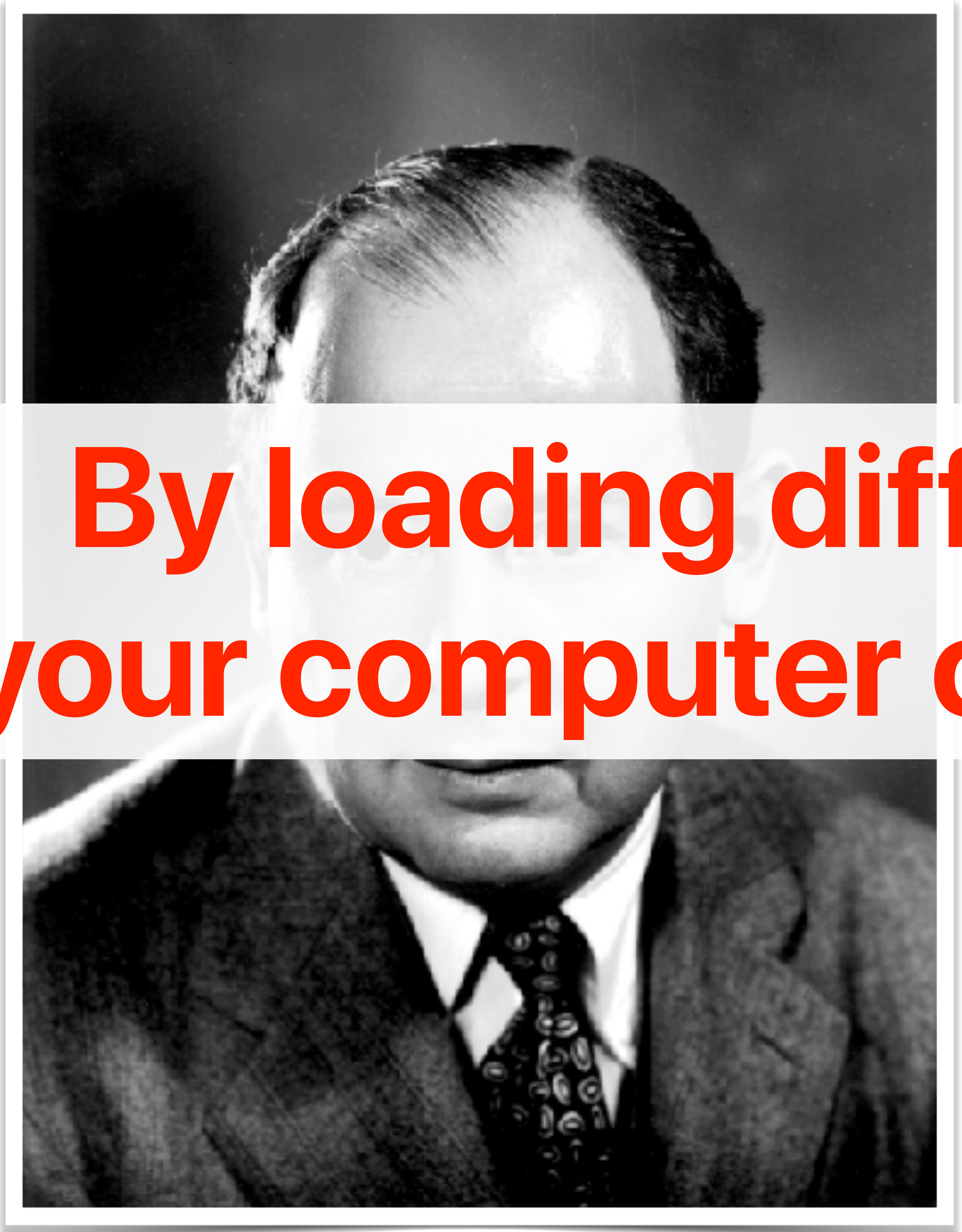
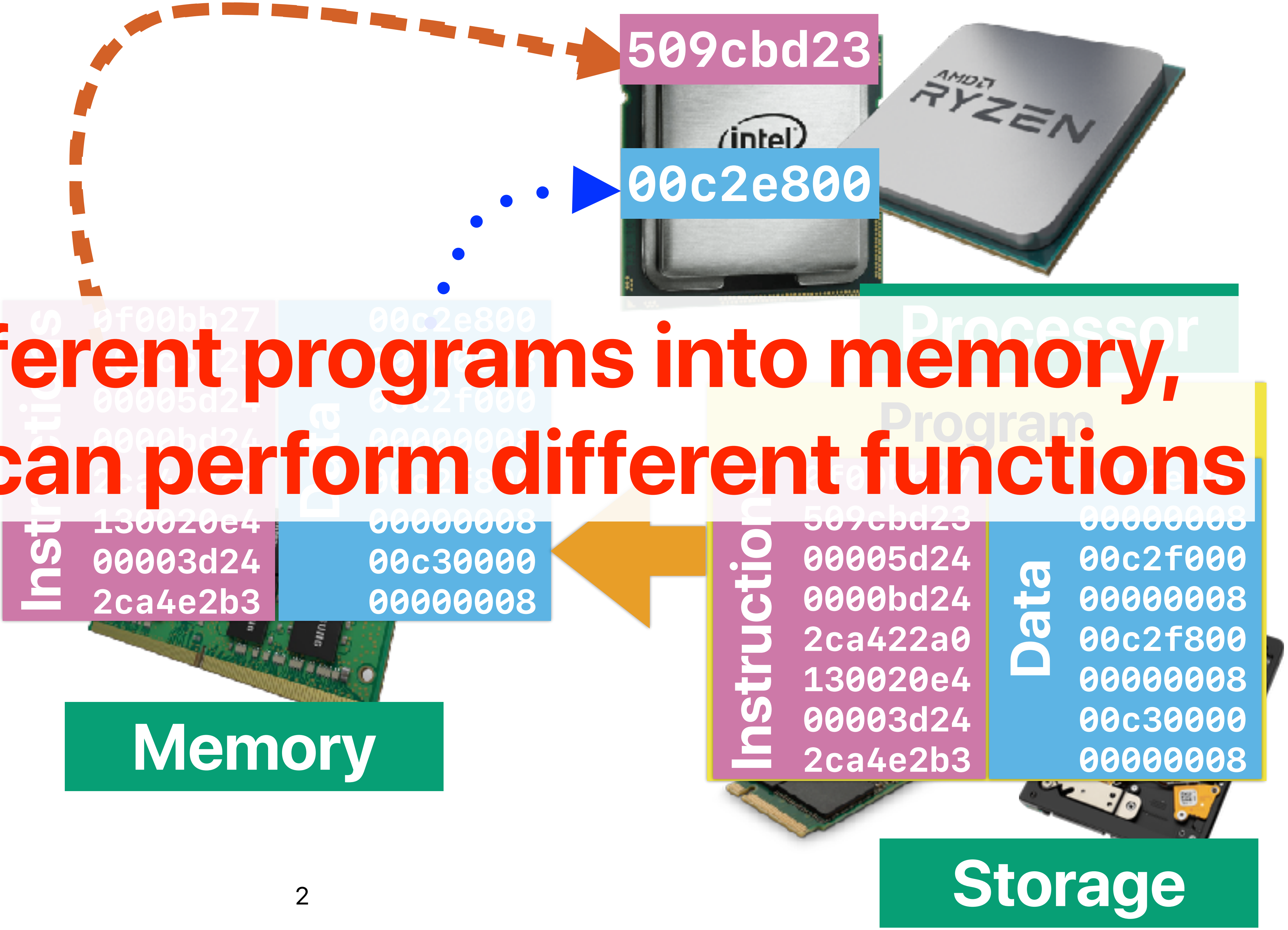
Performance (3): Do the right thing

Hung-Wei Tseng

Recap: von Neumman Architecture





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

Takeaways: What matters?

- Execution time is the most essential performance metric
- The following three factors define the execution time of a program
  - Instruction count
  - Cycles per instruction
  - Cycle time
- Programmers can control all factors that affect performance
- 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!

"Quantitive Approach" to computer performance

- CPU Performance Equation

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

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

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

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

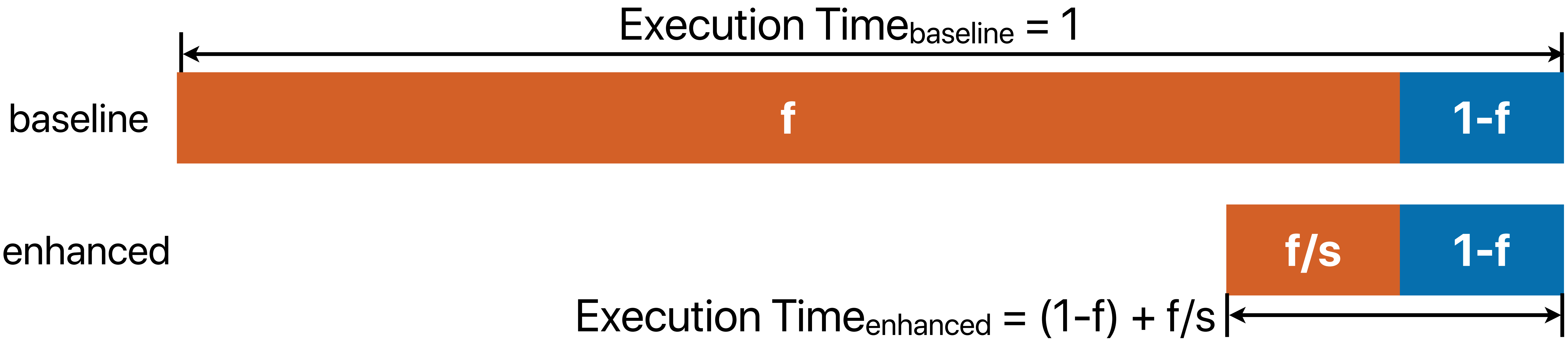
$$Speedup_{Y\_over\_X} = n = \frac{Execution\ Time_X}{Execution\ Time_Y}$$

- Amdahl's Law

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

Amdahl's Law

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



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

Outline

- Amdahl’s law and its implications

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?

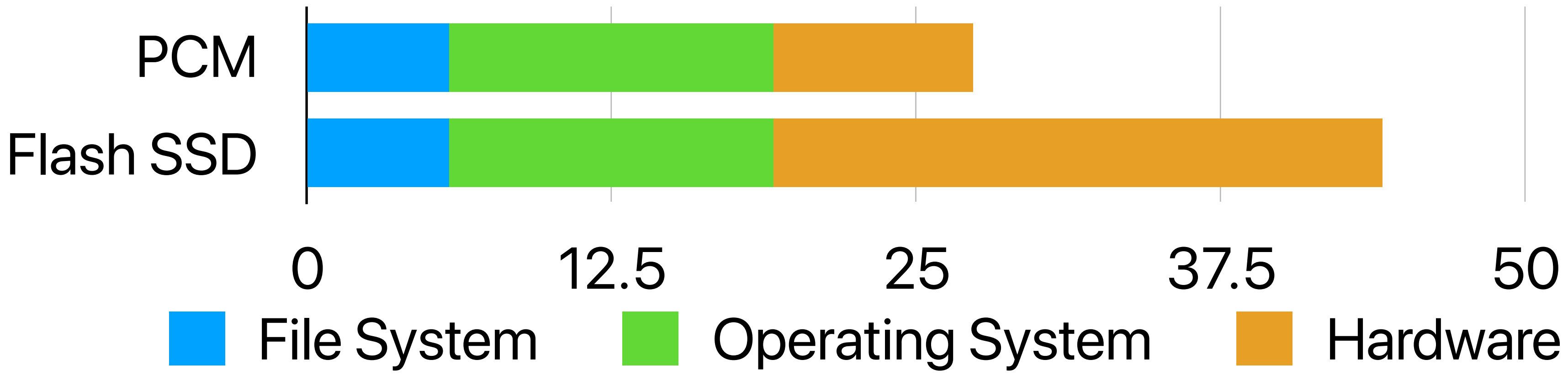
A. ~5x

B. ~10x

C. ~20x

D. ~100x

E. None of the above



$$Speedup_{enhanced}(16\%, x) = \frac{1}{(1-16\%) + \frac{16\%}{x}} = 2$$

$$x = 0.47$$

Does this make sense?

Amdahl's Law Corollary #1

- The maximum speedup of an optimization is bounded by

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

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

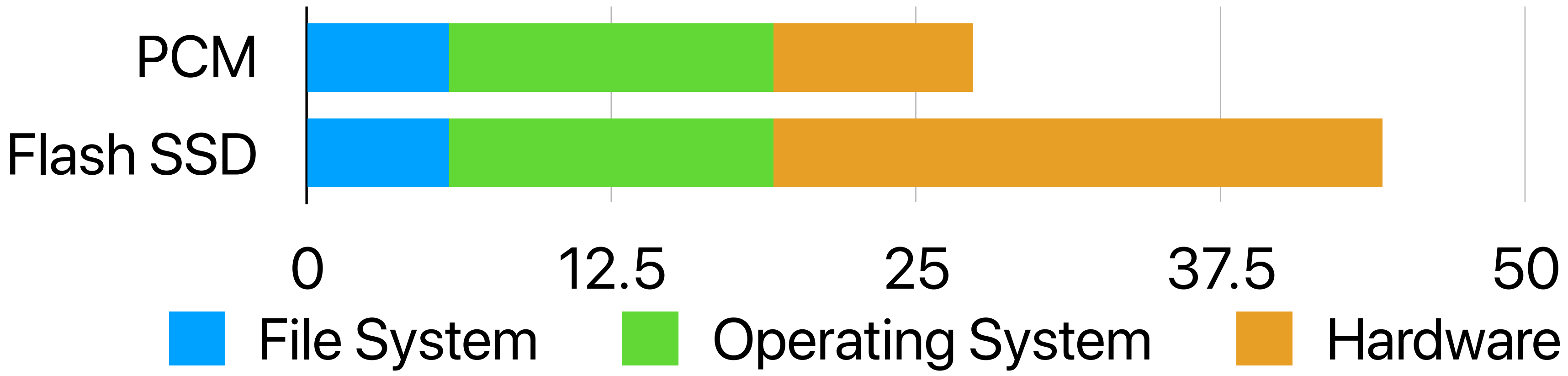
Takeaways: Are we there yet?

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

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

f_1	f_2	f_3	f_4	$1-f_1-f_2-f_3-f_4$

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

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

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

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

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
  - $ET = IC \times CPI \times CT$
- The uncommon case doesn't make much difference
- The common case can change based on inputs, compiler options, optimizations you've applied, etc.

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
  - Corollary 2 — make the common case (the most time consuming case) fast!

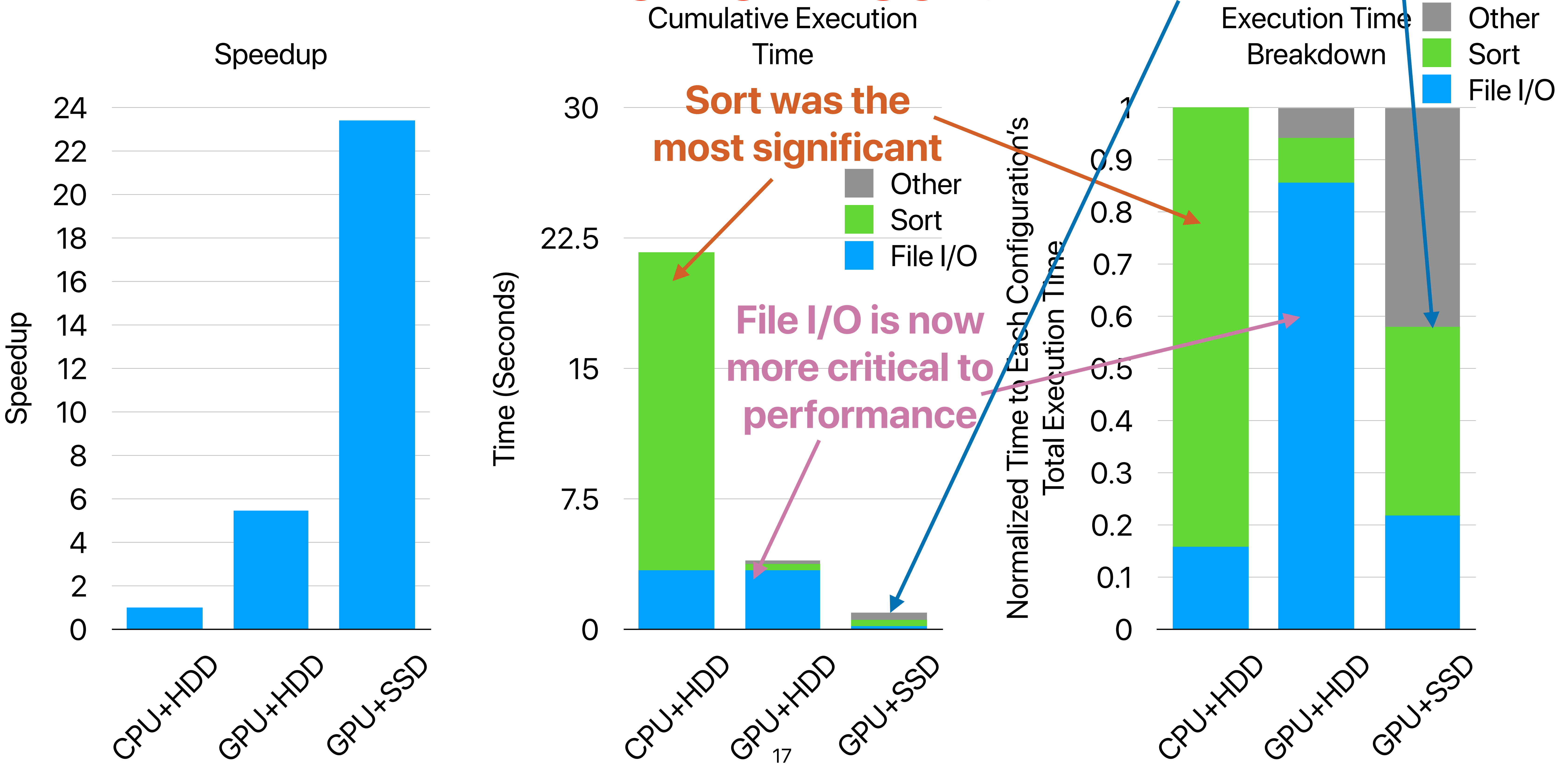
$$Speedup_{max}(f, \infty) = \frac{1}{(1-f)}$$

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

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



Something else (e.g., data movement) matters more
Speedup
Speedup
24
22
20
18
16
14
12
10
8
6
4
2
0
CPU+HDD
GPU+HDD
GPU+SSD
Cumulative Execution Time
Time (Seconds)
30
22.5
15
7.5
0
Sort was the most significant
Other
Sort
File I/O
File I/O is now more critical to performance
Execution Time Breakdown
Normalized Time to Each Configuration's Total Execution Time
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0

Corollary #3: optimization has a moving target

- 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

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

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

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

Amdahl's Law on Multicore Architectures

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

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

https://www.pollev.com/hungweitseng close in 1:30

Amdahl's Law on Multicore Architectures

- Regarding Amdahl's Law on multicore architectures, how many of the following statements is/are correct?
  ① If we have unlimited parallelism, the performance of executing each parallel partition does not matter as long as the performance slowdown in each piece is bounded
  ② With unlimited amount of parallel hardware units, single-core performance does not matter anymore
  ③ With unlimited amount of parallel hardware units, the maximum speedup will be bounded by the fraction of parallel parts
  ④ With unlimited amount of parallel hardware units, the effect of scheduling and data exchange overhead is minor

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

Amdahl's Law on Multicore Architectures

- Regarding Amdahl's Law on multicore architectures, how many of the following statements is/are correct?

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

① If we have unlimited parallelism, the performance of executing each parallel partition does not matter as long as the performance slowdown in each piece is bounded

② With unlimited amount of parallel hardware units, single-core performance does not matter anymore

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

speedup is determined by 1-f

③ With unlimited amount of parallel hardware units, the maximum speedup will be bounded by the fraction of parallel parts

④ With unlimited amount of parallel hardware units, the effect of scheduling and data exchange overhead is minor

A. 0

B. 1

C. 2

D. 3

E. 4

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

Quick Sort

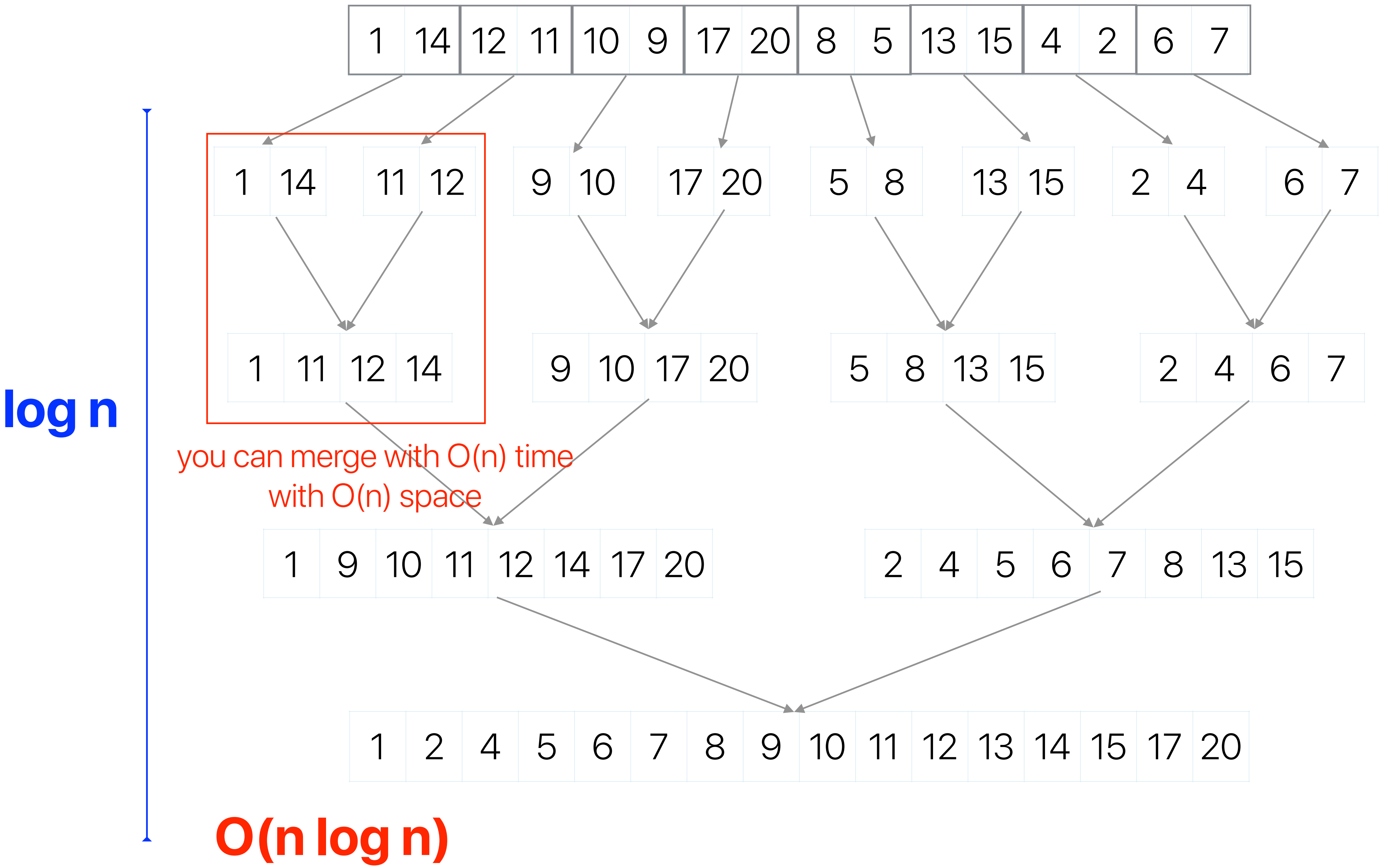
$$O(nlog_2n)$$

Bitonic Sort

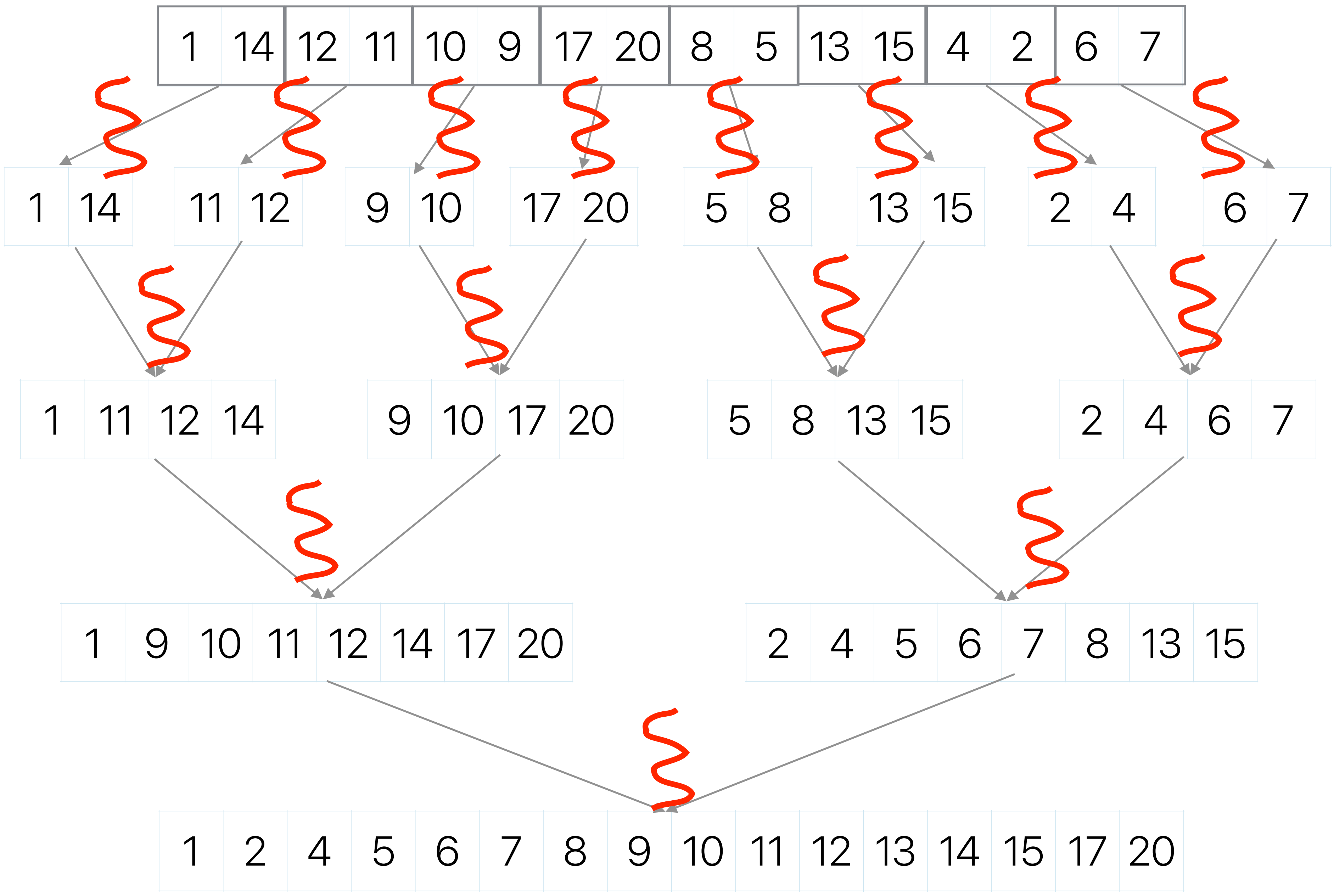
```
void BitonicSort() {

    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);
                }
            }
        }
    }
}
```

$$O(nlog_2^2n)$$

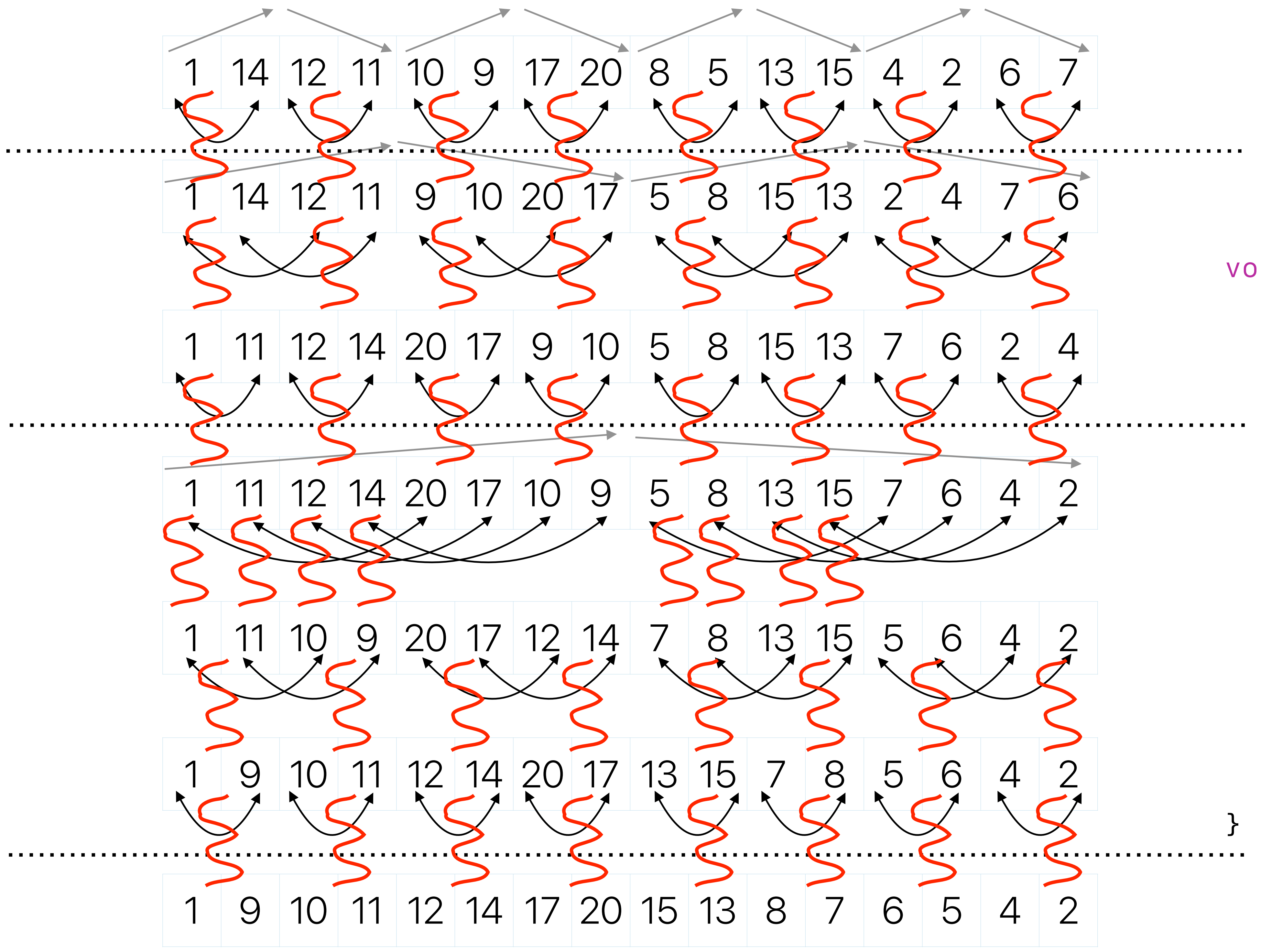
Merge sort


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
log n
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
O(n log n)

Parallel merge sort

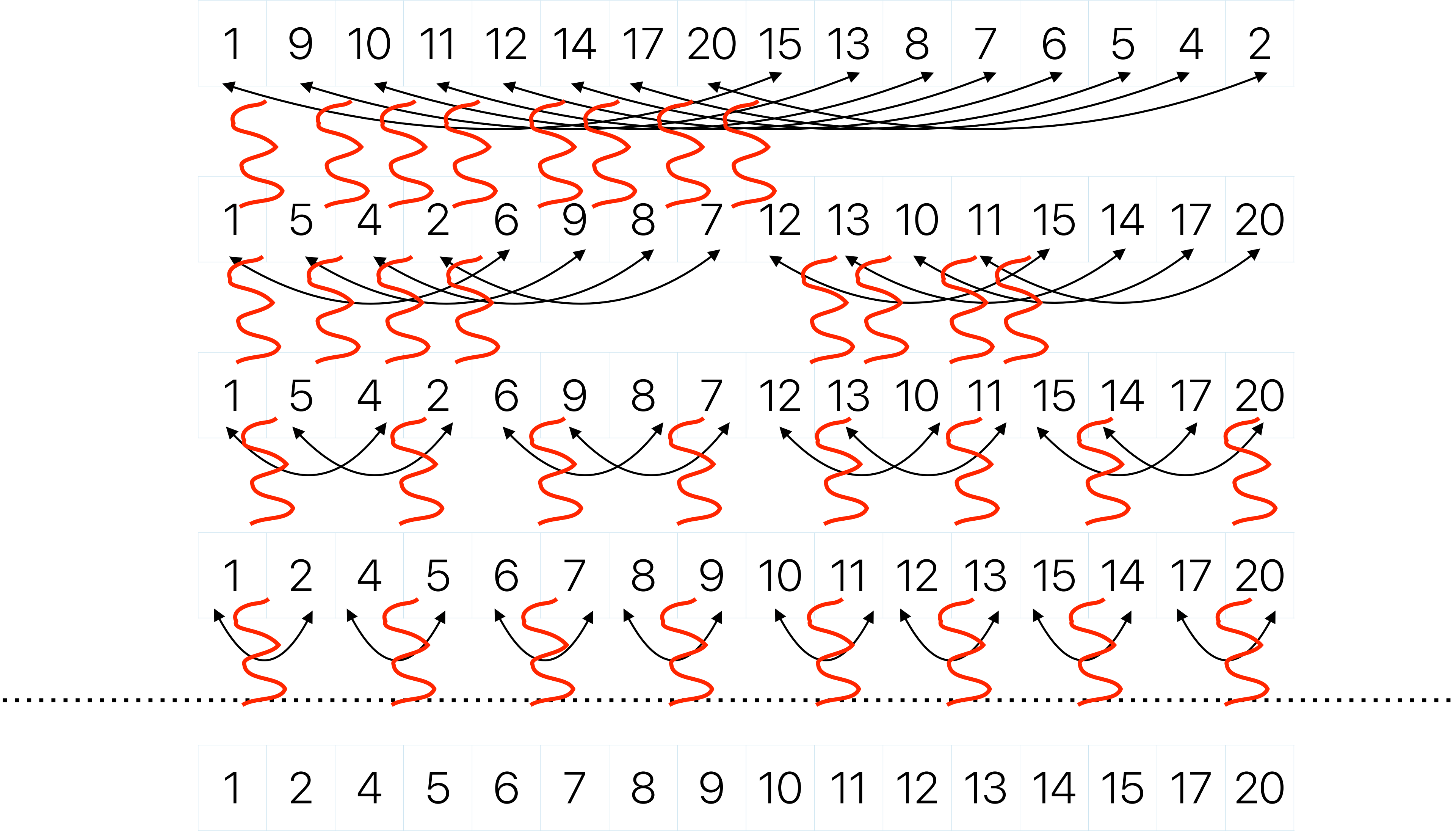


Bitonic sort




```
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**

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!
    $$Speedup_{max}(f_1, \infty) = \frac{1}{(1-f_1)}$$
    $$Speedup_{max}(f_2, \infty) = \frac{1}{(1-f_2)}$$
    $$Speedup_{max}(f_3, \infty) = \frac{1}{(1-f_3)}$$
    $$Speedup_{max}(f_4, \infty) = \frac{1}{(1-f_4)}$$
  - Corollary 3: Optimization has a moving target
  - Corollary 4: Exploiting more parallelism from a program is the key to performance gain in modern architectures
    $$Speedup_{parallel}(f_{parallelizable}, \infty) = \frac{1}{(1-f_{parallelizable})}$$

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

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!
    $Speedup_{max}(f_1, \infty) = \frac{1}{(1-f_1)}$
    $Speedup_{max}(f_2, \infty) = \frac{1}{(1-f_2)}$
    $Speedup_{max}(f_3, \infty) = \frac{1}{(1-f_3)}$
    $Speedup_{max}(f_4, \infty) = \frac{1}{(1-f_4)}$
  - Corollary 3: Optimization has a moving target
  - 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
    $Speedup_{parallel}(f_{parallelizable}, \infty) = \frac{1}{(1-f_{parallelizable})}$

However, parallelism is not “tax-free”

However, parallelism is not "tax-free"

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

Execution Time$_{baseline}$ = 1

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

Amdahl's Law considering overhead

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

- r is some other parameter that affects the overhead
  - input size?
  - degree of parallelism?
- The overhead may scale differently than the original problem — that's why we introduce "perf()" function

Corollary #6: 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 (i.e., an overhead that is 9x longer than the original execution time)...

$$Speedup = \frac{1}{(1-f) + perf(r) + \frac{f}{s}} = \frac{1}{(1-0.9) + (1-0.9) \times 9 + \frac{0.9}{9}} = 0.91\times$$

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!

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

    - Corollary 3: Optimization has a moving target
    - 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

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

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

V. Sze, Y. -H. Chen, T. -J. Yang and J. S. Emer. How to Evaluate Deep Neural Network Processors: TOPS/W (Alone) Considered Harmful. In IEEE Solid-State Circuits Magazine, vol. 12, no. 3, pp. 28-41, Summer 2020.

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)

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!

nvidia.com

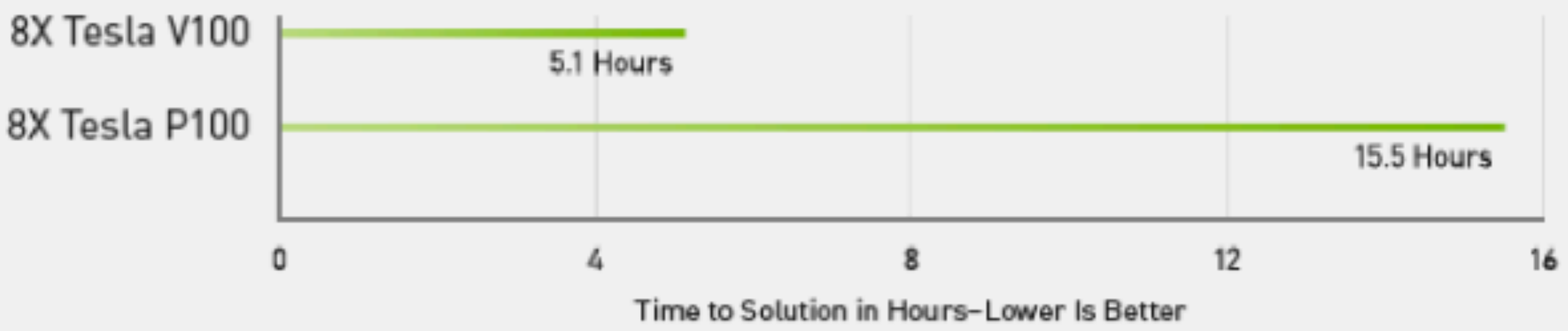
NVIDIA

Artificial Intelligence Computing Leadership from NVIDIA

CLOUD & DATA CENTER   PRODUCTS   SOLUTIONS   APPS   FOR DEVELOPERS   TECHNOLOGIES

Tesla V100   AI TRAINING   AI INFERENCE   HPC   DATA CENTER GPUs   SPECIFICATIONS

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.

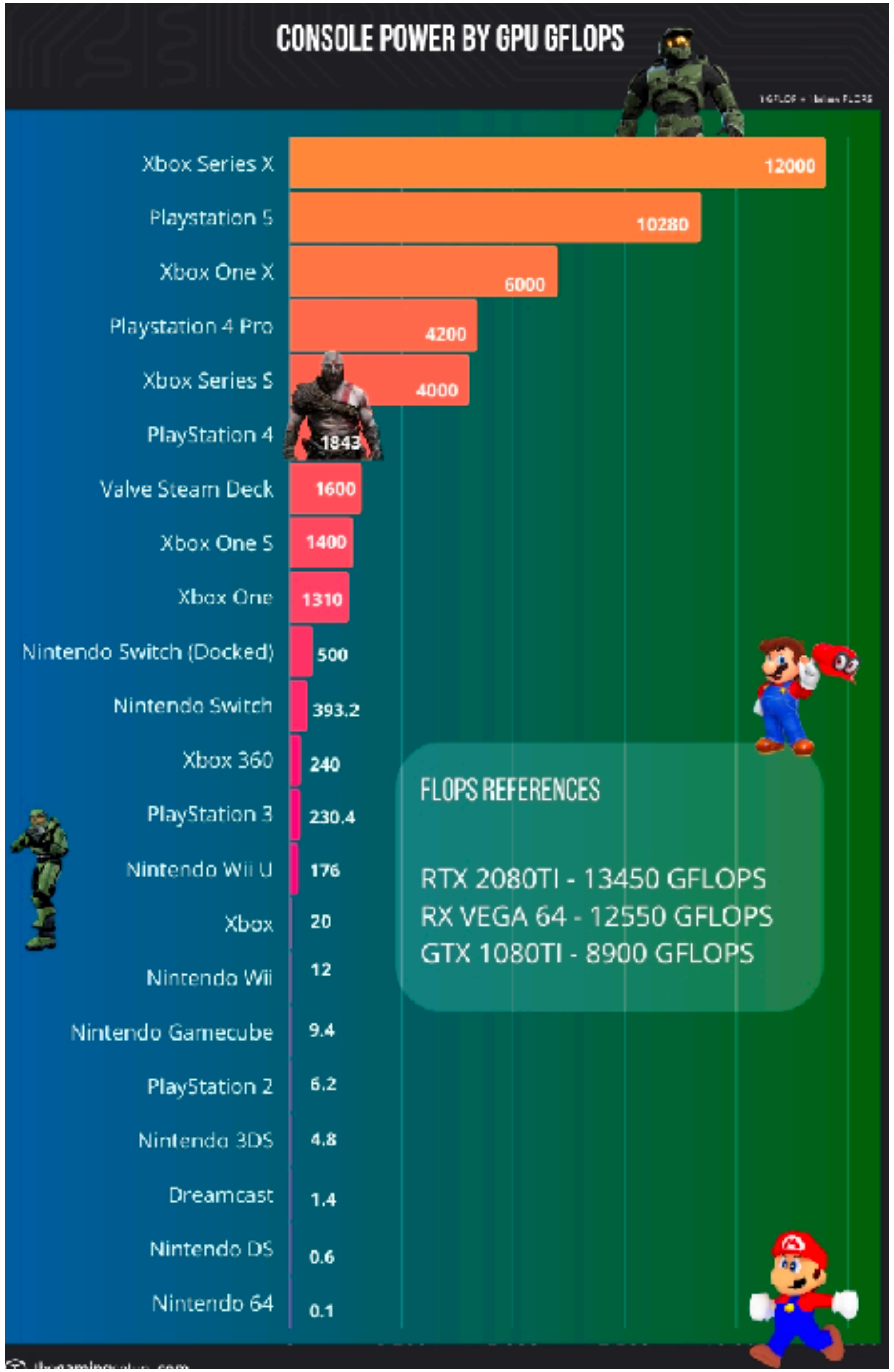
AI TRAINING

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



TFLOPS (Tera FLoating-point Operations Per Second)

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

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 $i = j = k = 2048$

$2^{3\times 11} \times 2 = 2^{34}$ **FLOPs in total** https://www.pollev.com/hungweitseng close in 1:30

How reflective is FLOPS?

- Given the FLOPS number measured, how many of the followings are true?
  ① The FLOPS number remains the same on each architecture if we change the data size
  ② The FLOPS number remains the same on each architecture if we change the data type to double
  ③ The FLOPS number remains the same on each architecture if we change the algorithm implementation
  ④ The FLOPS number reflects the performance ratio of different architectures when executing floating point applications

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

How reflective is FLOPS?

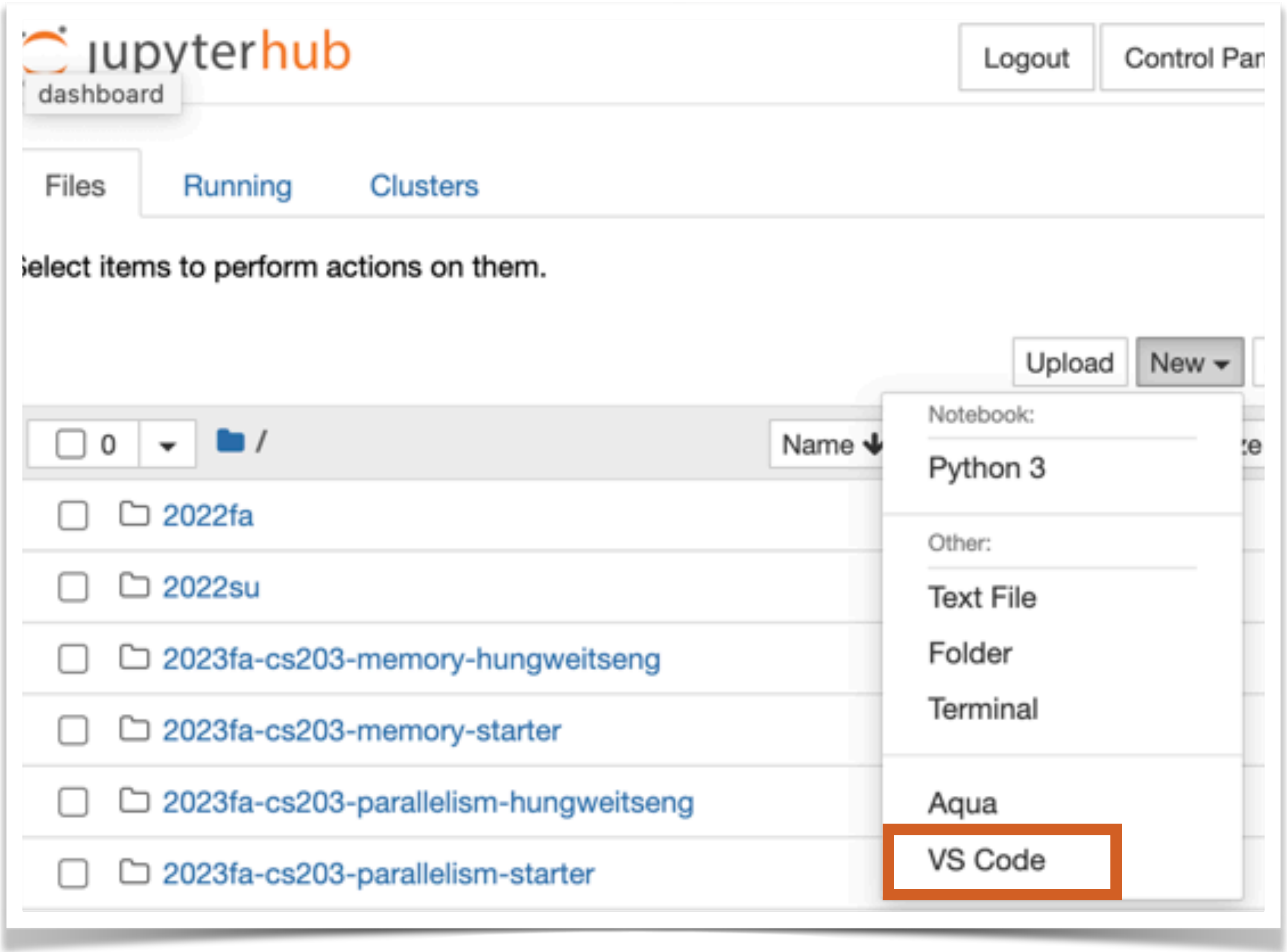
- Given the FLOPS number measured, how many of the followings are true?
  ① The FLOPS number remains the same on each architecture if we change the data size
  ② The FLOPS number remains the same on each architecture if we change the data type to double
  ③ The FLOPS number remains the same on each architecture if we change the algorithm implementation
  ④ The FLOPS number reflects the performance ratio of different architectures when executing floating point applications
  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

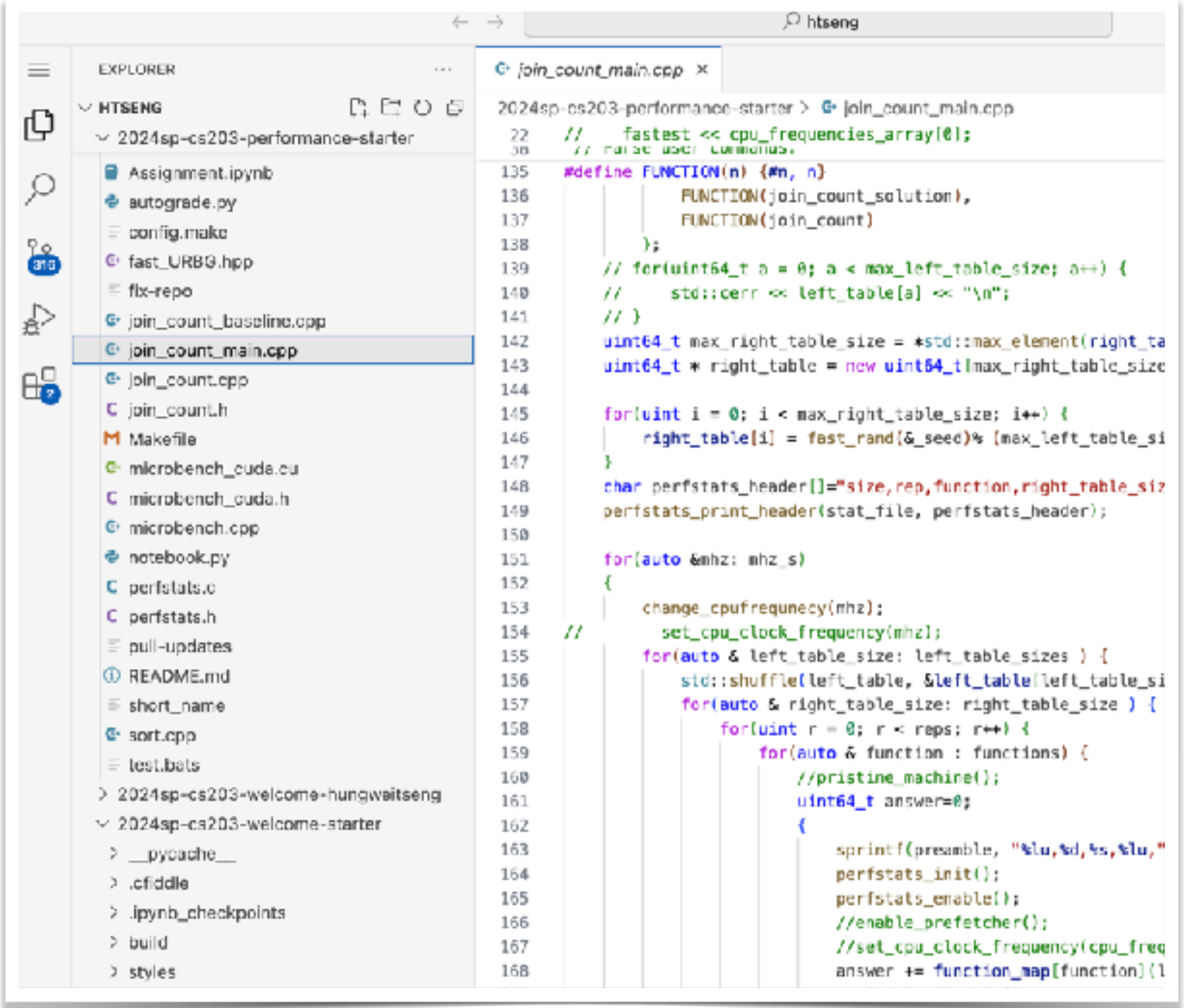
How reflective is FLOPS?

- Given the FLOPS number measured, how many of the followings are true?
  ① The FLOPS number remains the same on each architecture if we change the data size
  ② The FLOPS number remains the same on each architecture if we change the data type to double
  ③ The FLOPS number remains the same on each architecture if we change the algorithm implementation
  ④ The FLOPS number reflects the performance ratio of different architectures when executing floating point applications

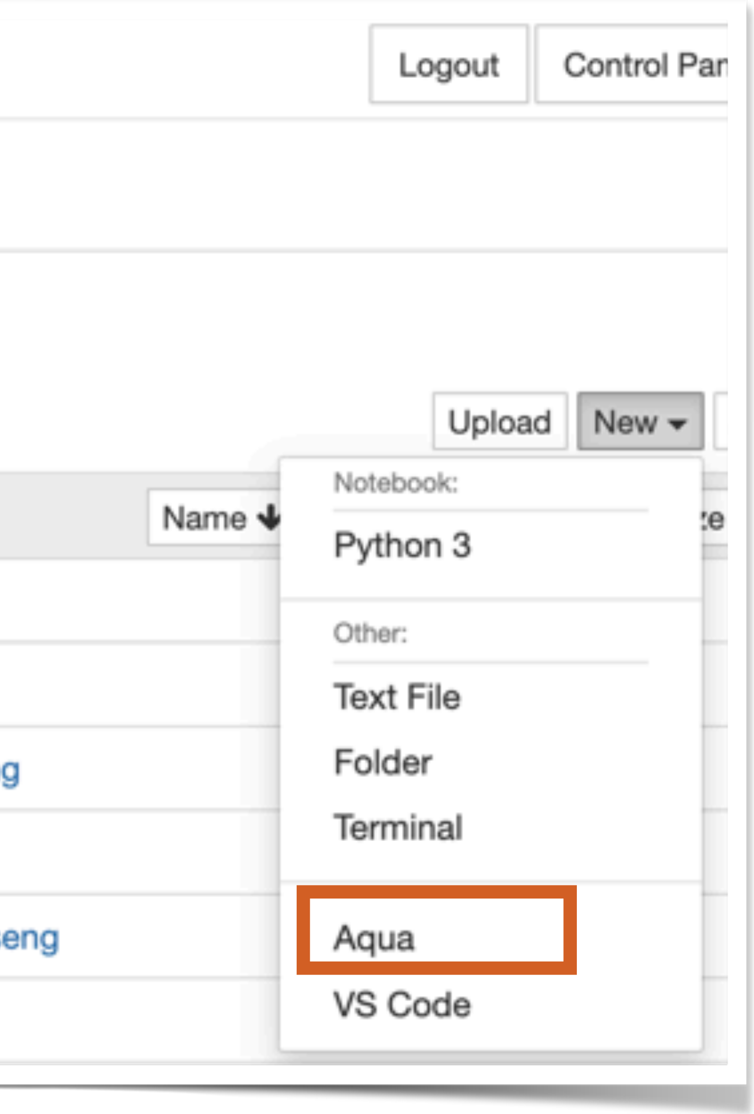
A. 0
B. 1
C. 2
D. 3
E. 4

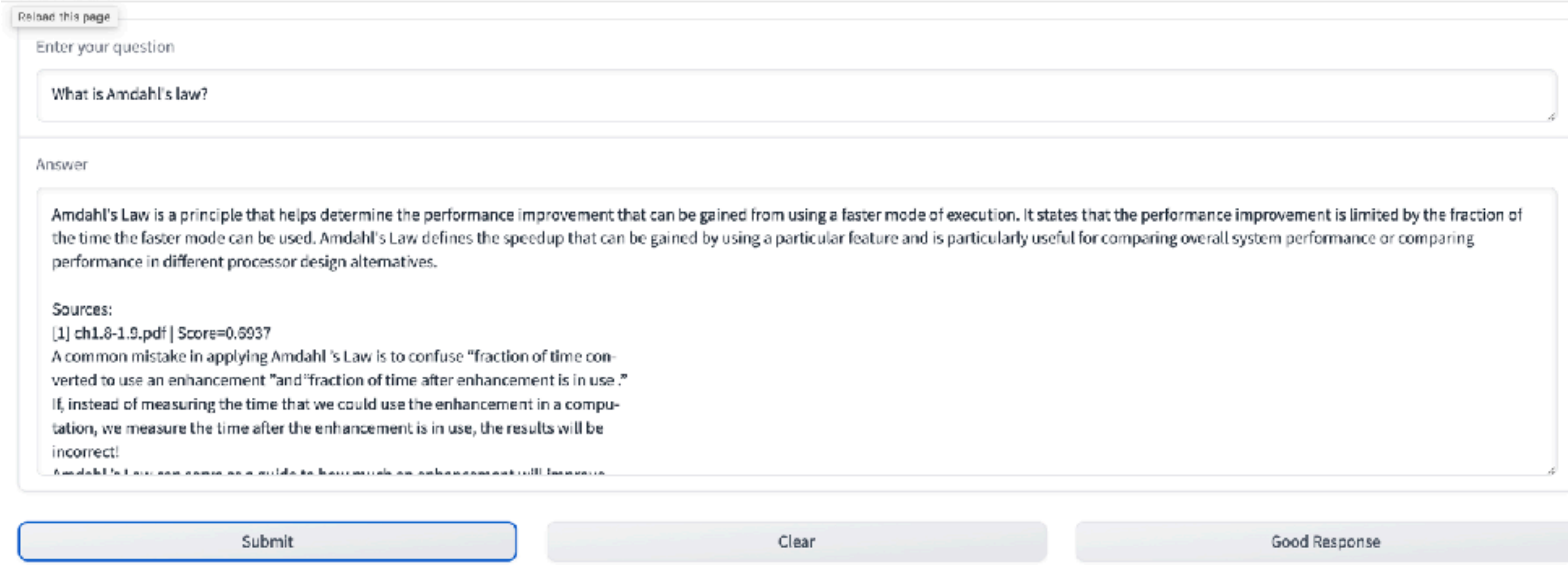
Use Jupyterhub features!





Help us improve!





Announcement

- Assignment #1 due tonight
  - We never allow late submission and we will never have deadline extension
  - We autograde everything in your assignment + unlimited submissions before the deadline = start early, receive feedback early, and get good grades
  - No regrading — please make sure your submission can be parsed by the grader correctly
- Assignment #2 due on 4/18
- Check our
  - **website** for slides/grades/schedules
  - **gradescope** for quizzes/assignments
  - **piazza** for discussions
  - **escalab.org/datahub** as the place to work on your assignments
  - there is **no** eLearn for this class
- Youtube channel for lecture recordings: https://www.youtube.com/c/ProfUsagi/playlists

Computer
Science &
Engineering

203

つづく