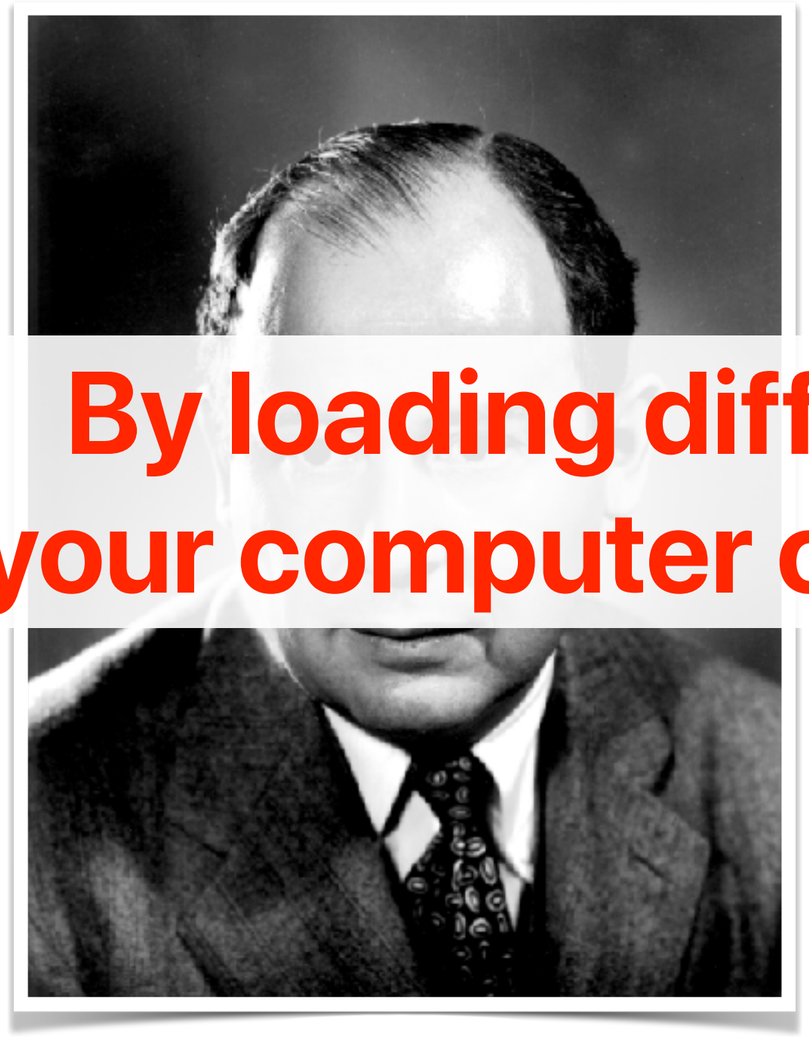
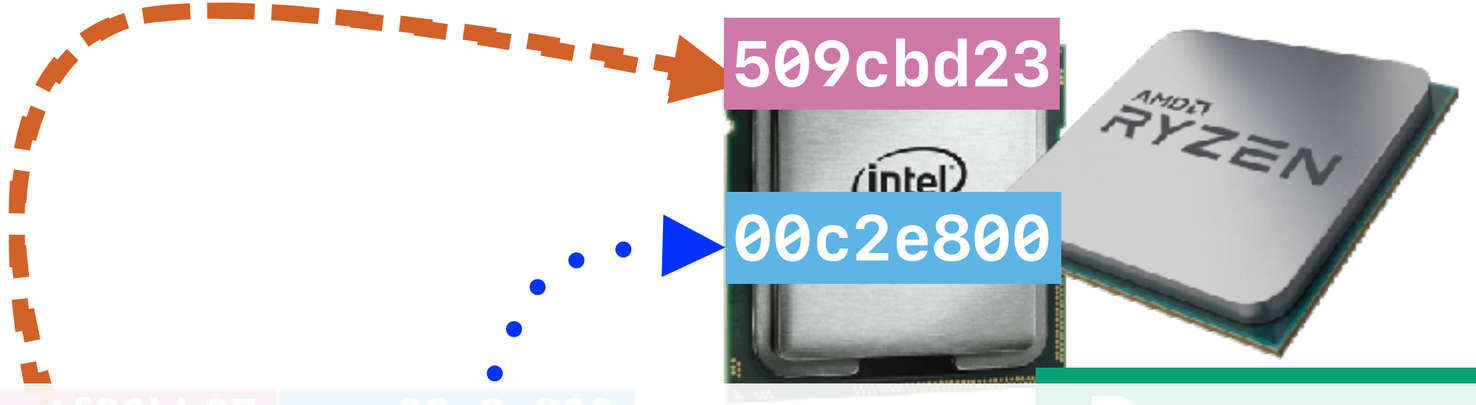
Performance (3): What's the right thing to do?

Hung-Wei Tseng

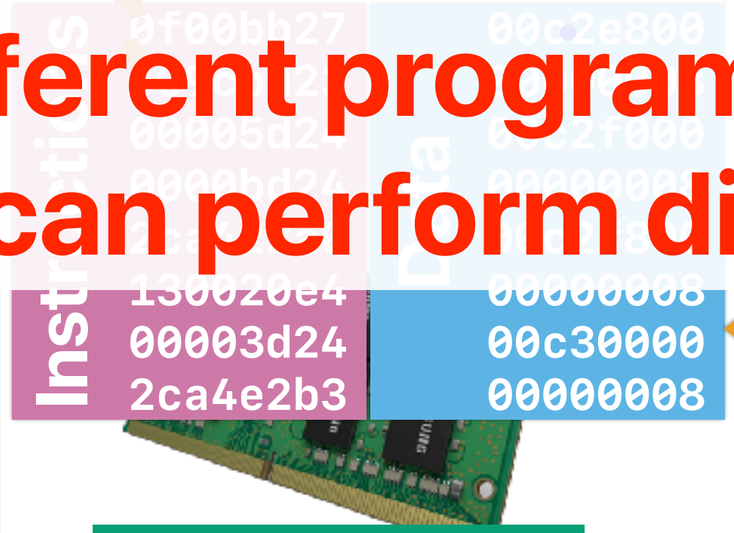
Recap: von Neumman Architecture





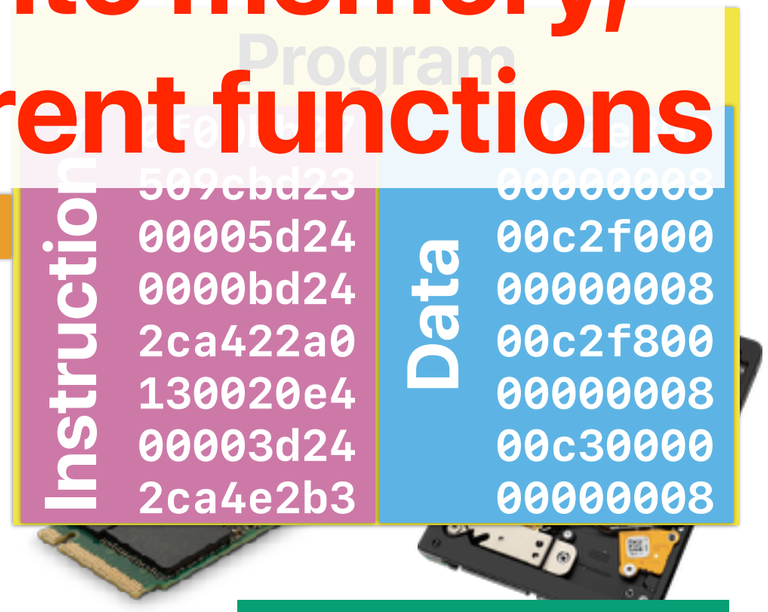
Processor

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



Memory

Program



Storage

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!

How compilers affect performance

- If we apply compiler optimizations for 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!

Compiler 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];
  }
}
```

Recap: 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 — only if programmers write code in a way facilitating optimizations!
- Complexity does not provide good assessment on real machines due to the idealized assumptions

3:00

If there is one life-changing thing that you can do, what & why is that?

Outline

- The definition of Speedup
- Amdahl’s Law and its implications

Quantitive Analysis of “Better”

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

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%	4	20%	3	60%	1
Machine Y	6 GHz	5000000000	20%	6	20%	3	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	Instructions	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%	4	20%	3	60%	1
Machine Y	6 GHz	5000000000	20%	6	20%	3	60%	1

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

$$ET_X = (5 \times 10^9) \times (20\% \times 4 + 20\% \times 3 + 60\% \times 1) \times \frac{1}{4 \times 10^9}\ sec = 2.5\ sec$$

$$ET_Y = (5 \times 10^9) \times (20\% \times 6 + 20\% \times 3 + 60\% \times 1) \times \frac{1}{6 \times 10^9}\ secs = 2\ secs$$

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

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

Takeaways: 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 — and It's Implication in the Multicore Era

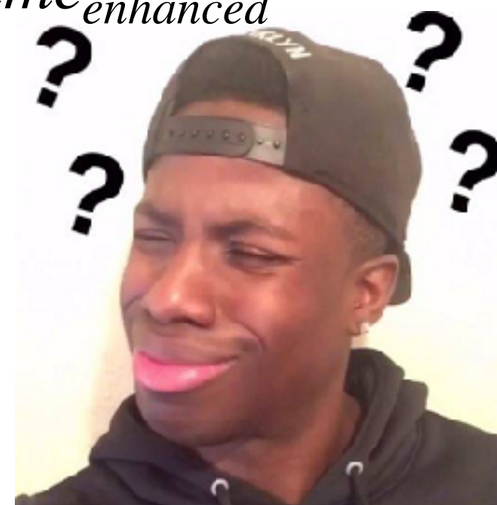
Amdahl's Law

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

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

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

Execution Time$_{baseline}$ = 1

enhanced

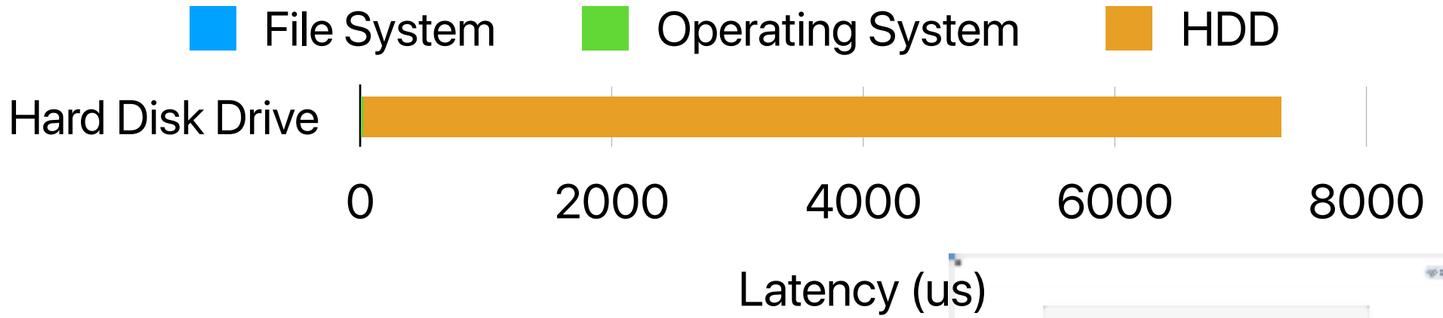
Execution Time$_{enhanced}$ = $(1 - f) + \frac{f}{s}$

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

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

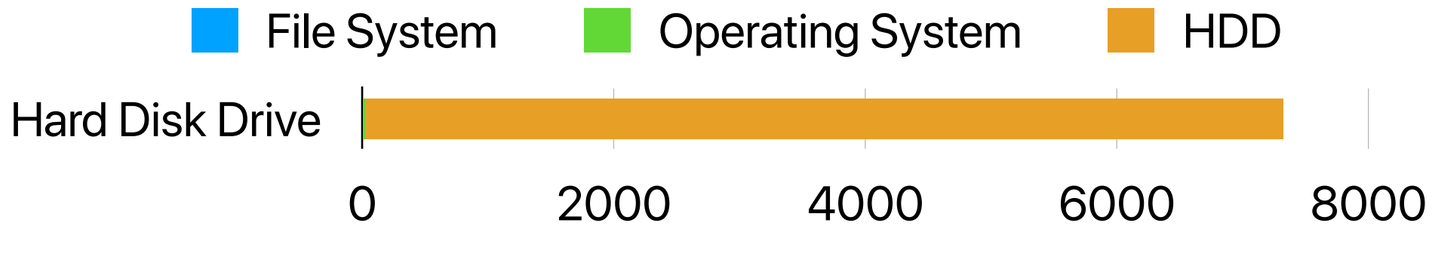
Practicing Amdahl's Law

- 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:

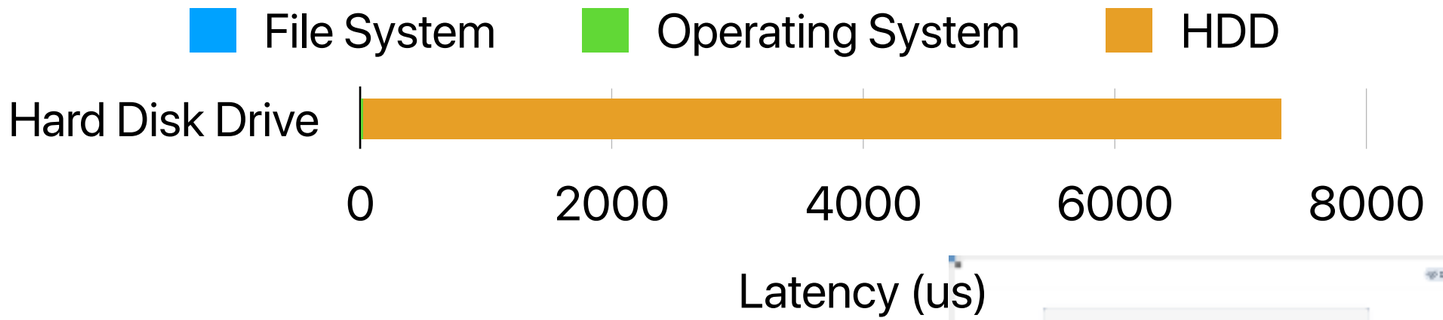
$$Speedup_{enhanced}(f_{Opt1}, f_{Opt2}, s_{Opt1}, s_{Opt2}) = \frac{1}{(1 - f_{Opt1} - f_{Opt2}) + \frac{f\_Opt1}{s\_Opt1} + \frac{f\_Opt2}{s\_Opt2}}$$

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

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

Practicing Amdahl's Law (2)

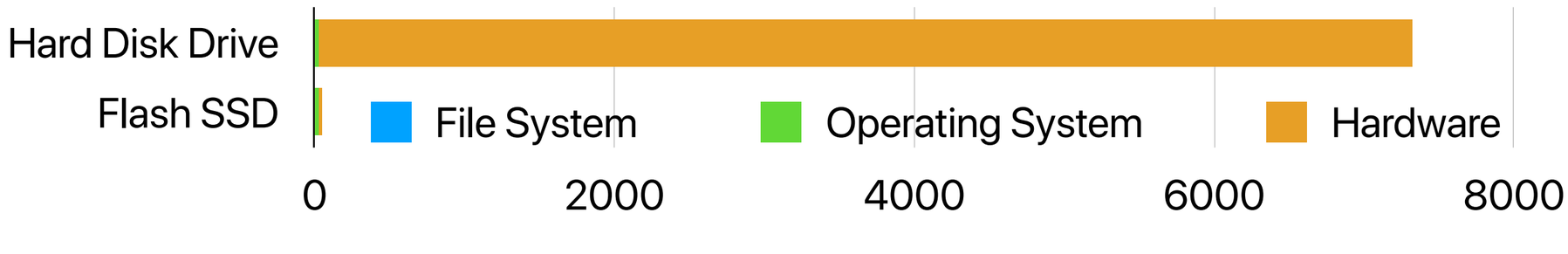
- Final Fantasy XV spends lots of time loading a map — within which period that 95% of the time on the accessing the H.D.D., the rest in the operating system, file system and the I/O protocol. If we replace the H.D.D. with a flash drive, which provides 100x faster access time and a better processor to accelerate the software overhead by 2x. 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 (2)

- 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 and a better processor to accelerate the software overhead by 2x. By how much can we speed up the map loading process?

A. ~7x
B. ~10x
C. ~17x
D. ~29x
E. ~100x

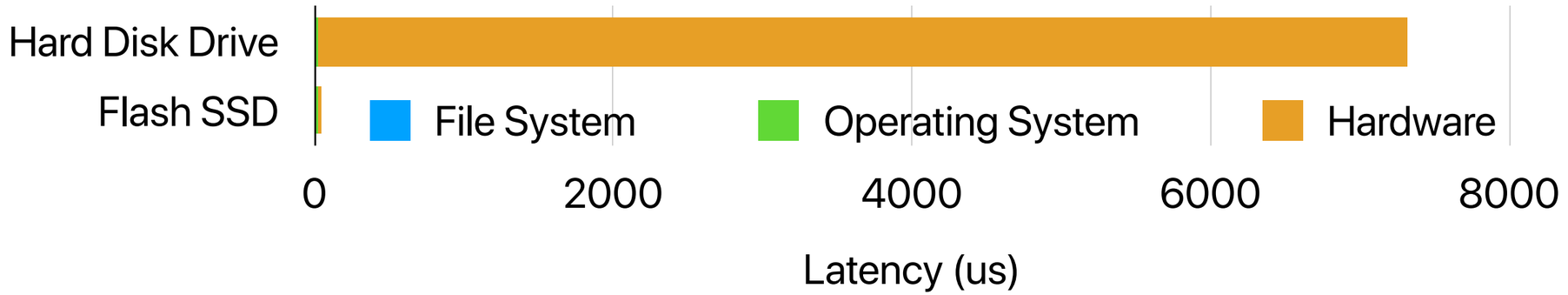


$$Speedup_{enhanced}(95\%, 5\%, 100, 2) = \frac{1}{(1 - 95\% - 5\%) + \frac{95\%}{100} + \frac{5\%}{2}} = 28.98\times$$

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?

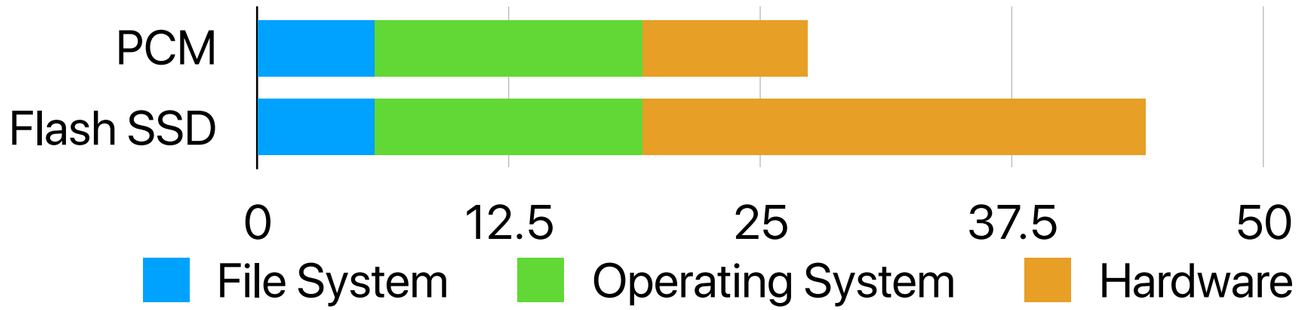
A. ~5x
B. ~10x
C. ~20x
D. ~100x
E. None of the above



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

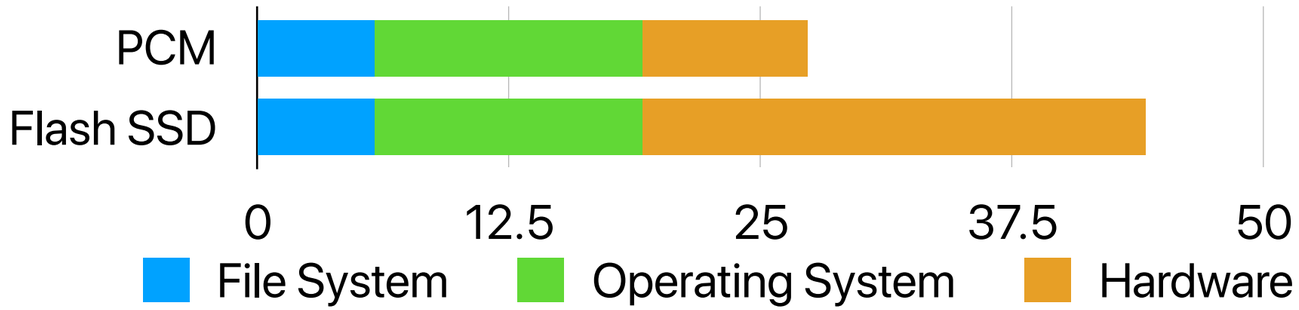
$$Speedup_{max}(f, \infty) = \frac{1}{(1-f) + \frac{f}{\infty}}$$

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

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

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!

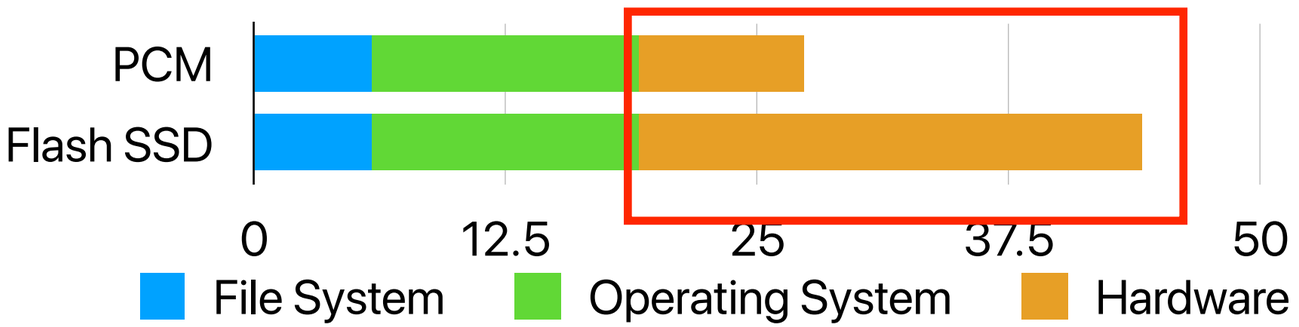
- When f is small, optimizations will have little effect.
- Common == **most time consuming** not necessarily the most frequent
- The uncommon case doesn’t make much difference
- The common case can change based on inputs, compiler options, optimizations you’ve applied, etc.

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!

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

Speedup further!

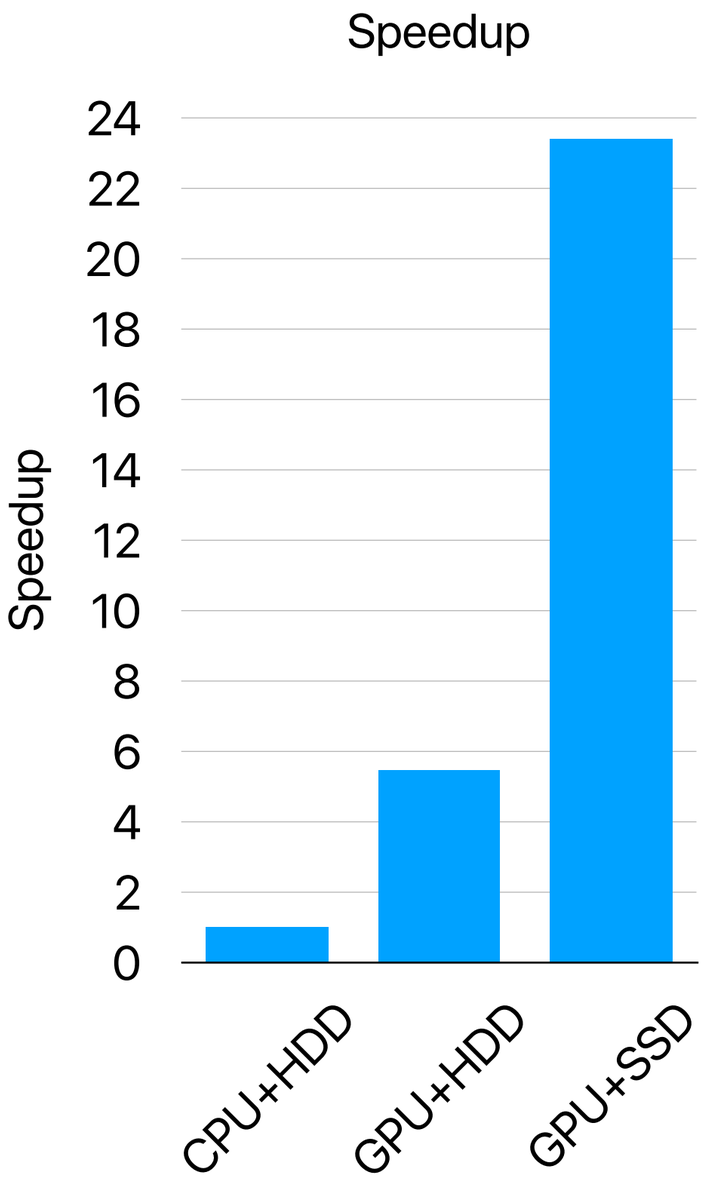


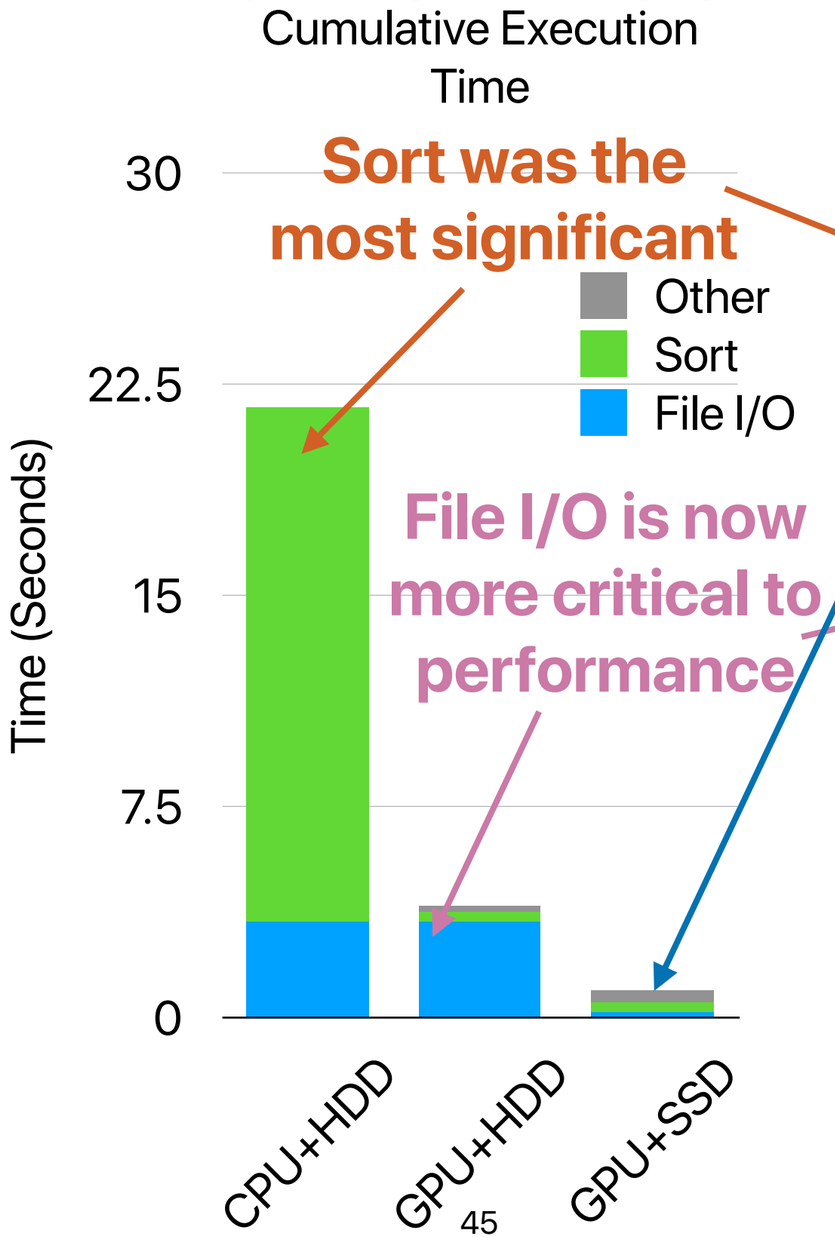
With optimization, the common becomes uncommon.

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

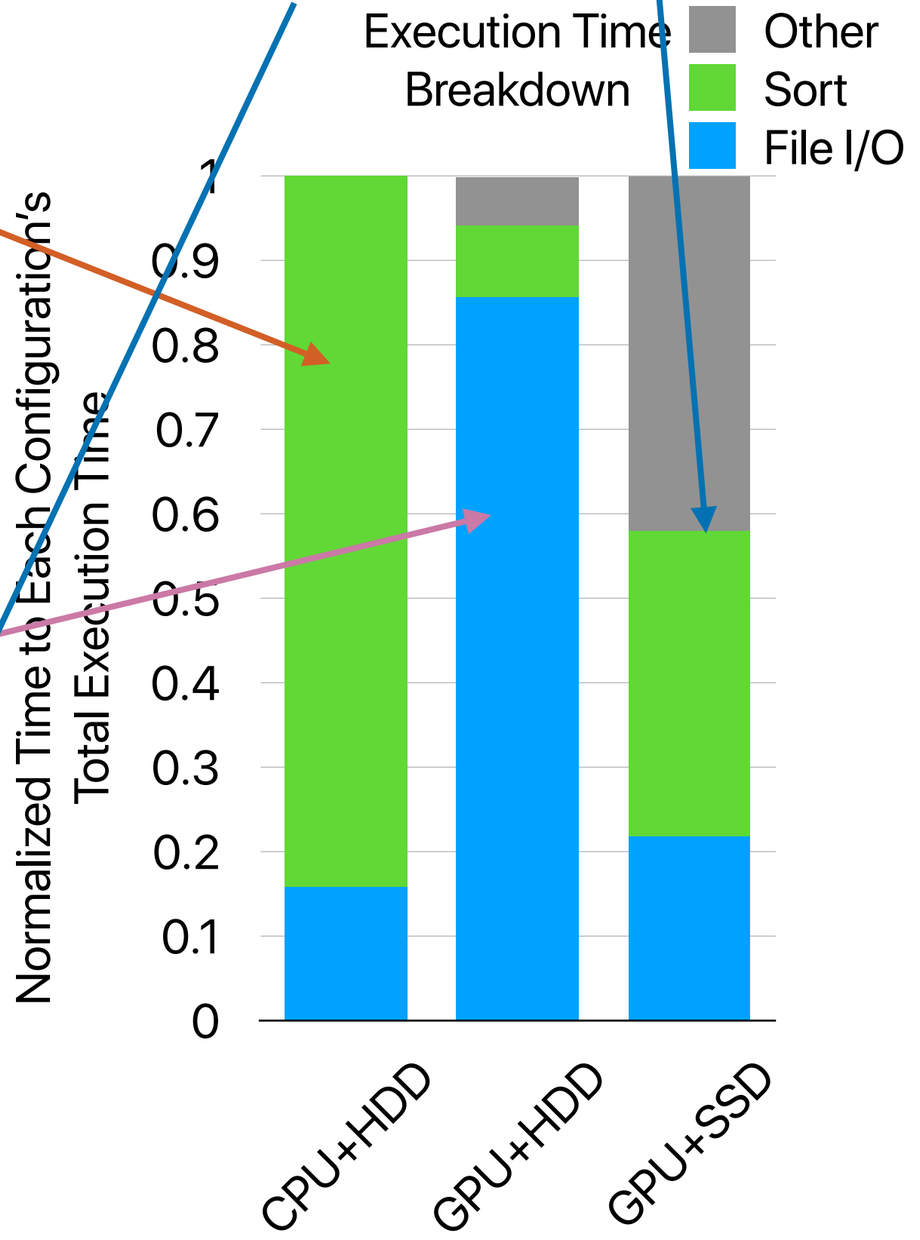




Sort was the most significant

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

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

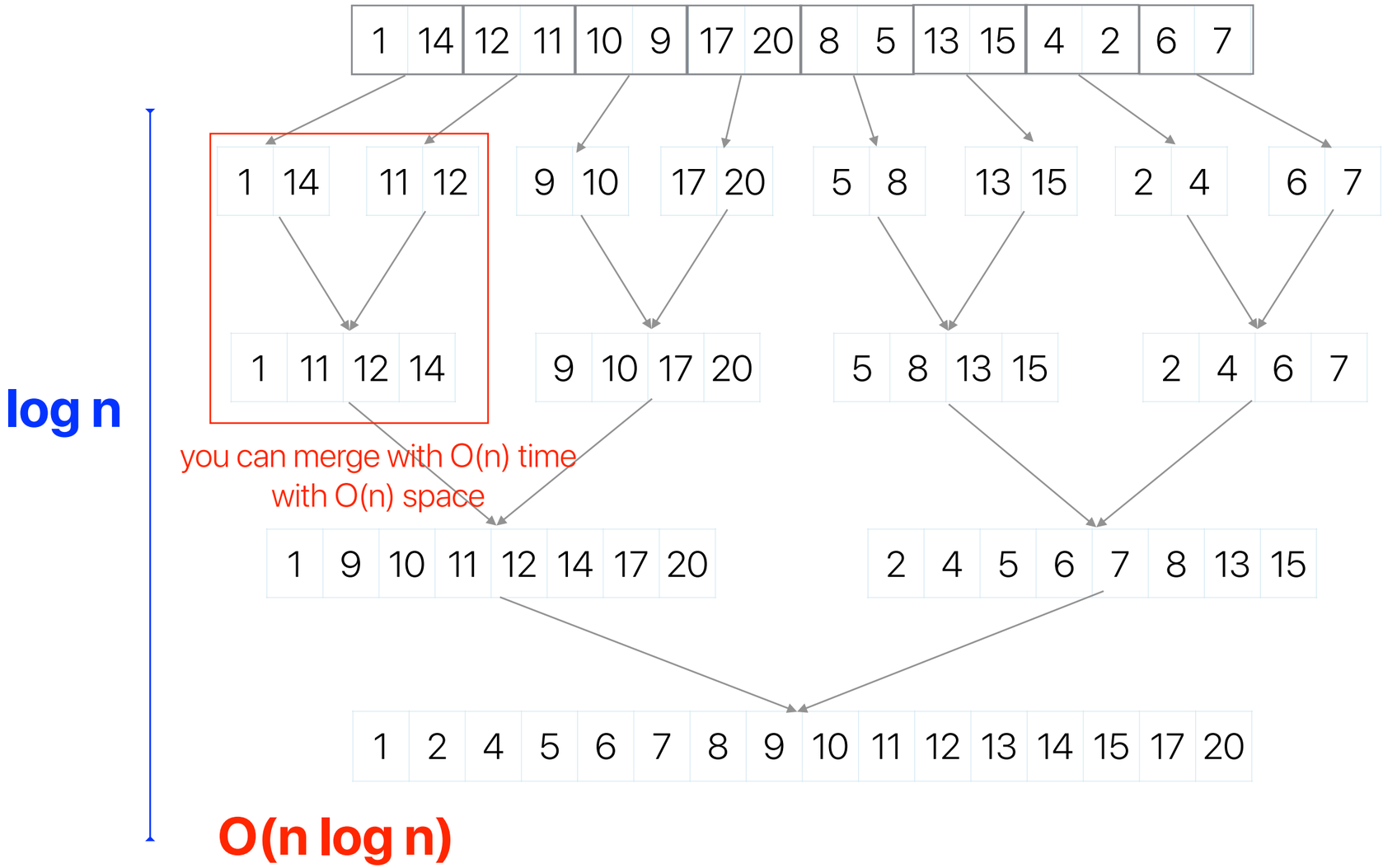
$O(nlog_2n)$

Bitonic Sort

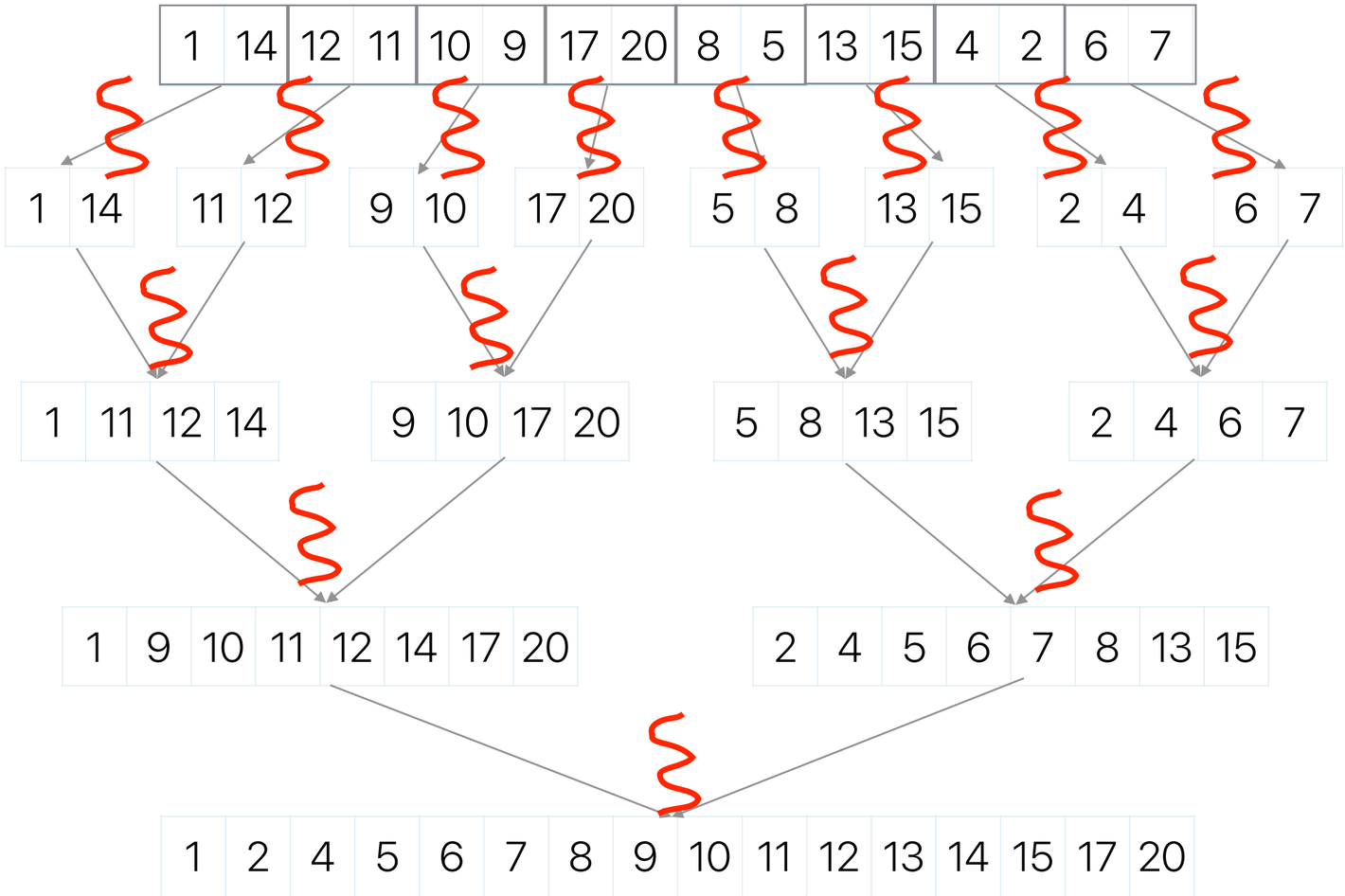
$O(nlog_2^2n)$

```
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);
                }
            }
        }
    }
}
```

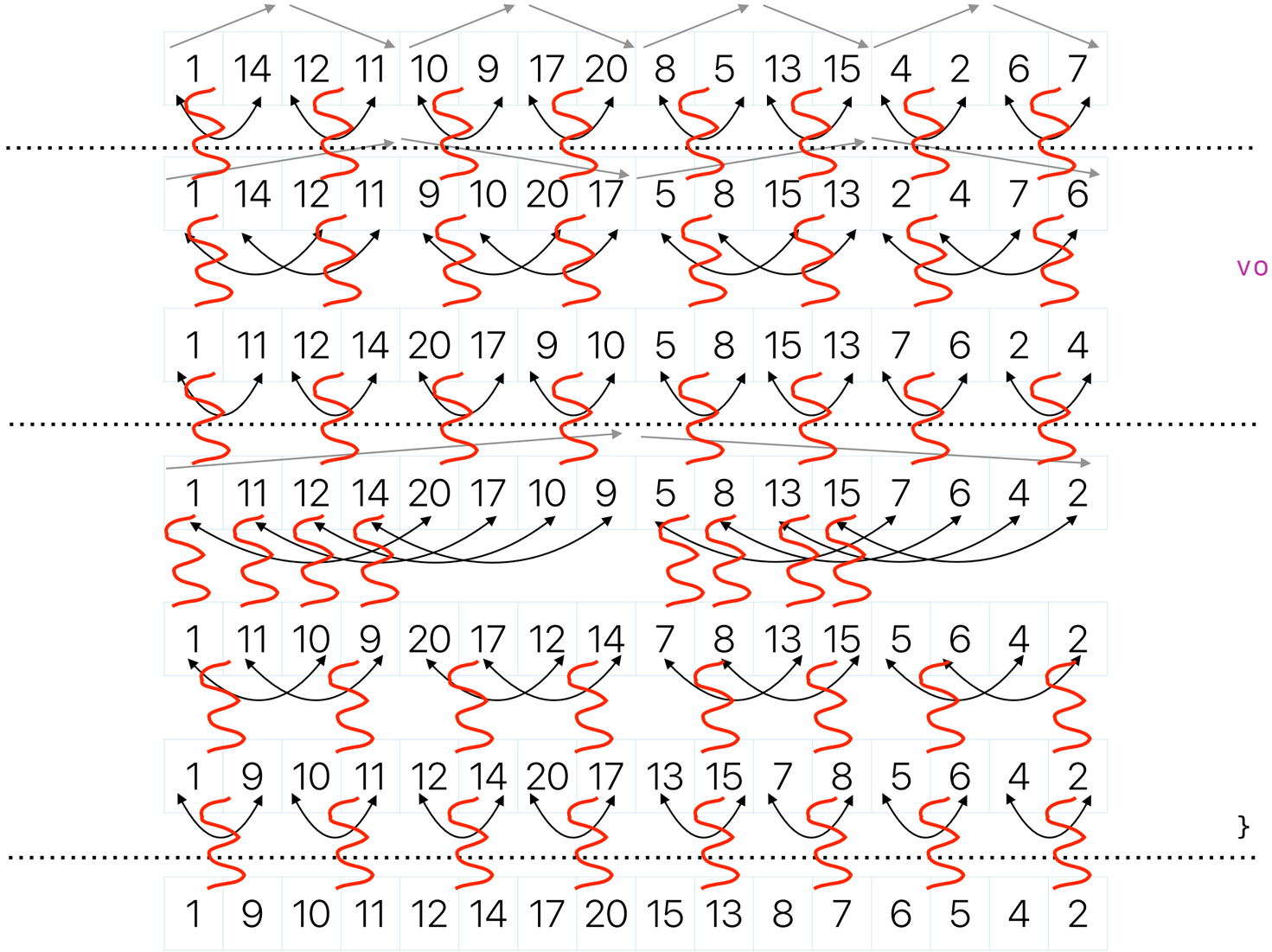
Merge sort


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

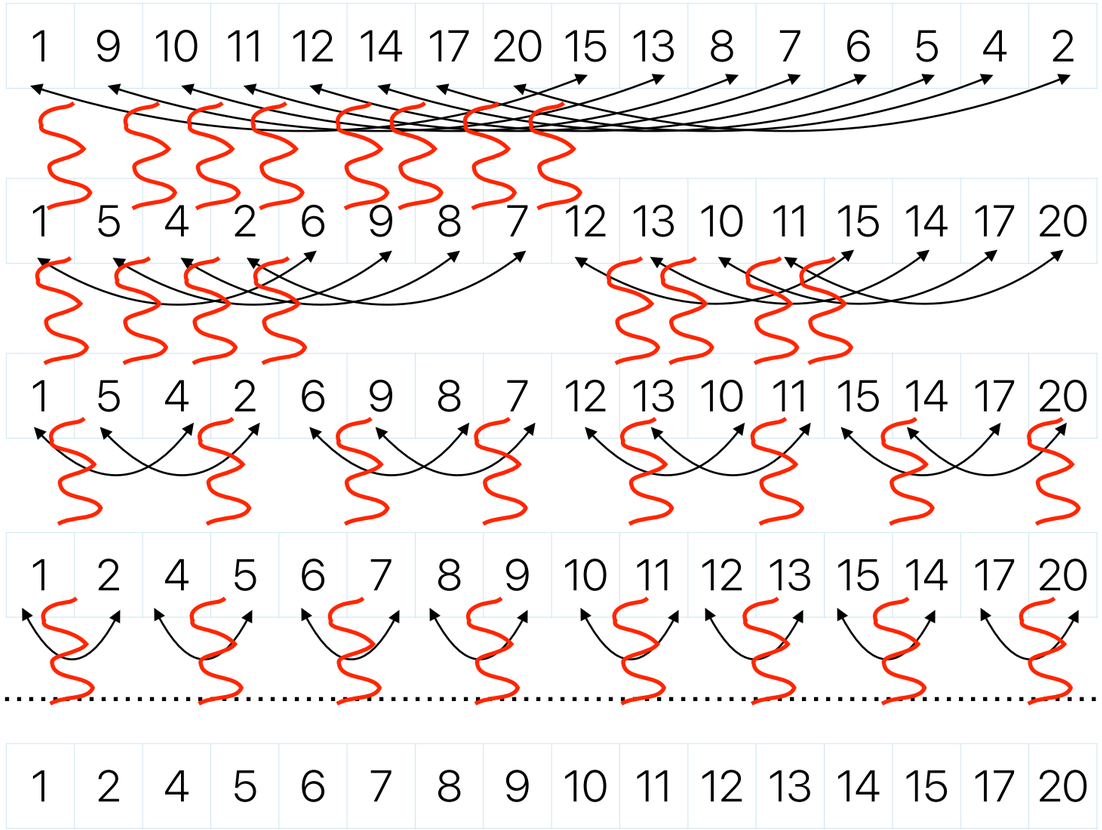


Bitonic sort




```
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

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

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

  - 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"

- 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: 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

- Reading quiz due next Monday before the lecture — we will drop two of your least performing reading quizzes
- Assignment due this Saturday
  - Last office hours by tutors on Friday
  - Discussion session might be helpful
- Check our website for slides, Gradescope for assignments, piazza for discussions
  - Don’t forget to check your access to escalab.org/datahub and piazza
  - Check your grades on escalab.org/my_grades
- Youtube channel for lecture recordings: https://www.youtube.com/c/ProfUsagi/playlists

Computer
Science &
Engineering

142

つづく