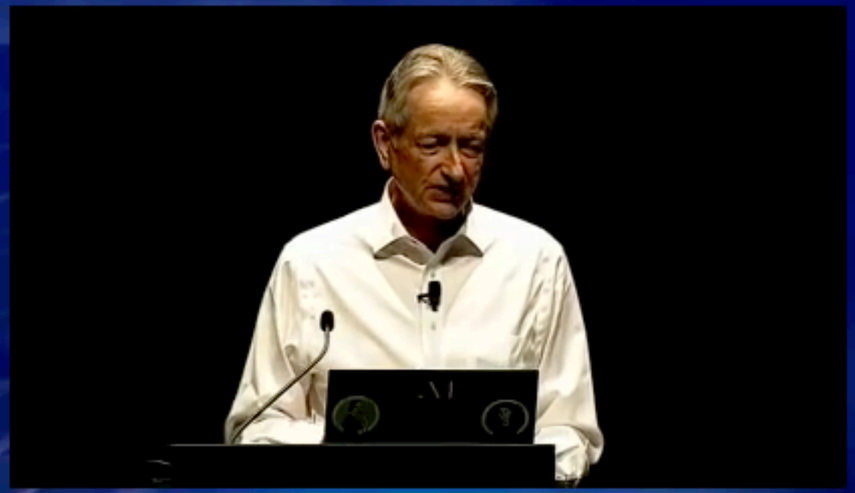
Performance (3): Do the right thing

Hung-Wei Tseng

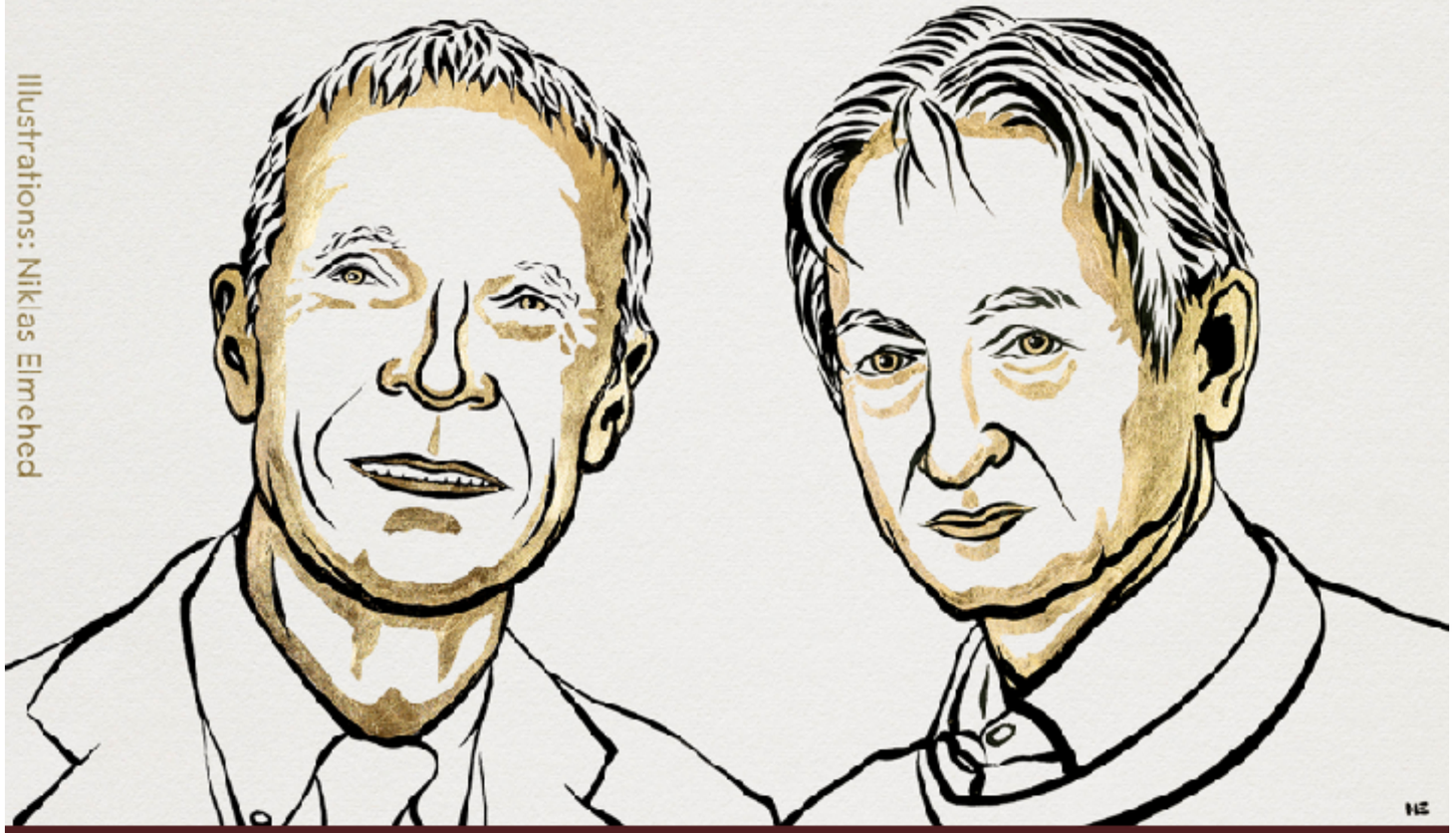
The return of backpropagation

- Between 2005 and 2009 researchers (in Canada!) made several technical advances that enabled backpropagation to work better in feed-forward nets.
  - Unsupervised pre-training; random dropout of units; rectified linear units.
  - The technical details of these advances are very important to the researchers but they are not the main message.
  - The main message is that backpropagation now works amazingly well if you have two things:
    - a lot of labeled data
    - a lot of convenient compute power (*e.g.* GPUs)



acm FCRC

THE NOBEL PRIZE IN PHYSICS 2024



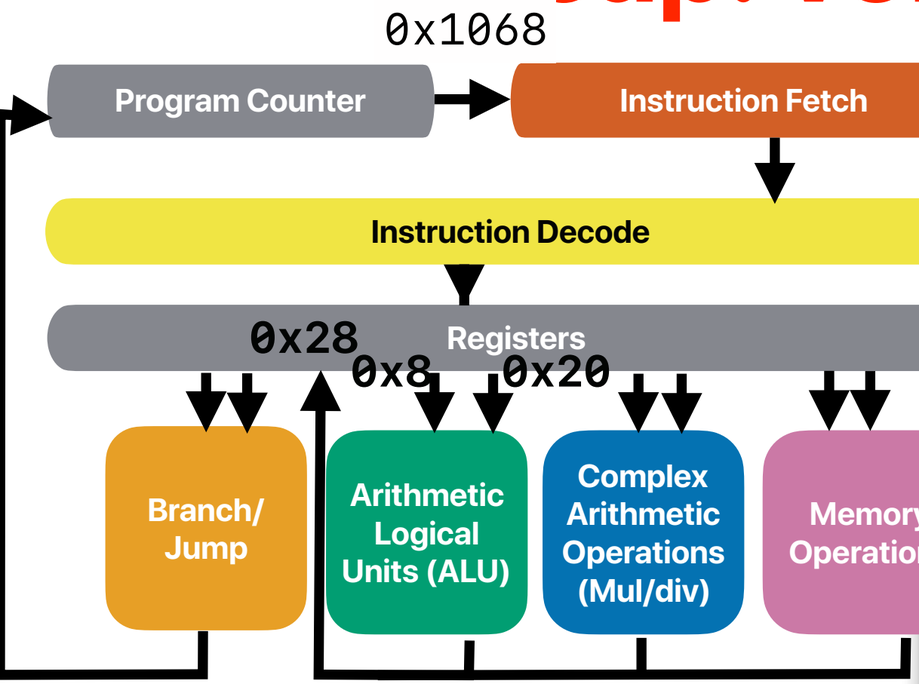
Illustrations: Niklas Elmehed

John J. Hopfield Geoffrey E. Hinton

"for foundational discoveries and inventions
that enable machine learning
with artificial neural networks"

THE ROYAL SWEDISH ACADEMY OF SCIENCES

Recap: von Neumann architecture

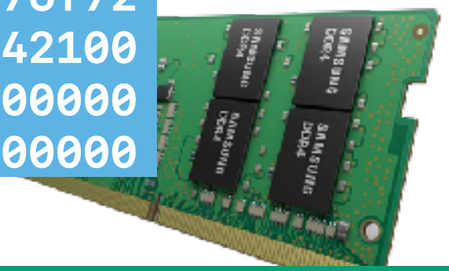


Processor

4883ec08

sub $0x8,%rsp

Instructions	Data
f30f1efa	08400000
4883ec08	00000100
488d3d95	02004865
0f0000e8	6c6c6f2c
dcffffff	20776f72
31c04883	6c642100
c408c30f	00000000
1f440000	00000000



Memory

```
int main(){
    pr
}
```

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

Storage

How compilers affect performance

- If we turn on "-O3" flag when using gcc to compile both code snippets **A** and **B**, how many of the following can we expect?
  - ① Compiler optimizations can reduce IC for both
  - ② Compiler optimizations can make the CPI lower for both
  - ③ Compiler optimizations can make the ET lower for both
  - ④ Compiler optimizations can transform code B into code A

Compiler can apply loop unrolling, constant propagation naively to reduce IC

Reduced IC does not necessarily mean lower CPI — compiler may pick one longer instruction to replace a few shorter ones

Compiler cannot guarantee the combined effects lead to better performance!

"Most compilers" will not significantly change programmer's code since compiler cannot guarantee if doing that would affect the correctness

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

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];
  }
}
```

What matters to the classical 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**

Outline

- What does better mean?
- Amdahl's Law and its implications

Quantitive Analysis of “Better”

Speedup of Y over X

- Consider the same program on the following two machines, X and Y. By how much Y is faster than X?

	Clock Rate	Dynamic Instruction Count	Percentage of Type-A	CPI of Type-A	Percentage of Type-B	CPI of Type-B	Percentage of Type-C	CPI of Type-C
Machine X	4 GHz	5000000000	20%	5	20%	2	60%	1
Machine Y	6 GHz	5000000000	20%	7	20%	2	60%	1

A. 0.2
B. 0.25
C. 0.8
D. 1.25
E. No changes

Speedup

- The relative performance between two machines, X and Y. Y is n times faster than X

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

- The speedup of Y over X

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

Speedup of Y over X

- Consider the same program on the following two machines, X and Y. By how much Y is faster than X?

	Clock Rate	Dynamic Instruction	Percentage of Type-A	CPI of Type-A	Percentage of Type-B	CPI of Type-B	Percentage of Type-C	CPI of Type-C
Machine X	4 GHz	5000000000	20%	5	20%	2	60%	1
Machine Y	6 GHz	5000000000	20%	7	20%	2	60%	1

A. 0.2
B. 0.25
C. 0.8
D. 1.25
E. No changes

$$ET_X = (5 \times 10^9) \times (20\% \times 5 + 20\% \times 2 + 60\% \times 1) \times \frac{1}{4 \times 10^9}\ sec = 2.5\ sec$$

$$ET_Y = (5 \times 10^9) \times (20\% \times 7 + 20\% \times 2 + 60\% \times 1) \times \frac{1}{6 \times 10^9}\ secs = 2\ secs$$

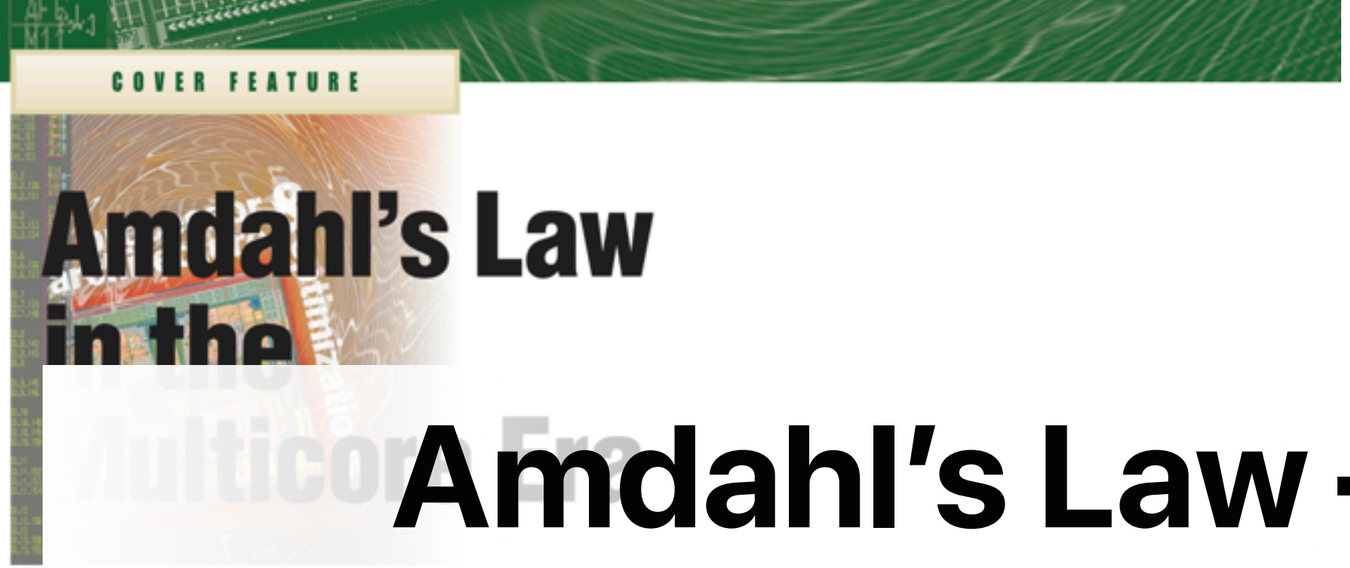
$$Speedup = \frac{Execution\ Time_X}{Execution\ Time_Y}$$

$$= \frac{2.5}{2} = 1.25$$

Takeaways: What matters?

- Different programming languages can generate machine operations with different orders of magnitude performance — programmers need to make wise choice of that!
- Programmers can control all three factors in the classic performance equation when composing the program
- Compiler optimization does not always help
  - Compiler optimizes code based on some assumptions that may not be true on all computers
  - Programmers' can write code in a way facilitating optimizations!
- 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

Augmenting Amdahl's law with a corollary for multicore hardware makes it relevant to future generations of chips with multiple processor cores. Obtaining optimal multicore performance will require further research in both extracting more parallelism and making sequential cores faster.

Mark D. Hill, University of Wisconsin-Madison
Michael R. Marty, Google

In IEEE Computer, vol. 41, no. 7

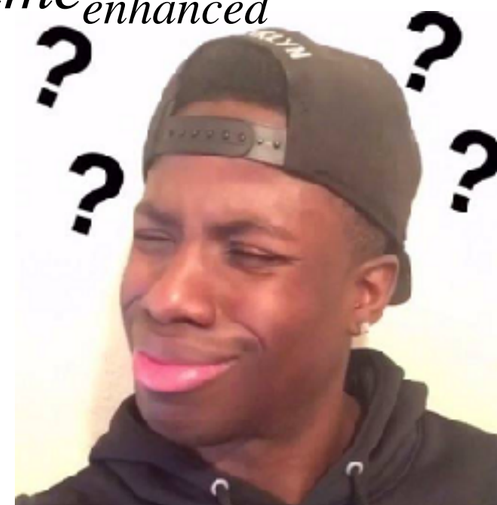
Amdahl's Law

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

f — The fraction of time in the original program

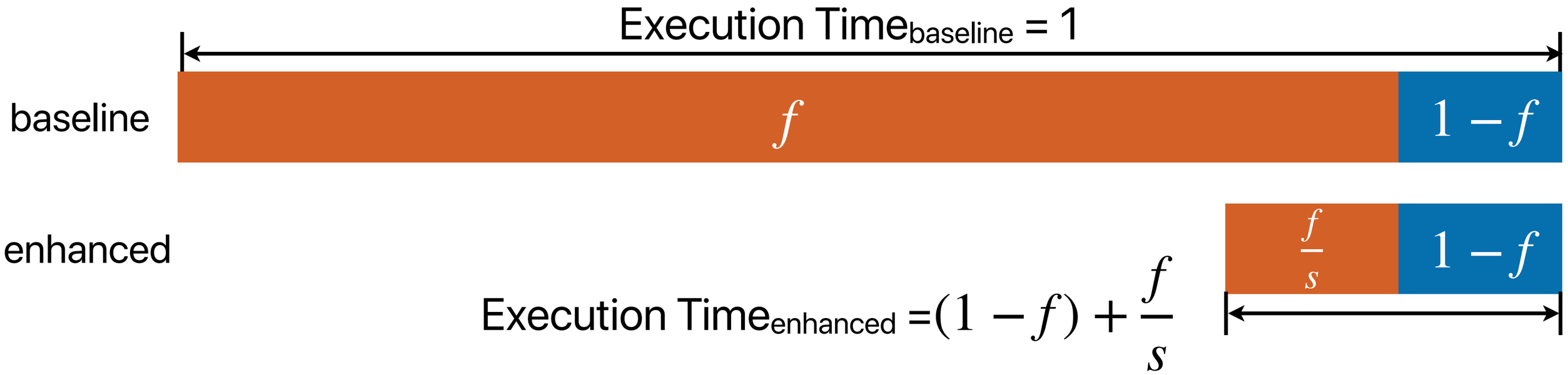
s — The speedup we can achieve on f

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



Amdahl's Law

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

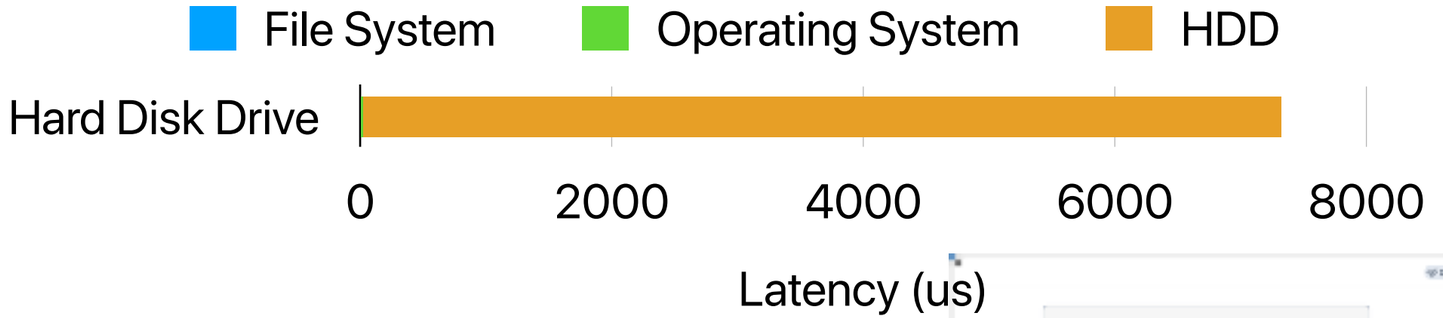


$$Speedup_{enhanced} = \frac{Execution\ Time_{baseline}}{Execution\ Time_{enhanced}} = \frac{1}{(1 - f) + \frac{f}{s}}$$

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

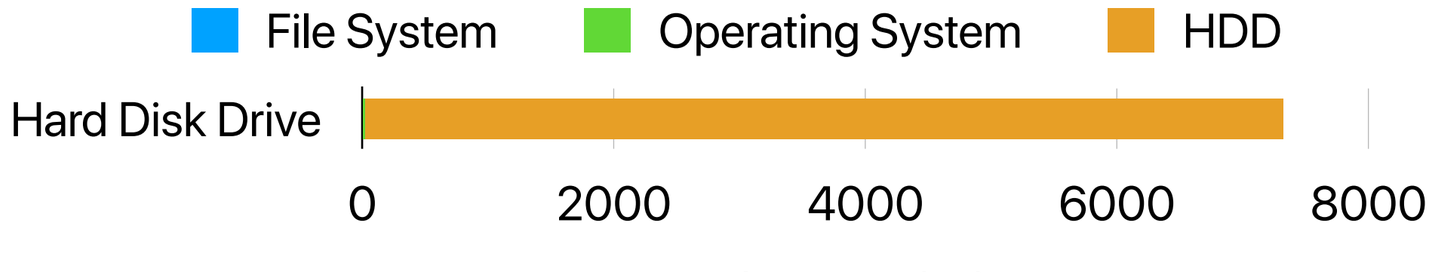
Practicing Amdahl's Law

- Final Fantasy XV spends lots of time loading a map — within which period that 95% of the time on the accessing the H.D.D., the rest in the operating system, file system and the I/O protocol. If we replace the H.D.D. with a flash drive, which provides 100x faster access time. By how much can we speed up the map loading process?
  - A. ~7x
  - B. ~10x
  - C. ~17x
  - D. ~29x
  - E. ~100x



Practicing Amdahl's Law

- Final Fantasy XV spends lots of time loading a map — within which period that 95% of the time on the accessing the H.D.D., the rest in the operating system, file system and the I/O protocol. If we replace the H.D.D. with a flash drive, which provides 100x faster access time. By how much can we speed up the map loading process?
  - A. ~7x
  - B. ~10x
  - C. ~17x
  - D. ~29x
  - E. ~100x



$$Speedup_{enhanced}(95\%, 100) = \frac{1}{(1 - 95\%) + \frac{95\%}{100}} = 16.81\times$$

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:

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

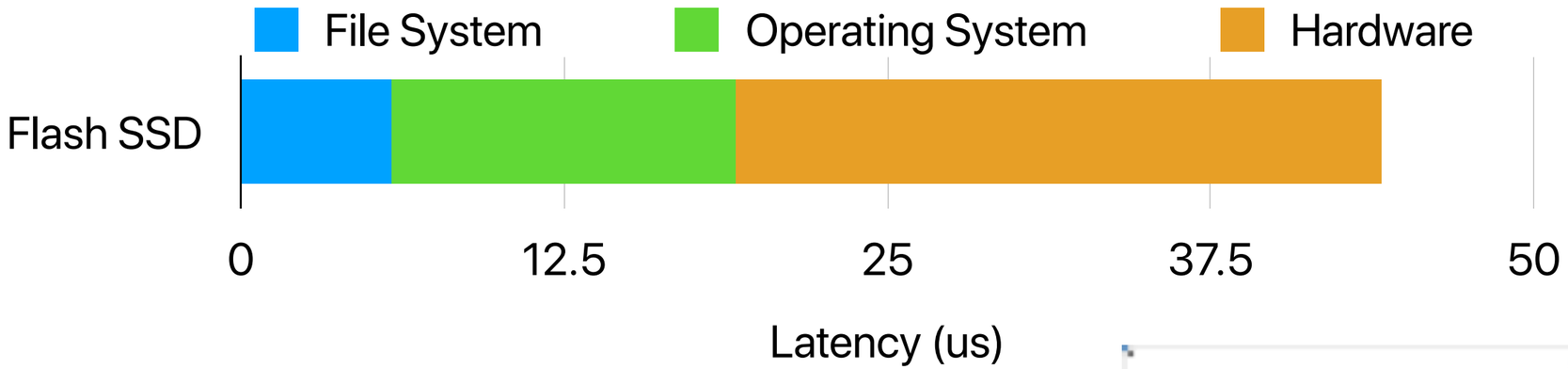
  - If optimization #1 and optimization #2 are not dis-joint:

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

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?

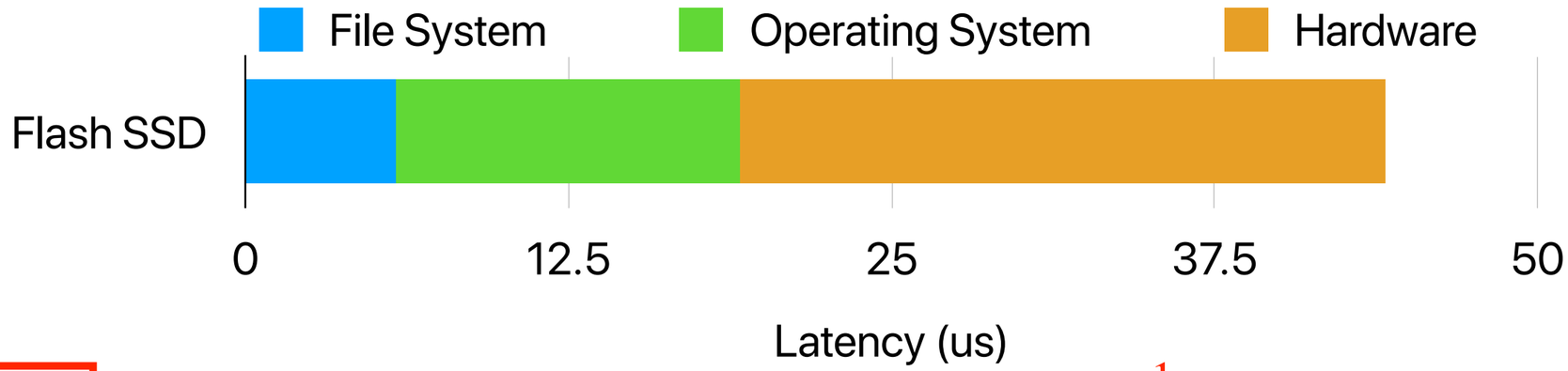
A. ~5x
B. ~10x
C. ~20x
D. ~100x
E. None of the above



Speedup 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 is bounded by

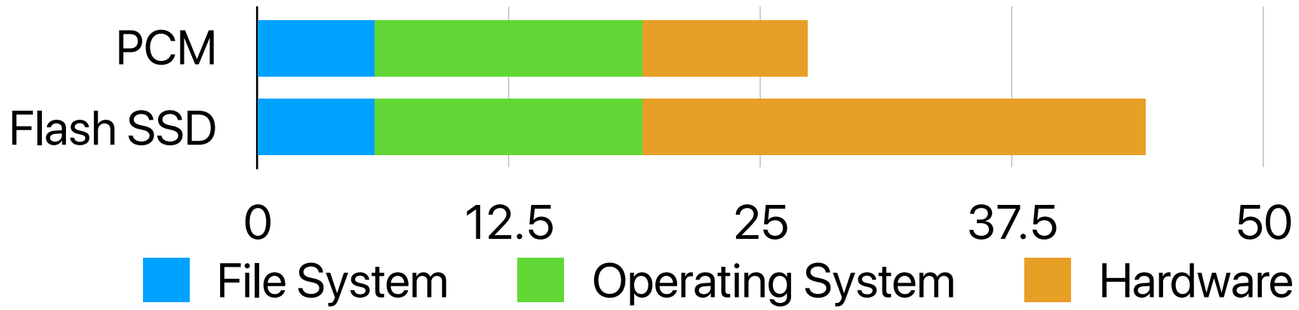
$$Speedup_{max}(f, \infty) = \frac{1}{(1-f) + \frac{f}{\infty}}$$

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

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

PCWorld NEWS BEST PICKS REVIEWS HOW-TO DEALS LAPTOPS WINDOWS SECURITY MORE NEWS

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

You spent a lot of money but can only get 20% performance gain!

Image: Intel

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

Practicing Amdahl's Law (2)

- After applying an SSD, Final Fantasy XV now spends 16% in accessing SSD, the rest in the operating system, file system and the I/O protocol. Which of the following proposals would give as the largest performance gain?
  A. Replacing the CPU to speed up the rest by 2x
  B. Replacing the CPU to speed up the rest by 1.2x and replacing the SSD to speed up the SSD part by 100x
  C. Replacing the CPU to speed up the rest by 1.5x and replacing the SSD to speed up the SSD part by 20x
  D. Replacing the SSD to speed up the SSD part by 200x
  E. They are about the same

Practicing Amdahl's Law (2)

- After applying an SSD, Final Fantasy XV now spends 16% in accessing SSD, the rest in the operating system, file system and the I/O protocol. Which of the following proposals would give as the largest performance gain?

$$Speedup_{enhanced}(84\%, 16\%, 2, 1) = \frac{1}{(1-16\%) + \frac{84\%}{2}} = 1.72\times$$

  A. Replacing the CPU to speed up the rest by 2x
  B. Replacing the CPU to speed up the rest by 1.2x and replacing the SSD to speed up the SSD part by 100x

$$Speedup_{enhanced}(84\%, 16\%, 1.2, 100) = \frac{1}{(1-84\%-16\%) + \frac{84\%}{1.2} + \frac{16\%}{100}} = 1.43\times$$

  C. Replacing the CPU to speed up the rest by 1.5x and replacing the SSD to speed up the SSD part by 20x

$$Speedup_{enhanced}(84\%, 16\%, 1.5, 20) = \frac{1}{(1-84\%-16\%) + \frac{84\%}{1.5} + \frac{16\%}{20}} = 1.76\times$$

  D. Replacing the SSD to speed up the SSD part by 200x

$$Speedup_{enhanced}(84\%, 16\%, 1, 200) = \frac{1}{(1-16\%) + \frac{16\%}{200}} = 1.19\times$$

  E. They are about the same

If we don't touch the 84% part, it's hard to speedup!

Corollary #1 on Multiple Optimizations

- If we want to make significant impact on performance —

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

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

Corollary #2 — make the common case fast!

- Common == **most time consuming** == largest f
  $\neq$ the most frequently invoked, executed
- When f is small, optimizations will have very little effect.
- When f is large, small change can make huge difference!
- The uncommon case doesn't make much difference
- The common case can change based on inputs, compiler options, optimizations you've applied, etc.

Takeaways: find the right thing to do

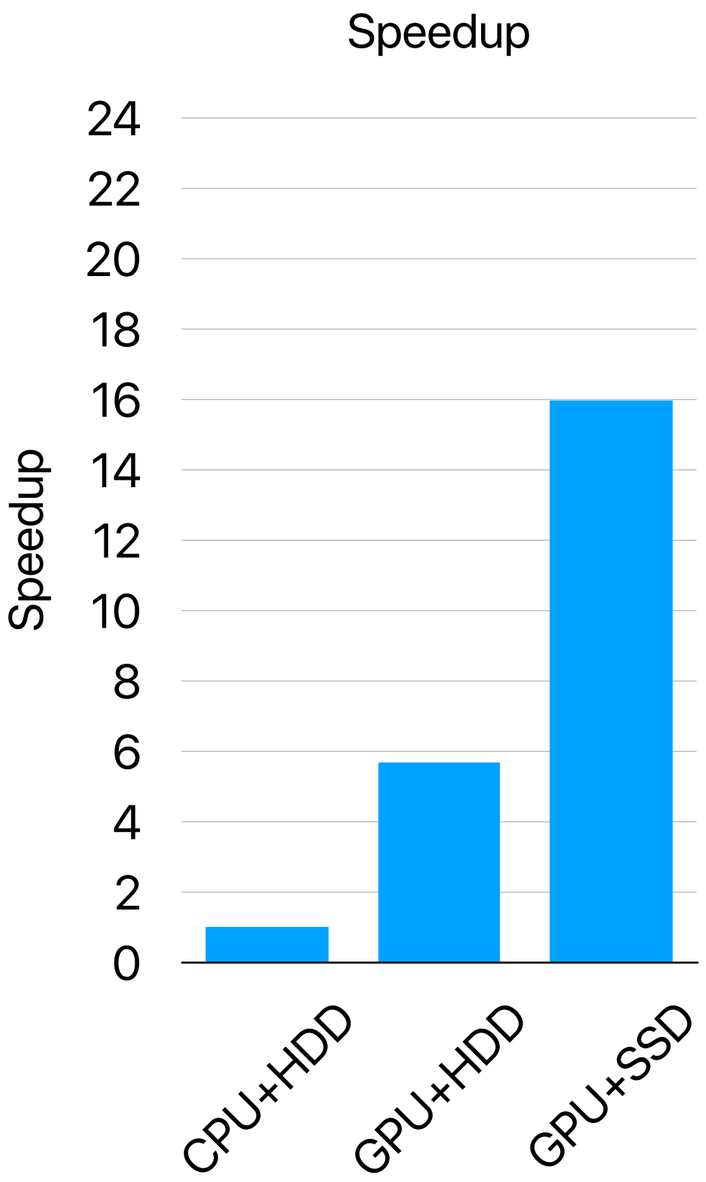
- Definition of "Speedup of Y over X" or say Y is n times faster than X: $speedup_{Y\_over\_X} = n = \frac{Execution\ Time_X}{Execution\ Time_Y}$
- Amdahl's Law — $Speedup_{enhanced}(f, s) = \frac{1}{(1-f) + \frac{f}{s}}$ $Speedup_{max}(f, \infty) = \frac{1}{(1-f)}$
  - Corollary 1 — each optimization has an upper bound
  - Corollary 2 — make the common case (the most time consuming case) fast!

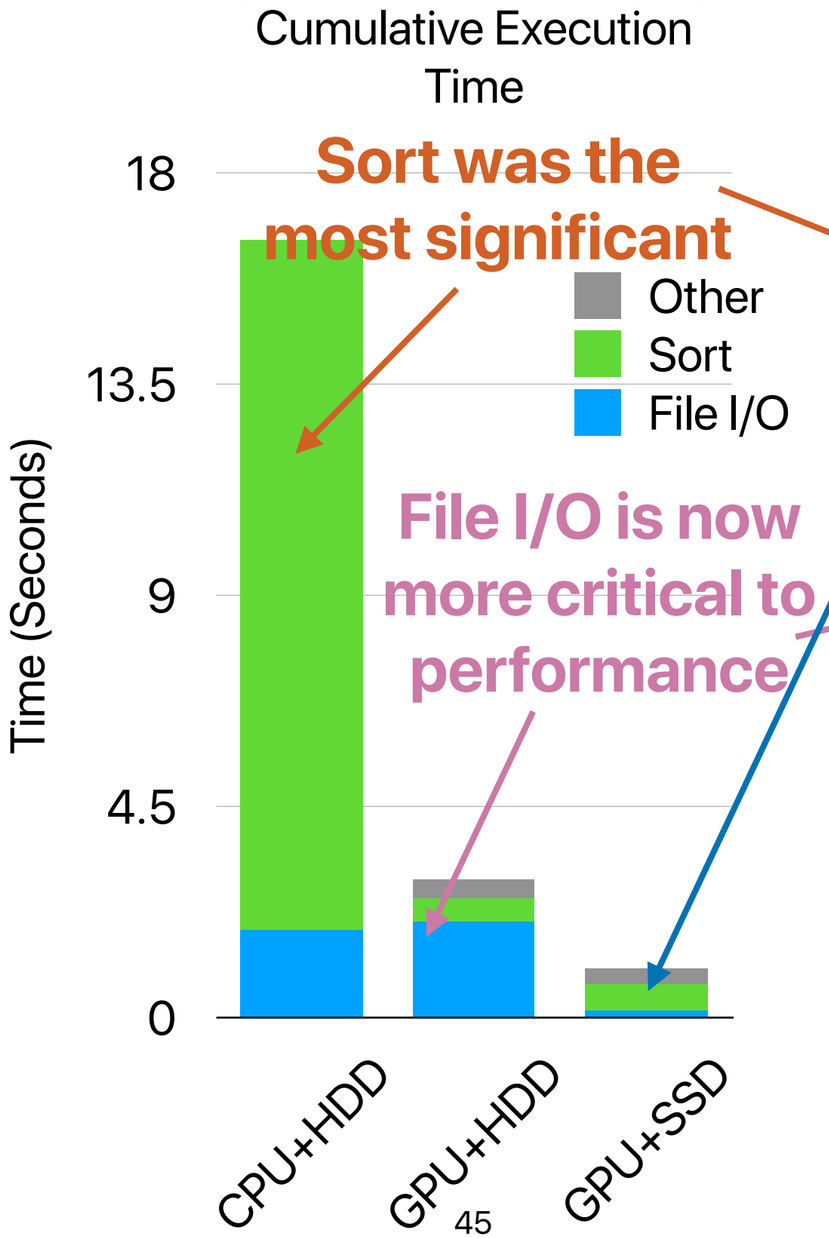
$$Speedup_{max}(f_1, \infty) = \frac{1}{(1-f_1)}$$
$$Speedup_{max}(f_2, \infty) = \frac{1}{(1-f_2)}$$
$$Speedup_{max}(f_3, \infty) = \frac{1}{(1-f_3)}$$
$$Speedup_{max}(f_4, \infty) = \frac{1}{(1-f_4)}$$

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

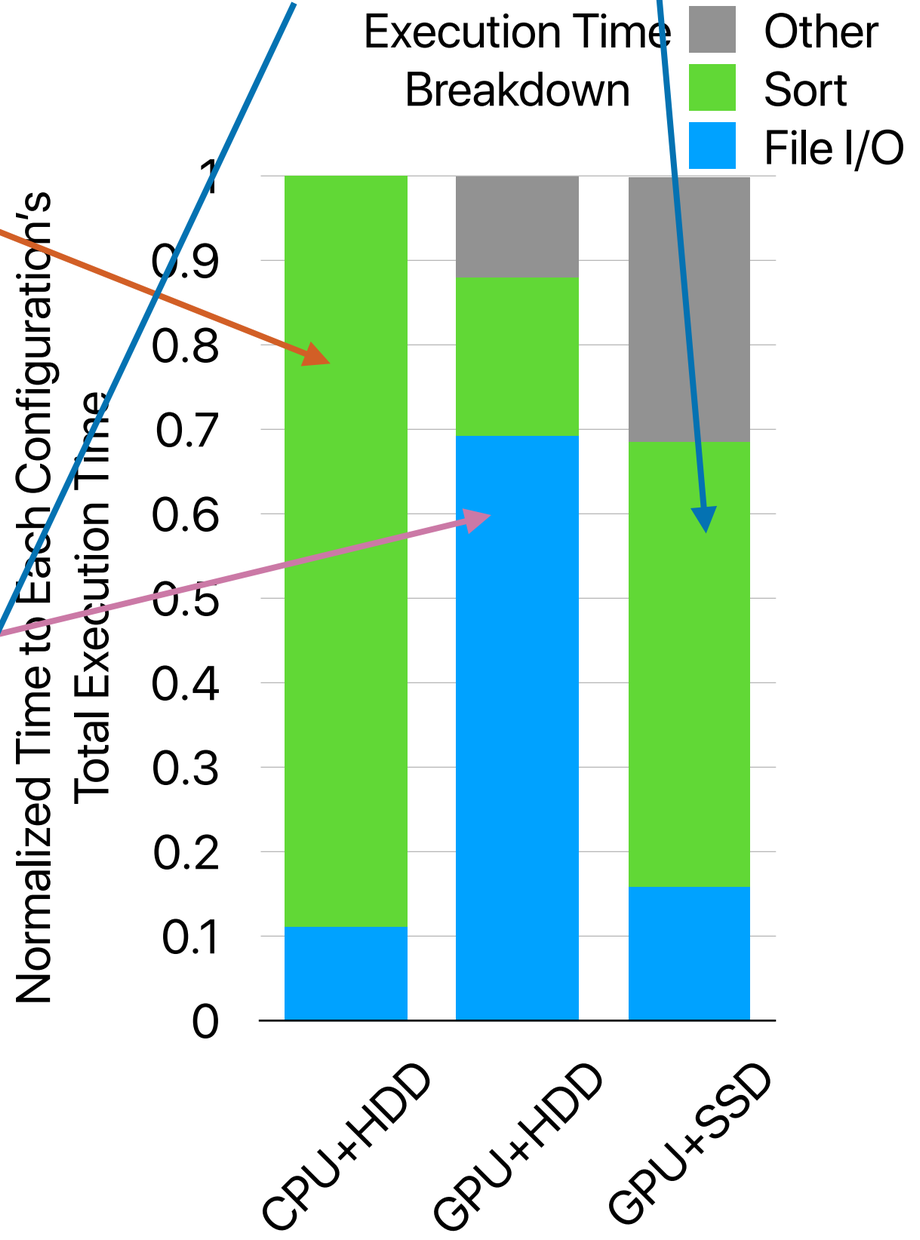




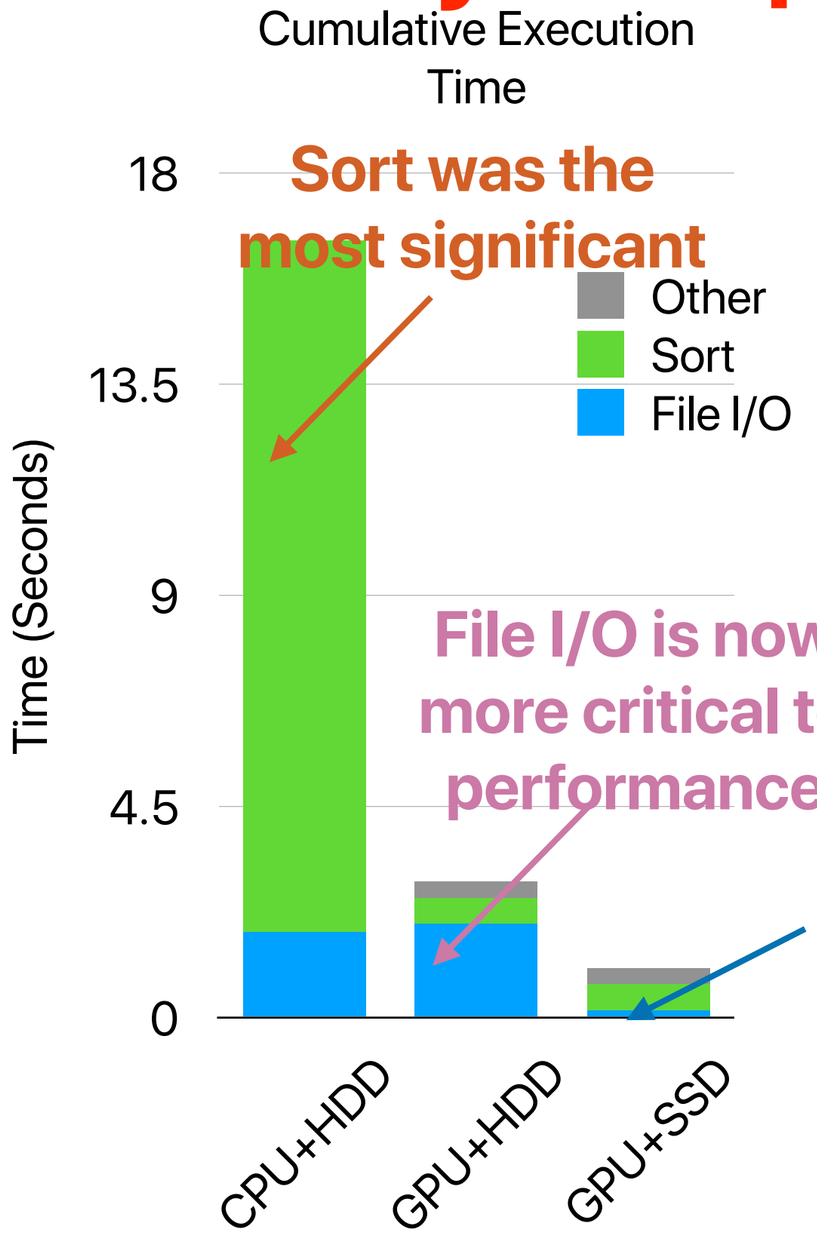
Sort was the most significant

File I/O is now more critical to performance

GPU kernel is again more important



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

Takeaways: find the right thing to do

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

Merge Sort

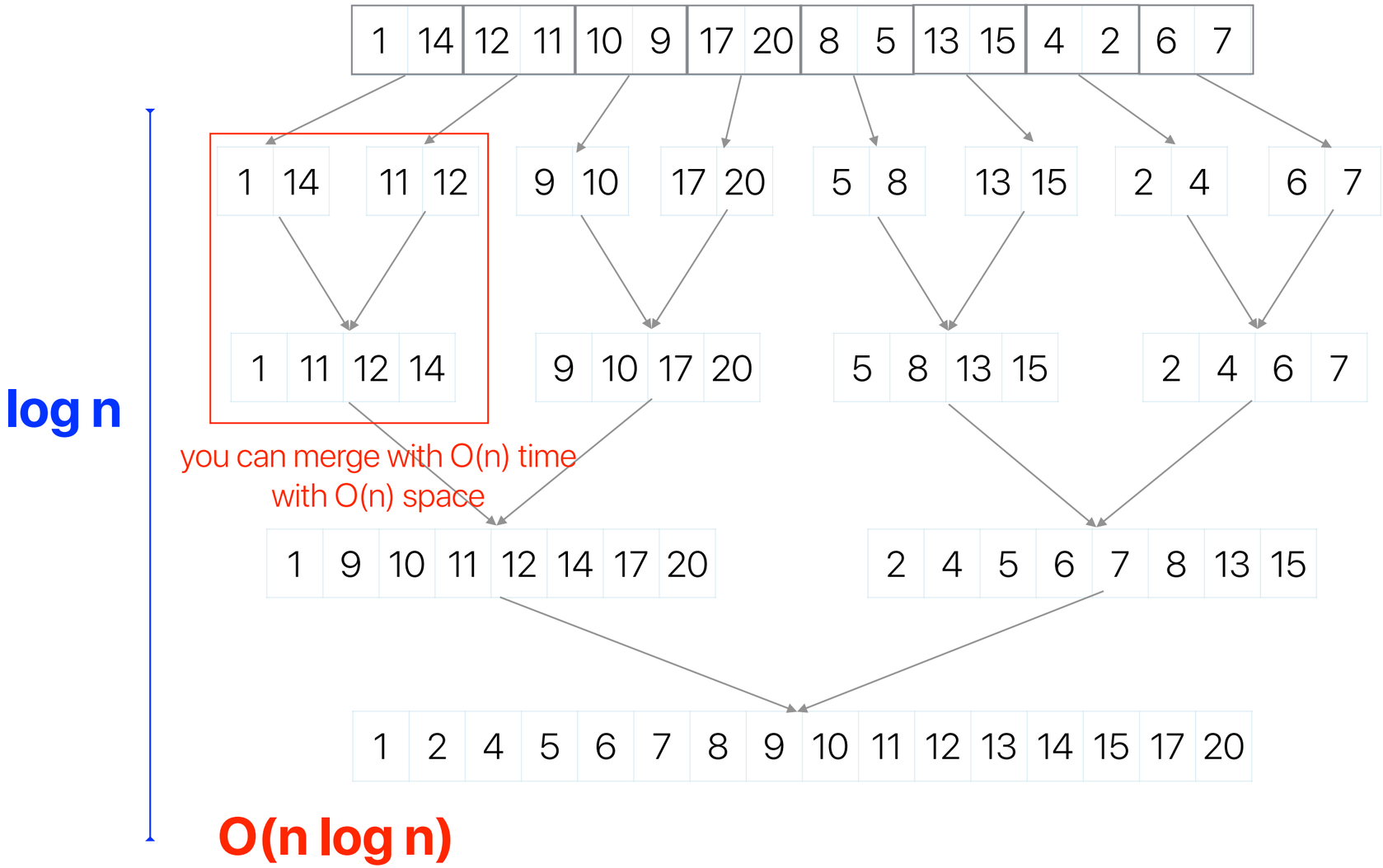
$O(nlog_2n)$

Bitonic Sort

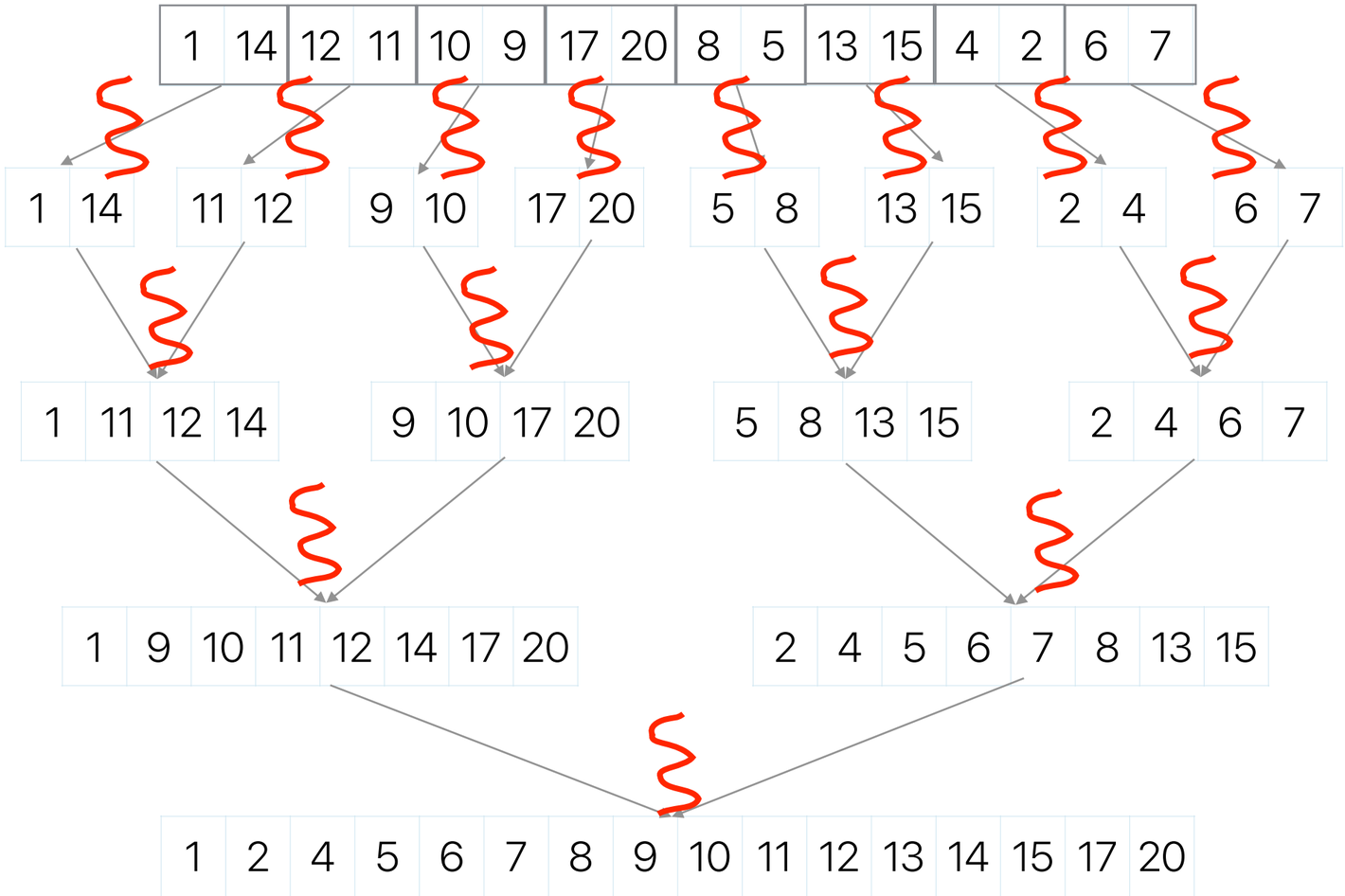
$O(nlog_2^2n)$

```
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



What's the speedup of merge sort using Amdahl's Law

The degree of parallelism is $1, 2, 4, \ldots, \frac{n}{2}$

at step $1, 2, 3, \ldots, log_2(n)$

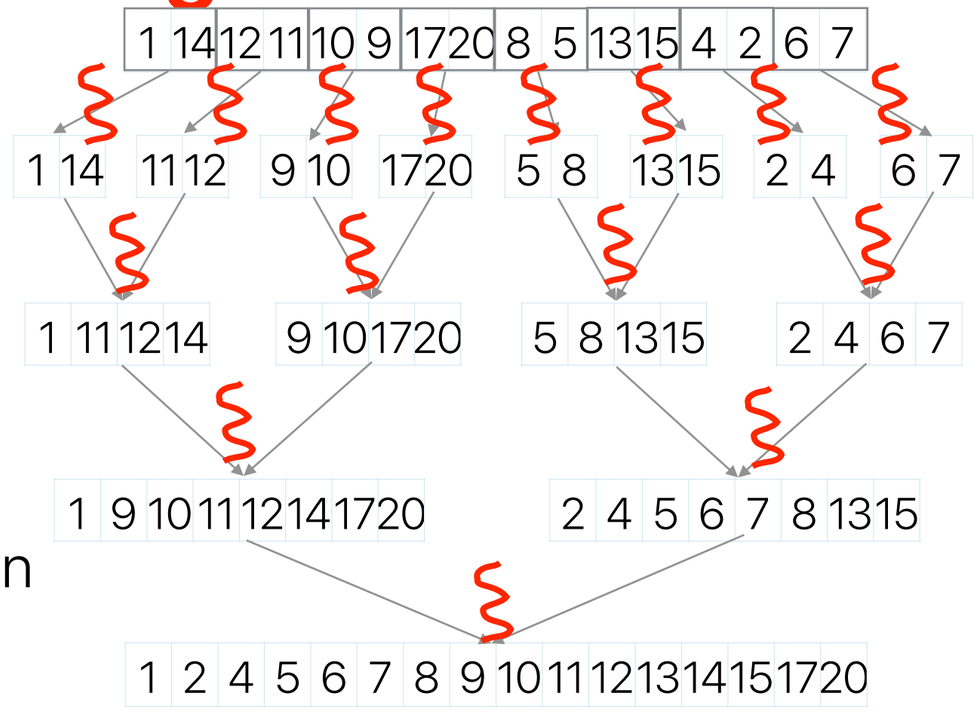
The ideal speedup of each step is $1, 2, 4, \ldots, \frac{n}{2}$ or say

$1, 2, 4, \ldots, 2^{log_2(n)-1}$ if we have **unlimited** parallelism

if we assume equal amount of time in each step in the baseline,

each step is going to take $\frac{1}{lg(n)}$ portion of time in the baseline, an

the first $\frac{1}{lg(n)}$ is not parallelizable (i.e., $(1-x) = \frac{1}{lg(n)}$)

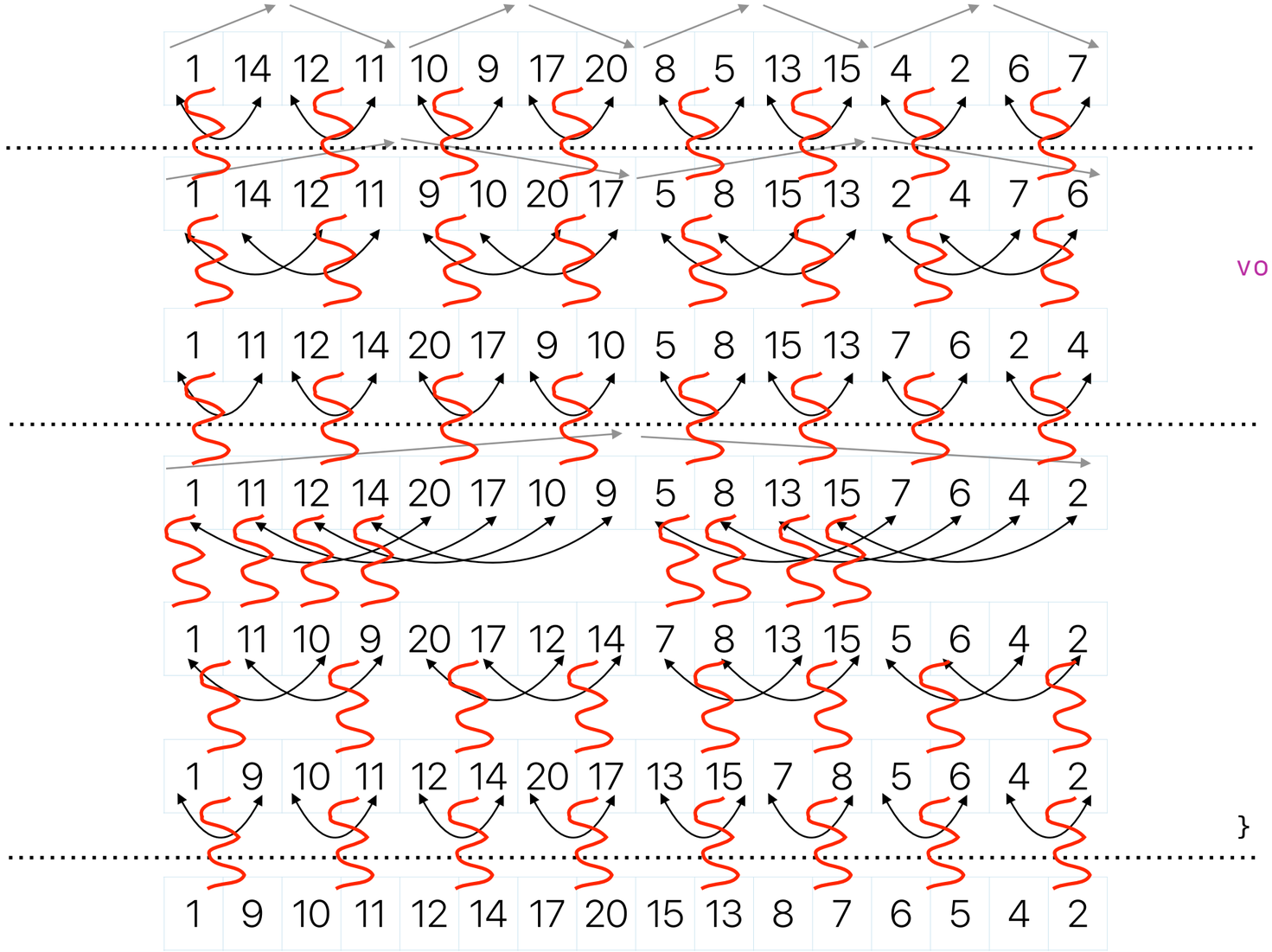


So the Amdahl's Law's evaluation will become

$$\frac{1}{\frac{1}{lg(n)} + \frac{1}{lg(n)\times 2} + \frac{1}{lg(n)\times 4} + \ldots + \frac{1}{lg(n)\times 2^{lg(n)-1}}} = \frac{1}{\frac{1}{lg(n)}(1 + \frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{2^{lg(n)-1}})}$$

$$= \frac{1}{\frac{1}{lg(n)}(1 + 1 - \frac{1}{2^{lg(n)-1}})} = \frac{1}{\frac{2}{lg(n)}} = \frac{lg(n)}{2}$$

Bitonic sort

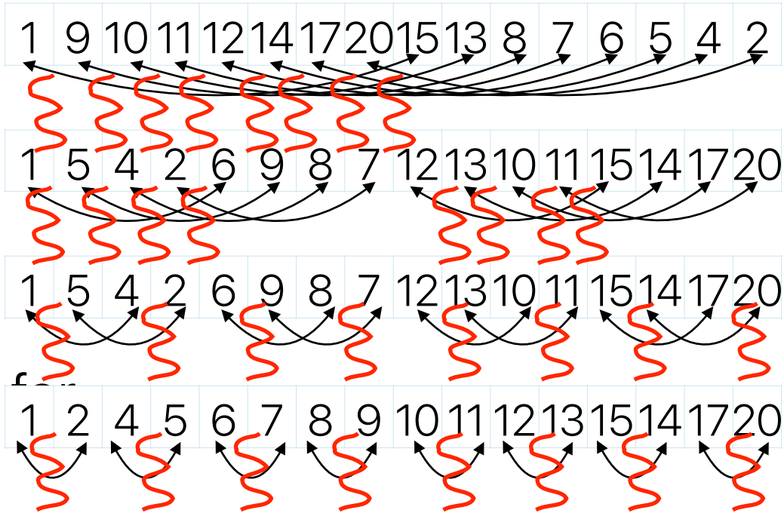



```
void BitonicSort() {

    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);
                }
            }
        }
    }
}
```

What's the speedup of bitonic sort using Amdahl's Law

The degree of parallelism is always $\frac{n}{2}$

at step $1,\ 2,\ 3,\ \ldots,\ log_2^2(n)$



The ideal speedup of each step is $\frac{n}{2}$ if we have **unlimited** parallelism & for each step of bitonic sort. However, bitonic sort will have lg(n)x more steps than merge sort.

If the baseline is merge sort — the speedup of using bitonic sort is

$$\frac{1}{\frac{1 \times lg(n)}{\frac{n}{2}}} = \frac{n}{2lg(n)} > \frac{lg(n)}{2}$$

What if we have only p processors?

For **bitonic sort**,The degree of parallelism is always $\frac{n}{2}$

at step $1,\ 2,\ 3,\ \ldots,\ log_2^2(n)$, but we have only p processors...

The theoretical speedup of each step is $max(\frac{n}{2}, p) = p$ since n is very likely to

be larger than **p** & assume equal amount of time in each step in the baseline

So the Amdahl's Law's evaluation will become $\frac{1}{\frac{p}{lg(n)}} = \frac{p}{lg(n)} = \frac{1024}{30} = 34.1333333$

What about merge sort? $= \frac{1}{\frac{1}{lg(n)}(1 + \frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{p})}$

What if p=1024, n=1M=2^{30}? $= \frac{1}{\frac{1}{30}(1 + \frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{1024} + 20 \times \frac{1}{30})} = 14.862119$

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 algorithm complexity becomes less important 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: find the right thing to do

- 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 — $Speedup_{enhanced}(f, s) = \frac{1}{(1-f) + \frac{f}{s}}$ $Speedup_{max}(f, \infty) = \frac{1}{(1-f)}$
  - 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
  - Corollary 4 — Exploiting more parallelism from a program is the key to performance gain in modern architectures

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

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

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: find the right thing to do

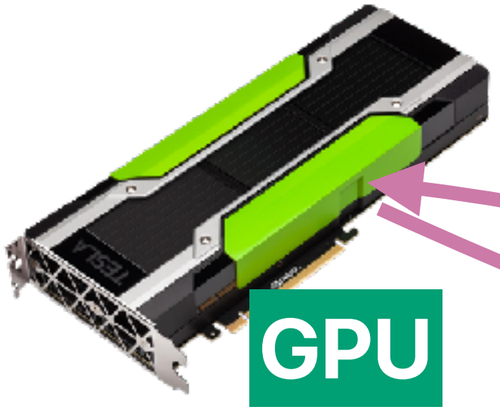
- Definition of "Speedup of Y over X" or say Y is n times faster than X: $speedup_{Y\_over\_X} = n = \frac{Execution\ Time_X}{Execution\ Time_Y}$
- Amdahl's Law — $Speedup_{enhanced}(f, s) = \frac{1}{(1-f) + \frac{f}{s}}$ $Speedup_{max}(f, \infty) = \frac{1}{(1-f)}$
  - Corollary 1 — each optimization has an upper bound
  - Corollary 2 — make the common case (the most time consuming case) fast!
  - Corollary 3 — Optimization has a moving target
  - Corollary 4 — Exploiting more parallelism from a program is the key to performance gain in modern architectures
  - Corollary 5 — Single-core performance still matters

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

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

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

Most cases, parallelism is not “tax-free”



GPU

16GB/sec

Memory

Processor

38.4GB/sec

Storage

Additional data movements

Process of calling GPU function in your applications

```
extern void vector_mul_double_cuda(double *va, double *vb, unsigned long int size, int p, int iter)
{
    double *d_va, *d_vb;

    // Allocate GPU memory for input a & b
    cudaMalloc((void **) &d_va, sizeof(double)*size);
    cudaMalloc((void **) &d_vb, sizeof(double)*size);

    // Copy a & b into GPU memory
    cudaMemcpy(d_va, va, sizeof(double)*size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_vb, vb, sizeof(double)*size, cudaMemcpyHostToDevice);

    unsigned int grid_size = 1024;
    unsigned int grid_cols = (size + grid_size - 1) / (p*1024);
    perf_stats[NUM_OF_THREADS] = grid_cols;
    dim3 dimGrid(grid_cols, 1);
    dim3 dimBlock(grid_size, 1);

    // Launch kernel
    vector_mul_double_cuda_kernel<<<(size + grid_size - 1) / (p*1024), 1024>>>(d_va, d_vb, size, p, iter);

    // Copy a & b back to the main memory
    cudaMemcpy(va, d_va, sizeof(double)*size,cudaMemcpyDeviceToHost);

    cudaFree(d_va);
    cudaFree(d_vb);
   return;
}
```

Additional data movements

Additional data movements

What if parallelism is not "tax-free"?

- Parallelization Overhead
  - Preparing/exchanging/synchronizing data
  - Additional function calls/control overhead
- The "$1 - f$" will potentially slowdown by a factor of $perf(r)$

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

Amdahl's Law considering overhead

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

- ***r*** is some other parameter that affects the overhead
  - input size?
  - degree of parallelism?
- *perf(r)* should be <=1 (i.e., slowdown)
- 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., a speedup of $\frac{1}{10}$)...

$$Speedup = \frac{1}{\frac{(1-f)}{perf(r)} + \frac{f}{s}} = \frac{1}{\frac{(1-0.9)}{\frac{1}{10}} + \frac{0.9}{9}} = 0.91\times$$

Takeaways: find the right thing to do!

- 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

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

Announcement

- Assignment 1 **due Thursday**
  - We cannot help you at the last minute — please start early
  - Watch before you start https://youtu.be/m7OoY8y_lsk
  - Please always make sure you follow the exact steps in the readme and the notebook
  - Submit to the right item on Gradescope
  - Please visit an office hour if you need more assistance
- Reading quiz 3 due next **Tuesday before the lecture** — we will drop two of your least performing reading quizzes
- Check our website for slides, Gradescope for assignments, discord for discussions
- Check your grades at https://www.escalab.org/my_grades
  - If you don't have any grade, you need to make sure your gradescope account is associated with your UCRNetID@ucr.edu
  - You have to submit a course agreement to receive scores
- Youtube channel for lecture recordings: https://www.youtube.com/c/ProfUsagi/playlists

Computer
Science &
Engineering

203

つづく