE; j++)
{
  for(i = 0; i < ARRAY_SIZE; i++)
  {
    c[i][j] = a[i][j]+b[i][j];
  }
}
```

Takeaways: What matters?

- Programmers can control all three factors in the classic performance equation
- Different programming languages can generate machine operations with different orders of magnitude performance — programmers need to make wise choice of that!
- Compiler optimization can help — but programmers need to write code in a way facilitate optimizations!

How about complexity?

How about “computational complexity”

- Algorithm complexity provides a good estimate on the performance if —
  - Every instruction takes exactly the same amount of time
  - Every operation takes exactly the same amount of instructions

These are unlikely to be true

Summary of CPU Performance Equation

$$Performance = \frac{1}{Execution\ Time}$$

$$Execution\ Time = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Seconds}{Cycle}$$

$$ET = IC \times CPI \times CT$$

- IC (Instruction Count)
  - ISA, Compiler, algorithm, programming language, **programmer**
- CPI (Cycles Per Instruction)
  - Machine Implementation, microarchitecture, compiler, application, algorithm, programming language, **programmer**
- Cycle Time (Seconds Per Cycle)
  - Process Technology, microarchitecture, **programmer**

Takeaways: What matters?

- Programmers can control all three factors in the classic performance equation
- Different programming languages can generate machine operations with different orders of magnitude performance — programmers need to make wise choice of that!
- Compiler optimization can help — but programmers need to write code in a way facilitate optimizations!
- Complexity does not provide good assessment on real machines due to the idealized assumptions

Quantitive Analysis of “Better”

Takeaways: What matters?

- Programmers can control all three factors in the classic performance equation
- Different programming languages can generate machine operations with different orders of magnitude performance — programmers need to make wise choice of that!
- Compiler optimization can help — but programmers need to write code in a way facilitate optimizations!
- Complexity does not provide good assessment on real machines due to the idealized assumptions
- The only definition of Y is Speedup times faster than X —

$$Speedup = \frac{Execution\ Time_X}{Execution\ Time_Y}$$

Amdahl's Law — and It's Implication in the Multicore Era

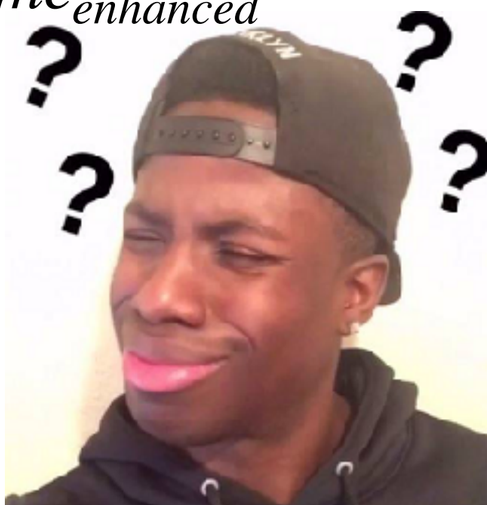
Amdahl's Law

$$Speedup_{enhanced}(f, s) = \frac{1}{(1 - f) + \frac{f}{s}}$$

f — The fraction of time in the original program

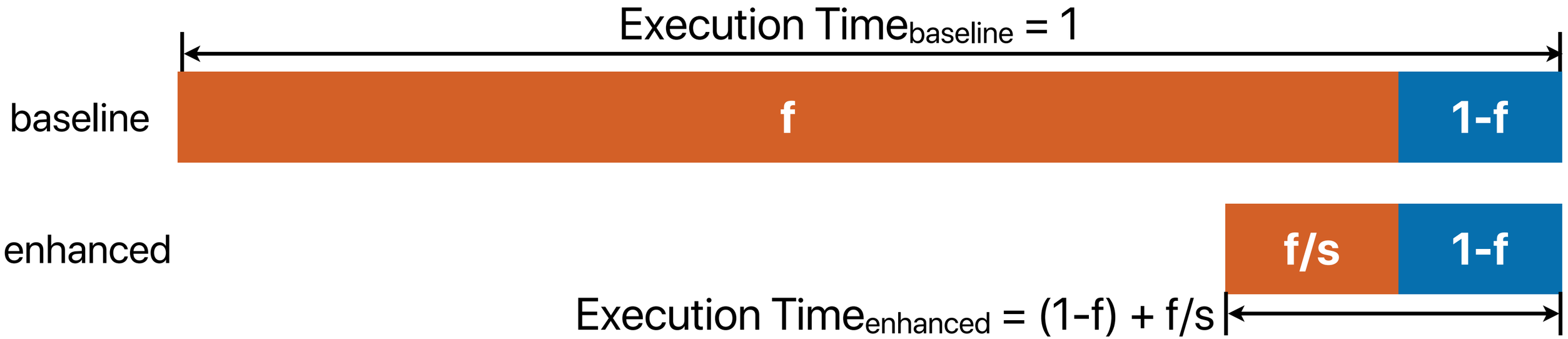
s — The speedup we can achieve on f

$$Speedup_{enhanced} = \frac{Execution\ Time_{baseline}}{Execution\ Time_{enhanced}}$$



Amdahl's Law

$$Speedup_{enhanced}(f, s) = \frac{1}{(1 - f) + \frac{f}{s}}$$



$$Speedup_{enhanced} = \frac{Execution\ Time_{baseline}}{Execution\ Time_{enhanced}} = \frac{1}{(1 - f) + \frac{f}{s}}$$

Amdahl's Law on Multiple Optimizations

- We can apply Amdahl's law for multiple optimizations
- These optimizations must be dis-joint!
  - If optimization #1 and optimization #2 are dis-joint:

$$Speedup_{enhanced}(f_{Opt1}, f_{Opt2}, s_{Opt1}, s_{Opt2}) = \frac{1}{(1 - f_{Opt1} - f_{Opt2}) + \frac{f\_Opt1}{s\_Opt1} + \frac{f\_Opt2}{s\_Opt2}}$$

  - If optimization #1 and optimization #2 are not dis-joint:

$$Speedup_{enhanced}(f_{OnlyOpt1}, f_{OnlyOpt2}, f_{BothOpt1Opt2}, s_{OnlyOpt1}, s_{OnlyOpt2}, s_{BothOpt1Opt2}) = \frac{1}{(1 - f_{OnlyOpt1} - f_{OnlyOpt2} - f_{BothOpt1Opt2}) + + \frac{f\_BothOpt1Opt2}{s\_BothOpt1Opt2} + \frac{f\_OnlyOpt1}{s\_OnlyOpt1} + \frac{f\_OnlyOpt2}{s\_OnlyOpt2}}$$

Amdahl's Law Corollary #1

- The maximum speedup is bounded by

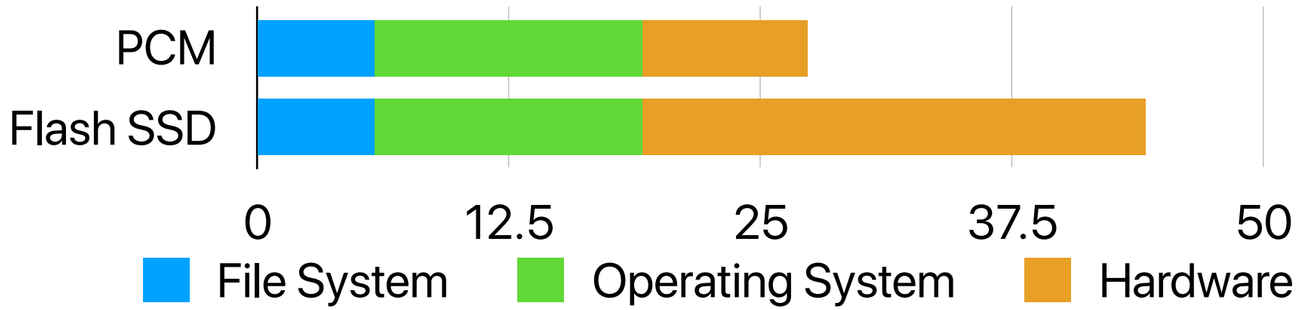
$$Speedup_{max}(f, \infty) = \frac{1}{(1-f) + \frac{f}{\infty}}$$

$$Speedup_{max}(f, \infty) = \frac{1}{(1-f)}$$

Speedup further!

- With the latest flash memory technologies, the system spends 16% of time on accessing the flash, and the software overhead is now 84%. If your company ask you and your team to invent a new memory technology that replaces flash to achieve 2x speedup on loading maps, how much faster the new technology needs to be?

A. ~5x
B. ~10x
C. ~20x
D. ~100x
E. None of the above



$$Speedup_{max}(16\,\%\,, \infty) = \frac{1}{(1 - 16\%)} = 1.19$$

2x is not possible

NEWS

Intel kills the remnants of Optane memory

The speed-boosting storage tech was already on the ropes.

By Michael Crider
Staff Writer, PCWorld | JUL 29, 2022 6:59 AM PDT

Image: Intel

Corollary #1 on Multiple Optimizations

- If we can pick just one thing to work on/optimize

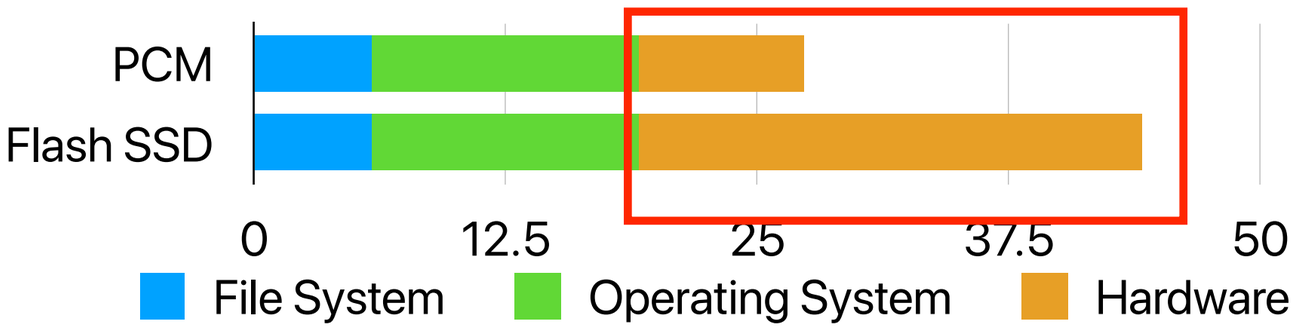
$$Speedup_{max}(f_1, \infty) = \frac{1}{(1 - f_1)}$$
$$Speedup_{max}(f_2, \infty) = \frac{1}{(1 - f_2)}$$
$$Speedup_{max}(f_3, \infty) = \frac{1}{(1 - f_3)}$$
$$Speedup_{max}(f_4, \infty) = \frac{1}{(1 - f_4)}$$

The biggest f_x would lead to the largest $Speedup_{max}$!

Corollary #2 — make the common case fast!

- When f is small, optimizations will have little effect.
- Common == **most time consuming** not necessarily the most frequent
- The uncommon case doesn’t make much difference
- The common case can change based on inputs, compiler options, optimizations you’ve applied, etc.

Speedup further!

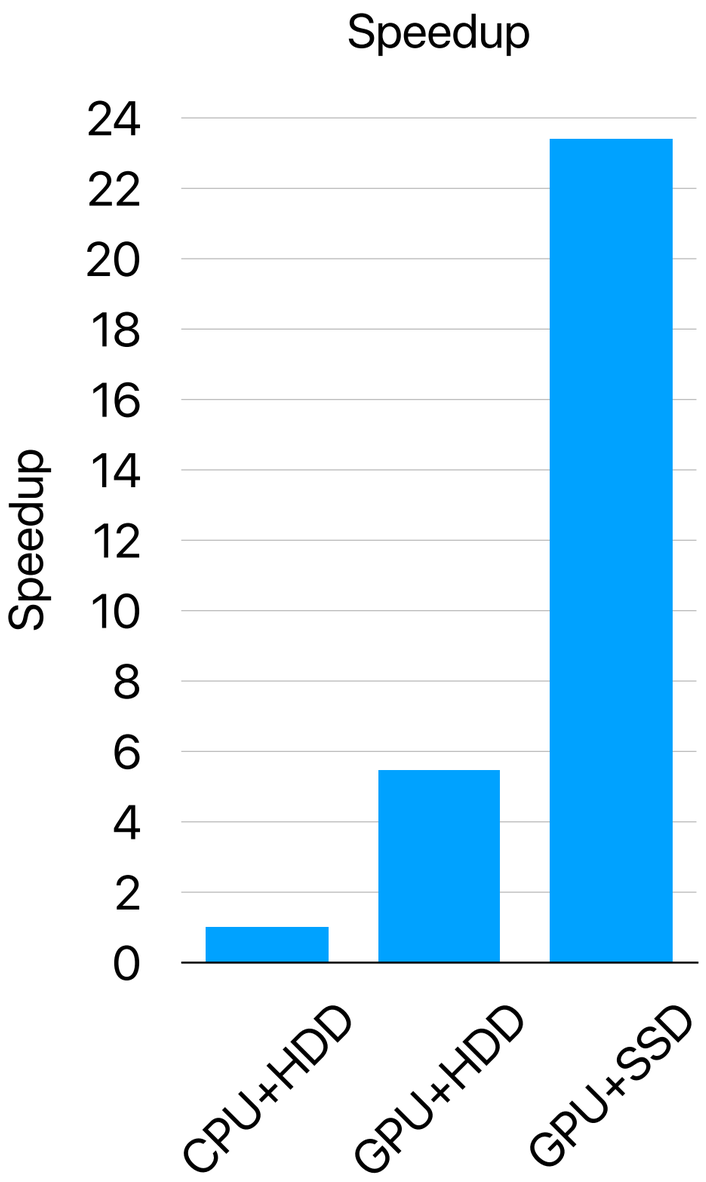


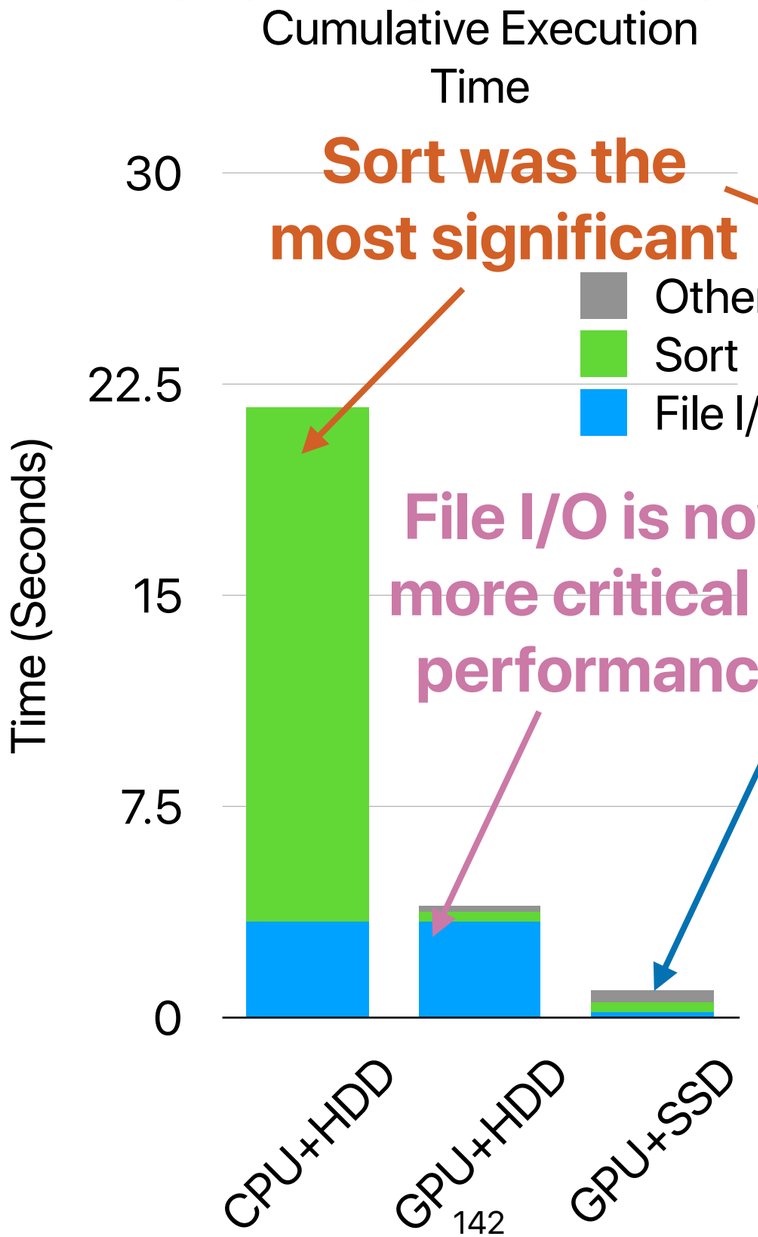
With optimization, the common becomes uncommon.

Identify the most time consuming part

- Compile your program with -pg flag
- Run the program
  - It will generate a gmon.out
  - gprof your_program gmon.out > your_program.prof
- It will give you the profiled result in your_program.prof

Demo — sort

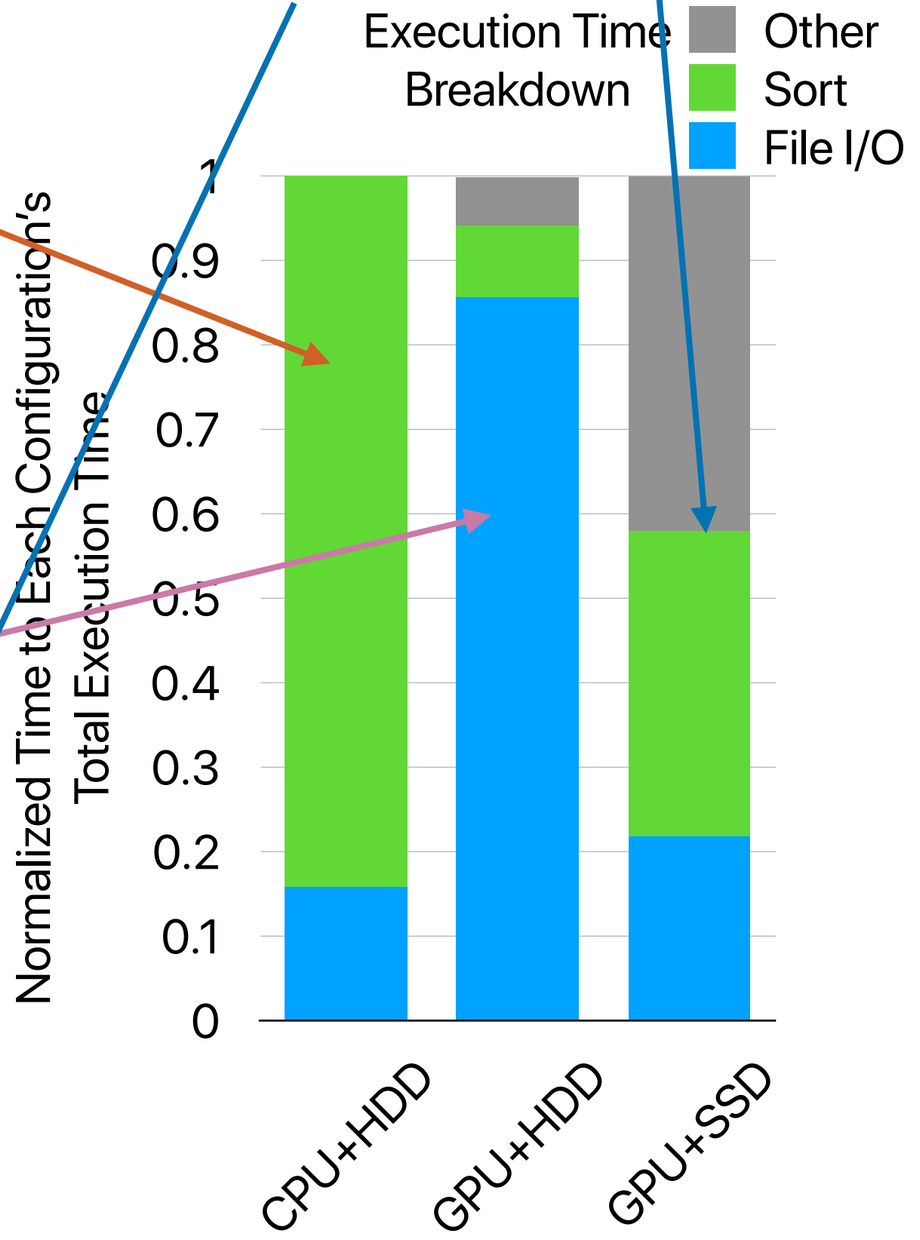




Sort was the most significant

File I/O is now more critical to performance

Something else (e.g., data movement) matters more



If we repeatedly optimizing our design based on Amdahl's law…

- With optimization, the common becomes uncommon.
- An uncommon case will (hopefully) become the new common case.
- Now you have a new target for optimization — You have to revisit "Amdahl's Law" every time you applied some optimization

Amdahl's Law on Multicore Architectures

- Symmetric multicore processor with n cores (if we assume the processor performance scales perfectly)

$$Speedup_{parallel}(f_{parallelizable}, n) = \frac{1}{(1 - f_{parallelizable}) + \frac{f\_parallelizable}{n}}$$

Demo — merge sort v.s. bitonic sort on GPUs

Merge Sort

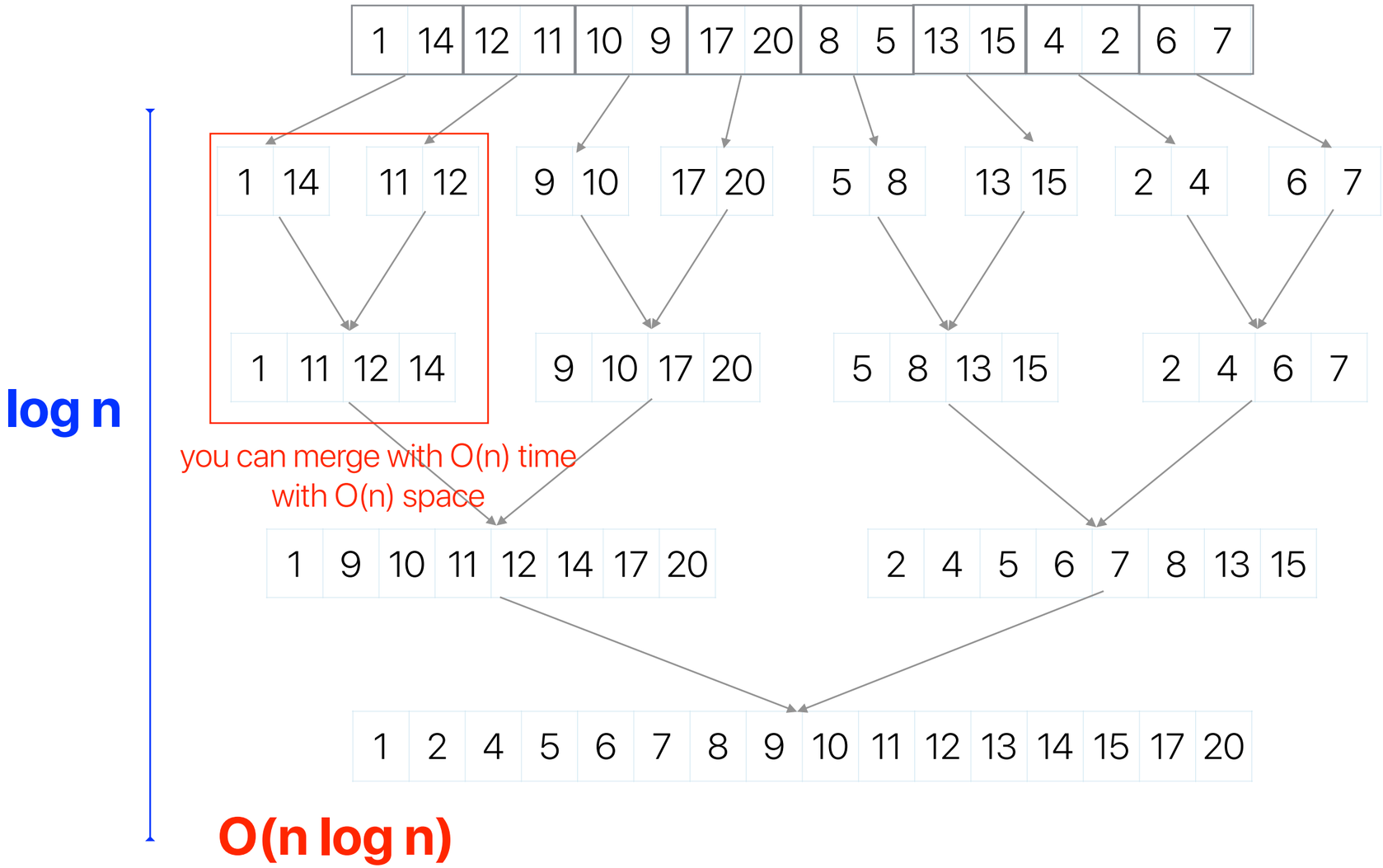
$O(nlog_2n)$

Bitonic Sort

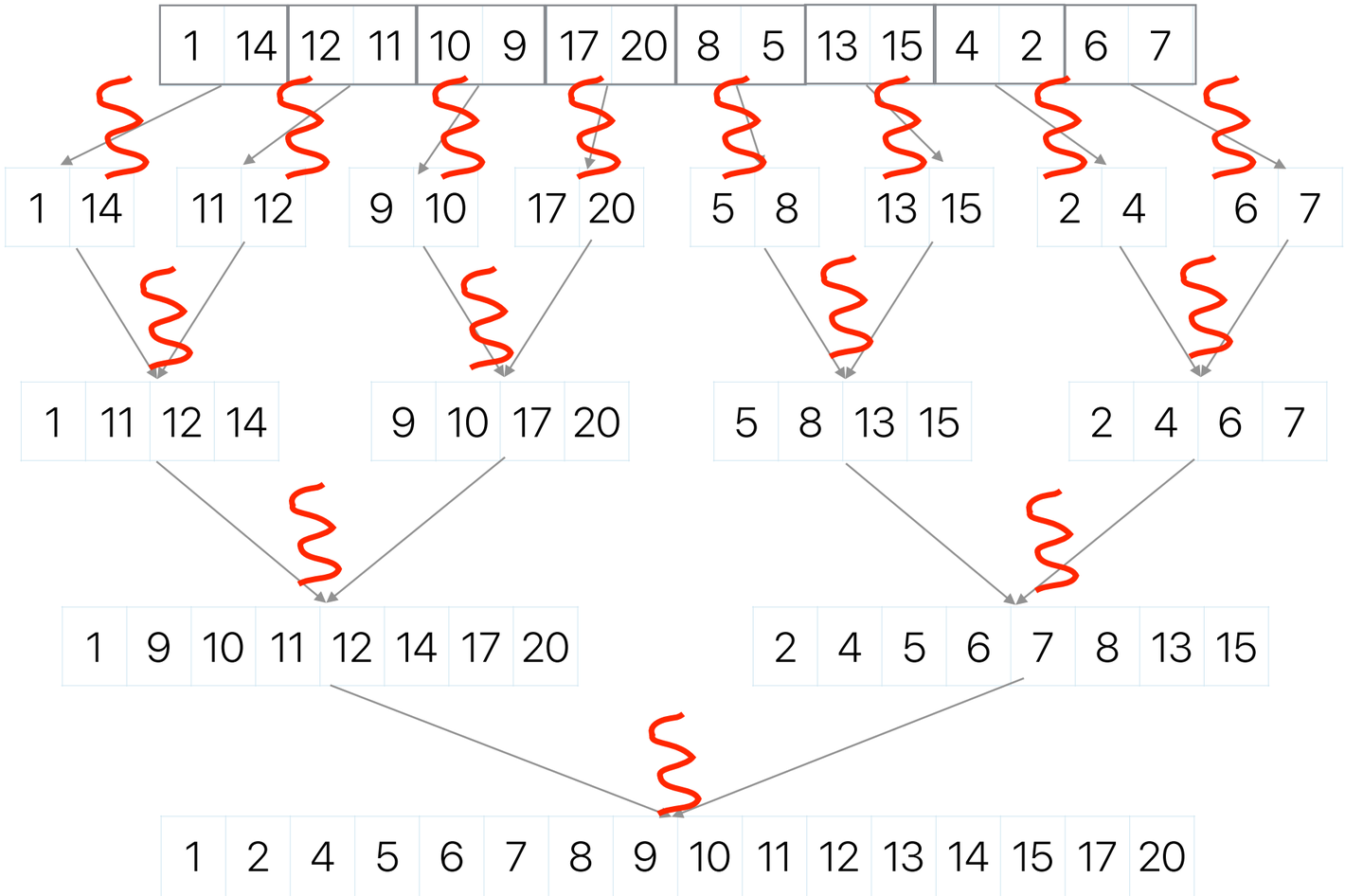
$O(nlog_2^2n)$

```
void BitonicSort() {

    int i,j,k;

    for (k=2; k<=N; k=2*k) {
        for (j=k>>1; j>0; j=j>>1) {
            for (i=0; i<N; i++) {
                int ij=i^j;
                if ((ij)>i) {
                    if ((i&k)==0 && a[i] > a[ij])
                        exchange(i,ij);
                    if ((i&k)!=0 && a[i] < a[ij])
                        exchange(i,ij);
                }
            }
        }
    }
}
```

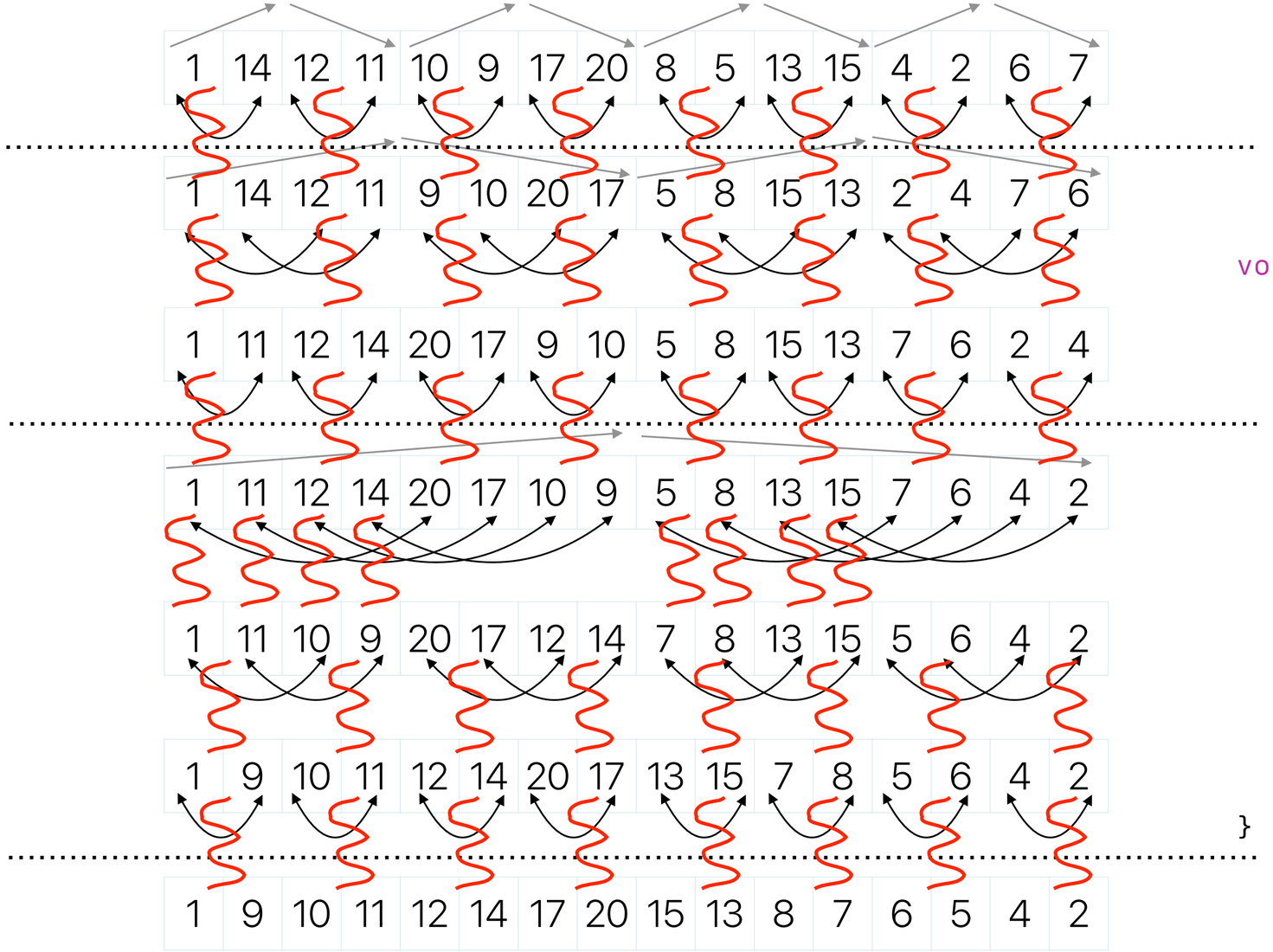
Merge sort


1 14 12 11 10 9 17 20 8 5 13 15 4 2 6 7
1 14 11 12 9 10 17 20 5 8 13 15 2 4 6 7
1 11 12 14 9 10 17 20 5 8 13 15 2 4 6 7
you can merge with O(n) time
with O(n) space
1 9 10 11 12 14 17 20 2 4 5 6 7 8 13 15
1 2 4 5 6 7 8 9 10 11 12 13 14 15 17 20
log n
O(n log n)

Parallel merge sort

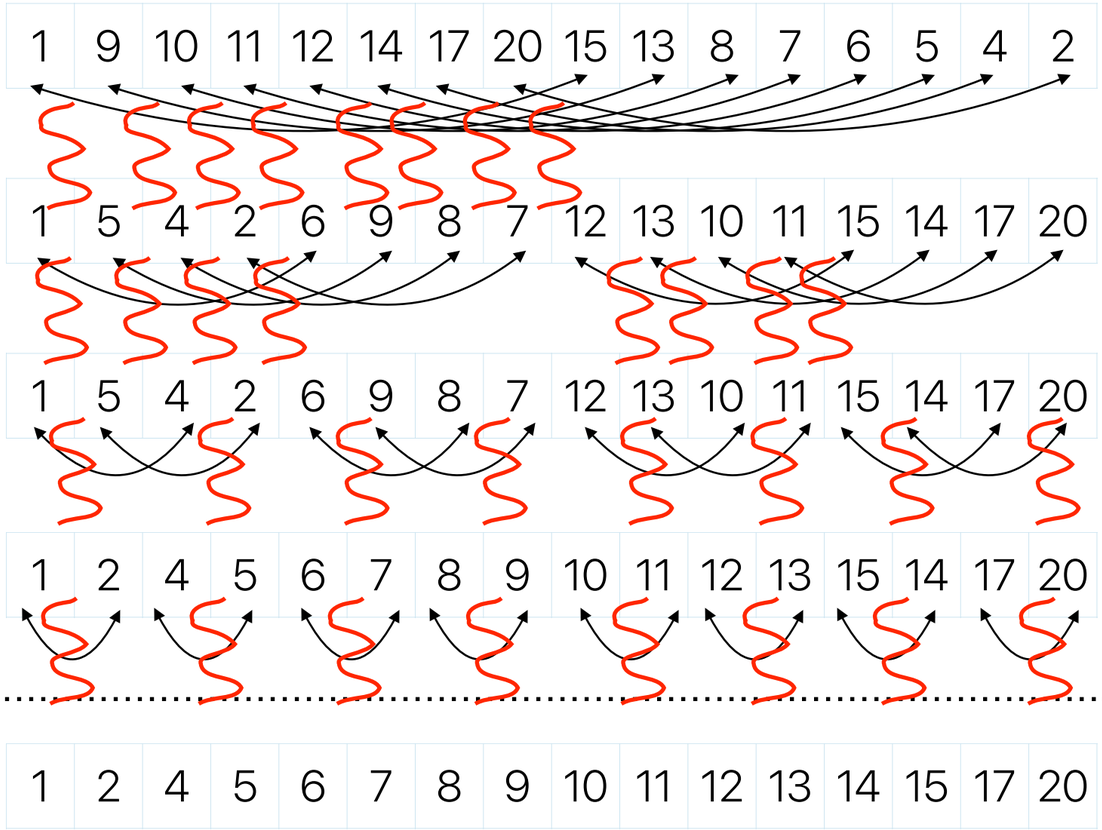


Bitonic sort




```
void BitonicSort() {

    int i,j,k;

    for (k=2; k<=N; k=2*k) {
        for (j=k>>1; j>0; j=j>>1) {
            for (i=0; i<N; i++) {
                int ij=i^j;
                if ((ij)>i) {
                    if ((i&k)==0 && a[i] > a[ij])
                        exchange(i,ij);
                    if ((i&k)!=0 && a[i] < a[ij])
                        exchange(i,ij);
                }
            }
        }
    }
}
```

Bitonic sort (cont.)




```
void BitonicSort() {

    int i,j,k;

    for (k=2; k<=N; k=2*k) {
        for (j=k>>1; j>0; j=j>>1) {
            for (i=0; i<N; i++) {
                int ij=i^j;
                if ((ij)>i) {
                    if ((i&k)==0 && a[i] > a[ij])
                        exchange(i,ij);
                    if ((i&k)!=0 && a[i] < a[ij])
                        exchange(i,ij);
                }
            }
        }
    }
}
```

benefits — in-place merge (no additional space is necessary), very stable comparison patterns

$O(n \log^2 n)$ — hard to beat $n(\log n)$ if you can't parallelize this a lot!

Corollary #4

$$Speedup_{parallel}(f_{parallelizable}, \infty) = \frac{1}{(1 - f_{parallelizable}) + \frac{f\_parallelizable}{\infty}}$$

$$Speedup_{parallel}(f_{parallelizable}, \infty) = \frac{1}{(1 - f_{parallelizable})}$$

- If we can build a processor with unlimited parallelism
  - The complexity doesn't matter as long as the algorithm can utilize all parallelism
  - That's why bitonic sort or MapReduce works!
- **The future trend of software/application design is seeking for more parallelism rather than lower the computational complexity**

Is it the end of computational complexity?

Corollary #5

$$Speedup_{parallel}(f_{parallelizable}, \infty) = \frac{1}{(1 - f_{parallelizable}) + \frac{f\_parallelizable}{\infty}}$$

$$Speedup_{parallel}(f_{parallelizable}, \infty) = \frac{1}{(1 - f_{parallelizable})}$$

- Single-core performance still matters
  - It will eventually dominate the performance
  - If we cannot improve single-core performance further, finding more "parallelizable" parts is more important
  - Algorithm complexity still gives some "insights" regarding the growth of execution time in the same algorithm, though still not accurate

However, parallelism is not "tax-free"

- Synchronization
- Preparing data
- Addition function calls
- Data exchange if the parallel hardware has its own memory hierarchy

$$\text{Execution Time}_{\text{enhanced}} = \frac{(1-f)}{perf(r)} + \frac{f}{n}$$

Don't hurt non-common part too mach

- If the program spend 90% in A, 10% in B. Assume that an optimization can accelerate A by 9x, by hurts B by 10x…
- Assume the original execution time is T. The new execution time

$$ET_{new} = \frac{ET_{old} \times 90\%}{9} + ET_{old} \times 10\% \times 10$$

$$ET_{new} = 1.1 \times ET_{old}$$

$$Speedup = \frac{ET_{old}}{ET_{new}} = \frac{ET_{old}}{1.1 \times ET_{old}} = 0.91 \times$$ ……slowdown!

You may not use Amdahl's Law for this case as Amdahl's Law does NOT
(1) consider overhead
(2) bound to slowdown

Takeaways: Are we there yet?

- Definition of "Speedup of Y over X" or say Y is n times faster than X:

$$speedup_{Y\_over\_X} = n = \frac{Execution\ Time_X}{Execution\ Time_Y}$$

- Amdahl's Law — $Speedup_{enhanced}(f, s) = \frac{1}{(1-f) + \frac{f}{s}}$
  - Corollary 1 — each optimization has an upper bound $Speedup_{max}(f, \infty) = \frac{1}{(1-f)}$
  - Corollary 2 — make the common case (the most time consuming case) fast!
  - Corollary 3: Optimization has a moving target

$$Speedup_{max}(f_1, \infty) = \frac{1}{(1-f_1)}$$
$$Speedup_{max}(f_2, \infty) = \frac{1}{(1-f_2)}$$
$$Speedup_{max}(f_3, \infty) = \frac{1}{(1-f_3)}$$
$$Speedup_{max}(f_4, \infty) = \frac{1}{(1-f_4)}$$

  - Corollary 4: Exploiting more parallelism from a program is the key to performance gain in modern architectures $Speedup_{parallel}(f_{parallelizable}, \infty) = \frac{1}{(1-f_{parallelizable})}$
  - Corollary 5: Single-core performance still matters $Speedup_{parallel}(f_{parallelizable}, \infty) = \frac{1}{(1-f_{parallelizable})}$
  - Corollary 6: Don't hurt the non-common case too much

$$Speedup_{enhanced}(f, s, r) = \frac{1}{(1-f) + perf(r) + \frac{f}{s}}$$