Memory Hierarchy (1): Inside out of our memory hierarchy

Hung-Wei Tseng

Disney+

Disney · PIXAR

INSIDE OUT

GET DISNEY+

TRAILER

PG 2015 • 1h 35m • Coming of age, Family, Animation

When 11-year-old Riley moves to a new city, her Emotions team up to help her through the transition. Joy, Fear, Anger, Disgust and Sadness work together, but when Joy and Sadness get lost, they must journey through unfamiliar places to get back home.

Recap: von Neumman Architecture

509cbd23

00c2e800

Processor

Instructions	Data
0f00bb27	00c2e800
509cbd23	00000008
00005d24	00c2f000
0000bd24	00000008
2ca422a0	00c2f800
130020e4	00000008
00003d24	00c30000
2ca4e2b3	00000008

Memory

Program

Instructions	Data
0f00bb27	00c2e800
509cbd23	00000008
00005d24	00c2f000
0000bd24	00000008
2ca422a0	00c2f800
130020e4	00000008
00003d24	00c30000
2ca4e2b3	00000008

Storage

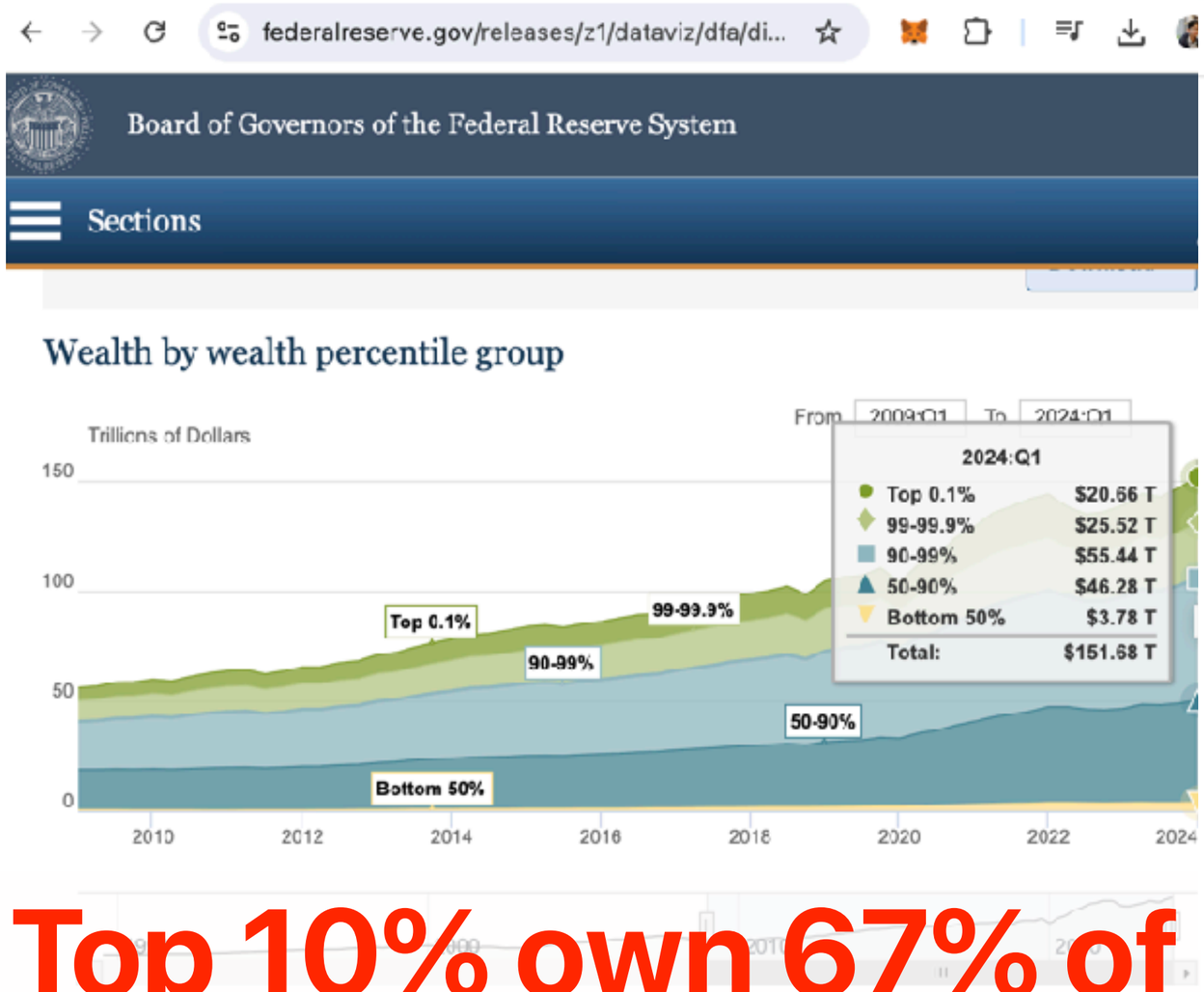
Recap: Speedup and Amdahl's Law?

- 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

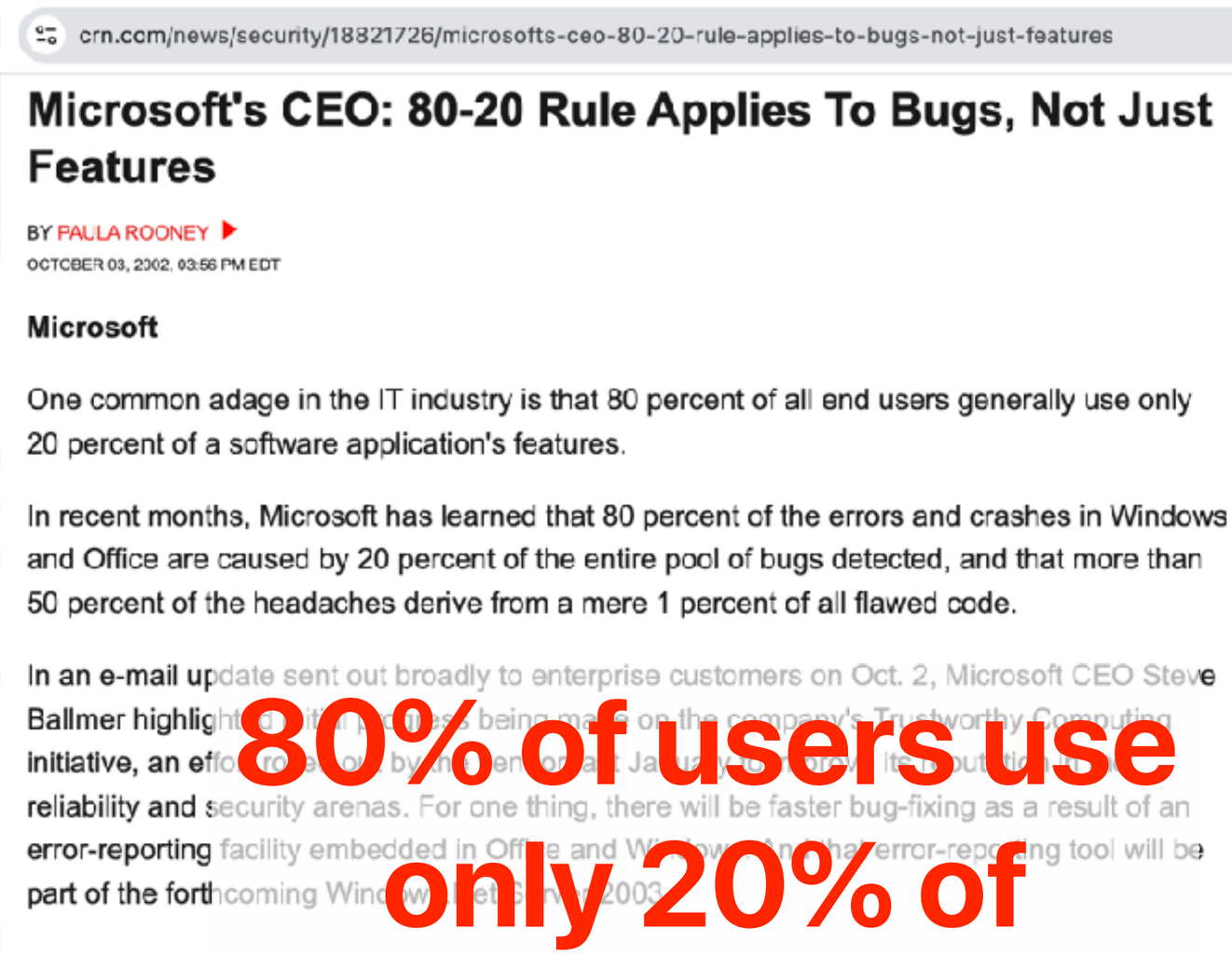
$Speedup_{max}(f_1, \infty) = \frac{1}{(1-f_1)}$
$Speedup_{max}(f_2, \infty) = \frac{1}{(1-f_2)}$
$Speedup_{max}(f_3, \infty) = \frac{1}{(1-f_3)}$
$Speedup_{max}(f_4, \infty) = \frac{1}{(1-f_4)}$

  - 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

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



Top 10% own 67% of the wealth in the U.S.



80% of users use only 20% of features

https://en.wikipedia.org/wiki/Pareto_principle

In the world | For students | Reddit

AI Overview Learn more

Native English speakers use around 2,000–3,000 words in daily life, which account for about 80% of their communication. These words include common verbs like "eat," "sleep," "work," "talk," and "walk," as well as pronouns like "I," "you," "he," and "she," and basic nouns like "house," "car," "food," and "water". The most commonly used words in English are "the," "be," and "to".

Eton Institute · LinkedIn · 1y
How many words are in the English language (95/5 RULE)?

GeeksforGeeks
Daily Used English Words: List of 100+ Most Common Words
May 6, 2024 — Words used in daily life often include common verbs like...

Show more

With 2,500 to 3,000 words, you can understand 90% of everyday English conversations, English newspaper and magazine articles, and English used in the workplace.

ef.edu
https://www.ef.edu › english-vocabulary › top-3000-w...
3000 most common words in English - Examples

AI Overview

The number of words in an English dictionary depends on the dictio whether it includes obsolete words, combinations, and phrases:

Oxford English Dictionary

The second edition of this 20-volume dictionary includes 171,476 words in c 47,156 obsolete words, and around 9,500 derivative words. It also includes combinations and derivatives, and 169,000 phrases and combinations, for a over 600,000 word forms. The dictionary is updated annually to include new and new meanings for existing words.

Webster's Third New International Dictionary

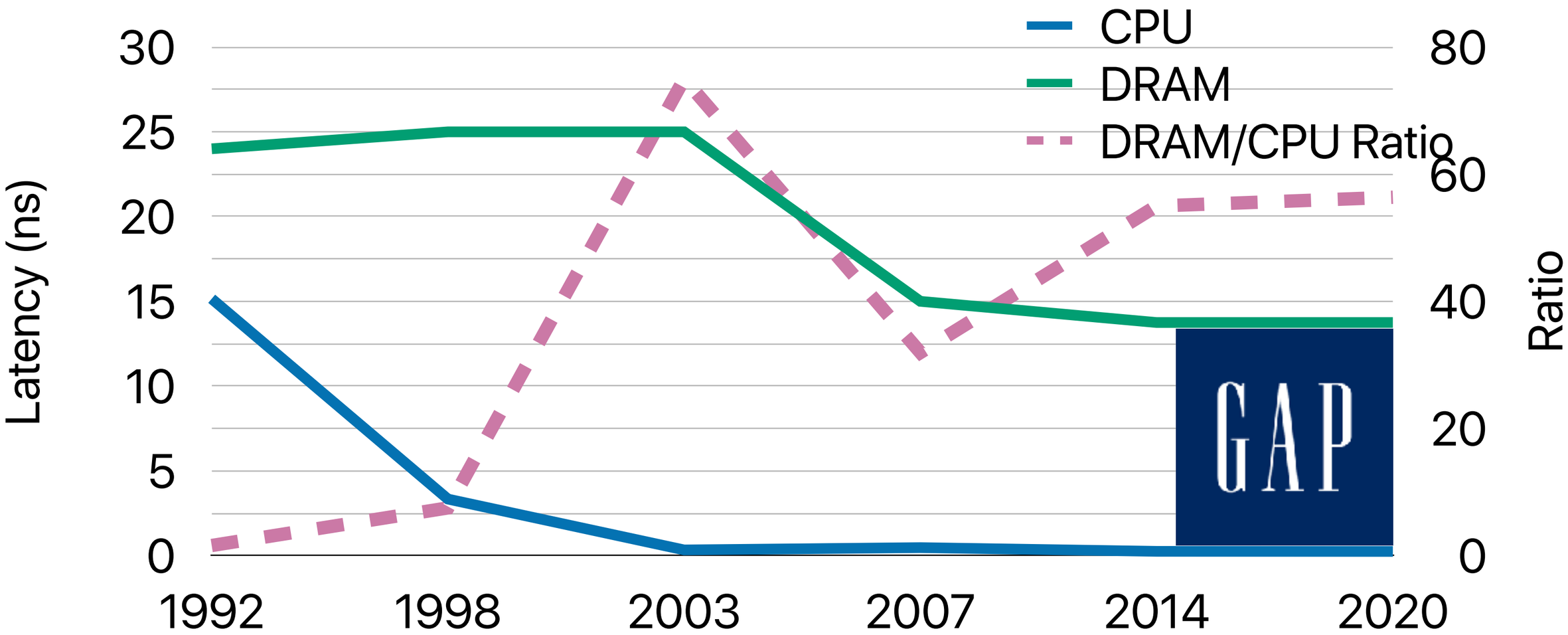
This dictionary, along with its 1993 Addenda Section, includes around 470,0 entries.

You only need to know 2% English words to understand 90% of conversations

Outline

- The memory-all problem
- The “predictable” code behavior
- Designing a cache that captures the predictability

The “latency” gap between CPU and DRAM



CPU	i486	Pentium II	Pentium 4	Core 2	Core i7-4790K	Core i5-10600K
DRAM	SDRAM	DDR	DDR2	DDR3	DDR4	DDR5

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

The impact of “slow” memory

- Assume that we have a processor running @ 4 GHz and a program with 20% of load/store instructions. If the instruction has no memory access and the processor already fetches the instruction, the CPI is just 1. Now, consider we have DDR5. The program is well-optimized so precharge is never necessary — the memory access latency is 13.75 ns. What’s the average CPI (pick the closest one)?
  A. 9
  B. 12
  C. 15
  D. 56
  E. 67

The impact of "slow" memory

- Assume that we have a processor running @ 4 GHz and a program with 20% of load/store instructions. If the instruction has no memory access and the processor already fetches the instruction, the CPI is just 1. Now, consider we have DDR5. The program is well-optimized so precharge is never necessary — the memory access latency is 13.75 ns. What's the average CPI (pick the closest one)?

  A. 9
  B. 12
  C. 15
  D. 56
  E. 67

$$CPU\ cycle\ time = \frac{1}{4 \times 10^9} = 0.25ns$$

$$Each\ DRAM\ access = \frac{13.75}{0.25} = 55\ cycles$$

$$CPI_{average} = 1 + 100\% \times 55 + 20\% \times 55 = 67\ cycles$$

Don't forget, instructions are also from "memory"

$$\frac{66}{67} = 98.5\%$$ **of time, we're dealing with memory accesses!**

20% is under-estimating …

Instruction class	MIPS examples	HLL correspondence	Frequency	
			Integer	Ft. pt.
Arithmetic	add, sub, addi	Operations in assignment statements	16%	48%
Data transfer	lw, sw, lb, lbu, lh, lhu, sb, lui	References to data structures, such as arrays	35%	36%
Logical	and, or, nor, andi, ori, sll, srl	Operations in assignment statements	12%	4%
Conditional branch	beq, bne, slt, slti, sltiu	*If* statements and loops	34%	8%
Jump	j, jr, jal	Procedure calls, r eturns, and *case/switch* statements	2%	0%

FIGURE 2.48 MIPS instruction classes, examples, correspondence to high-level program language constructs, and percentage of MIPS instructions executed by category for the average integer and floating point SPEC CPU2006 benchmarks.

Figure 3.24 in Chapter 3 shows average percentage of the individual MIPS instructions executed.

Recap: Speedup and Amdahl's Law?

- 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 $Speedup_{parallel}(f_{parallelizable}, \infty) = \frac{1}{(1-f_{parallelizable})}$
  - Corollary 5: Single-core performance still matters

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

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

Take-aways: inside out our memory hierarchy

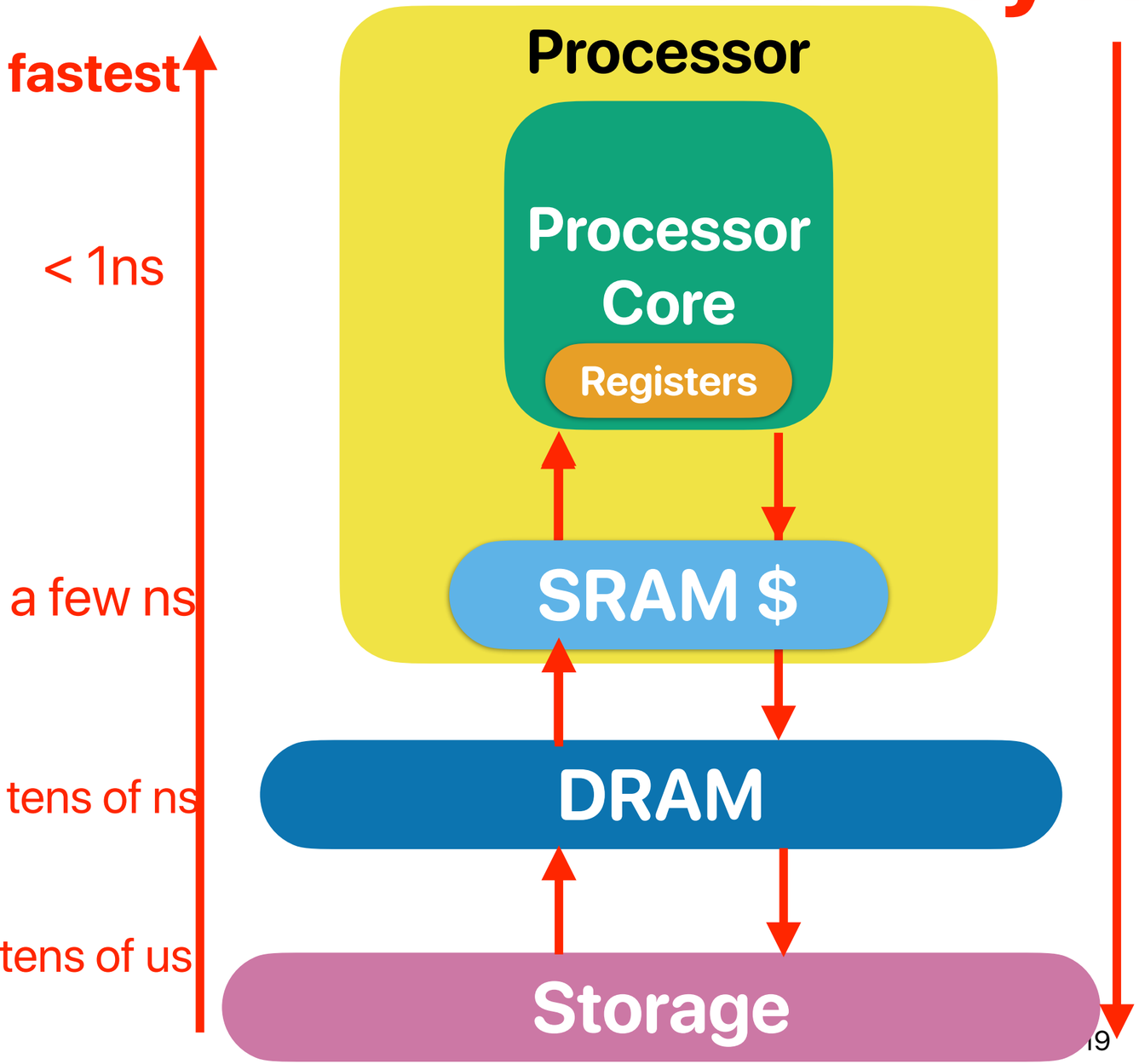
- Memory access time is the most critical performance problem
  - One memory operation is as expensive as 50 arithmetic operations
  - Processor has to fetch instructions from memory
  - We have an average of 33% of data memory access instructions!

Alternatives?

Memory technology	Typical access time	$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$10–$20
Flash semiconductor memory	5,000–50,000 ns	$0.75–$1.00
Magnetic disk	5,000,000–20,000,000 ns	$0.05–$0.10

Fast, but expensive $$$

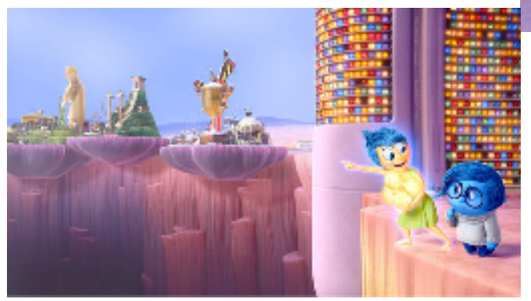
Memory Hierarchy



32 or 64 words

KBs ~ MBs

GBs



TBs

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

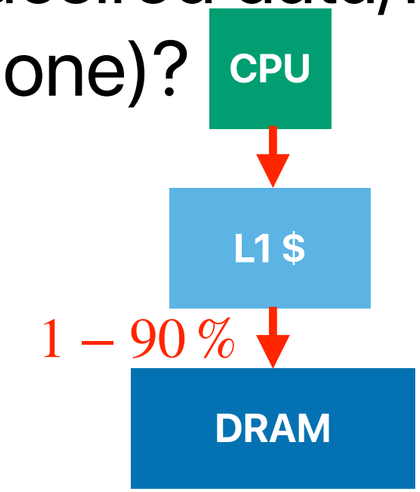
How can “memory hierarchy” help in performance?

- Assume that we have a processor running @ 4 GHz and a program with 20% of load/store instructions. If the instruction has no memory access, the CPI is just 1. Now, in addition to we DDR5, whose latency 13.75 ns, we also got an SRAM cache with latency of just at 0.5 ns and can capture **90%** of the desired data/instructions. what’s the average CPI (pick the closest one)?
  A. 6
  B. 8
  C. 10
  D. 12
  E. 67

How can "memory hierarchy" help in performance?

- Assume that we have a processor running @ 4 GHz and a program with 20% of load/store instructions. If the instruction has no memory access, the CPI is just 1. Now, in addition to we DDR5, whose latency 13.75 ns, we also got an SRAM cache with latency of just at 0.5 ns and can capture **90%** of the desired data/instructions. what's the average CPI (pick the closest one)?

  A. 6
  B. 8
  C. 10
  D. 12
  E. 67



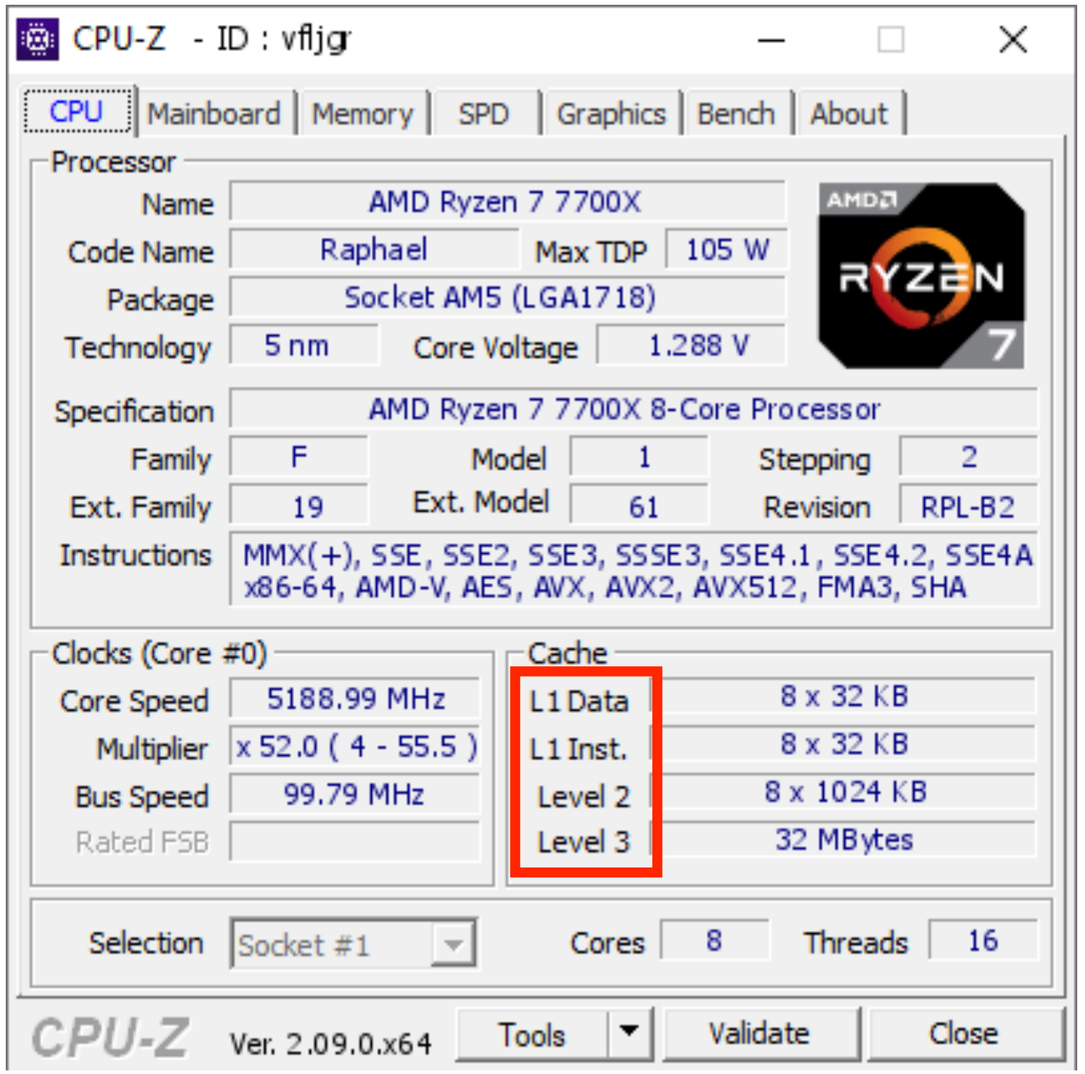
$$CPU\ cycle\ time = \frac{1}{4 \times 10^9} = 0.25ns$$

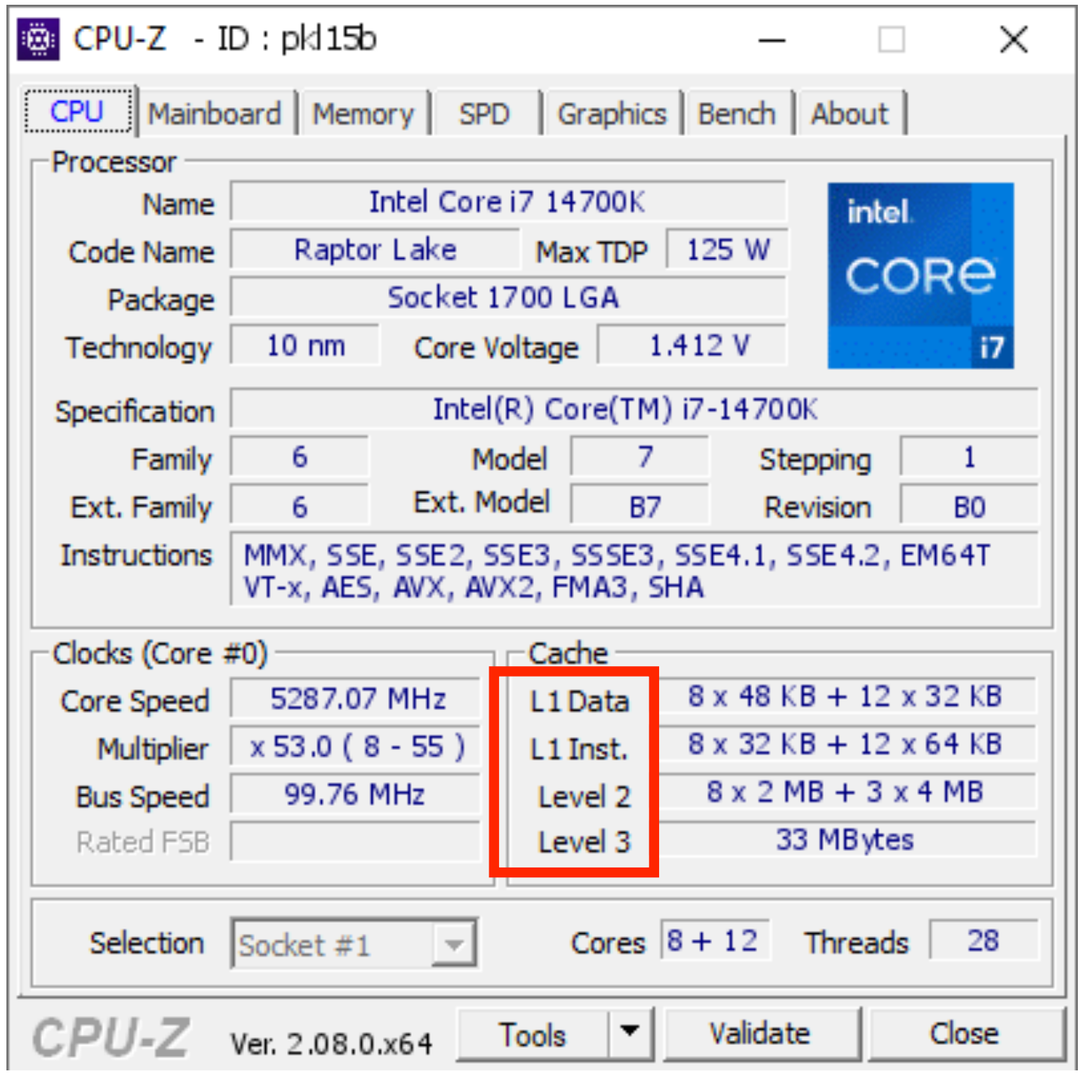
$$Each\ \$\ access = \frac{0.5}{0.25} = 2\ cycles$$

$$Each\ DRAM\ access = \frac{13.75}{0.25} = 55\ cycles$$

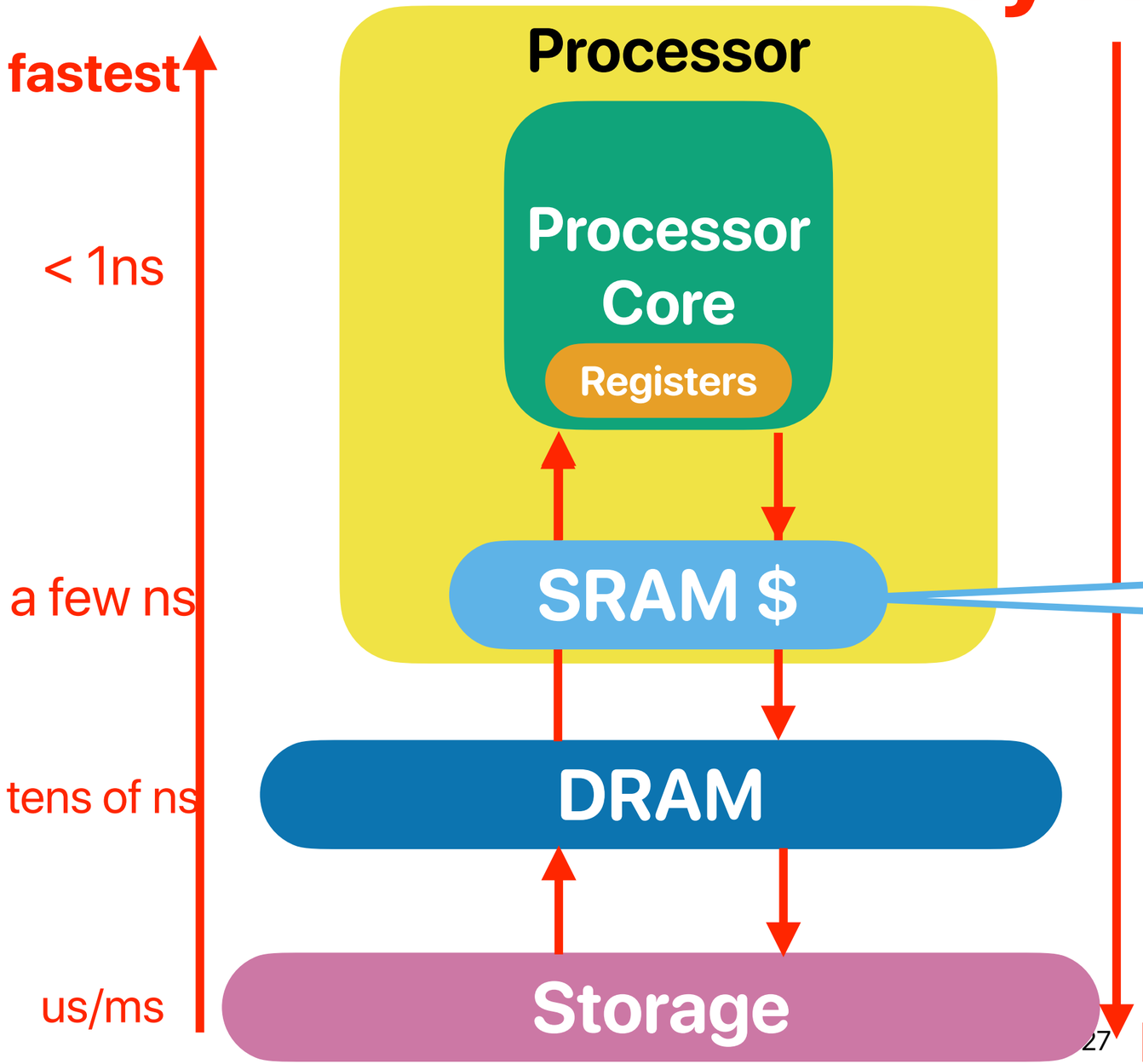
$$CPI_{average} = 1 + 100\% \times [2 + (1 - 90\%) \times 55] + 20\% \times [2 + (1 - 90\%) \times 55] = 10\ cycles$$

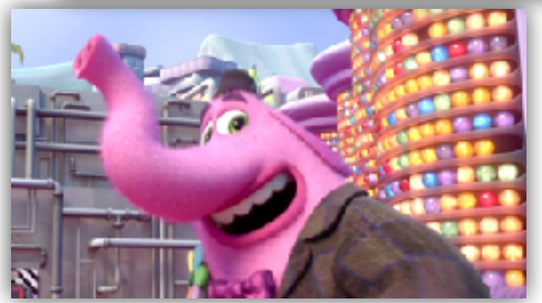
L1? L2? L3?


CPU-Z - ID : vfljg
CPU
Mainboard
Memory
SPD
Graphics
Bench
About
Processor
Name
AMD Ryzen 7 7700X
Code Name
Raphael
Max TDP
105 W
Package
Socket AM5 (LGA1718)
Technology
5 nm
Core Voltage
1.288 V
Specification
AMD Ryzen 7 7700X 8-Core Processor
Family
F
Model
1
Stepping
2
Ext. Family
19
Ext. Model
61
Revision
RPL-B2
Instructions
MMX(+), SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, SSE4A
x86-64, AMD-V, AES, AVX, AVX2, AVX512, FMA3, SHA
Clocks (Core #0)
Core Speed
5188.99 MHz
Multiplier
x 52.0 (4 - 55.5)
Bus Speed
99.79 MHz
Rated FSB
Cache
L1 Data
8 x 32 KB
L1 Inst.
8 x 32 KB
Level 2
8 x 1024 KB
Level 3
32 MBytes
Selection
Socket #1
Cores
8
Threads
16
CPU-Z
Ver. 2.09.0.x64
Tools
Validate
Close


CPU-Z - ID : pkl15b
CPU
Mainboard
Memory
SPD
Graphics
Bench
About
Processor
Name
Intel Core i7 14700K
Code Name
Raptor Lake
Max TDP
125 W
Package
Socket 1700 LGA
Technology
10 nm
Core Voltage
1.412 V
Specification
Intel(R) Core(TM) i7-14700K
Family
6
Model
7
Stepping
1
Ext. Family
6
Ext. Model
B7
Revision
B0
Instructions
MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T
VT-x, AES, AVX, AVX2, FMA3, SHA
Clocks (Core #0)
Core Speed
5287.07 MHz
Multiplier
x 53.0 (8 - 55)
Bus Speed
99.76 MHz
Rated FSB
Cache
L1 Data
8 x 48 KB + 12 x 32 KB
L1 Inst.
8 x 32 KB + 12 x 64 KB
Level 2
8 x 2 MB + 3 x 4 MB
Level 3
33 MBytes
Selection
Socket #1
Cores
8 + 12
Threads
28
CPU-Z
Ver. 2.08.0.x64
Tools
Validate
Close

Memory Hierarchy





How can a deeper memory hierarchy help in performance?

- Assume that we have a processor running @ 4 GHz and a program with 20% of load/store instructions. If the instruction has no memory access, the CPI is just 1. Now, in addition to we DDR5, whose latency 13.75 ns, we also got a 2-level SRAM caches with
  - it's 1st-level one at latency of 0.5ns and can capture 90% of the desired data/ instructions.
  - the 2nd-level at latency of 5 ns and can capture 60% of the desired data/instructions

  What's the average CPI (pick the closest one)?

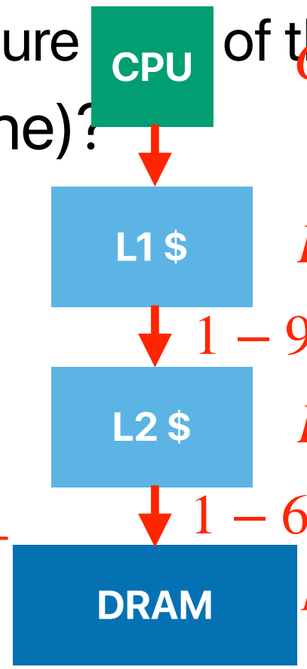
  A. 6
  B. 8
  C. 10
  D. 12
  E. 67

How can a deeper memory hierarchy help in performance?

- Assume that we have a processor running @ 4 GHz and a program with 20% of load/store instructions. If the instruction has no memory access, the CPI is just 1. Now, in addition to we DDR5, whose latency 13.75 ns, we also got a 2-level SRAM caches with
  - it's 1st-level one at latency of 0.5ns and can capture 90% of the desired data/ instructions.
  - the 2nd-level at latency of 5 ns and can capture 60% of the desired data/instructions

What's the average CPI (pick the closest one)?

A. 6
B. 8
C. 10
D. 12
E. 67



$$CPU\ cycle\ time = \frac{1}{4 \times 10^9} = 0.25ns$$

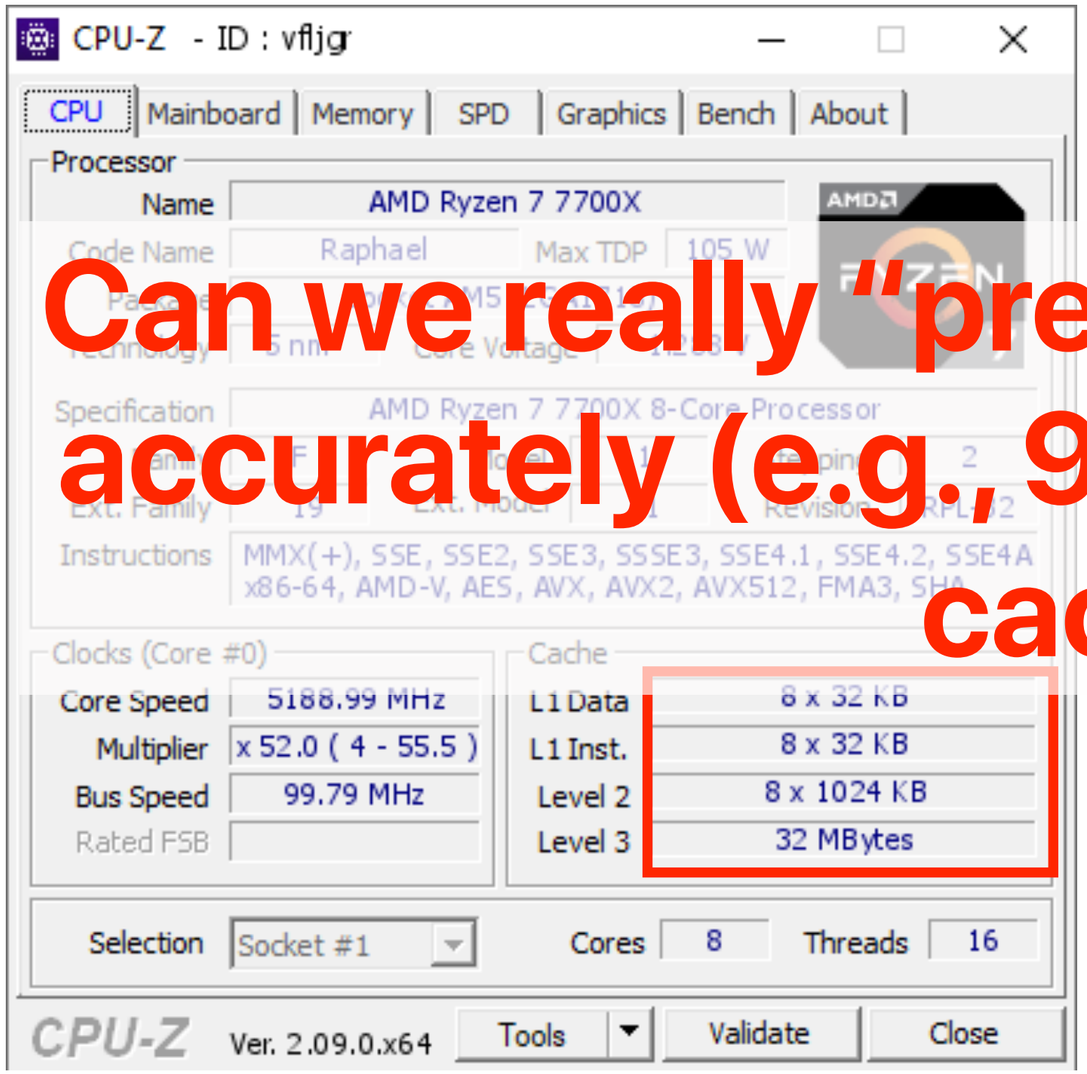
$$Each\ L1_{\$}\ access = \frac{0.5}{0.25} = 2\ cycles$$

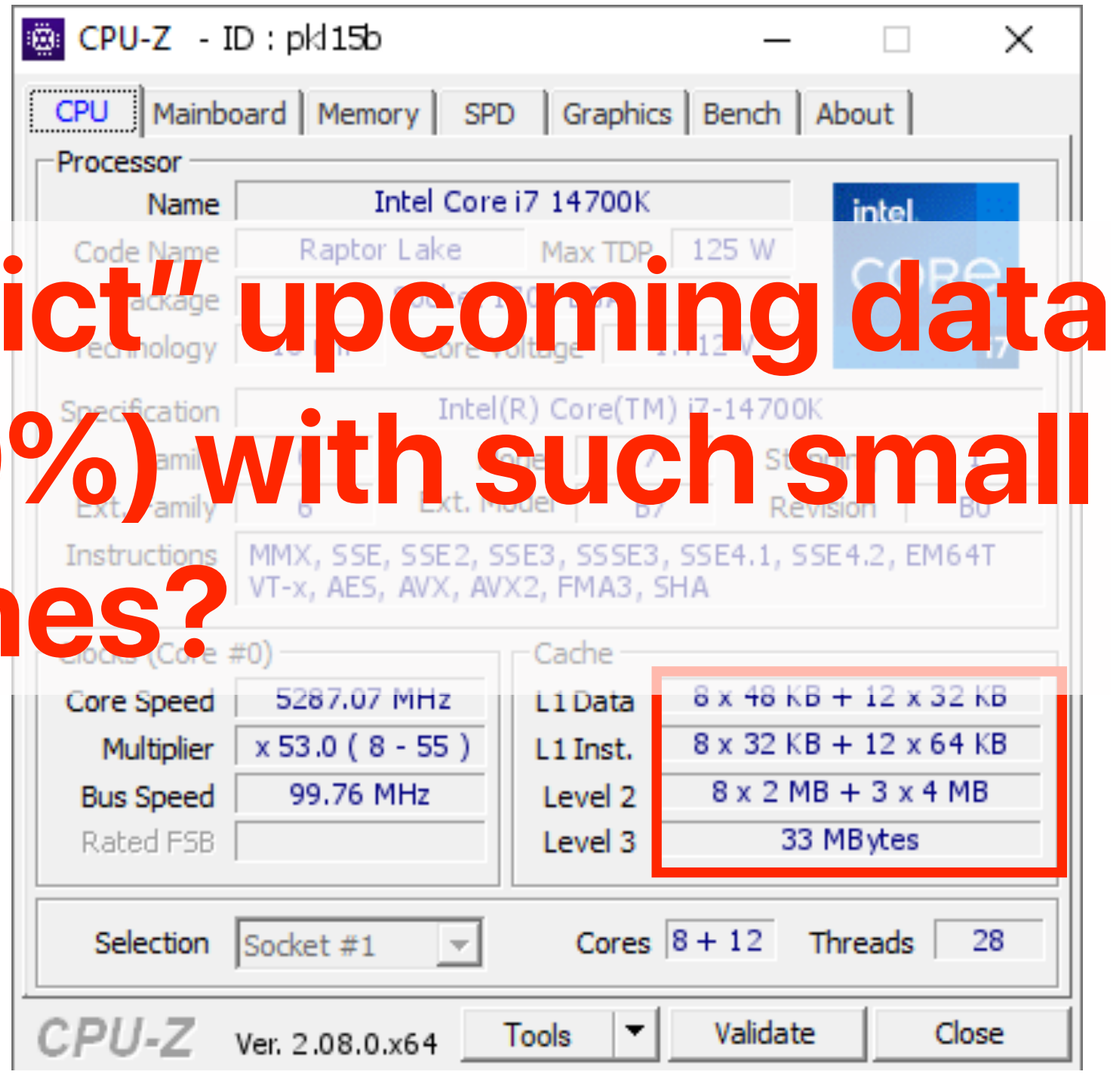
$$Each\ L2_{\$}\ access = \frac{5}{0.25} = 20\ cycles$$

$$Each\ DRAM\ access = \frac{13.75}{0.25} = 55\ cycles$$

$$CPI_{average} = 100\% \times [2 + (1 - 90\%) \times (20 + (1 - 60\%) \times 55] + 20\% \times [2 + (1 - 90\%) \times (20 + (1 - 60\%) \times 55)] = 8.44\ cycles$$

L1? L2? L3?





Can we really "predict" upcoming data accurately (e.g., 90%) with such small caches?

Take-aways: inside out our memory hierarchy

- Memory access time is the most critical performance problem
  - One memory operation is as expensive as 50 arithmetic operations
  - Processor has to fetch instructions from memory
  - We have an average of 33% of data memory access instructions!
- Hierarchical caching with small amount of SRAMs will work if we can efficiently capture data and instructions

The predictability of your code

The Machine Learning Inference Pipeline

Feature Extraction

Classification

Convolution + ReLU

Pooling

Convolution + ReLU

Pooling

Fully Connected

Output Prediction

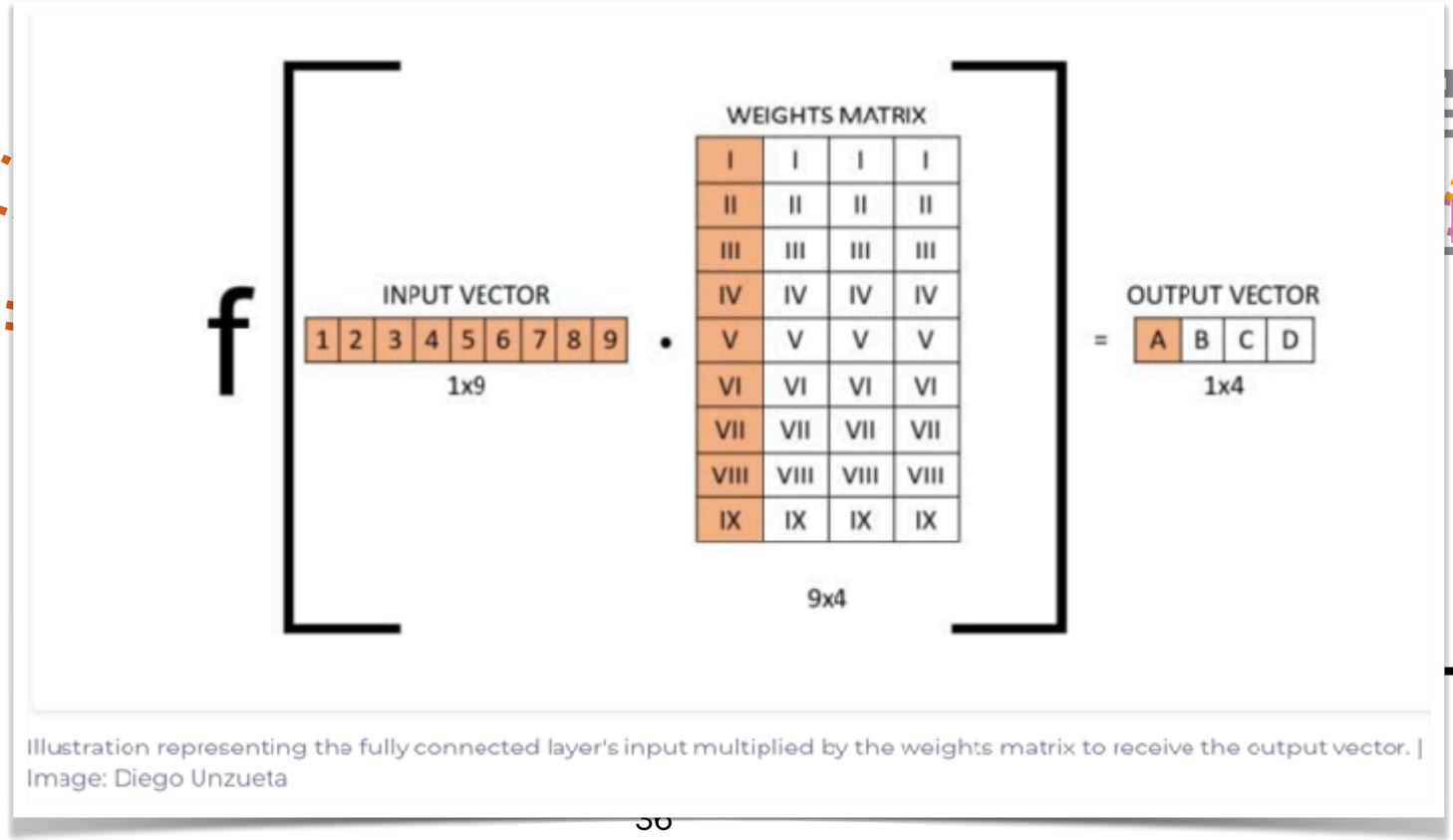
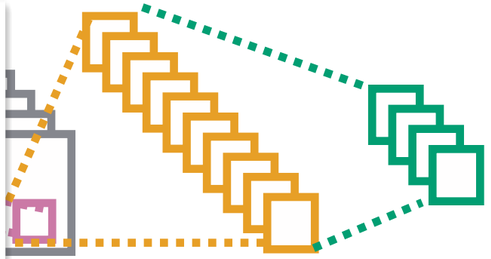


Illustration representing the fully connected layer's input multiplied by the weights matrix to receive the output vector. | Image: Diego Unzueta



Triton (1)

Cat (0)

Human (0)

Tiger (0)

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

Locality of data

- Which description about locality of arrays `matrix` and `vector` in the following code is the **most accurate**?

```
for(uint64_t i = 0; i < m; i++) {
     result = 0;
    for(uint64_t j = 0; j < n; j++) {
          result += matrix[i][j]*vector[j];
    }
    output[i] = result;
}
```

A. Access of `matrix` has temporal locality, `vector` has spatial locality
B. Both `matrix` and `vector` have temporal locality, and `vector` also has spatial locality
C. Access of `matrix` has spatial locality, `vector` has temporal locality
D. Both `matrix` and `vector` have spatial locality and temporal locality
E. Both `matrix` and `vector` have spatial locality, and `vector` also has temporal locality

Data locality

- Which description about locality of arrays `matrix` and `vector` in the following code is the **most accurate**?

```
for(uint64_t i = 0; i < m; i++) {
     result = 0;
    for(uint64_t j = 0; j < n; j++) {
          result += matrix[i][j]*vector[j];
    }
    output[i] = result;
}
```

spatial locality:
matrix[0][0], matrix[0][1], matrix[0][2], ...
vector[0], vector[1], ... , vector[n]
temporal locality:
reuse of vector[0], vector[1], ... ,

A. Access of `matrix` has temporal locality, `vector` has spatial locality
B. Both `matrix` and `vector` have temporal locality, and `vector` also has spatial locality
C. Access of `matrix` has spatial locality, `vector` has temporal locality
D. Both `matrix` and `vector` have spatial locality and temporal locality
E. Both `matrix` and `vector` have spatial locality, and `vector` also has temporal locality

Code also has locality

```
for(uint64_t i = 0; i < m; i++) {
    result = 0;
    for(uint64_t j = 0; j < n; j++) {
        result += matrix[i][j]*vector[j];
    }
    output[i] = result;
}
```

```
i = 0;
while(i < m) {
    result = 0;
    j = 0;
    while(j < n) {
        a = matrix[i][j];
        b = vector[j];
        temp = a*b;
        result = result + temp;
    }
    output[i] = result;
    i++;
}
```

keep going to the next instruction — spatial locality

repeat many times — temporal locality!

Locality

- Spatial locality — application tends to visit nearby stuffs in the memory
  - Code — the current instruction, and then the next instruction
  - Data — the current data, and then the next data
- Temporal locality — application revisit the same thing again and again
  - Code — loops, frequently invoked functions
    - Typically tens of static instructions — at most several KBs
  - Data — program can read/write the same data many times (e.g., vectors in matrix-vector product)

Most of time, your program is just visiting a very small amount of data/instructions within a given window

Take-aways: inside out our memory hierarchy

- Memory access time is the most critical performance problem
  - One memory operation is as expensive as 50 arithmetic operations
  - Processor has to fetch instructions from memory
  - We have an average of 33% of data memory access instructions!
- Hierarchical caching with small amount of SRAMs will work if we can efficiently capture data and instructions
- Caching is possible! Most of time, we only work on a small amount of data!
  - Spatial locality
  - Temporal locality

Designing a hardware to exploit locality

- Spatial locality — application tends to visit nearby stuffs in the memory

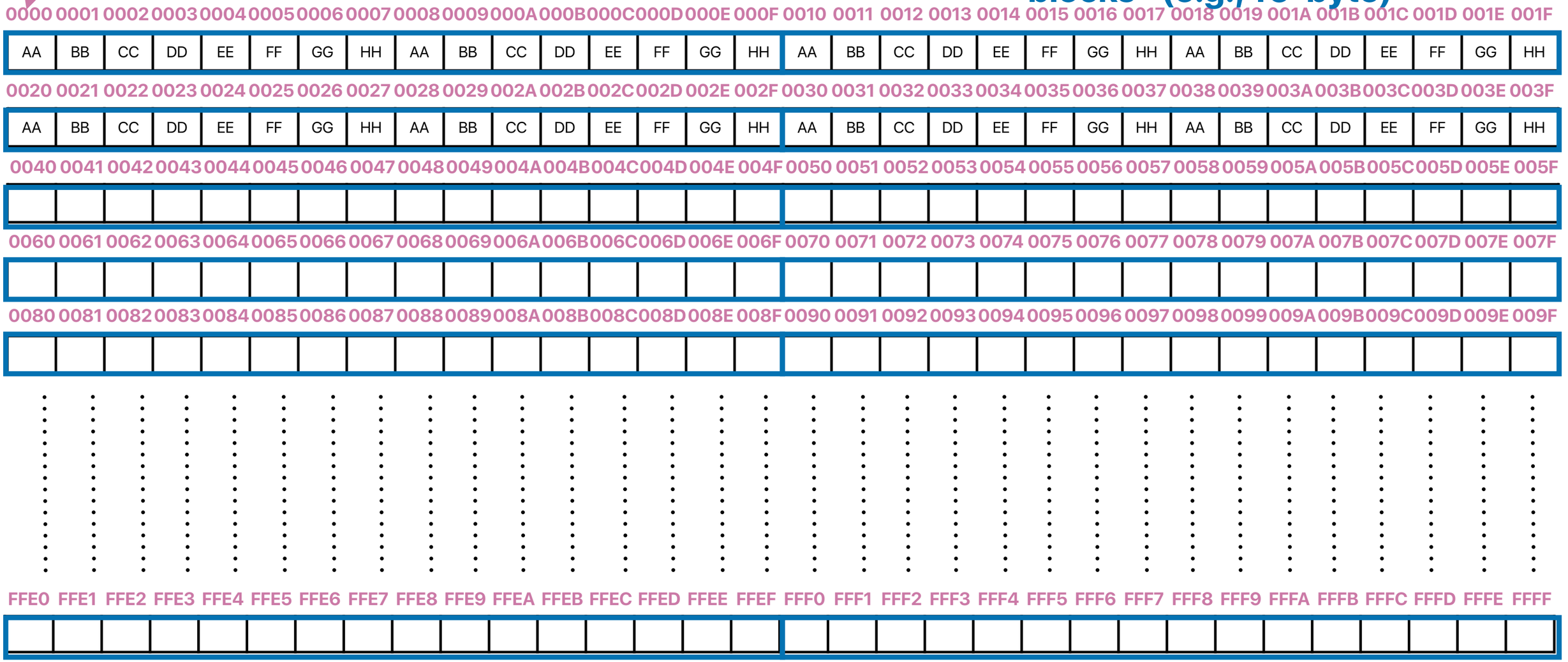
  We need to "cache consecutive memory locations" every time — the cache should store a "block" of code/data

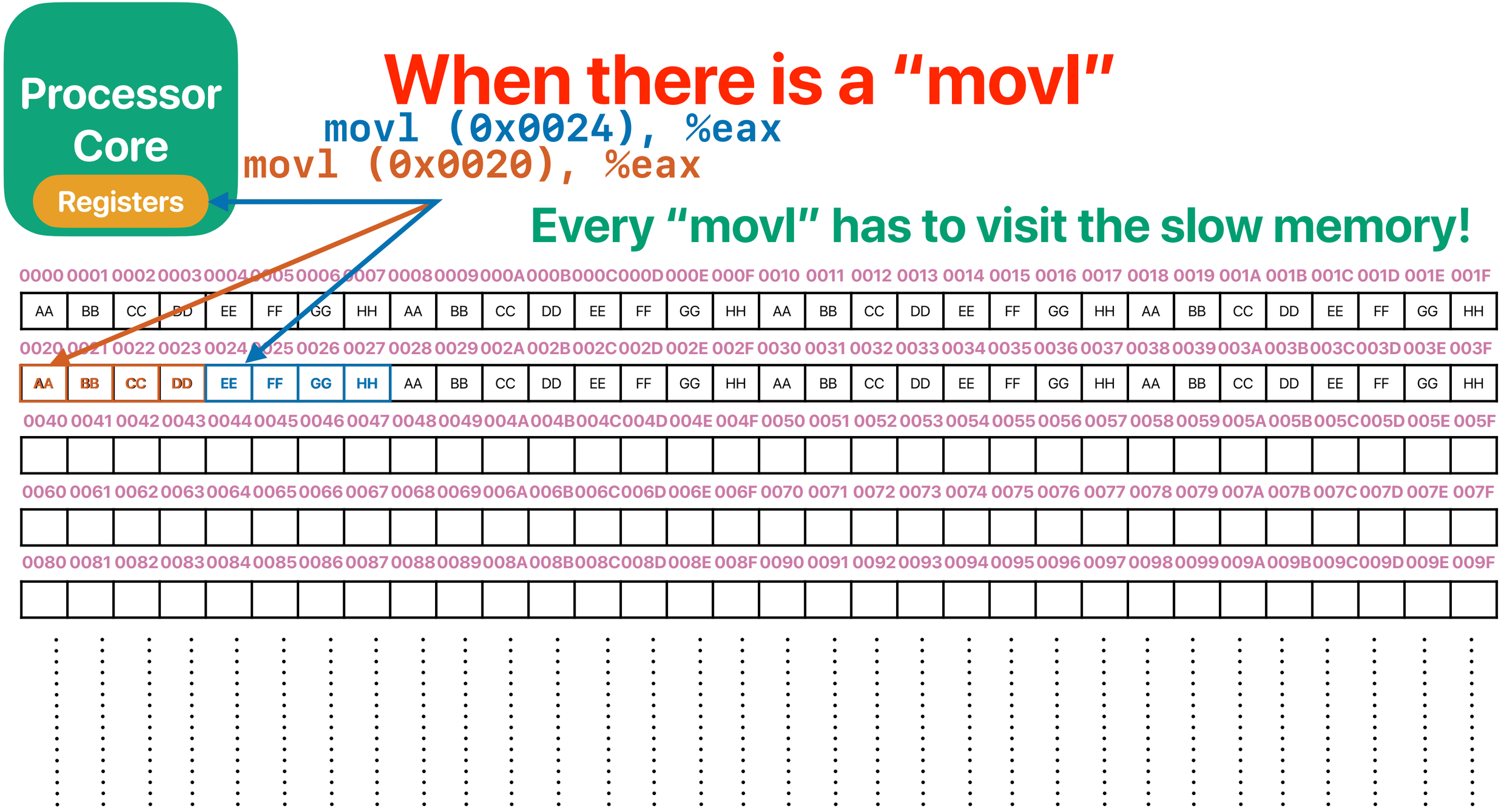
- Temporal locality — application revisit the same thing again and again
  - Code — loops, frequently invoked functions
    - Typically tens of static instructions — at most several KBs
  - Data — program can read/write the same data many times (e.g., vectors in matrix-vector product)

Block and the memory space

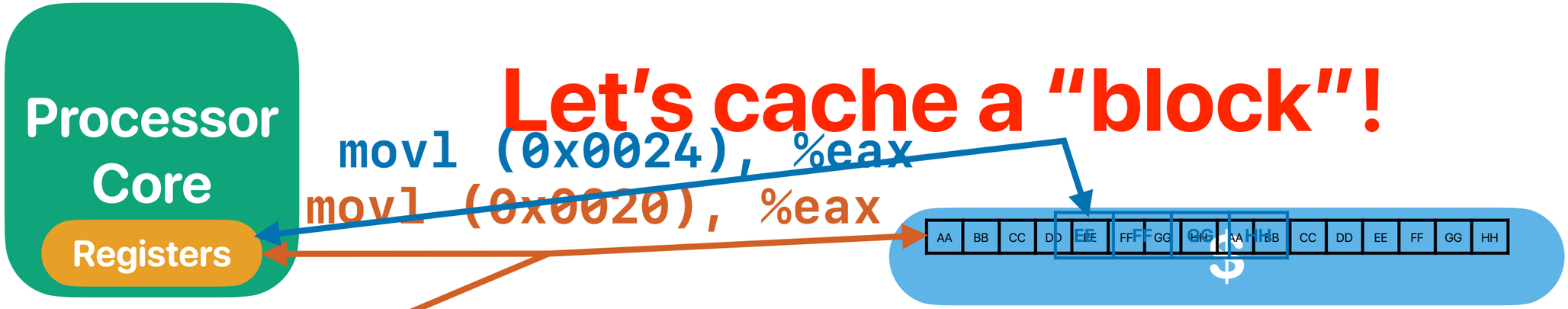
Each byte of memory location has an "address"

Partition the space with fixed-size "blocks" (e.g., 16-byte)




Processor Core
Registers
When there is a “movl”
movl (0x0024), %eax
movl (0x0020), %eax
Every “movl” has to visit the slow memory!
0000 0001 0002 0003 0004 0005 0006 0007 0008 0009 000A 000B 000C 000D 000E 000F 0010 0011 0012 0013 0014 0015 0016 0017 0018 0019 001A 001B 001C 001D 001E 001F
AA BB CC DD EE FF GG HH AA BB CC DD EE FF GG HH AA BB CC DD EE FF GG HH AA BB CC DD EE FF GG HH
0020 0021 0022 0023 0024 0025 0026 0027 0028 0029 002A 002B 002C 002D 002E 002F 0030 0031 0032 0033 0034 0035 0036 0037 0038 0039 003A 003B 003C 003D 003E 003F
AA BB CC DD EE FF GG HH AA BB CC DD EE FF GG HH AA BB CC DD EE FF GG HH AA BB CC DD EE FF GG HH
0040 0041 0042 0043 0044 0045 0046 0047 0048 0049 004A 004B 004C 004D 004E 004F 0050 0051 0052 0053 0054 0055 0056 0057 0058 0059 005A 005B 005C 005D 005E 005F
0060 0061 0062 0063 0064 0065 0066 0067 0068 0069 006A 006B 006C 006D 006E 006F 0070 0071 0072 0073 0074 0075 0076 0077 0078 0079 007A 007B 007C 007D 007E 007F
0080 0081 0082 0083 0084 0085 0086 0087 0088 0089 008A 008B 008C 008D 008E 008F 0090 0091 0092 0093 0094 0095 0096 0097 0098 0099 009A 009B 009C 009D 009E 009F

Let's cache a "block"!



Caching a block helps exploit "spatial locality"!

0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	000A	000B	000C	000D	000E	000F	0010	0011	0012	0013	0014	0015	0016	0017	0018	0019	001A	001B	001C	001D	001E	001F
AA	BB	CC	DD	EE	FF	GG	HH	AA	BB	CC	DD	EE	FF	GG	HH	AA	BB	CC	DD	EE	FF	GG	HH	AA	BB	CC	DD	EE	FF	GG	HH
0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	002A	002B	002C	002D	002E	002F	0030	0031	0032	0033	0034	0035	0036	0037	0038	0039	003A	003B	003C	003D	003E	003F
AA	BB	CC	DD	EE	FF	GG	HH	AA	BB	CC	DD	EE	FF	GG	HH	AA	BB	CC	DD	EE	FF	GG	HH	AA	BB	CC	DD	EE	FF	GG	HH
0040	0041	0042	0043	0044	0045	0046	0047	0048	0049	004A	004B	004C	004D	004E	004F	0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	005A	005B	005C	005D	005E	005F
0060	0061	0062	0063	0064	0065	0066	0067	0068	0069	006A	006B	006C	006D	006E	006F	0070	0071	0072	0073	0074	0075	0076	0077	0078	0079	007A	007B	007C	007D	007E	007F
0080	0081	0082	0083	0084	0085	0086	0087	0088	0089	008A	008B	008C	008D	008E	008F	0090	0091	0092	0093	0094	0095	0096	0097	0098	0099	009A	009B	009C	009D	009E	009F

Recap: Locality

- Which description about locality of arrays `matrix` and `vector` in the following code is the **most accurate**?

```
for(uint64_t i = 0; i < m; i++) {
     result = 0;
    for(uint64_t j = 0; j < n; j++) {
          result += matrix[i][j]*vector[j];
    }
     output[i] = result;
 }
```

Simply caching one block isn't enough

Designing a hardware to exploit locality

- Spatial locality — application tends to visit nearby stuffs in the memory

  We need to "cache consecutive memory locations" every time

  — the cache should store a "block" of code/data

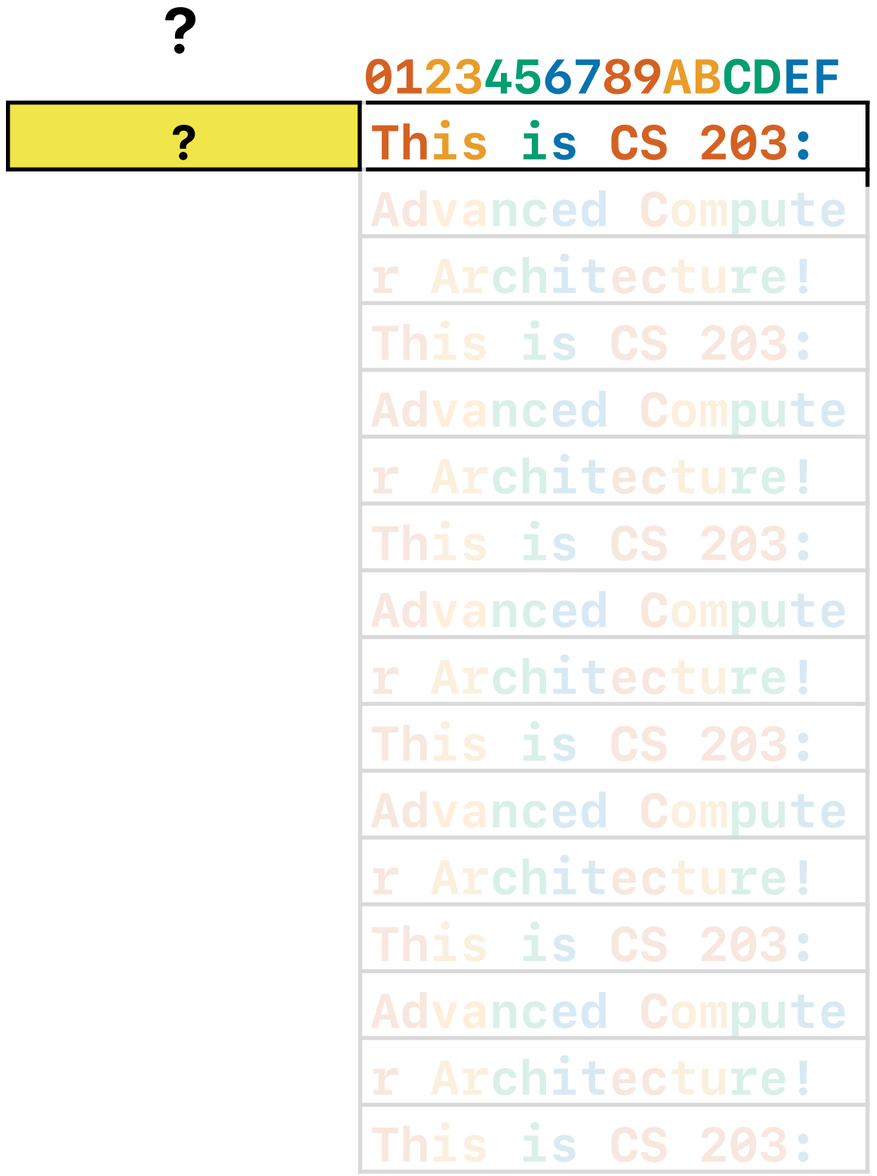
- Temporal locality — application revisit the same thing again and again

  We need to "cache frequently used memory blocks"

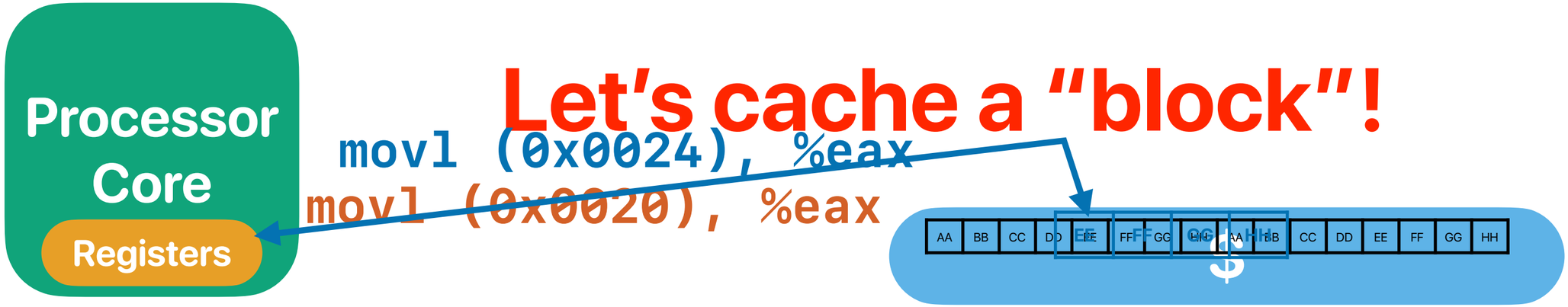
  — the cache should store a few blocks

  — the cache must be able to distinguish blocks

How to tell who is there?



Let's cache a "block"!



0000 0001 0002 0003 0004 0005 0006 0007 0008 0009 000A 000B 000C 000D 000E 000F 0010 0011 0012 0013 0014 0015 0016 0017 0018 0019 001A 001B 001C 001D 001E 001F

AA BB CC DD EE FF GG HH AA BB CC DD EE FF GG HH AA BB CC DD EE FF GG HH AA BB CC DD EE FF GG HH

0020 0021 0022 0023 0024 0025 0026 0027 0028 0029 002A 002B 002C 002D 002E 002F 0030 0031 0032 0033 0034 0035 0036 0037 0038 0039 003A 003B 003C 003D 003E 003F

AA BB CC DD EE FF GG HH AA BB CC DD EE FF GG HH AA BB CC DD EE FF GG HH AA BB CC DD EE FF GG HH

0040 0041 0042 0043 0044 0045 0046 0047 0048 0049 004A 004B 004C 004D 004E 004F 0050 0051 0052 0053 0054 0055 0056 0057 0058 0059 005A 005B 005C 005D 005E 005F

0060 0061 0062 0063 0064 0065 0066 0067 0068 0069 006A 006B 006C 006D 006E 006F 0070 0071 0072 0073 0074 0075 0076 0077 0078 0079 007A 007B 007C 007D 007E 007F

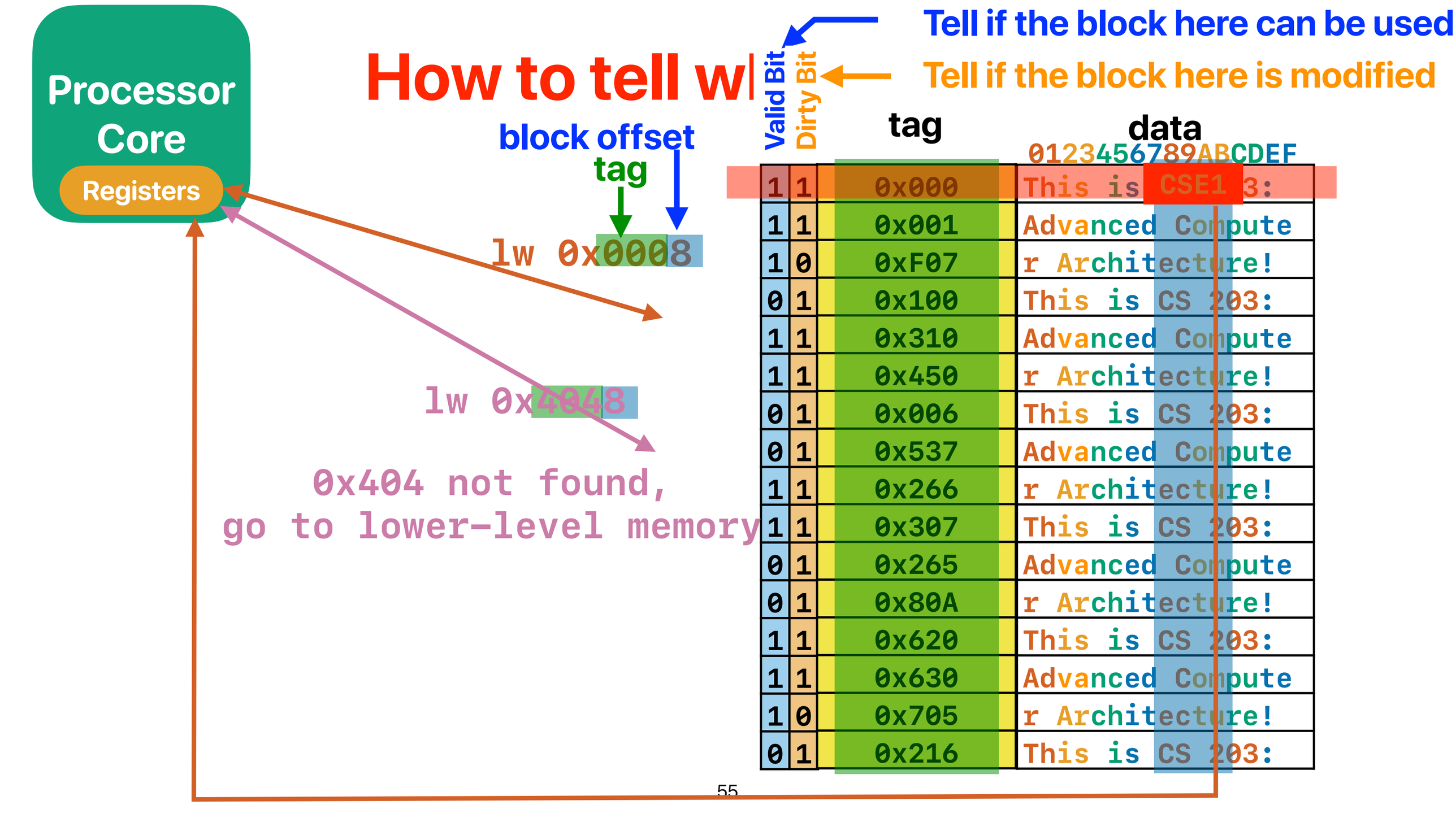
0080 0081 0082 0083 0084 0085 0086 0087 0088 0089 008A 008B 008C 008D 008E 008F 0090 0091 0092 0093 0094 0095 0096 0097 0098 0099 009A 009B 009C 009D 009E 009F

the address in each block starts with the same "prefix"

How to tell who is there?

the common address prefix in each block

tag array	0x0000 0x0001 0x0002 0x0003 0x0004 0x0005 0x0006 0x0007 0x0008 0x0009 0x000A 0x000B 0x000C 0x000D 0x000E 0x000F 0123456789ABCDEF
0x000	This is CS 203:
0x001	Advanced Compute
0xF07	r Architecture!
0x100	This is CS 203:
0x310	Advanced Compute
0x450	r Architecture!
0x006	This is CS 203:
0x537	Advanced Compute
0x266	r Architecture!
0x307	This is CS 203:
0x265	Advanced Compute
0x80A	r Architecture!
0x620	This is CS 203:
0x630	Advanced Compute
0x705	r Architecture!
0x216	This is CS 203:


Processor Core
Registers
How to tell w
Valid Bit
Dirty Bit
Tell if the block here can be used
Tell if the block here is modified
block offset
tag
lw 0x0008
lw 0x4048
0x404 not found,
go to lower-level memory
tag
data
0123456789ABCDEF
1 1 0x000 This is CSE1 3:
1 1 0x001 Advanced Compute
1 0 0xF07 r Architecture!
0 1 0x100 This is CS 203:
1 1 0x310 Advanced Compute
1 1 0x450 r Architecture!
0 1 0x006 This is CS 203:
0 1 0x537 Advanced Compute
1 1 0x266 r Architecture!
1 1 0x307 This is CS 203:
0 1 0x265 Advanced Compute
0 1 0x80A r Architecture!
1 1 0x620 This is CS 203:
1 1 0x630 Advanced Compute
1 0 0x705 r Architecture!
0 1 0x216 This is CS 203:

Take-aways: inside out our memory hierarchy

- Memory access time is the most critical performance problem
  - One memory operation is as expensive as 50 arithmetic operations
  - Processor has to fetch instructions from memory
  - We have an average of 33% of data memory access instructions!
- Hierarchical caching with small amount of SRAMs will work if we can efficiently capture data and instructions
- Caching is possible! Most of time, we only work on a small amount of data!
  - Spatial locality
  - Temporal locality
- Basic cache structures —
  - Caching in granularity of a block to capture spatial locality
  - Caching multiple blocks to keep frequently used data — temporal locality
  - Tags to distinguish cached blocks

Announcement

- Reading quiz #4 due **Wednesday** before the lecture
- Assignment #2 due **this Saturday**
  - You should run the performance measurement yourself and calculate results based on that — everyone should have a different answer
  - All questions this time require **correct** estimations in cache performance to help you better prepare the examines

Computer
Science &
Engineering

142

つづく