

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

Hung-Wei Tseng



# Disney·PIXAR INSIDE OUT

GET DISNEY+

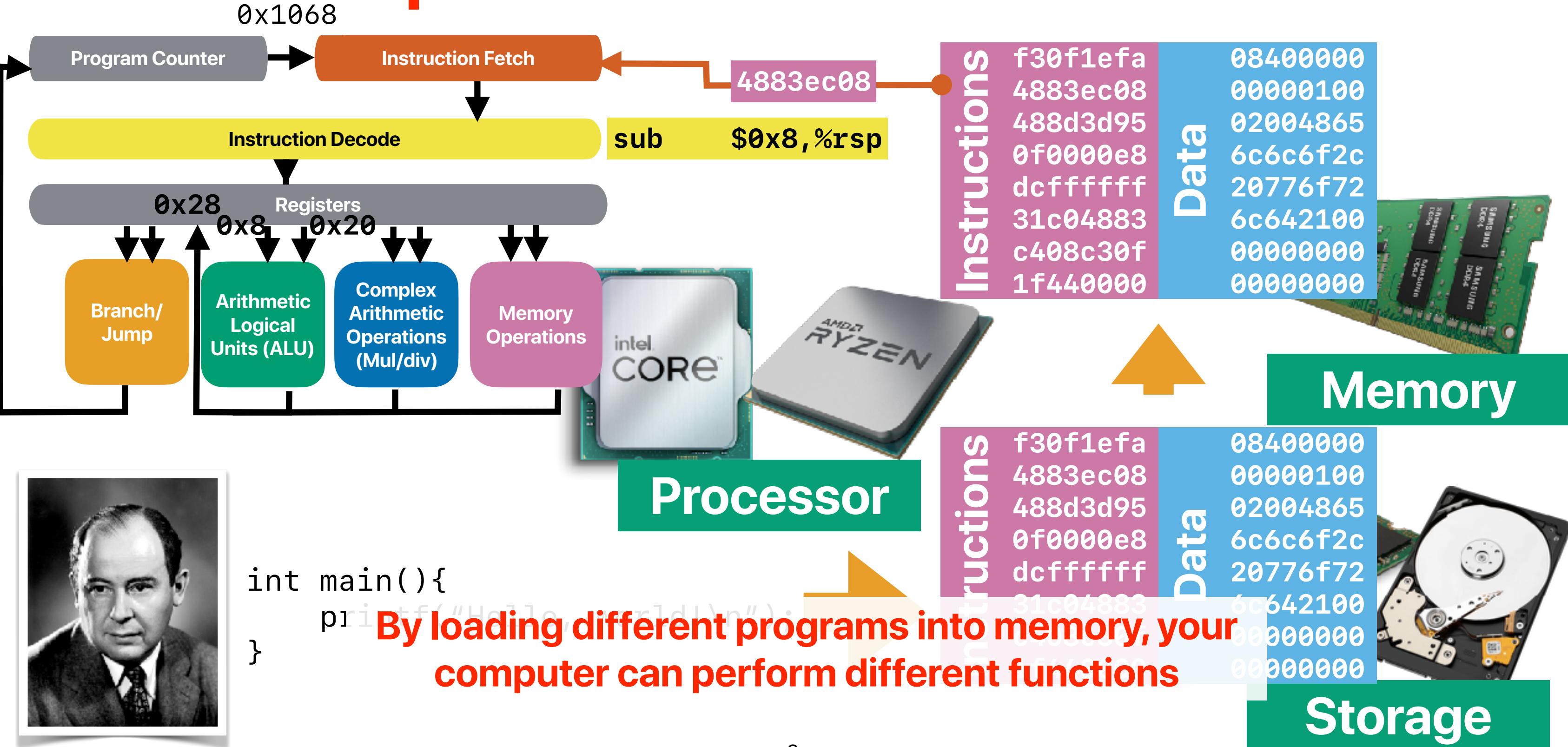
► TRAILER

**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 Neumann architecture



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

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

- Corollary 3 — Optimization has a moving target  $Speedup_{max}(f_2, \infty) = \frac{1}{(1-f_2)}$

- 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

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

**What percentage of “English words” from the dictionary do you think someone has to know for their everyday life?**

[federalreserve.gov/releases/z1/datavizdfa/di...](https://federalreserve.gov/releases/z1/datavizdfa/di...)

[crn.ccm/news/security/18821726/microsofts-ceo-80-20-rule-applies-to-bugs-not-just-features](https://crn.ccm/news/security/18821726/microsofts-ceo-80-20-rule-applies-to-bugs-not-just-features)

Board of Governors of the Federal Reserve System

Sections

### Wealth by wealth percentile group

Trillions of Dollars

The chart displays the total wealth in Trillions of Dollars over time, broken down into five percentile groups. The legend indicates:

- Top 0.1%: \$20.66 T
- 99-99.9%: \$25.52 T
- 90-99%: \$55.44 T
- 50-90%: \$46.28 T
- Bottom 50%: \$3.78 T

Total: \$151.68 T

Percentile Group	Value (Trillions of Dollars)
Top 0.1%	\$20.66 T
99-99.9%	\$25.52 T
90-99%	\$55.44 T
50-90%	\$46.28 T
Bottom 50%	\$3.78 T
Total	\$151.68 T

2010 2012 2014 2016 2018 2020 2022 2024

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

**Microsoft's CEO: 80-20 Rule Applies To Bugs, Not Just Features**

BY PAULA ROONEY ▶ OCTOBER 03, 2002, 03:56 PM EDT

### Microsoft

One common adage in the IT industry is that 80 percent of all end users generally use only 20 percent of a software application's features.

In recent months, Microsoft has learned that 80 percent of the errors and crashes in Windows and Office are caused by 20 percent of the entire pool of bugs detected, and that more than 50 percent of the headaches derive from a mere 1 percent of all flawed code.

In an e-mail update sent out broadly to enterprise customers on Oct. 2, Microsoft CEO Steve Ballmer highlighted critical progress being made on the company's Trustworthy Computing initiative, an effort to expand by the end of next January to improve its reputation in reliability and security arenas. For one thing, there will be faster bug-fixing as a result of an error-reporting facility embedded in Office and Windows. And that error-reporting tool will be part of the forthcoming Windows Server 2003.

**80% of users use only 20% of features**

[https://en.wikipedia.org/wiki/Pareto\\_principle](https://en.wikipedia.org/wiki/Pareto_principle)

6

 AI Overview[Learn more](#) 

Native English speakers use around 1,200–3,000 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)?

How Many Words Do Native Speakers Use In Daily Life? Native speakers of...

 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

<http://www.ef.edu/english-vocabulary/1000-3000-w...>

3000 most common words in English | EF English Charts

 AI Overview

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

**Oxford English Dictionary**

The second edition of this 20-volume dictionary includes 171,476 words in current use, 47,156 obsolete words, and around 9,500 derivative words. It also includes 169,000 combinations and derivatives, and 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**

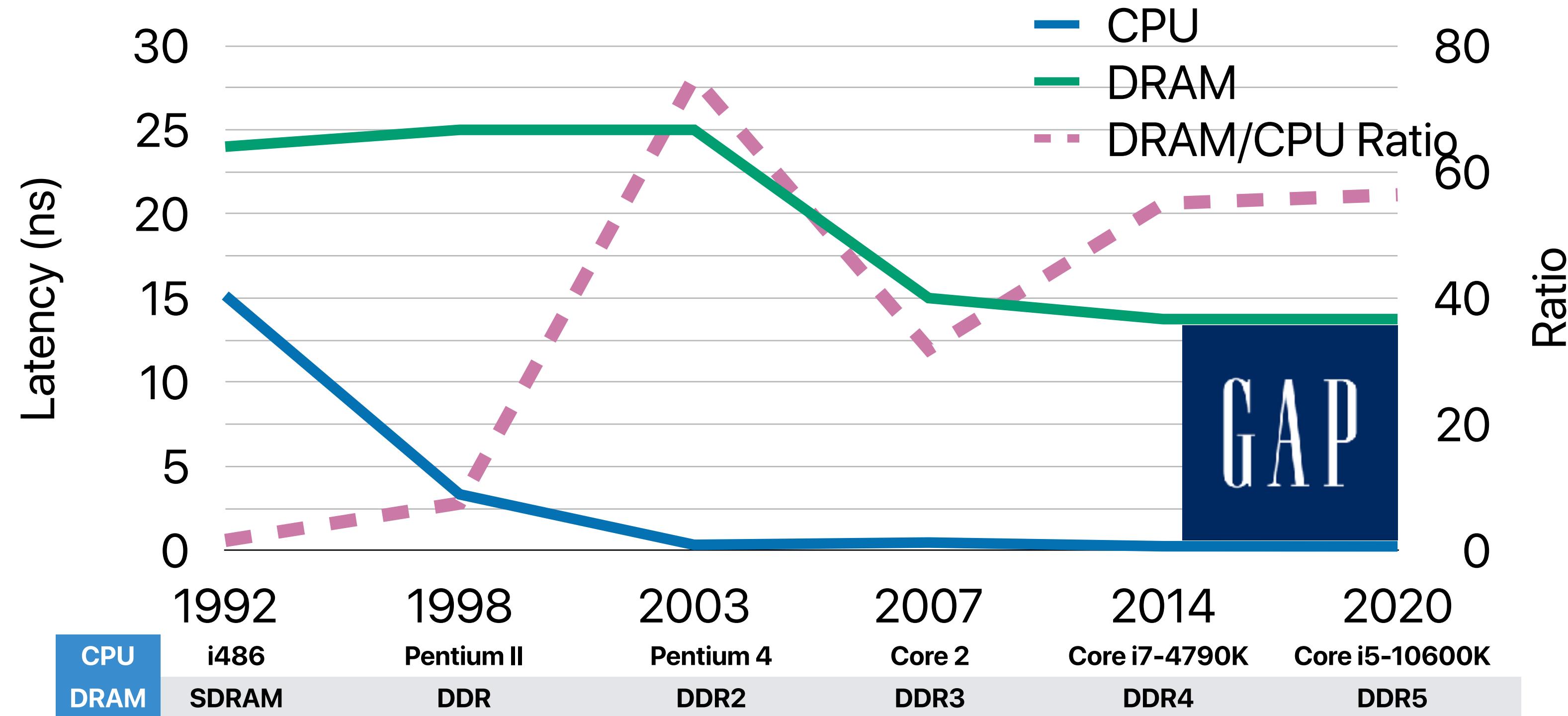
This dictionary, along with its 1993 Addenda Section, includes around 470,000 entries. 

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

# Outline

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

# The “latency” gap between CPU and DRAM





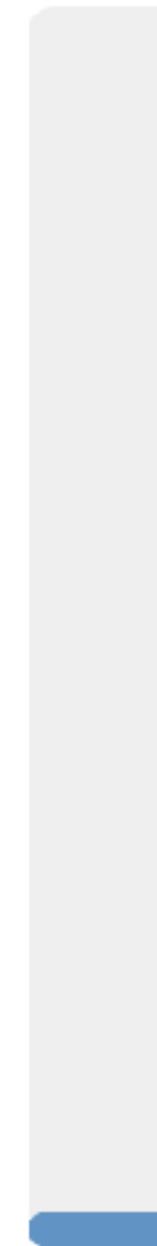
# The impact of “slow” memory

- 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)?

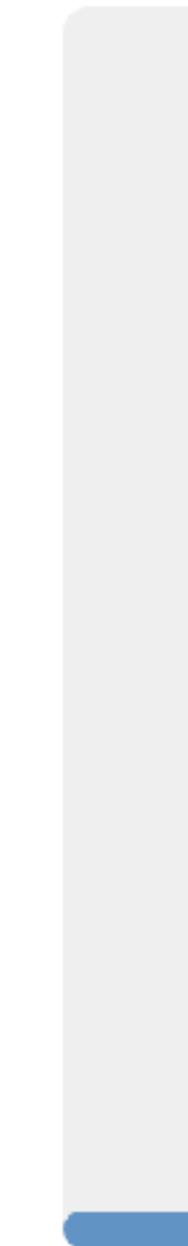
- A. 5
- B. 10
- C. 30
- D. 50
- E. 70

 0

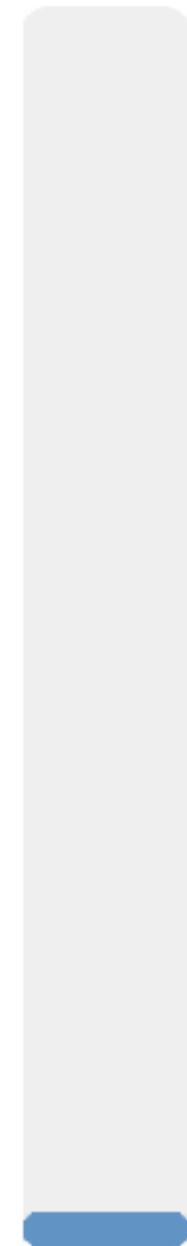
0



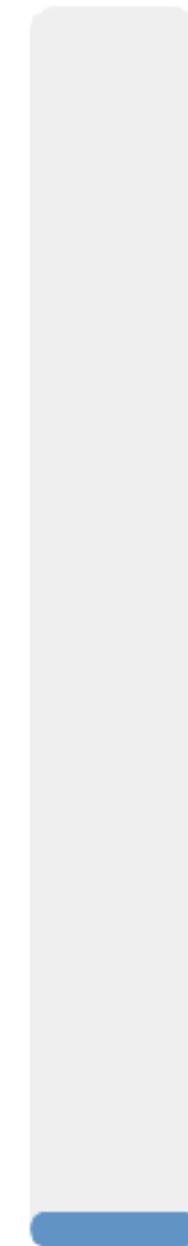
0



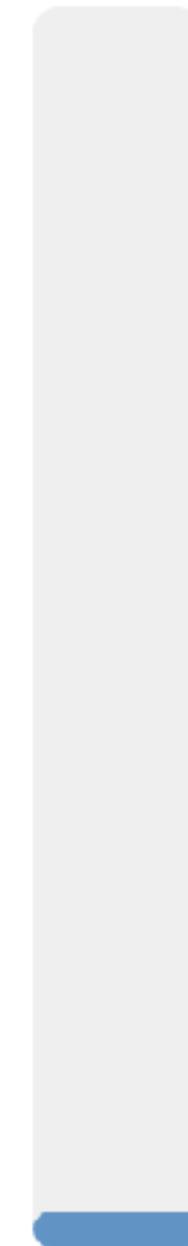
0



0



0



A

B

C

D

E



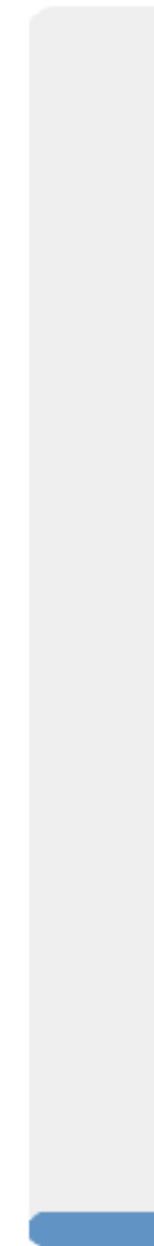
# The impact of “slow” memory

- 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. 5
- B. 10
- C. 30
- D. 50
- E. 70

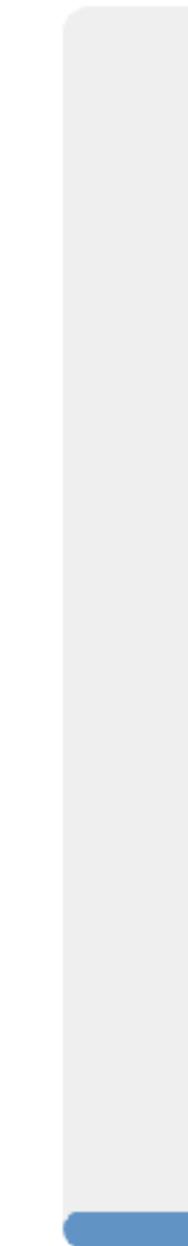
 0

0



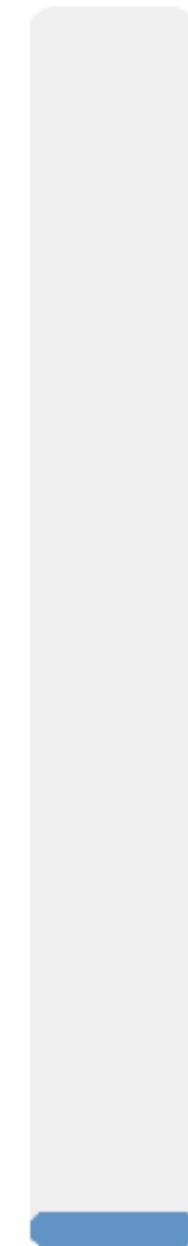
A

0



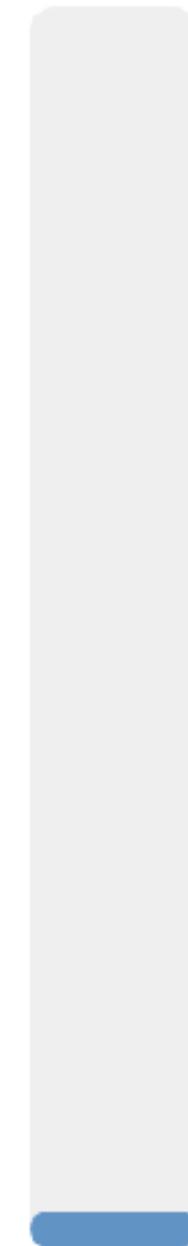
B

0



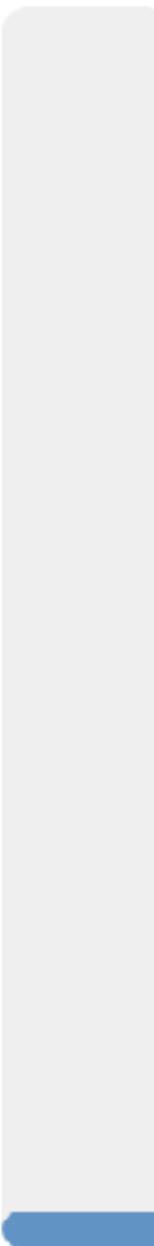
C

0



D

0



E

# 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. 5

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

B. 10

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

C. 30

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

D. 50

**Don't forget, instructions are also from “memory”**

E. 70

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

# 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	jr, jal	Procedure calls, returns, 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

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

# Take-aways: inside out our memory hierarchy

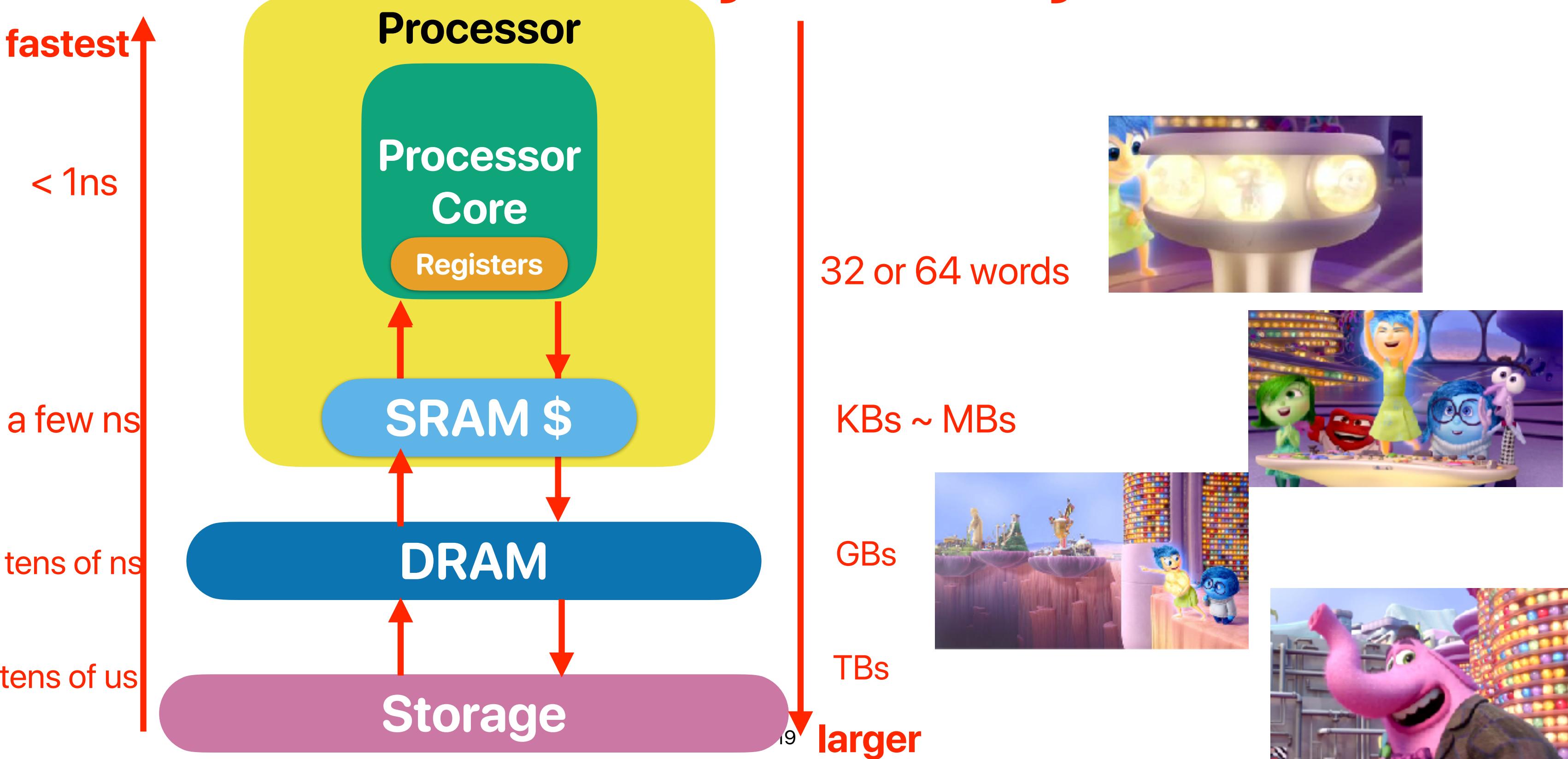
- Memory access time is the most critical performance problem
  - One memory operation is as expensive as 50 arithmetic operations
  - Processor has to fetch instructions from memory
  - We have an average of 33% of data memory access instructions!

# 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





## How can “memory hierarchy” help in performance?

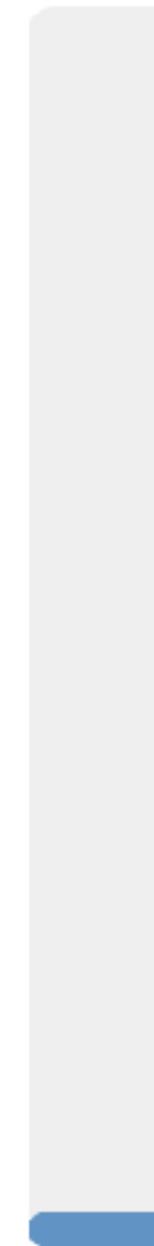
- Assume that we have a processor running @ 4 GHz and a program with 20% of load/store instructions. If the instruction has no memory access, the CPI is just 1. Now, in addition to we DDR5, whose latency 13.75 ns, we also got an SRAM cache with latency of just at 0.5 ns and can capture **90%** of the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

- A. 5
- B. 10
- C. 30
- D. 50
- E. 70



 0

0



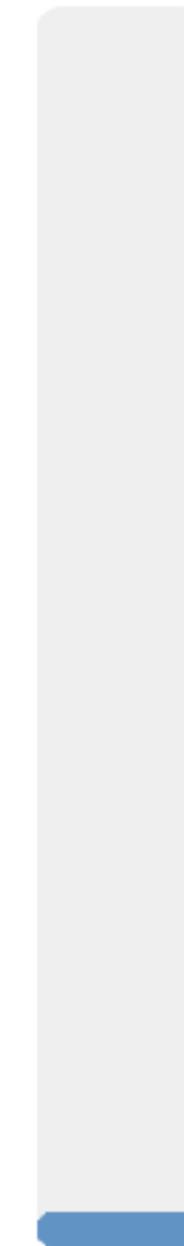
A

0



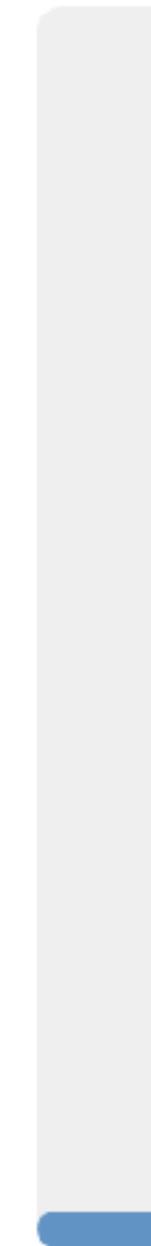
B

0



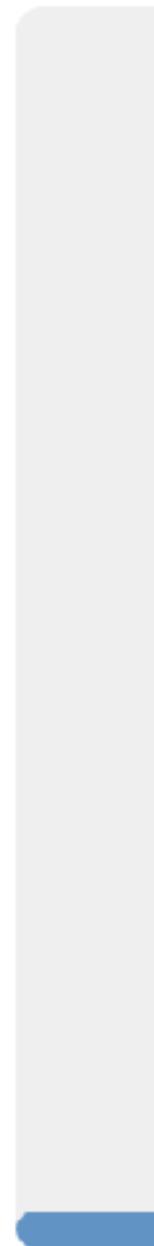
C

0



D

0



E



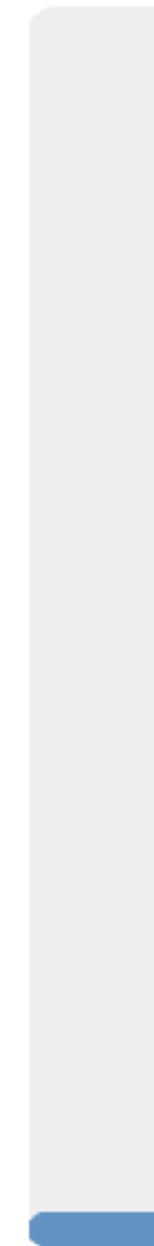
## How can “memory hierarchy” help in performance?

- 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. 5
- B. 10
- C. 30
- D. 50
- E. 70

 0

0



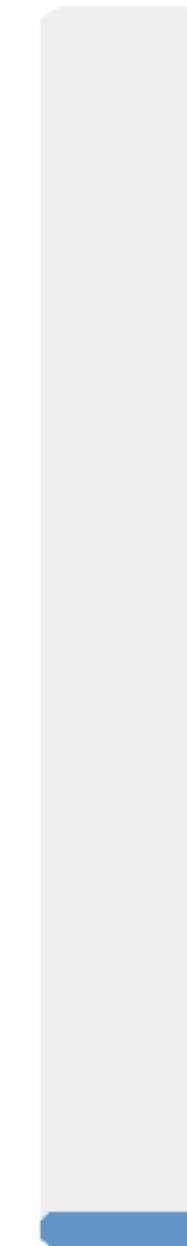
A

0



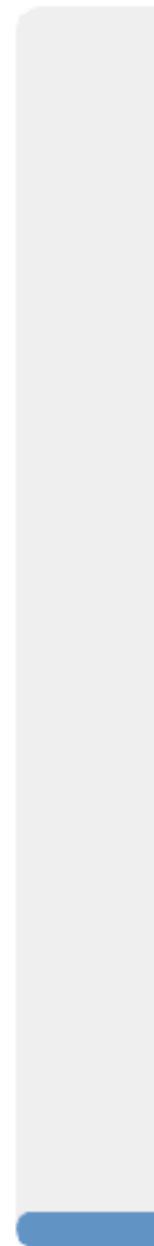
B

0



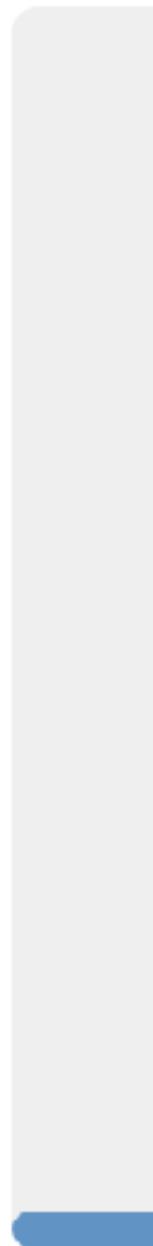
C

0



D

0



E

# 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)?

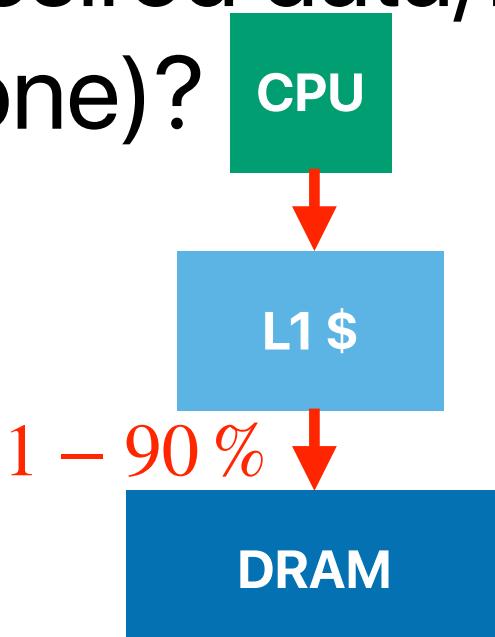
A. 5

**B. 10**

C. 30

D. 50

E. 70



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

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

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

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



Start the presentation to see live content. Still no live content? Install the app or get help at [PollEv.com/app](https://PollEv.com/app)

# 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
Ext. Family	19	Ext. Model	61
Stepping	2	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 : pk15b

**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
Ext. Family	6	Ext. Model	B7
Stepping	1	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	

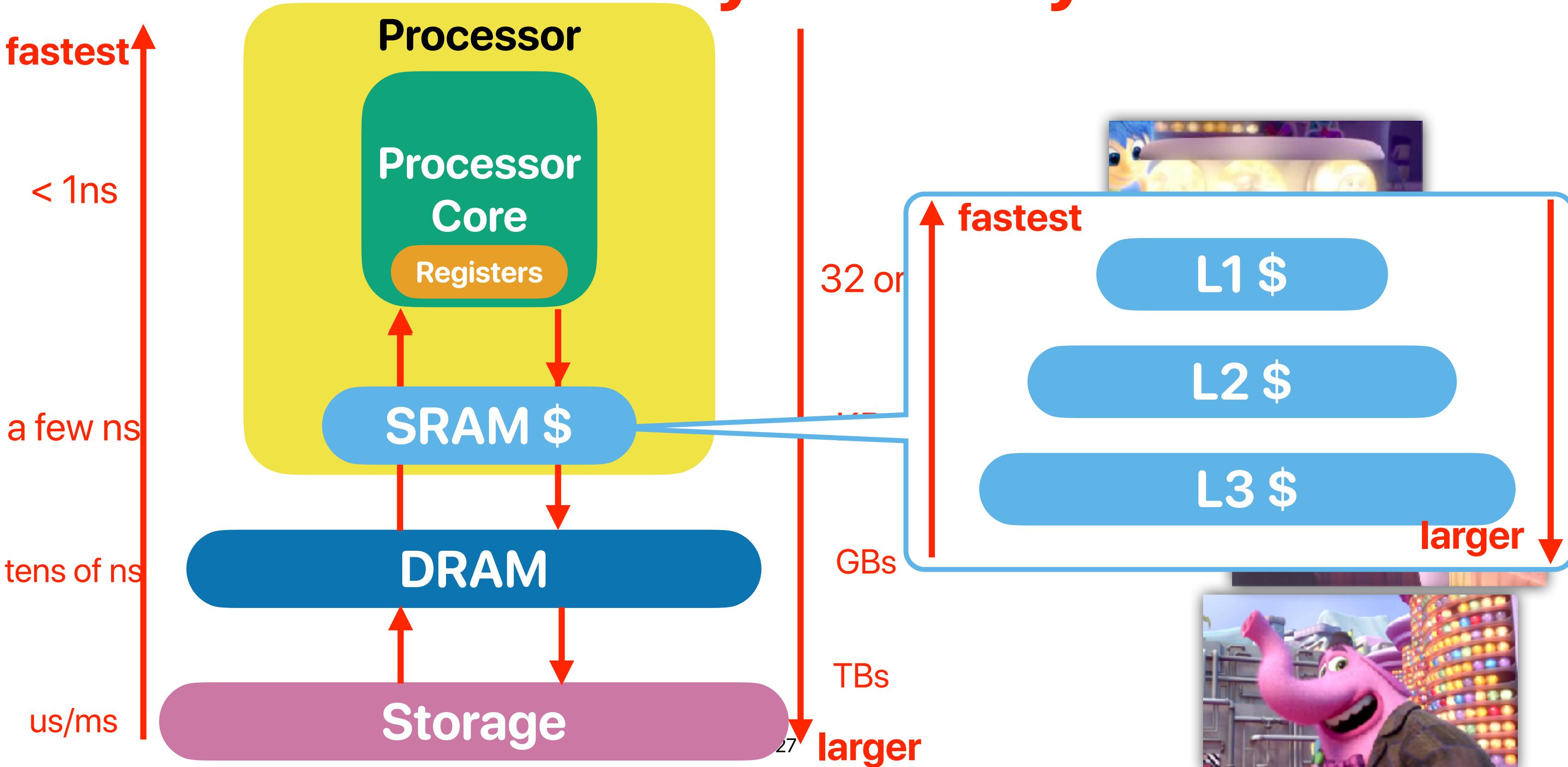
**Cache**

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

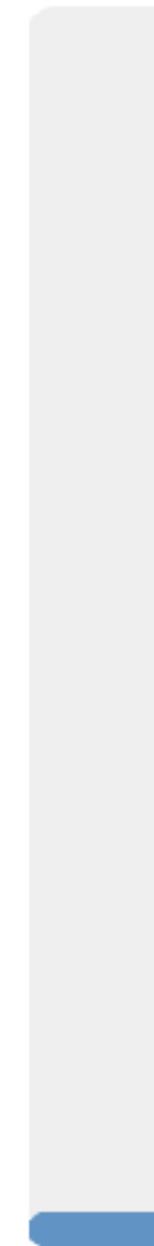
What's the average CPI (pick the closest one)?

- A. 5
- B. 10
- C. 30
- D. 50
- E. 70



 0

0



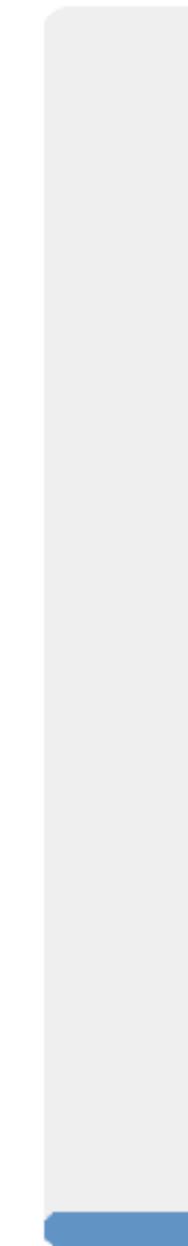
A

0



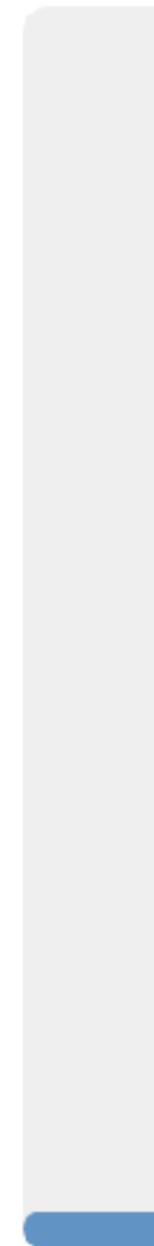
B

0



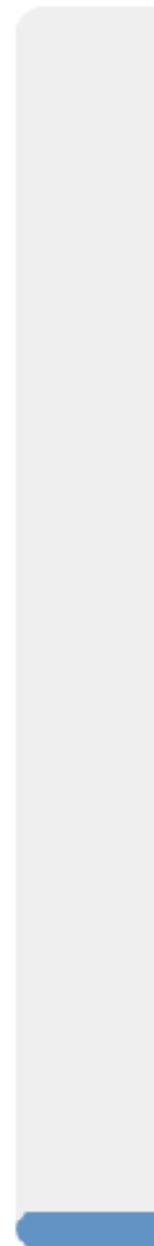
C

0



D

0



E



## 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

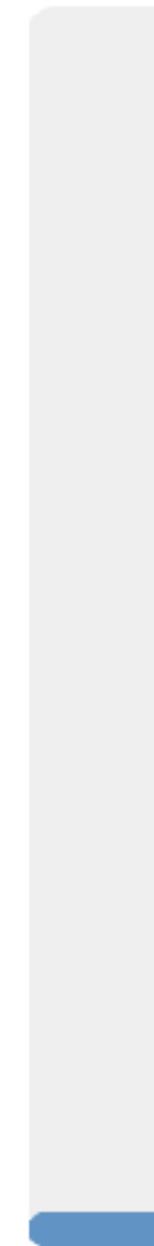
What's the average CPI (pick the closest one)?

- A. 5
- B. 10
- C. 30
- D. 50
- E. 70



 0

0



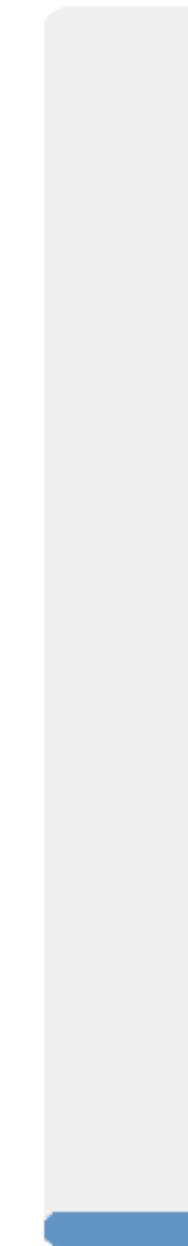
A

0



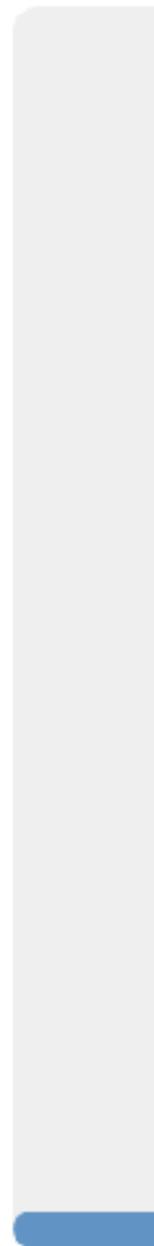
B

0



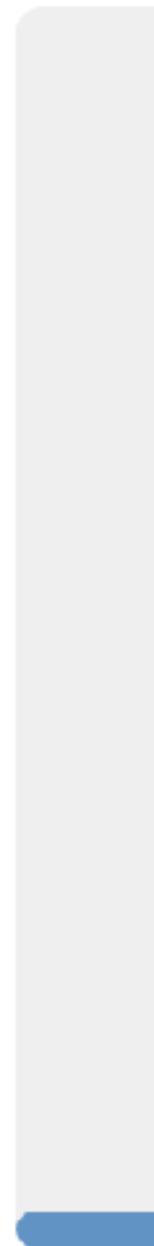
C

0



D

0



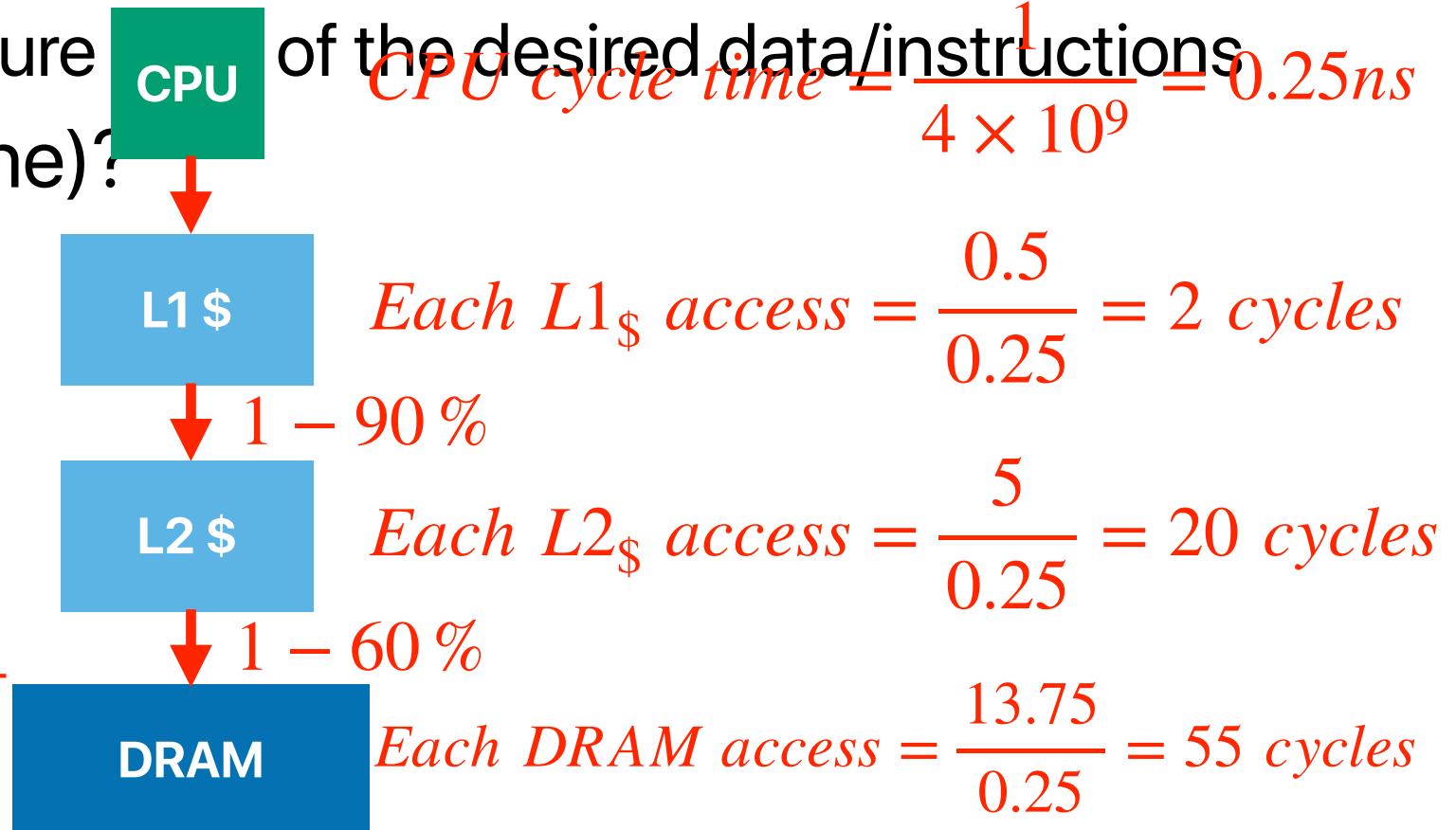
E

# 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 the DDR5, whose latency is 13.75 ns, we also got 2 levels SRAM caches where
    - the 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  $\frac{1}{CPU\ cycle\ time} = \frac{1}{4 \times 10^9} = 0.25\ ns$  of the desired data/instructions.
- What's the average CPI (pick the closest one)?

- A. 5
- B. 10
- C. 30
- D. 50

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



# L1? L2? L3?

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

The image displays two side-by-side CPU-Z software windows. The left window is for an AMD Ryzen 7 7700X (Code Name: Raphael) and the right is for an Intel Core i7 14700K (Code Name: Raptor Lake). Both windows show their respective processor details, including core speed, multiplier, bus speed, and rated FSB. The cache sections are highlighted with red boxes. For the Ryzen 7 7700X, the cache is listed as "8 x 32 KB" for L1 Data and "8 x 32 KB" for L1 Inst. For the Intel Core i7 14700K, the cache is listed as "8 x 48 KB + 12 x 32 KB" for L1 Data and "8 x 32 KB + 12 x 64 KB" for L1 Inst.

Processor	Name	Code Name	Max TDP	Core Voltage
AMD Ryzen 7 7700X	Raphael	105 W	1.123 V	
Intel Core i7 14700K	Raptor Lake	125 W	1.112 V	

Processor	Specification	Ext. Family	Ext. Model	Revision
AMD Ryzen 7 7700X	8-Core Processor	19	1	RPL-12
Intel Core i7 14700K	Intel(R) Core(TM) i7-14700K	6	B7	B0

Processor	Instructions
AMD Ryzen 7 7700X	MMX(+), SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, SSE4A x86-64, AMD-V, AES, AVX, AVX2, AVX512, FMA3, SHA
Intel Core i7 14700K	MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T VT-x, AES, AVX, AVX2, FMA3, SHA

Clocks (Core #0)	Cache
Core Speed	8 x 32 KB
Multiplier	8 x 32 KB
Bus Speed	8 x 1024 KB
Rated FSB	32 MBytes
Core Speed	8 x 48 KB + 12 x 32 KB
Multiplier	8 x 32 KB + 12 x 64 KB
Bus Speed	8 x 2 MB + 3 x 4 MB
Rated FSB	33 MBytes

Processor	Core Speed	Multiplier	Bus Speed	Rated FSB
AMD Ryzen 7 7700X	5188.99 MHz	x 52.0 ( 4 - 55.5 )	99.79 MHz	
Intel Core i7 14700K	5287.07 MHz	x 53.0 ( 8 - 55 )	99.76 MHz	

Processor	Cores	Threads
AMD Ryzen 7 7700X	8	16
Intel Core i7 14700K	8 + 12	28

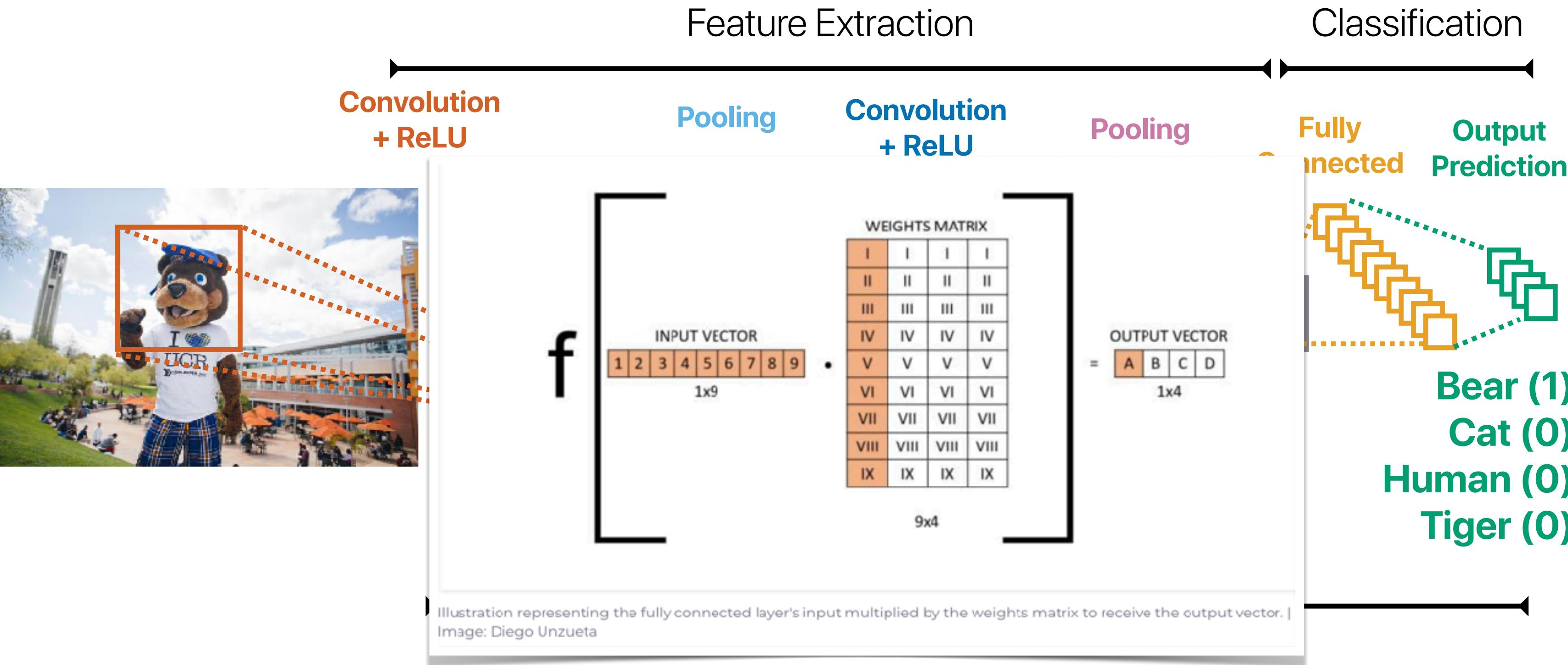
**CPU-Z** Ver. 2.09.0.x64    Tools    Validate    Close    **CPU-Z** Ver. 2.08.0.x64    Tools    Validate    Close

# 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





# Locality of data

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

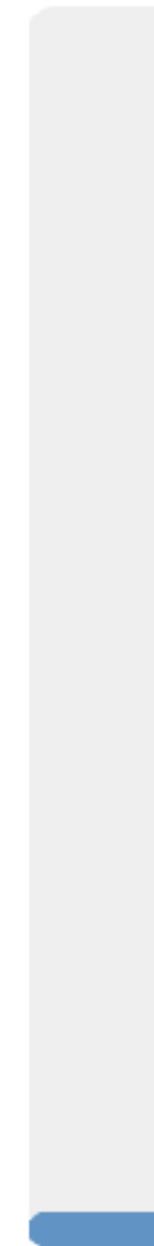
```
for(uint64_t i = 0; i < m; i++) {  
    result = 0;  
    for(uint64_t j = 0; j < n; j++) {  
        result += matrix[i][j]*vector[j];  
    }  
    output[i] = result;  
}
```



- 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

 0

0



A

0



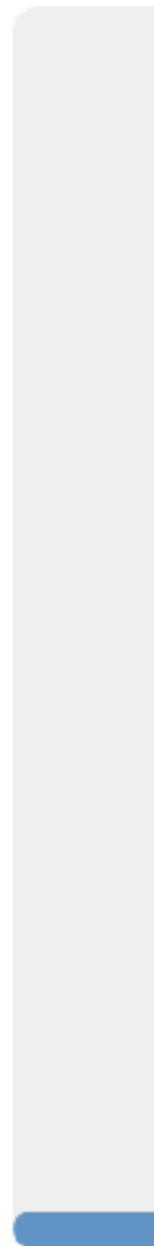
B

0



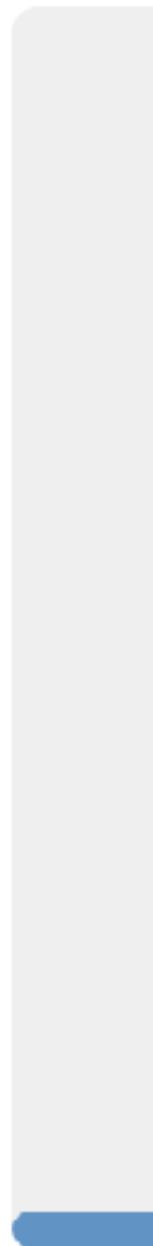
C

0



D

0



E



# Locality of data

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

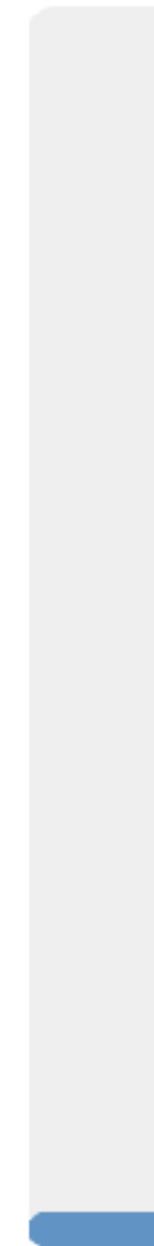
```
for(uint64_t i = 0; i < m; i++) {  
    result = 0;  
    for(uint64_t j = 0; j < n; j++) {  
        result += matrix[i][j]*vector[j];  
    }  
    output[i] = result;  
}
```



- 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

 0

0



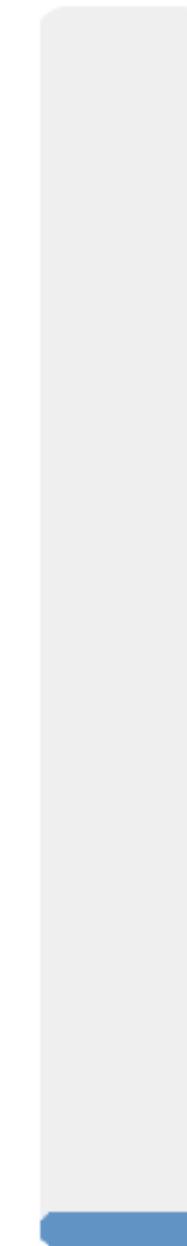
A

0



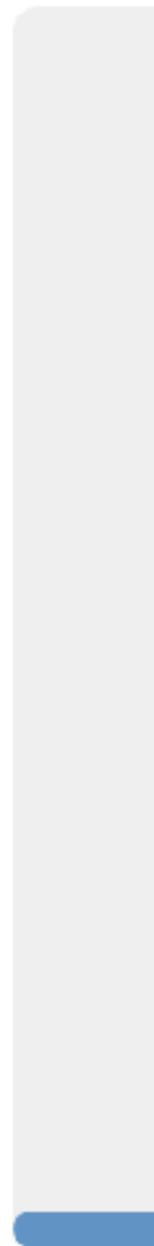
B

0



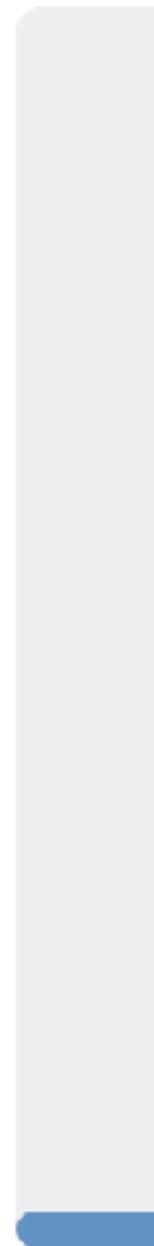
C

0



D

0



E

# 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;  
}
```

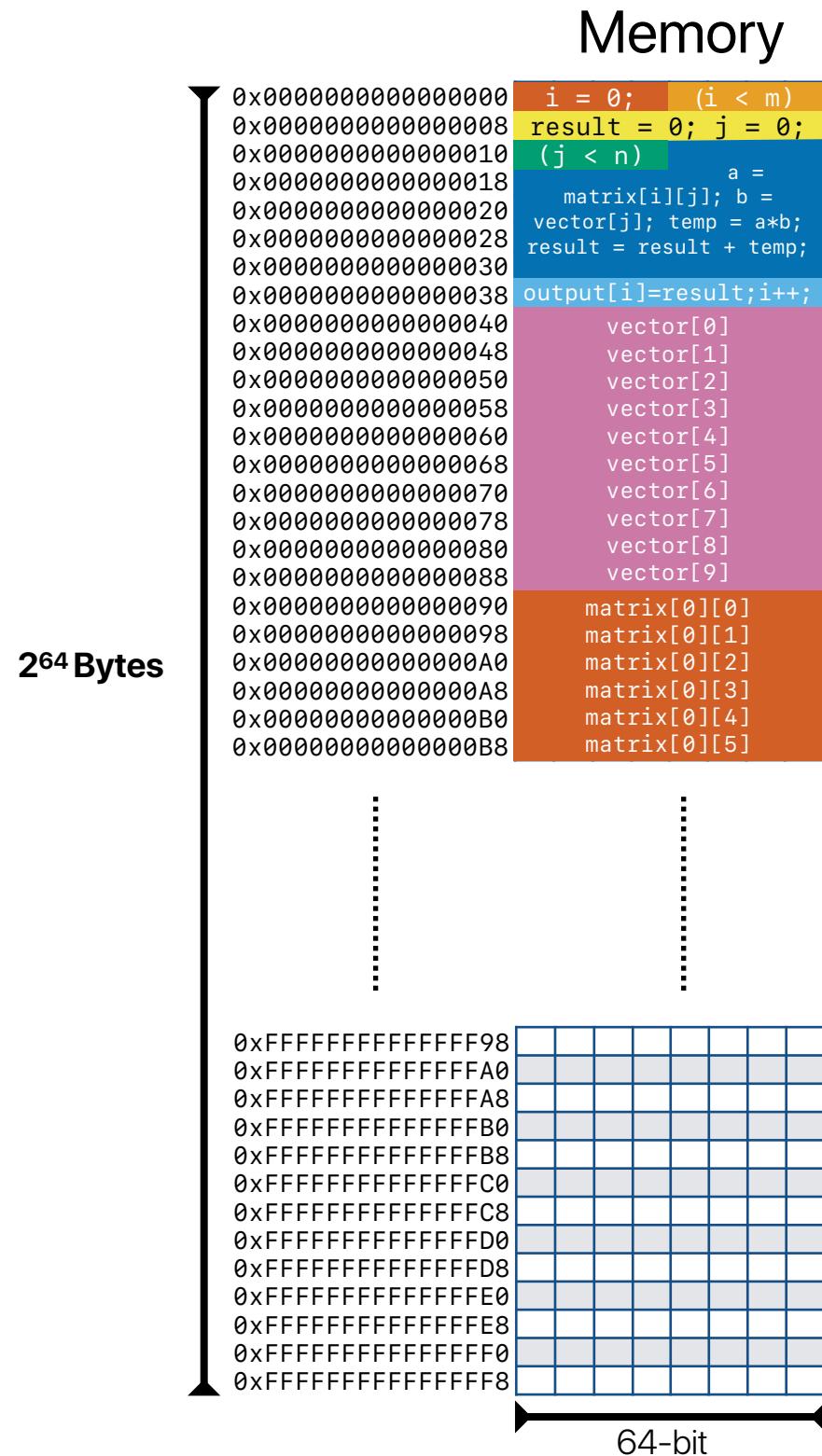
spatial locality:  
`matrix[0][0], matrix[0][1], matrix[0][2], ...`  
`vector[0], vector[1], ..., vector[n]`  
temporal locality:  
`reuse of vector[0], vector[1], ...`

- 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



Start the presentation to see live content. Still no live content? Install the app or get help at [PollEv.com/app](https://PollEv.com/app)

# Data/instructions in the memory



# 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;  
}
```

repeat many times —  
temporal locality!

```
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



# Locality

- Spatial locality — application tends to visit nearby stuffs in the memory
  - Code — the current instruction, and then the next instruction

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

- 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)

# 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

# Outline

- Designing a cache that captures the predictability
- The A, B, Cs of caches
- Estimating how cache friendly is our code

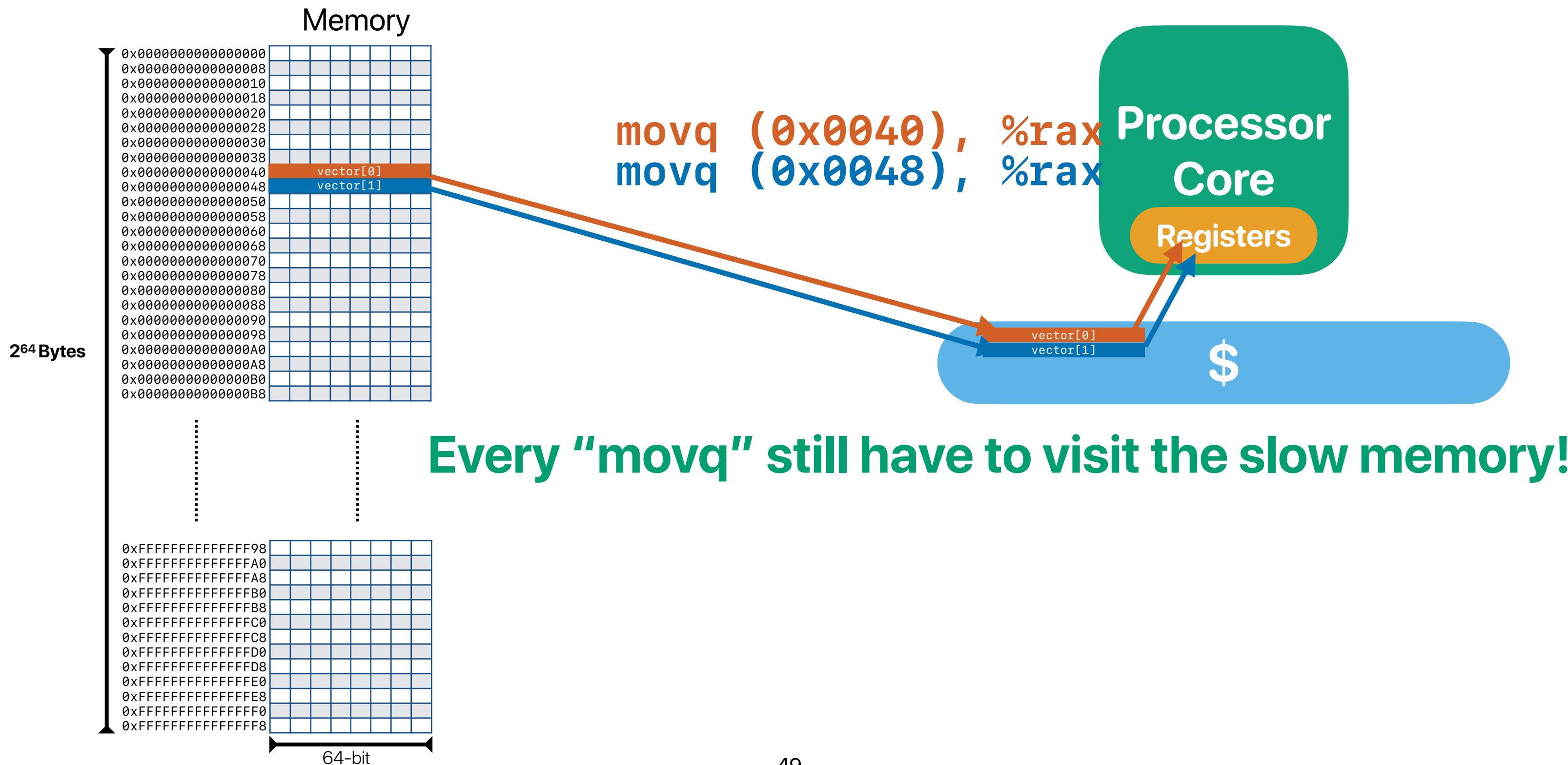
# Designing a hardware to exploit locality

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

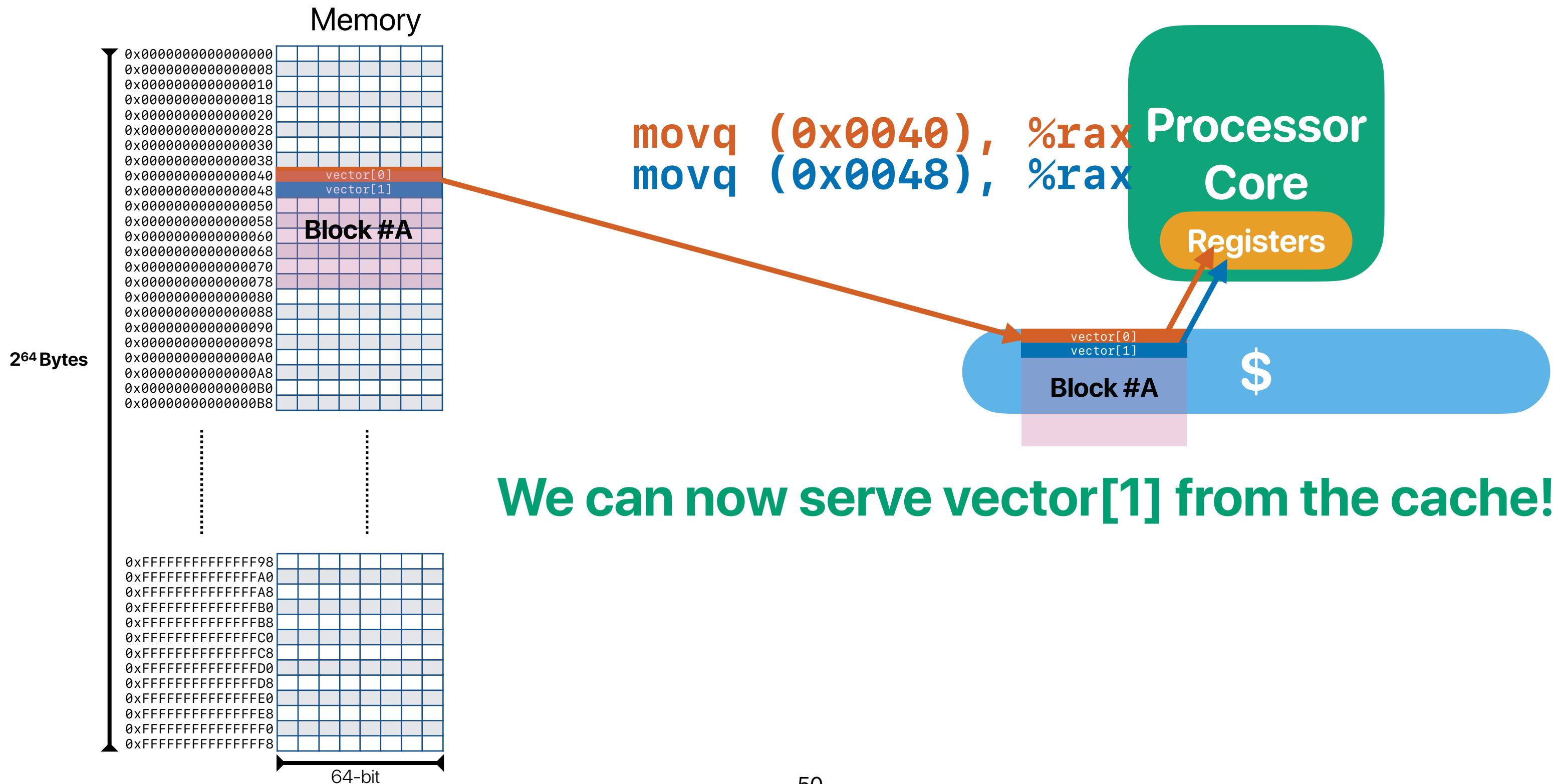
**We need to “cache consecutive memory locations” every time  
— the cache should store a “block” of code/data**

- 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)

# If we just cache what has requested



# If we can cache a block of data



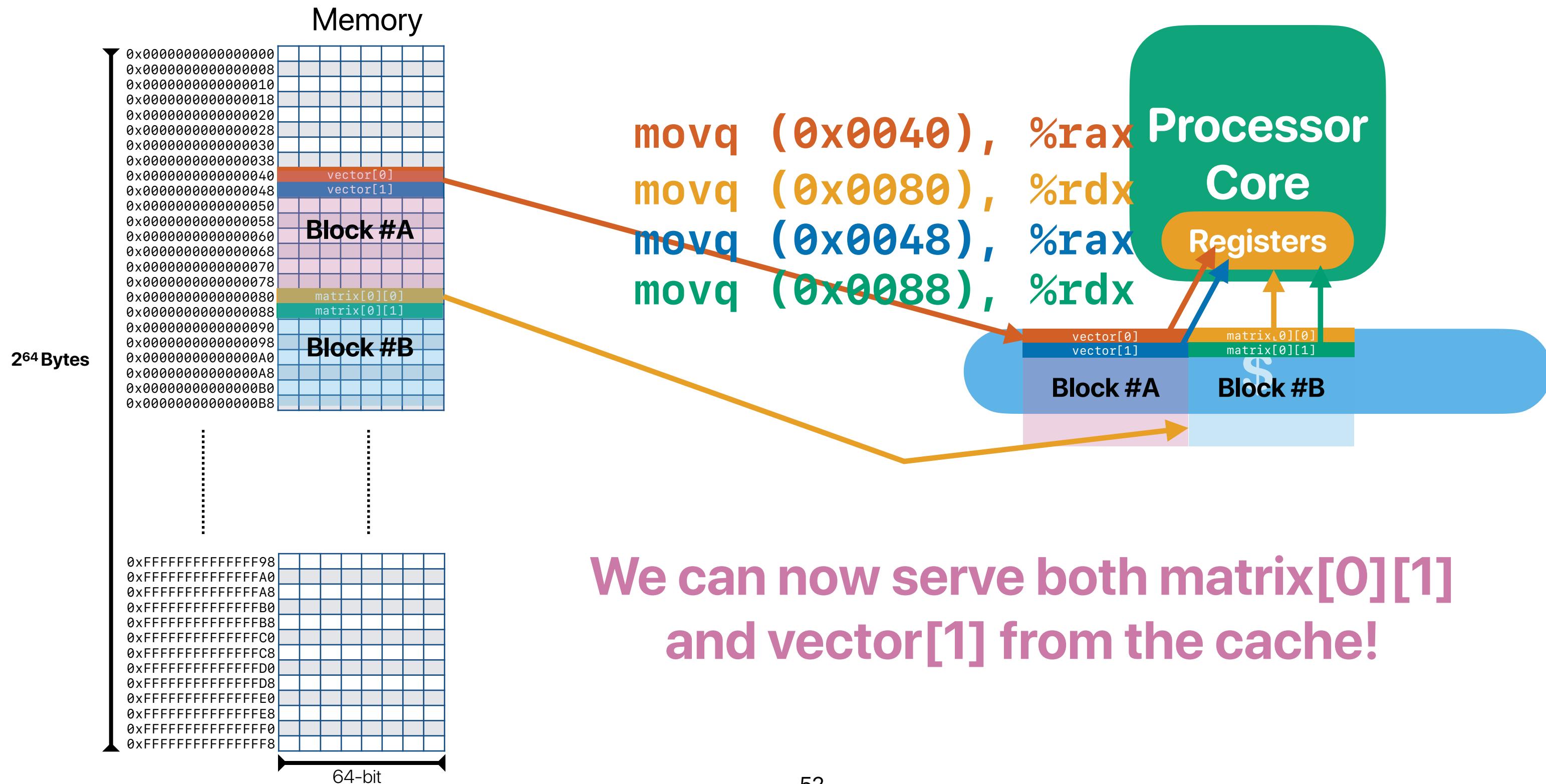
# Recap: Locality

- 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

# We need to cache multiple blocks of data!



# 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”**

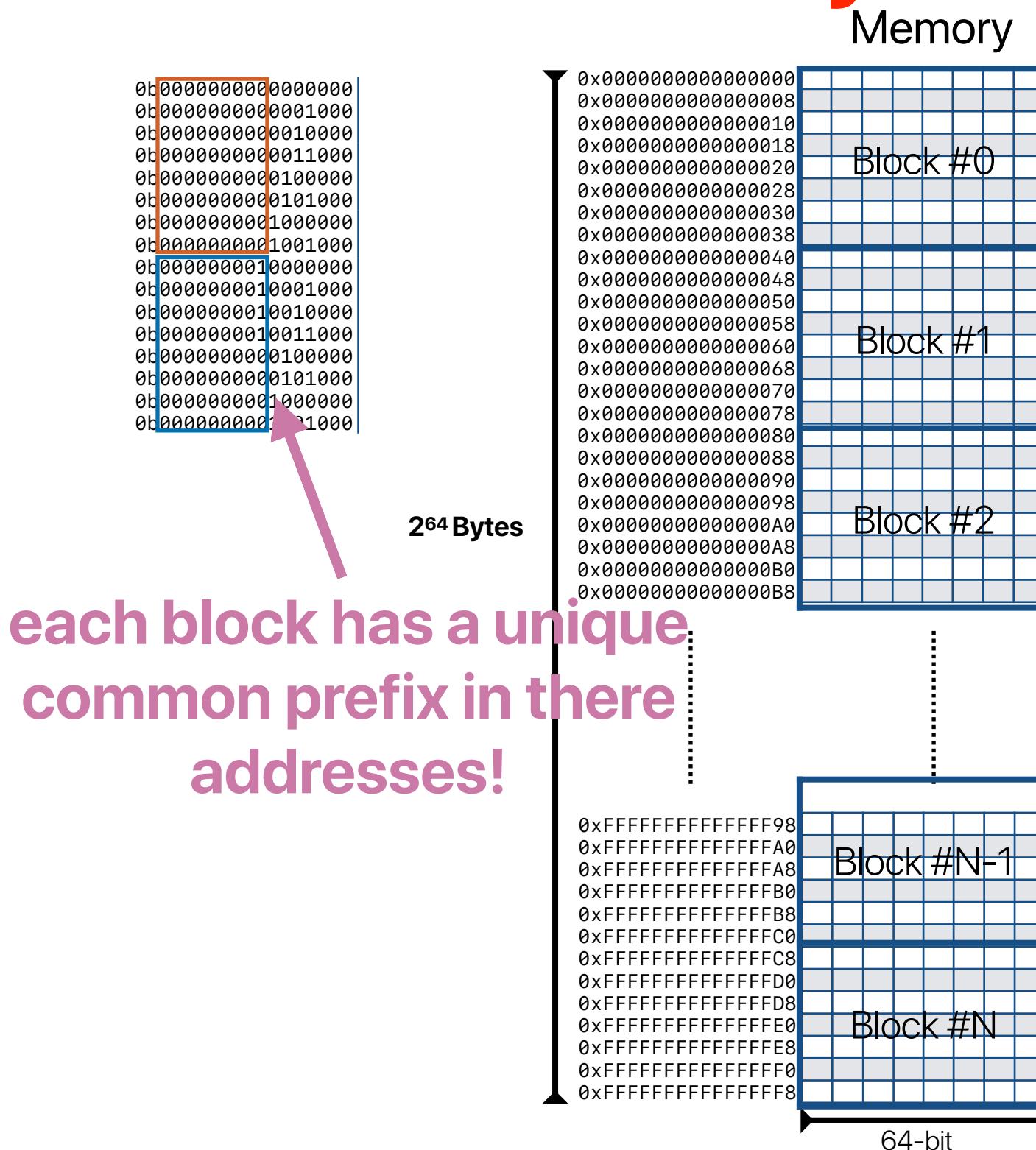
- Typically tens of static instructions — at most several KBs
- the cache should store a few blocks**
- Data — program can read/write the same data many times (e.g., vectors in matrix-vector product)
- the cache must be able to distinguish blocks**

# How to tell who is there?

?

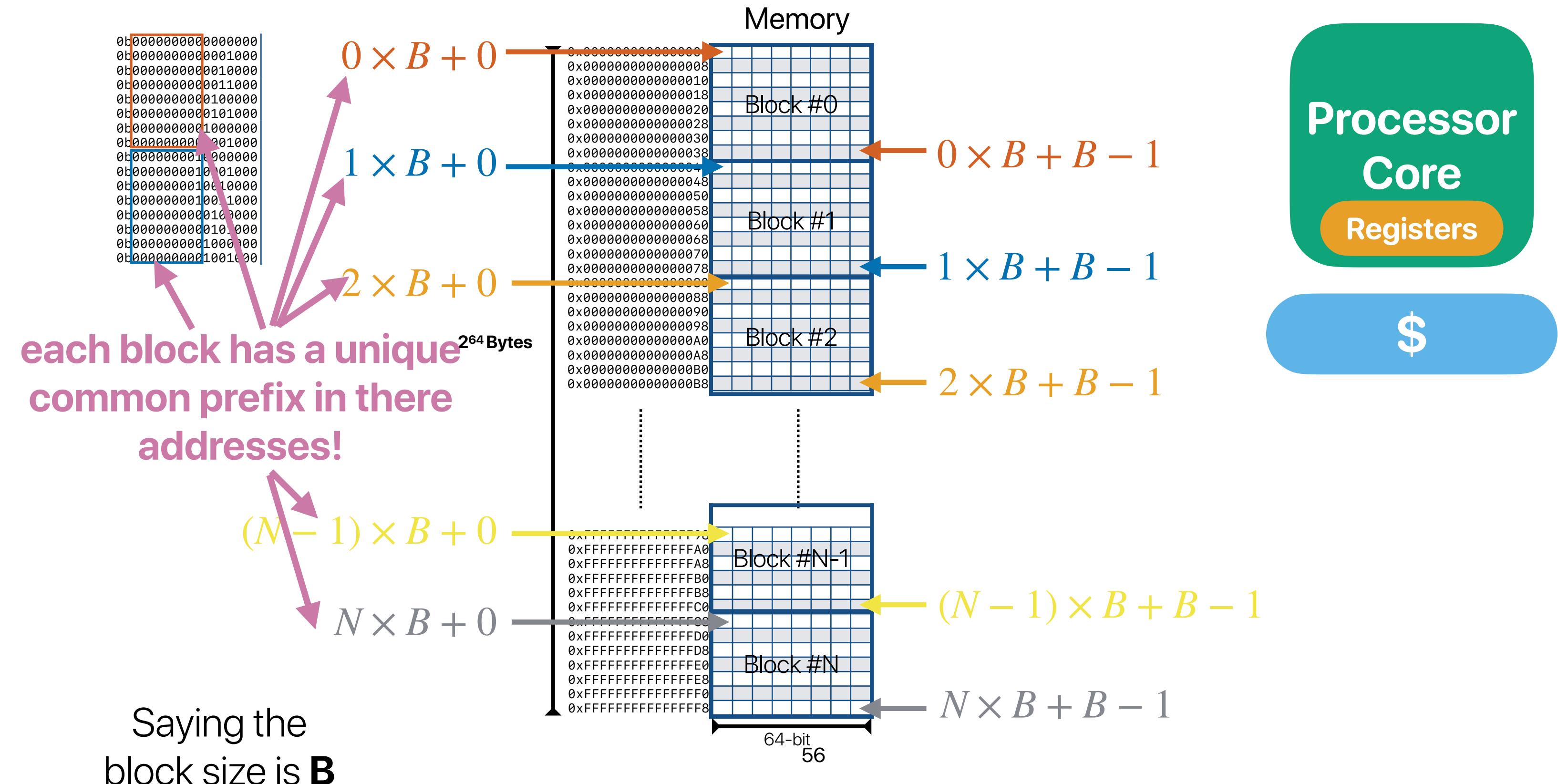
?	0123456789ABCDEF
?	This is CS 203:
	Advanced Compute
	r Architecture!
	This is CS 203:
	Advanced Compute
	r Architecture!
	This is CS 203:
	Advanced Compute
	r Architecture!
	This is CS 203:
	Advanced Compute
	r Architecture!
	This is CS 203:
	Advanced Compute
	r Architecture!
	This is CS 203:

# Partition memory addresses into fix-sized chunks



\$

# Partition memory addresses into fix-sized chunks



# How to tell who is there?

# the common address prefix in each block

# tag array

0x000	This is CS 203: Advanced Compute Architecture!
0x001	This is CS 203: Advanced Compute Architecture!
0xF07	This is CS 203: Advanced Compute Architecture!
0x100	This is CS 203: Advanced Compute Architecture!
0x310	This is CS 203: Advanced Compute Architecture!
0x450	This is CS 203: Advanced Compute Architecture!
0x006	This is CS 203: Advanced Compute Architecture!
0x537	This is CS 203: Advanced Compute Architecture!
0x266	This is CS 203: Advanced Compute Architecture!
0x307	This is CS 203: Advanced Compute Architecture!
0x265	This is CS 203: Advanced Compute Architecture!
0x80A	This is CS 203: Advanced Compute Architecture!
0x620	This is CS 203: Advanced Compute Architecture!
0x630	This is CS 203: Advanced Compute Architecture!
0x705	This is CS 203: Advanced Compute Architecture!
0x216	This is CS 203: Advanced Compute Architecture!



## Processor Core

Registers

# How to tell whether

block offset  
tag

1w 0x0008

1w 0x4048

0x404 not found,  
go to lower-level memory

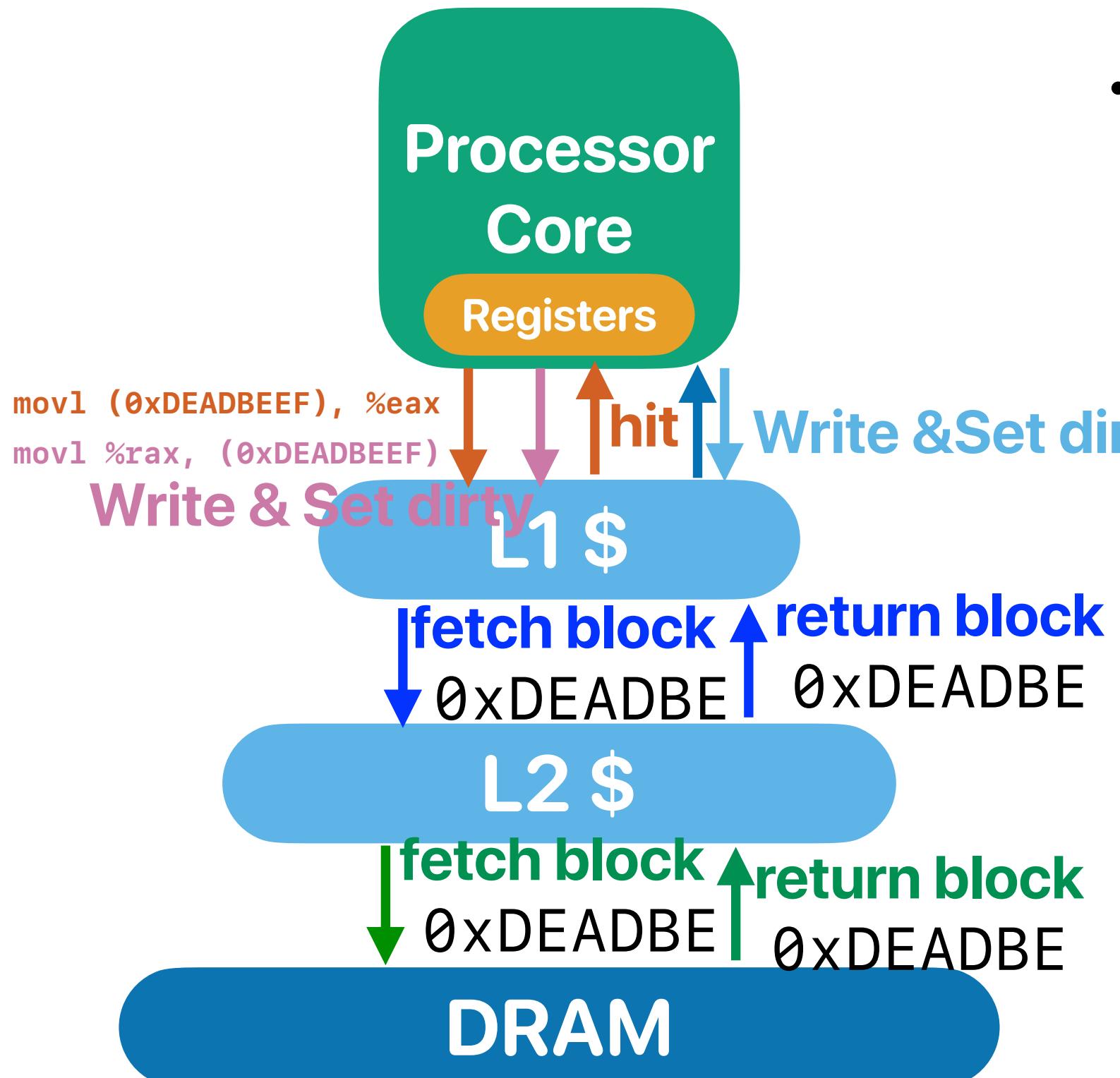
		Valid Bit	Dirty Bit	data
		tag		0123456789ABCDEF
1	1	0x000		This is CSE13:
1	1	0x001		Advanced Compute
1	0	0xF07		r Architecture!
0	1	0x100		This is CS 203:
1	1	0x310		Advanced Compute
1	1	0x450		r Architecture!
0	1	0x006		This is CS 203:
0	1	0x537		Advanced Compute
1	1	0x266		r Architecture!
1	1	0x307		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: designing caches

- 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

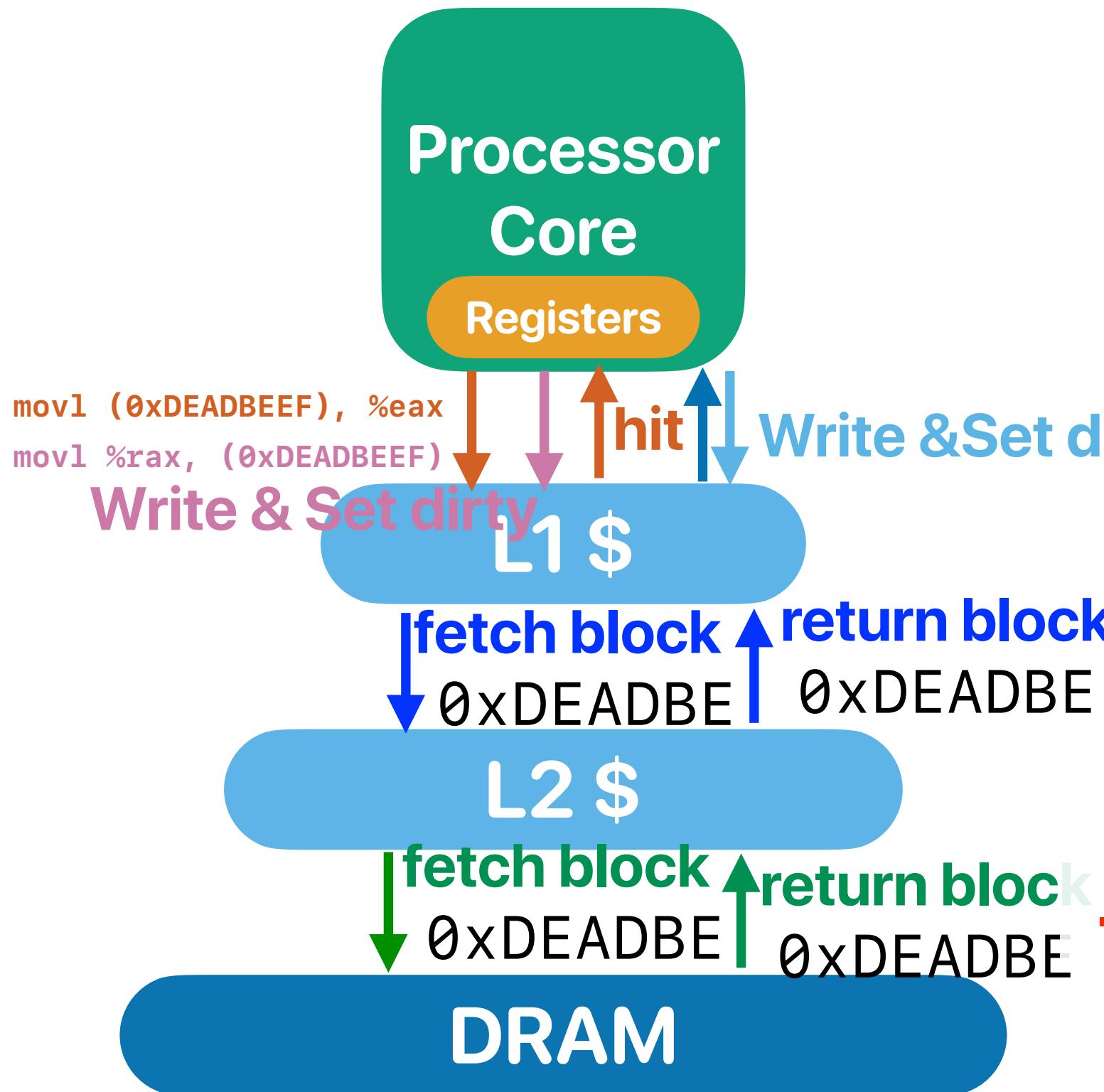
# **Put everything all together: How cache interacts with CPU**

# Processor/cache interaction



- Processor sends memory access request to L1-\$
  - if hit & it's a read**
    - Read: return data
    - Write: Update “ONLY” in L1 and set DIRTY **Why don't we write to L2?**
      - Too slow
  - if miss**
    - Fetch the requesting block from lower-level memory hierarchy and place in the cache
    - Present the write “ONLY” in L1 and set DIRTY **What if we run out of \$ blocks?**

# Processor/cache interaction

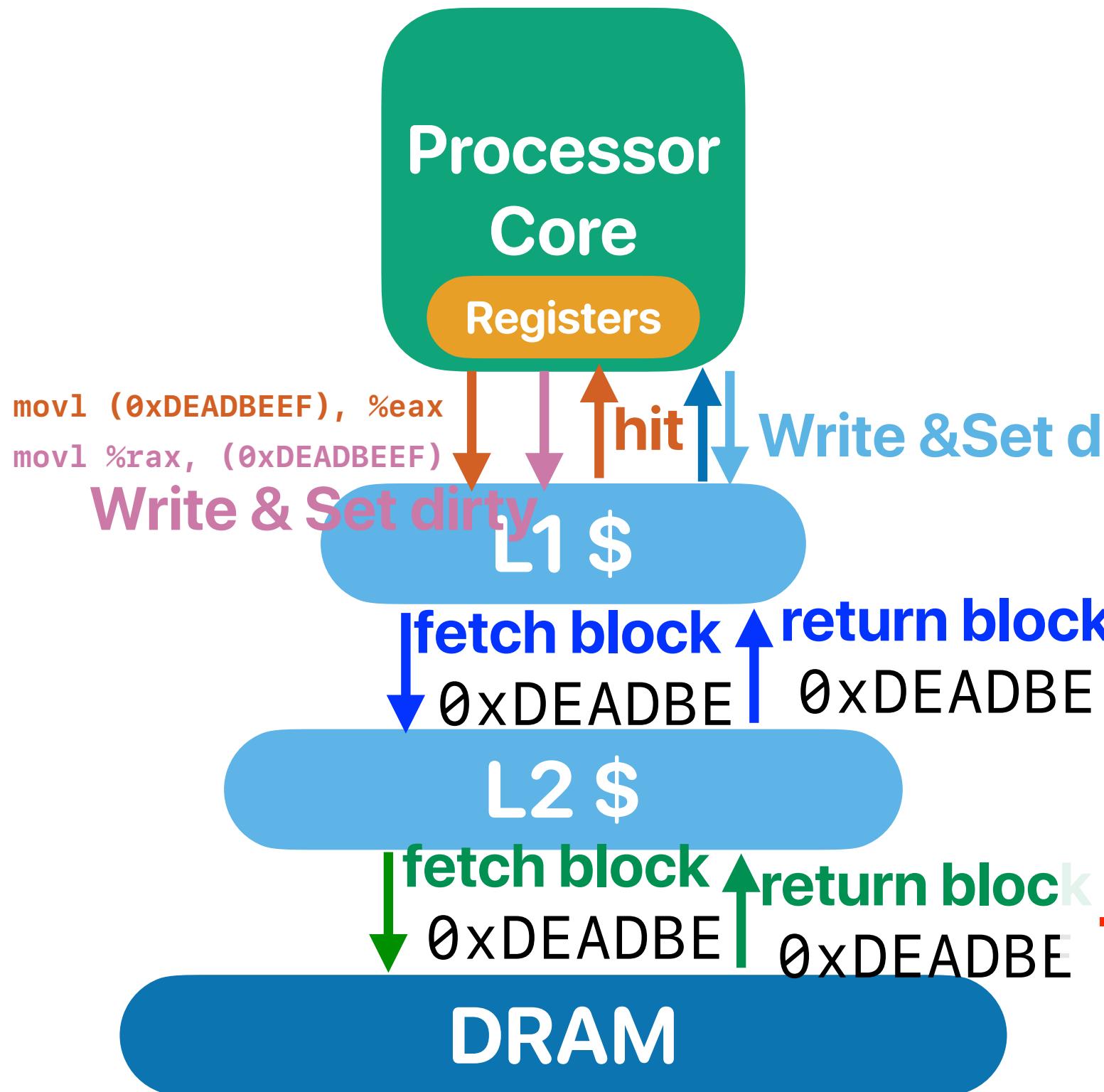


- Processor sends memory access request to L1-\$
  - if hit & it's a read**
    - Read:** return data
    - Write:** Update "ONLY" in L1 and set DIRTY
  - if miss**
    - If there an empty block — place the data there
    - If NOT (most frequent case) — select a **victim block**
      - Least Recently Used (LRU) policy

**What if the victim block is modified?**  
— ignoring the update is not acceptable!

DIRTY

# Processor/cache interaction

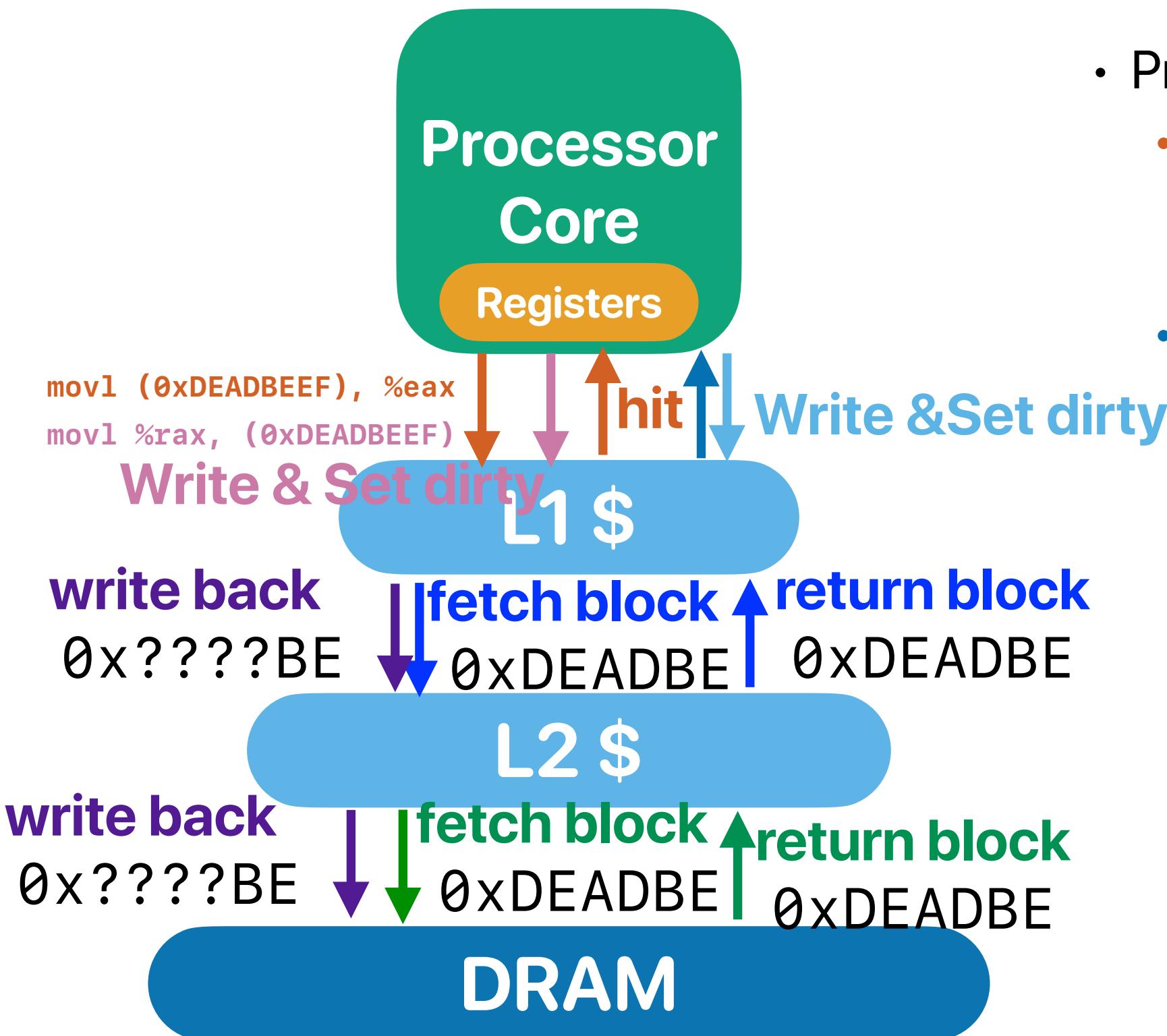


- Processor sends memory access request to L1-\$
  - if hit & it's a read**
    - Read:** return data
    - Write:** Update "ONLY" in L1 and set DIRTY
  - if miss**
    - If there an empty block — place the data there
    - If NOT (most frequent case) — select a **victim block**
      - Least Recently Used (LRU) policy

**What if the victim block is modified?**  
— ignoring the update is not acceptable!

DIRTY

# Processor/cache interaction



- Processor sends memory access request to L1-\$
  - **if hit & it's a read**
    - Read: **return data**
    - Write: **Update "ONLY" in L1 and set DIRTY**
  - **if miss**
    - If there an empty block — place the data there
    - If NOT (most frequent case) — select a **victim block**
      - Least Recently Used (LRU) policy
    - If the victim block is "dirty" & "valid"
      - **Write back** the block to lower-level memory hierarchy
      - If write-back or fetching causes any miss, repeat the same process
    - Fetch the requesting block from lower-level memory hierarchy and place in the cache
    - **Present the write "ONLY" in L1 and set DIRTY**

# Take-aways: designing caches

- 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
- Hierarchical caching — data must be presented on the top level (L1) before the processor can use

## Processor Core

Registers

# How to tell whether

block offset  
tag

1w 0x0008

1w 0x4048

0x404 not found,  
go to lower-level memory

The complexity of search the matching tag—

$O(n)$ —will be slow if our cache size grows!

Can we search things faster?

—hash table!  $O(1)$

Valid Bit		Dirty Bit	Tag	Data
1	1		0x000	This is CSE13:
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:

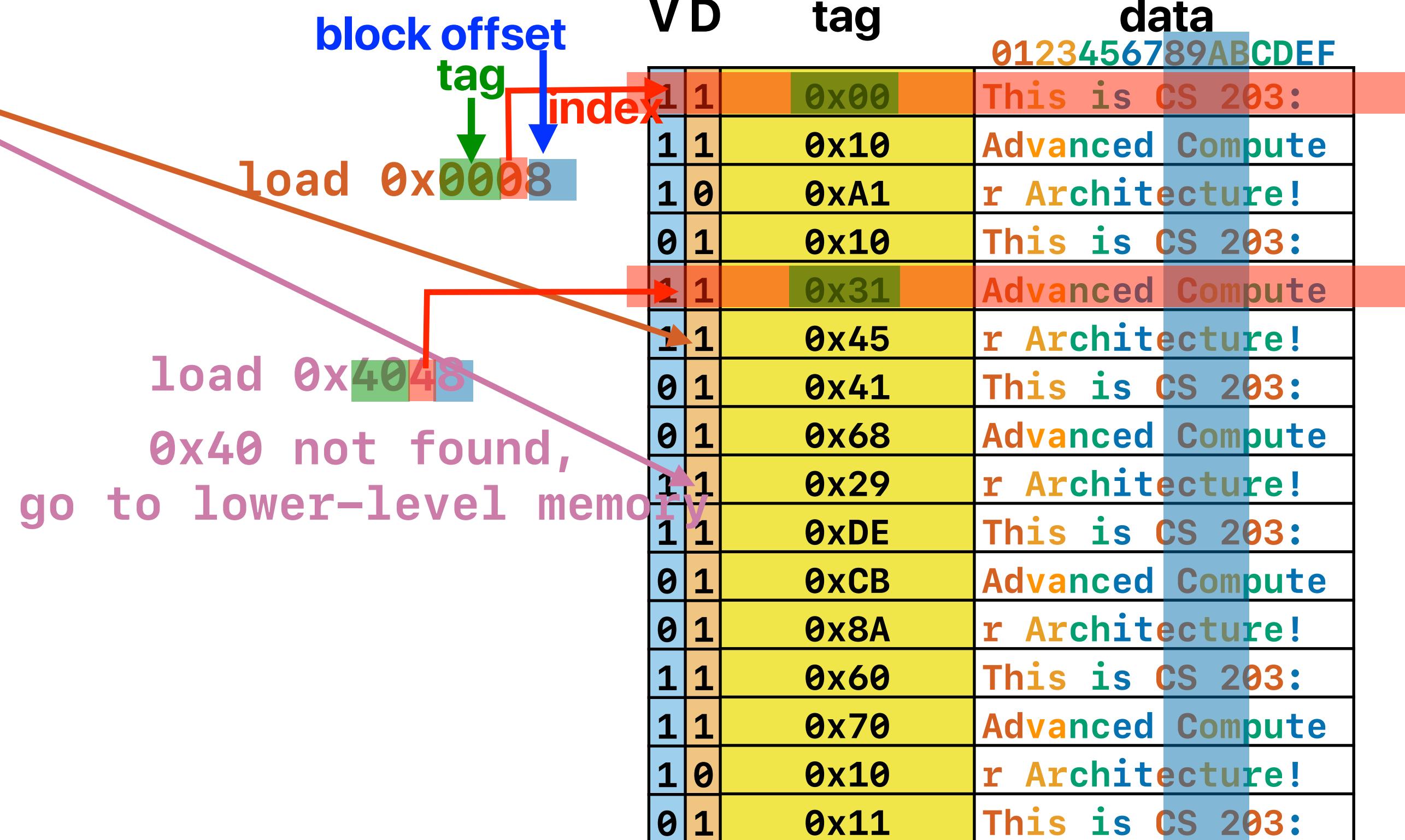
Tell if the block here can be used

Tell if the block here is modified

## Processor Core

Registers

# Hash-like structure — direct-mapped cache



# Take-aways: designing caches

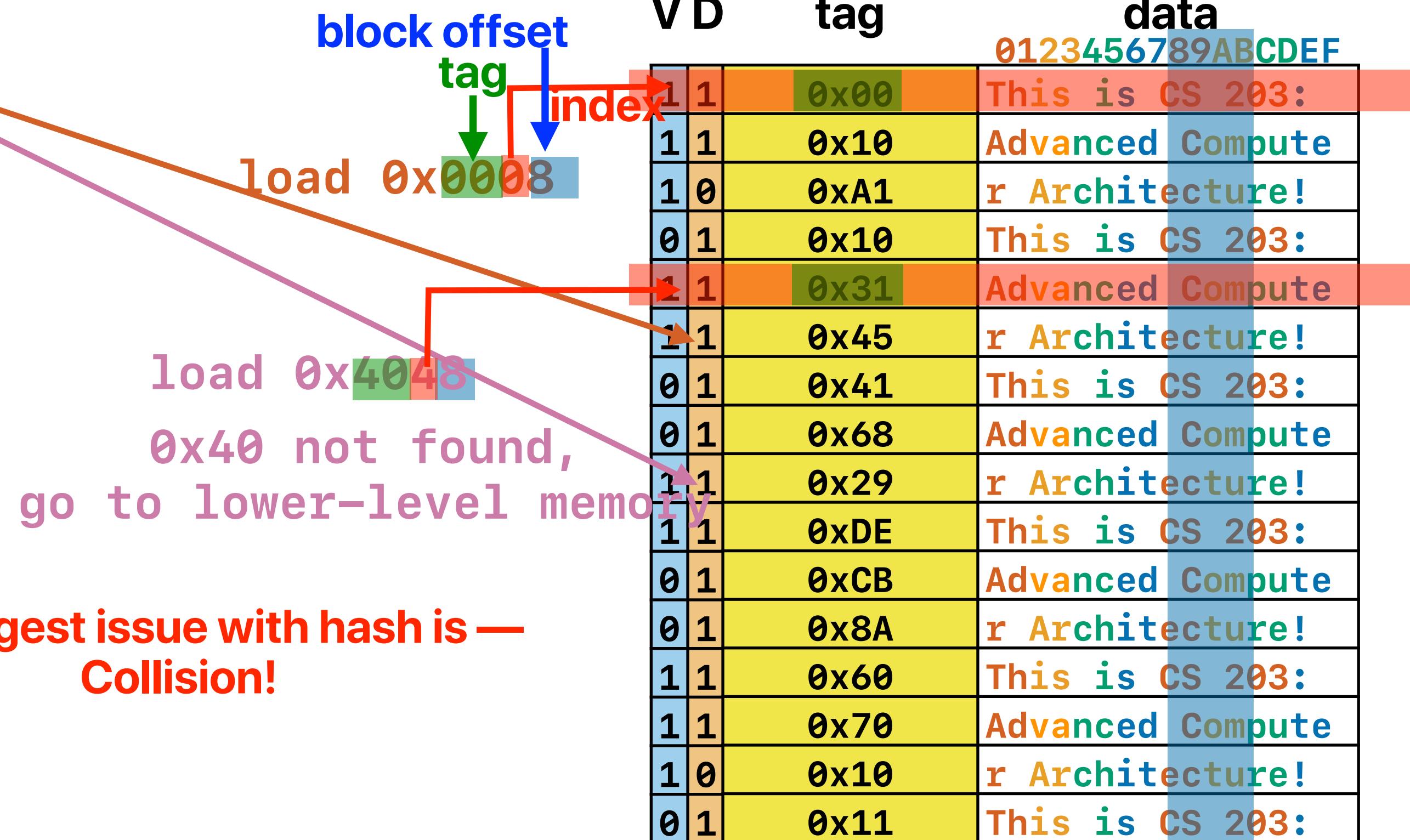
- Basic cache structures —
  - Caching in granularity of a block to capture spatial locality
  - Caching multiple blocks to keep frequently used data — temporal locality
  - Tags to distinguish cached blocks
- Hierarchical caching — data must be presented on the top level (L1) before the processor can use
- Optimizing cache structures
  - Hash block into “sets” to reduce the search time

Processor

Core

Registers

# Hash-like structure — direct-mapped cache



The biggest issue with hash is —  
Collision!

# Way-associative cache

memory address:

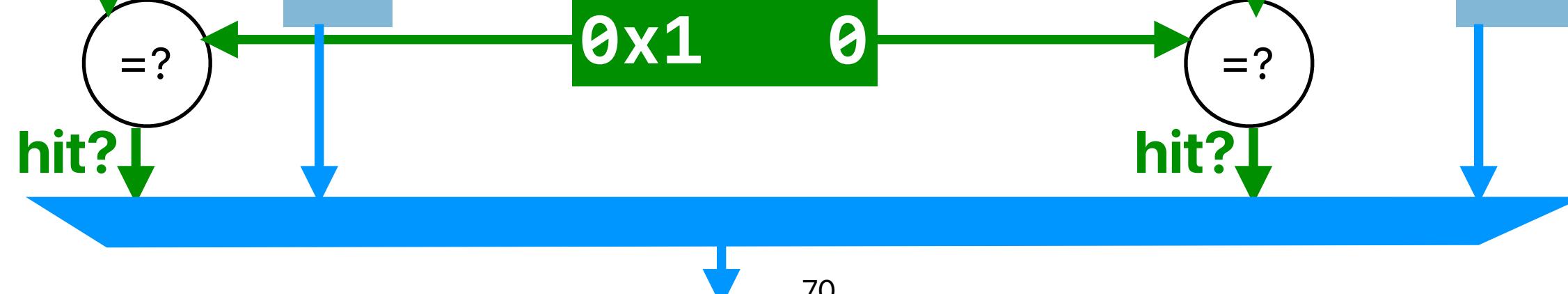
0x0      8      2      4  
 set    block  
 index    offset

memory address:

0b0000100000100100

V	D	tag	data
1	1	0x29	r Architecture!
1	1	0xDE	This is CS 203:
1	0	0x10	Advanced Compute
0	1	0x8A	r Architecture!
1	1	0x60	This is CS 203:
1	1	0x70	Advanced Compute
0	1	0x10	r Architecture!
0	1	0x11	This is CS 203:

V	D	tag	data
1	1	0x00	This is CS 203:
1	1	0x10	Advanced Compute
1	0	0xA1	r Architecture!
0	1	0x10	This is CS 203:
1	1	0x31	Advanced Compute
1	1	0x45	r Architecture!
0	1	0x41	This is CS 203:
0	1	0x68	Advanced Compute





Start the presentation to see live content. Still no live content? Install the app or get help at [PollEv.com/app](https://PollEv.com/app)

# Blocksize == Linesize

```
/sys/devices/system/cpu/cpu0/cache/index0/physical_line_partition:1
/sys/devices/system/cpu/cpu0/cache/index0/number_of_sets:64
/sys/devices/system/cpu/cpu0/cache/index0/ways_of_associativity:12
/sys/devices/system/cpu/cpu0/cache/index0/id:0
/sys/devices/system/cpu/cpu0/cache/index0/shared_cpu_list:0-1
/sys/devices/system/cpu/cpu0/cache/index0/type:Data
/sys/devices/system/cpu/cpu0/cache/index0/size:48K
/sys/devices/system/cpu/cpu0/cache/index0/level:1
/sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size:64
/sys/devices/system/cpu/cpu0/cache/index0/shared_cpu_map:000003
```

# Blocksize == Linesize

```
/sys/devices/system/cpu/cpu0/cache/index0/physical_line_partition:1
/sys/devices/system/cpu/cpu0/cache/index0/number of sets:64
/sys/devices/system/cpu/cpu0/cache/index0/ways of associativity:12
/sys/devices/system/cpu/cpu0/cache/index0/id:0
/sys/devices/system/cpu/cpu0/cache/index0/shared_cpu_list:0-1
/sys/devices/system/cpu/cpu0/cache/index0/type:Data
/sys/devices/system/cpu/cpu0/cache/index0/size:48K
/sys/devices/system/cpu/cpu0/cache/index0/level:1
/sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size:64
/sys/devices/system/cpu/cpu0/cache/index0/shared_cpu_map:000003
```

# Take-aways: designing caches

- 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
- Hierarchical caching — data must be presented on the top level (L1) before the processor can use
- Optimizing cache structures
  - Hash block into “sets” to reduce the search time
  - Set-associativity to reduce the “collision” problem

# Way-associative cache

memory address:

0x0      8      2      4  
 set    block  
 index    offset

memory address:

0b0000100000100100

V	D	tag	data
1	1	0x29	r Architecture!
1	1	0xDE	This is CS 203:
1	0	0x10	Advanced Compute
0	1	0x8A	r Architecture!
1	1	0x60	This is CS 203:
1	1	0x70	Advanced Compute
0	1	0x10	r Architecture!
0	1	0x11	This is CS 203:

V	D	tag	data
1	1	0x00	This is CS 203:
1	1	0x10	Advanced Compute
1	0	0xA1	r Architecture!
0	1	0x10	This is CS 203:
1	1	0x31	Advanced Compute
1	1	0x45	r Architecture!
0	1	0x41	This is CS 203:
0	1	0x68	Advanced ?



# The A, B, Cs of your cache

# C = ABS

- **C:** Capacity in data arrays
- **A:** Way-Associativity — how many blocks within a set
  - N-way: N blocks in a set, A = N
  - 1 for direct-mapped cache
- **B:** Block Size (Linesize)
  - How many bytes in a block
- **S:** Number of Sets:
  - A set contains blocks sharing the same index
  - 1 for fully associate cache



# Corollary of C = ABS



- number of bits in **block offset** —  $\lg(B)$
- number of bits in **set index**:  $\lg(S)$
- tag bits:  $\text{address\_length} - \lg(S) - \lg(B)$ 
  - $\text{address\_length}$  is N bits for N-bit machines (e.g., 64-bit for 64-bit machines)
- $(\text{address} / \text{block\_size}) \% S = \text{set index}$



# NVIDIA Tegra Orin

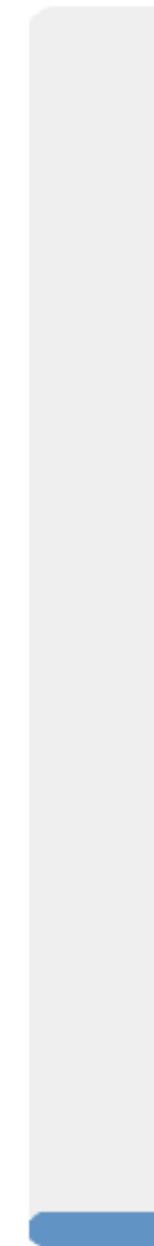
- L1 data (D-L1) cache configuration of NVIDIA Tegra Orin
  - Size 64KB, 4-way set associativity, 64B block
  - Assume 64-bit memory address

Which of the following is correct?

- A. Tag is 49 bits
- B. Index is 8 bits
- C. Offset is 7 bits
- D. The cache has 1024 sets
- E. None of the above

 0

0



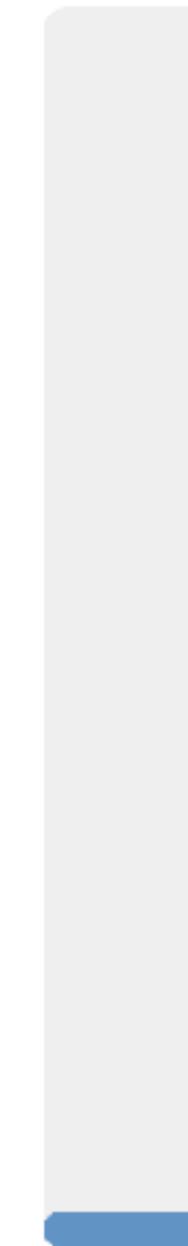
A

0



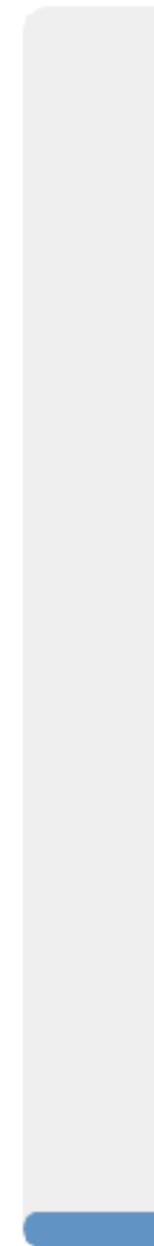
B

0



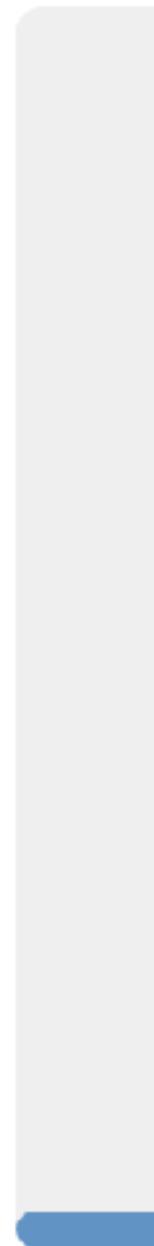
C

0



D

0



E



# NVIDIA Tegra Orin

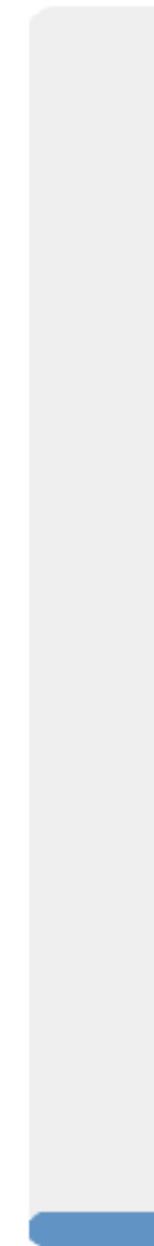
- L1 data (D-L1) cache configuration of NVIDIA Tegra Orin
  - Size 64KB, 4-way set associativity, 64B block
  - Assume 64-bit memory address

Which of the following is correct?

- A. Tag is 49 bits
- B. Index is 8 bits
- C. Offset is 7 bits
- D. The cache has 1024 sets
- E. None of the above

 0

0



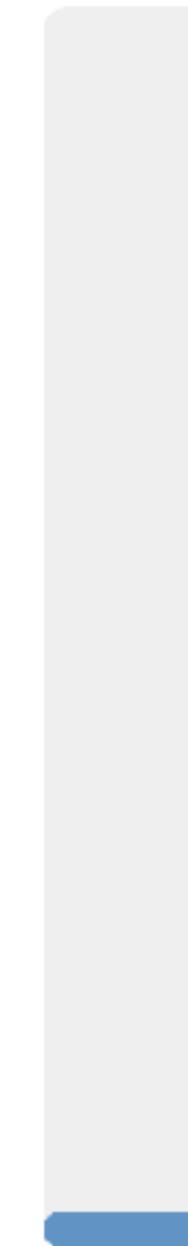
A

0



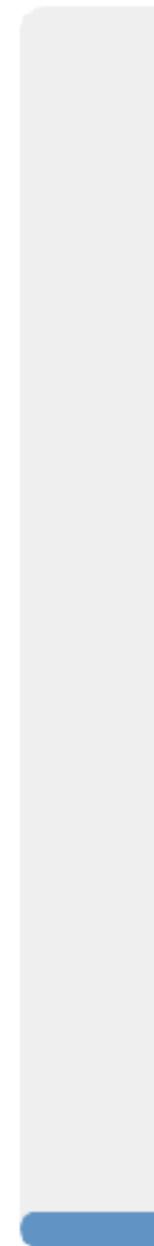
B

0



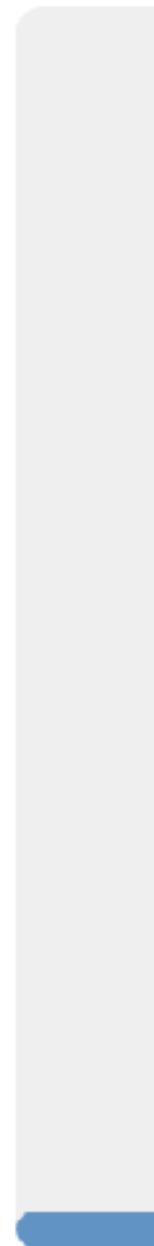
C

0



D

0



E

# NVIDIA Tegra Orin

- L1 data (D-L1) cache configuration of NVIDIA Tegra Orin
  - Size 64KB, 4-way set associativity, 64B block
  - Assume 64-bit memory address

Which of the following is correct?

- A. Tag is 49 bits
- B. Index is 8 bits
- C. Offset is 7 bits
- D. The cache has 1024 sets
- E. None of the above

$$C = A \times B \times S$$

$$64 \times 1024 = 4 \times 64 \times S$$

$$S = 256$$

$$\text{Offset} = \log_2(64) = 6$$

$$\text{Index} = \log_2(256) = 8$$

$$\text{Tag} = 64 - 8 - 6 = 50$$

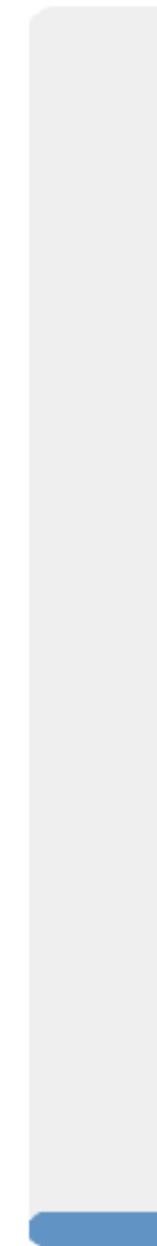


# intel Core i7 14700K

- L1 data (D-L1) cache configuration of Core i7 14700K's P-Core
  - Size 48KB, 12-way set associativity, 64B block
  - Assume 64-bit memory address
  - Which of the following is **NOT** correct?
    - A. Tag is 52 bits
    - B. Index is 6 bits
    - C. Offset is 6 bits
    - D. The cache has 128 sets
    - E. All of the above are correct

 0

0



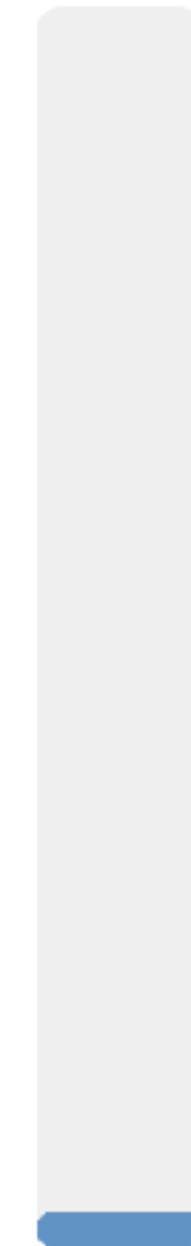
A

0



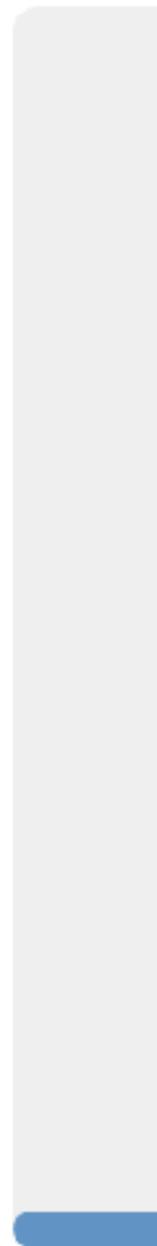
B

0



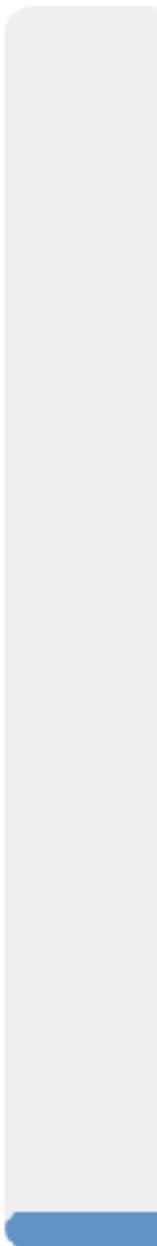
C

0



D

0



E

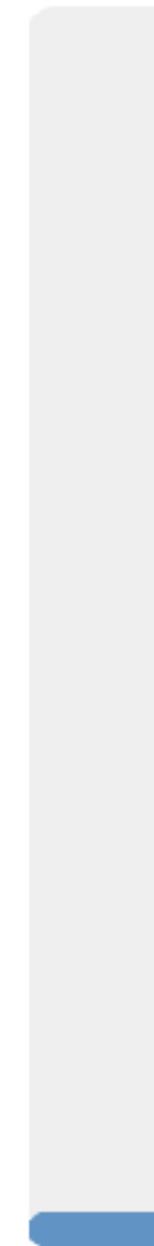


# intel Core i7 14700K

- L1 data (D-L1) cache configuration of Core i7 14700K's P-Core
  - Size 48KB, 12-way set associativity, 64B block
  - Assume 64-bit memory address
  - Which of the following is **NOT** correct?
    - A. Tag is 52 bits
    - B. Index is 6 bits
    - C. Offset is 6 bits
    - D. The cache has 128 sets
    - E. All of the above are correct

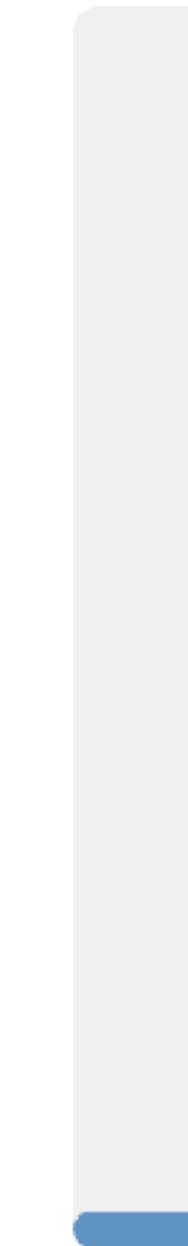
 0

0



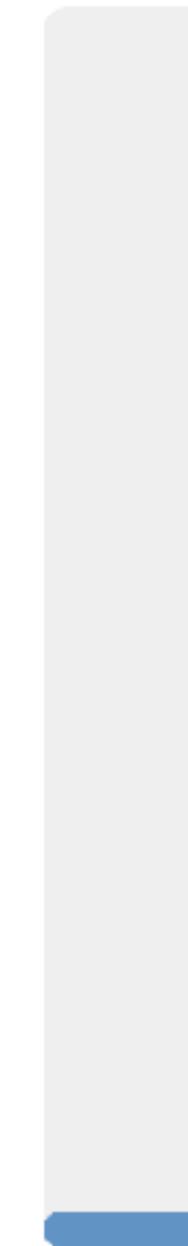
A

0



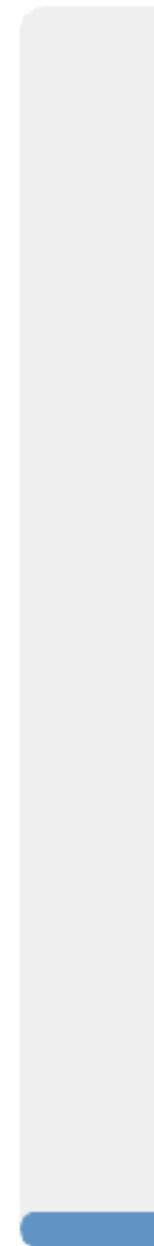
B

0



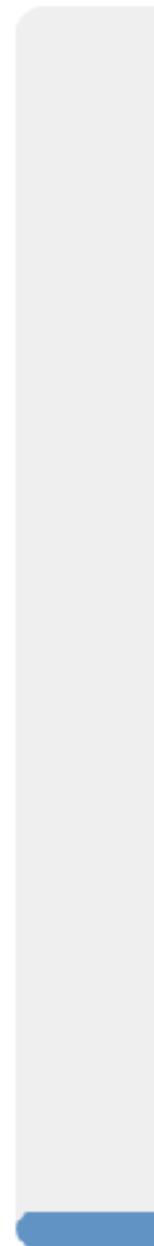
C

0



D

0



E

# intel Core i7 14700K

- L1 data (D-L1) cache configuration of Core i7 14700K's P-Core
  - Size 48KB, 12-way set associativity, 64B block
  - Assume 64-bit memory address
  - Which of the following is **NOT** correct?
    - A. Tag is 52 bits
    - B. Index is 6 bits
    - C. Offset is 6 bits
    - D. The cache has 128 sets
    - E. All of the above are correct

$$C = A \times B \times S$$

$$48 \times 1024 = 12 \times 64 \times S$$

$$S = 64$$

$$\text{Offset} = \log_2(64) = 6$$

$$\text{Index} = \log_2(64) = 6$$

$$\text{Tag} = 64 - 6 - 6 = 52$$

# Take-aways: designing caches

- 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
- Hierarchical caching — data must be presented on the top level (L1) before the processor can use
- Optimizing cache structures
  - Hash block into “sets” to reduce the search time
  - Set-associativity to reduce the “collision” problem
- $C = A B S$ 
  - C: capacity
  - A: Associativity
  - S: Number of sets
  - $\lg(S)$ : Number of bits in set index
  - $\lg(B)$ : Number of bits in block offset

# **Simulate the cache!**

# Simulate a direct-mapped cache

- A direct mapped (1-way) cache with 256 bytes total capacity, a block size of 16 bytes

- # of blocks =  $\frac{256}{16} = 16$
  - $\lg(16) = 4$  : 4 bits are used for the index
  - $\lg(16)$  = 4 : 4 bits are used for the byte offset
  - The tag is  $64 - (4 + 4) = 56$  bits
  - For example: 0x 8 0 0 0

```
: 0x    8    0    0    0    0    0    0    8    0
= 0b1000 0000 0000 0000 0000 0000 1000 0000 0000
          +-----+-----+
          |       |       |
          tag    index  offset
```



Start the presentation to see live content. Still no live content? Install the app or get help at [PollEv.com/app](https://PollEv.com/app)

# Matrix vector revisited

```
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;  
}
```

# Matrix vector revisited

tag index

```

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;
}

```

		Address (Hex)	Address (Binary)	tag	index
	&a[0][0]	0x558FE0A1D330	0b10101011000111111000010100011101001100110000		
	&b[0]	0x558FE0A1DC30	0b10101011000111111000010100011101110000110000		
	&a[0][1]	0x558FE0A1D338	0b10101011000111111000010100011101001100111000		
	&b[1]	0x558FE0A1DC38	0b10101011000111111000010100011101110000110000		
	&a[0][2]	0x558FE0A1D340	0b10101011000111111000010100011101001101000000		
	&b[2]	0x558FE0A1DC40	0b10101011000111111000010100011101110000100000		
	&a[0][3]	0x558FE0A1D348	0b10101011000111111000010100011101001101000000		
	&b[3]	0x558FE0A1DC48	0b1010101100011111100001010001110111000100000		
	&a[0][4]	0x558FE0A1D350	0b10101011000111111000010100011101001101000000		
	&b[4]	0x558FE0A1DC50	0b10101011000111111000010100011101110001010000		
	&a[0][5]	0x558FE0A1D358	0b10101011000111111000010100011101001101000000		
	&b[5]	0x558FE0A1DC58	0b10101011000111111000010100011101110001010000		
	&a[0][6]	0x558FE0A1D360	0b10101011000111111000010100011101001101100000		
	&b[6]	0x558FE0A1DC60	0b10101011000111111000010100011101110001100000		
	&a[0][7]	0x558FE0A1D368	0b10101011000111111000010100011101001101100000		
	&b[7]	0x558FE0A1DC68	0b10101011000111111000010100011101110001100000		
	&a[0][8]	0x558FE0A1D370	0b10101011000111111000010100011101001101100000		
	&b[8]	0x558FE0A1DC70	0b10101011000111111000010100011101110001110000		
	&a[0][9]	0x558FE0A1D378	0b10101011000111111000010100011101001101110000		
	&b[9]	0x558FE0A1DC78	0b101010110001111110000101000111011100011100000		

# Simulate a direct-mapped cache

tag index

	V	D	Tag	Data
0	0	0		
1	0	0		
2	0	0		
3	1	0	0x558FE0A1DC	b[0], b[1]
4	1	0	0x558FE0A1DC	b[2], b[3]
5	0	0		
6	0	0		
7	0	0		
8	0	0		
9	0	0		
10	0	0		
11	0	0		
12	0	0		
13	0	0		
14	0	0		
15	0	0		

This cache doesn't work!!!  
— collisions!

Address (Hex)	
&a[0][0]	0x558FE0A1D330 miss
&b[0]	0x558FE0A1DC30 miss
&a[0][1]	0x558FE0A1D338 miss
&b[1]	0x558FE0A1DC38 miss
&a[0][2]	0x558FE0A1D340 miss
&b[2]	0x558FE0A1DC40 miss
&a[0][3]	0x558FE0A1D348 miss
&b[3]	0x558FE0A1DC48 miss
&a[0][4]	0x558FE0A1D350 miss
&b[4]	0x558FE0A1DC50 miss
&a[0][5]	0x558FE0A1D358 miss
&b[5]	0x558FE0A1DC58 miss
&a[0][6]	0x558FE0A1D360 miss
&b[6]	0x558FE0A1DC60 miss
&a[0][7]	0x558FE0A1D368 miss
&b[7]	0x558FE0A1DC68 miss
&a[0][8]	0x558FE0A1D370 miss
&b[8]	0x558FE0A1DC70 miss
&a[0][9]	0x558FE0A1D378 miss
&b[9]	0x558FE0A1DC78

# Way-associative cache

memory address:

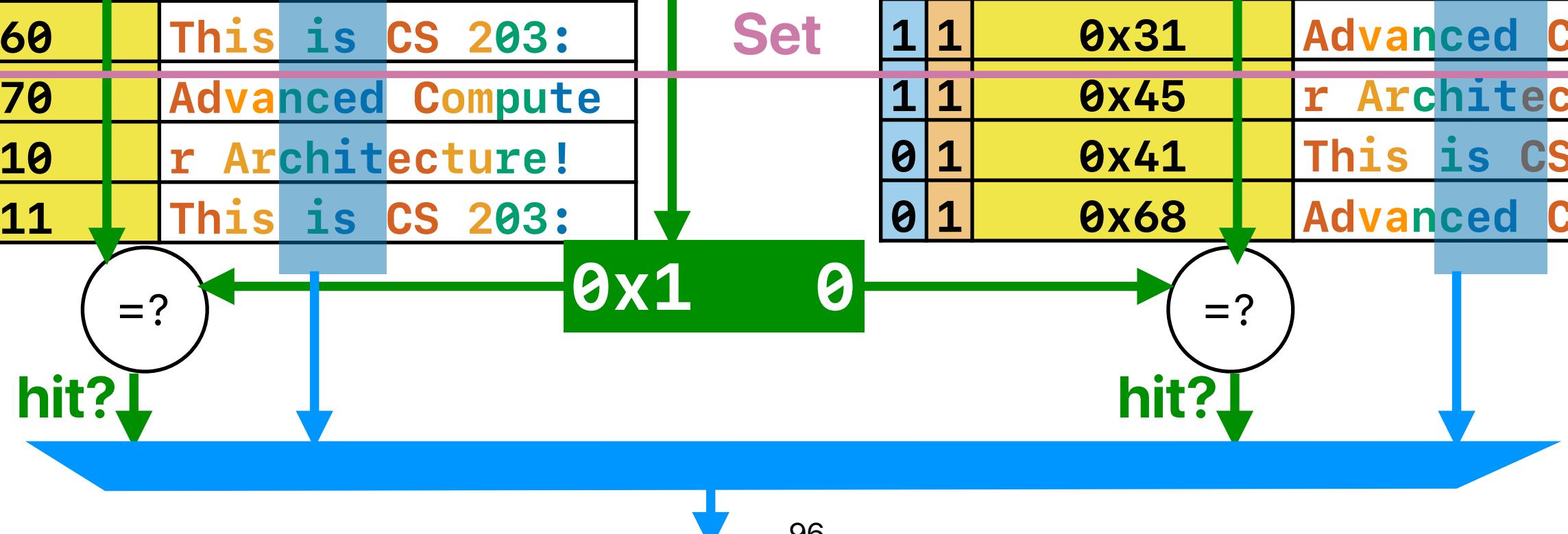
$0x0 \quad 8 \quad 2 \quad 4$   
**set**   **block**  
**tag**   **index** **offset**

memory address:

$0b0000100000100100$

V	D	tag	data
1	1	0x29	r Architecture!
1	1	0xDE	This is CS 203:
1	0	0x10	Advanced Compute
0	1	0x8A	r Architecture!
1	1	0x60	This is CS 203:
1	1	0x70	Advanced Compute
0	1	0x10	r Architecture!
0	1	0x11	This is CS 203:

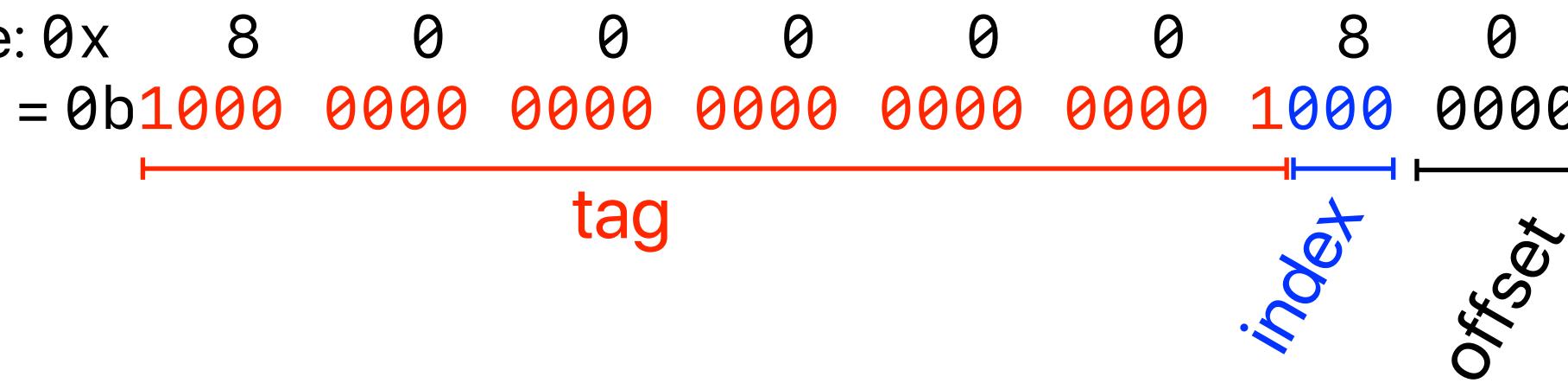
V	D	tag	data
1	1	0x00	This is CS 203:
1	1	0x10	Advanced Compute
1	0	0xA1	r Architecture!
0	1	0x10	This is CS 203:
1	1	0x31	Advanced Compute
1	1	0x45	r Architecture!
0	1	0x41	This is CS 203:
0	1	0x68	Advanced Compute



# Now, 2-way, same-sized cache

- A 2-way cache with 256 bytes total capacity, a block size of 16 bytes

- # of blocks =  $\frac{256}{16} = 16$
- # of sets =  $\frac{16}{2} = 8$  (2-way: 2 blocks in a set)
- $\lg(8) = 3$  : 3 bits are used for the index
- $\lg(16) = 4$  : 4 bits are used for the byte offset
- The tag is  $64 - (4 + 4) = 56$  bits
- For example: 0x 8 0 0 0 0 0 8 0





Start the presentation to see live content. Still no live content? Install the app or get help at [PollEv.com/app](https://PollEv.com/app)

# Matrix vector revisited

tag index

```

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;
}

```

		Address (Hex)	Address (Binary)	tag	index
	&a[0][0]	0x558FE0A1D330	0b1010101100011111100001010001110100110 0110000		
	&b[0]	0x558FE0A1DC30	0b1010101100011111100001010001110111000 0110000		
	&a[0][1]	0x558FE0A1D338	0b1010101100011111100001010001110100110 0111000		
	&b[1]	0x558FE0A1DC38	0b1010101100011111100001010001110111000 0111000		
	&a[0][2]	0x558FE0A1D340	0b1010101100011111100001010001110100110 1000000		
	&b[2]	0x558FE0A1DC40	0b1010101100011111100001010001110111000 1000000		
	&a[0][3]	0x558FE0A1D348	0b1010101100011111100001010001110100110 1001000		
	&b[3]	0x558FE0A1DC48	0b1010101100011111100001010001110111000 1001000		
	&a[0][4]	0x558FE0A1D350	0b1010101100011111100001010001110100110 1010000		
	&b[4]	0x558FE0A1DC50	0b1010101100011111100001010001110111000 1010000		
	&a[0][5]	0x558FE0A1D358	0b1010101100011111100001010001110100110 1011000		
	&b[5]	0x558FE0A1DC58	0b1010101100011111100001010001110111000 1011000		
	&a[0][6]	0x558FE0A1D360	0b1010101100011111100001010001110100110 1100000		
	&b[6]	0x558FE0A1DC60	0b1010101100011111100001010001110111000 1100000		
	&a[0][7]	0x558FE0A1D368	0b1010101100011111100001010001110100110 1101000		
	&b[7]	0x558FE0A1DC68	0b1010101100011111100001010001110111000 1101000		
	&a[0][8]	0x558FE0A1D370	0b1010101100011111100001010001110100110 1110000		
	&b[8]	0x558FE0A1DC70	0b1010101100011111100001010001110111000 1110000		
	&a[0][9]	0x558FE0A1D378	0b1010101100011111100001010001110100110 1111000		
	&b[9]	0x558FE0A1DC78	0b1010101100011111100001010001110111000 1111000		

# Simulate a 2-way cache

V	D	Tag	Data	V	D	Tag	Data
0	0			0	0		
0	0			0	0		
0	0			0	0		
1	0	0xAB1FC143A6	a[0][0], a[0][1]	1	0	0xAB1FC143B8	b[0], b[1]
1	0	0xAB1FC143A6	a[0][2], a[0][3]	1	0	0xAB1FC143B8	b[2], b[3]
0	0			0	0		
0	0			0	0		
0	0			0	0		

Address (Hex)	Tag	Index	
&a[0][0]	0xAB1FC143A6	0x3	miss
&b[0]	0xAB1FC143B8	0x3	miss
&a[0][1]	0xAB1FC143A6	0x3	hit
&b[1]	0xAB1FC143B8	0x3	hit
&a[0][2]	0xAB1FC143A6	0x4	miss
&b[2]	0xAB1FC143B8	0x4	miss
&a[0][3]	0xAB1FC143A6	0x4	hit
&b[3]	0xAB1FC143B8	0x4	hit
&a[0][4]	0xAB1FC143A6	0x5	miss
&b[4]	0xAB1FC143B8	0x5	miss
&a[0][5]	0xAB1FC143A6	0x5	hit
&b[5]	0xAB1FC143B8	0x5	hit
&a[0][6]	0xAB1FC143A6	0x6	miss
&b[6]	0xAB1FC143B8	0x6	miss
&a[0][7]	0xAB1FC143A6	0x6	hit
&b[7]	0xAB1FC143B8	0x6	hit
&a[0][8]	0xAB1FC143A6	0x7	miss
&b[8]	0xAB1FC143B8	0x7	miss
&a[0][9]	0xAB1FC143A6	0x7	hit
&b[9]	0xAB1FC143B8	0x7	hit



# NVIDIA Tegra Orin

- D-L1 Cache configuration of NVIDIA Tegra Orin
  - Size 64KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

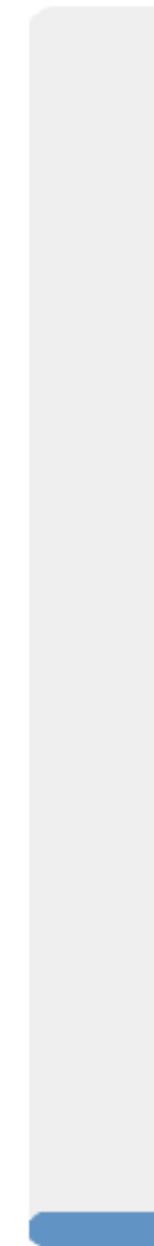
What's the data cache miss rate for this code?

- A. 12.5%
- B. 56.25%
- C. 66.67%
- D. 68.75%
- E. 100%



 0

0



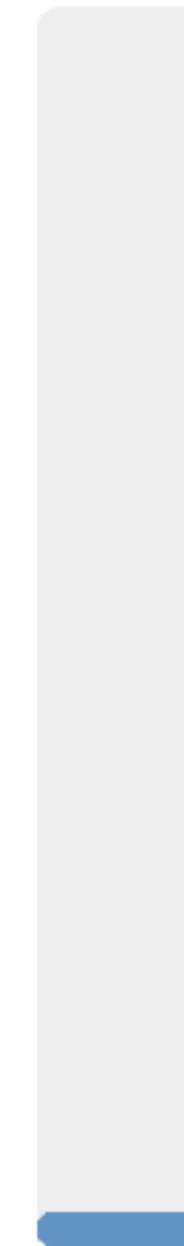
A

0



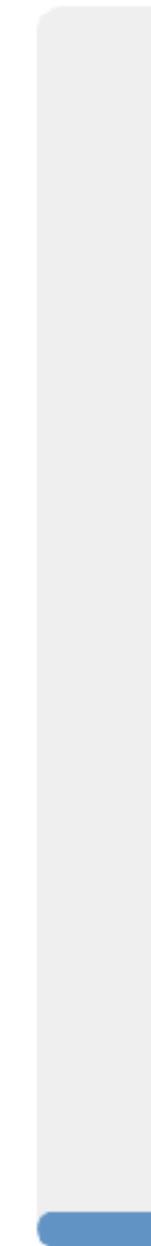
B

0



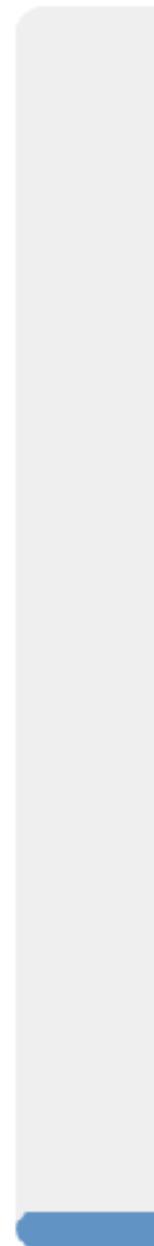
C

0



D

0



E



# NVIDIA Tegra Orin

- D-L1 Cache configuration of NVIDIA Tegra Orin
  - Size 64KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

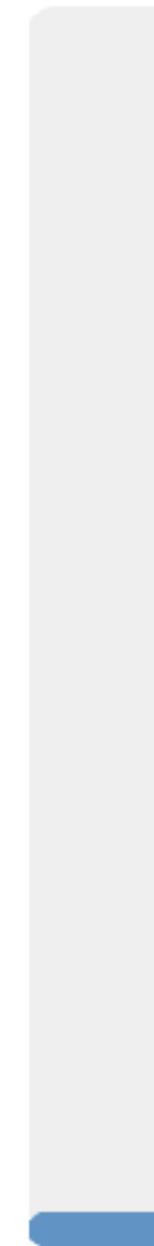
```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

What's the data cache miss rate for this code?

- A. 12.5%
- B. 56.25%
- C. 66.67%
- D. 68.75%
- E. 100%

 0

0



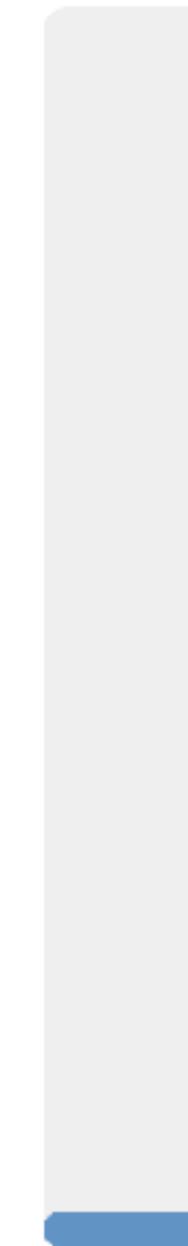
A

0



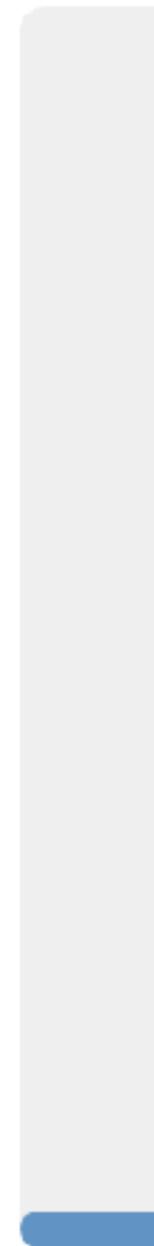
B

0



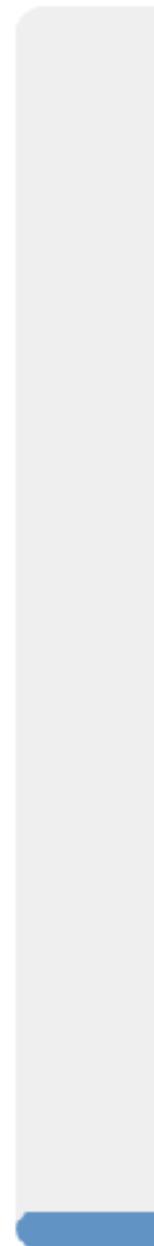
C

0



D

0



E

# NVIDIA Tegra Orin

## 100% miss rate!

- Size 64KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

$C = \text{ABS}$   
 $32\text{KB} = 4 * 64 * S$   
 $S = 256$   
 $\text{offset} = \lg(64) = 6 \text{ bits}$   
 $\text{index} = \lg(256) = 8 \text{ bits}$   
 $\text{tag} = \text{the rest bits}$

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[0]	0x10000	0b000100000000000000000000000000	0x4	0x0	Miss	
b[0]	0x20000	0b001000000000000000000000000000	0x8	0x0	Miss	
c[0]	0x30000	0b001100000000000000000000000000	0xC	0x0	Miss	
d[0]	0x40000	0b010000000000000000000000000000	0x10	0x0	Miss	
e[0]	0x50000	0b010100000000000000000000000000	0x14	0x0	Miss	a[0-7]
a[1]	0x10008	0b000100000000000000100000000000	0x4	0x0	Miss	b[0-7]
b[1]	0x20008	0b001000000000000000100000000000	0x8	0x0	Miss	c[0-7]
c[1]	0x30008	0b001100000000000000100000000000	0xC	0x0	Miss	d[0-7]
d[1]	0x40008	0b010000000000000000100000000000	0x10	0x0	Miss	e[0-7]
e[1]	0x50008	0b010100000000000000100000000000	0x14	0x0	Miss	a[0-7]

**During examines/interviews, why do you (occasionally) don't know the answer?**

# **Estimating code performance on caches (cont.)**

# NVIDIA Tegra Orin

## 100% miss rate!

- Size 64KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

$C = ABS$   
 $64KB = 4 * 64 * S$   
 $S = 256$   
 $offset = \lg(64) = 6 \text{ bits}$   
 $index = \lg(256) = 8 \text{ bits}$   
 $tag = \text{the rest bits}$

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[0]	0x10000	0b000100000000000000000000000000	0x4	0x0	Miss	
b[0]	0x20000	0b001000000000000000000000000000	0x8	0x0	Miss	
c[0]	0x30000	0b001100000000000000000000000000	0xC	0x0	Miss	
d[0]	0x40000	0b010000000000000000000000000000	0x10	0x0	Miss	
e[0]	0x50000	0b010100000000000000000000000000	0x14	0x0	Miss	a[0-7]
a[1]	0x10008	0b000100000000000000100000000000	0x4	0x0	Miss	b[0-7]
b[1]	0x20008	0b001000000000000000100000000000	0x8	0x0	Miss	c[0-7]
c[1]	0x30008	0b001100000000000000100000000000	0xC	0x0	Miss	d[0-7]
d[1]	0x40008	0b010000000000000000100000000000	0x10	0x0	Miss	e[0-7]
e[1]	0x50008	0b010100000000000000100000000000	0x14	0x0	Miss	a[0-7]

# NVIDIA Tegra Orin

- D-L1 Cache configuration of NVIDIA Tegra Orin
  - Size 64KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

What's the data cache miss rate for this code?

- A. 12.5%
- B. 56.25%
- C. 66.67%
- D. 68.75%
- E. 100%



Start the presentation to see live content. Still no live content? Install the app or get help at [PollEv.com/app](https://PollEv.com/app)



# intel Core i7 14700K

- D-L1 Cache configuration of intel Core i7 14700K's P-Core
  - Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

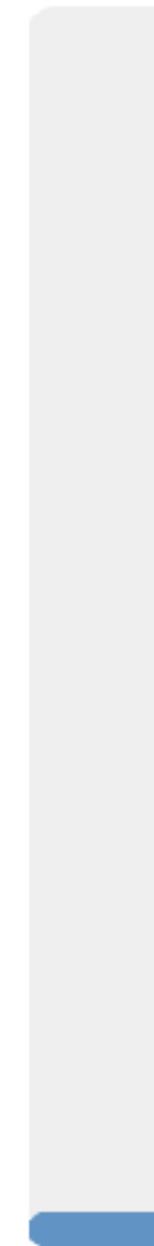
```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

What's the data cache miss rate for this code?

- A. 12.5%
- B. 56.25%
- C. 66.67%
- D. 68.75%
- E. 100%

 0

0



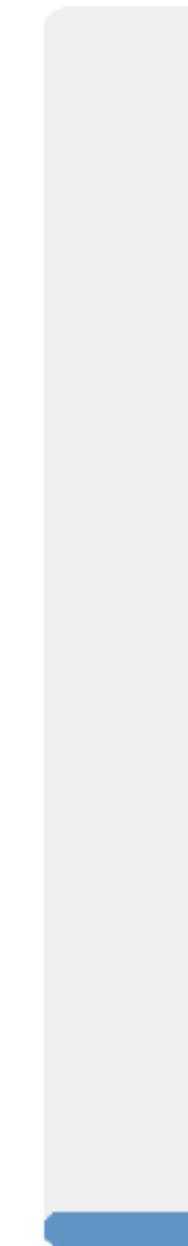
A

0



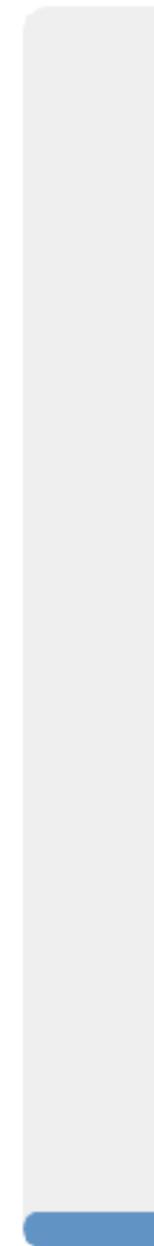
B

0



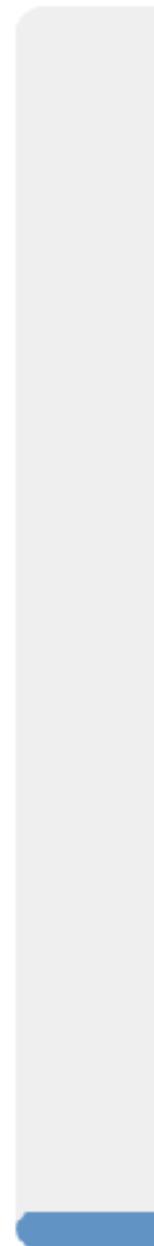
C

0



D

0



E



# intel Core i7 14700K

- D-L1 Cache configuration of intel Core i7 14700K's P-Core
  - Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

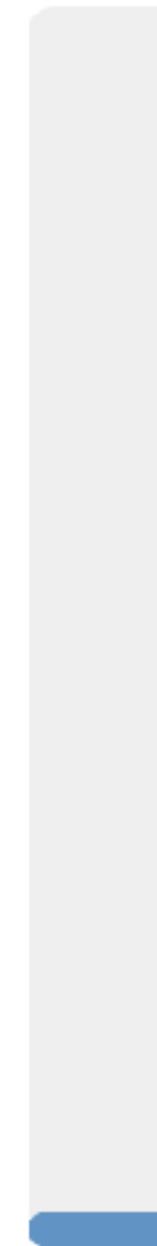
```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

What's the data cache miss rate for this code?

- A. 12.5%
- B. 56.25%
- C. 66.67%
- D. 68.75%
- E. 100%

 0

0



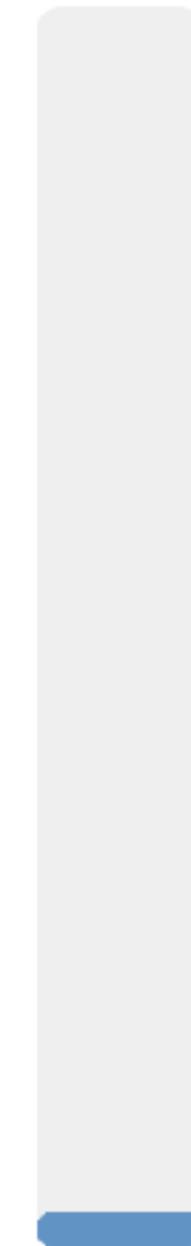
A

0



B

0



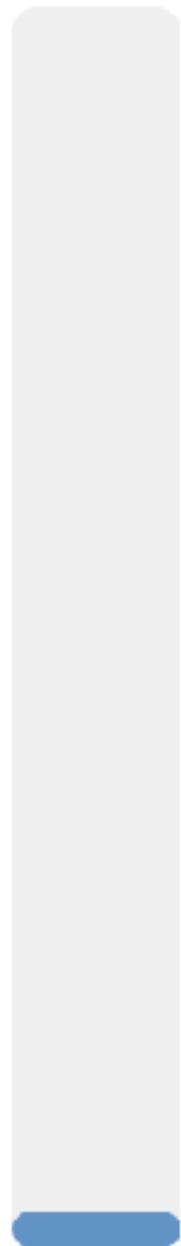
C

0



D

0



E

# intel Core i7 14700K

- Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

$C = ABS$   
 $48KB = 12 * 64 * S$   
 $S = 64$   
 $offset = \lg(64) = 6 \text{ bits}$   
 $index = \lg(64) = 6 \text{ bits}$   
 $tag = \text{the rest bits}$

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[0]	0x10000	0b000100000000000000000000000000	0x10	0x0	Miss	
b[0]	0x20000	0b001000000000000000000000000000	0x20	0x0	Miss	
c[0]	0x30000	0b001100000000000000000000000000	0x30	0x0	Miss	
d[0]	0x40000	0b010000000000000000000000000000	0x40	0x0	Miss	
e[0]	0x50000	0b010100000000000000000000000000	0x50	0x0	Miss	
a[1]	0x10008	0b000100000000000000100000000000	0x10	0x0	Hit	
b[1]	0x20008	0b001000000000000000100000000000	0x20	0x0	Hit	
c[1]	0x30008	0b001100000000000000100000000000	0x30	0x0	Hit	
d[1]	0x40008	0b010000000000000000100000000000	0x40	0x0	Hit	
e[1]	0x50008	0b010100000000000000100000000000	0x50	0x0	Hit	

# intel Core i7 14700K (cont.)

- Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

$C = ABS$   
 $48KB = 12 * 64 * S$   
 $S = 64$   
 $offset = \lg(64) = 6 \text{ bits}$   
 $index = \lg(64) = 6 \text{ bits}$   
 $tag = \text{the rest bits}$

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[7]	0x10038	0b00010000000000111000	0x10	0x0	Hit	
b[7]	0x20038	0b00100000000000111000	0x20	0x0	Hit	
c[7]	0x30038	0b00110000000000111000	0x30	0x0	Hit	
d[7]	0x40038	0b01000000000000111000	0x40	0x0	Hit	
e[7]	0x50038	0b01010000000000111000	0x50	0x0	Hit	
a[8]	0x10040	0b00010000000001000000	0x10	0x1	Miss	
b[8]	0x20040	0b00100000000001000000	0x20	0x1	Miss	
c[8]	0x30040	0b00110000000001000000	0x30	0x1	Miss	
d[8]	0x40040	0b01000000000001000000	0x40	0x1	Miss	
e[8]	0x50040	0b01010000000001000000	0x50	0x1	Miss	$5 \times \frac{512}{8} = 1$
a[9]	0x10048	0b00010000000001001000	0x10	0x1	Hit	$\frac{5 \times 512}{8} = 12.5\%$
b[9]	0x20048	0b00100000000001001000	0x20	0x1	Hit	
c[9]	0x30048	0b00110000000001001000	0x30	0x1	Hit	
d[9]	0x40048	0b01000000000001001000	0x40	0x1	Hit	

Miss when the array index is a multiply of 8!



Start the presentation to see live content. Still no live content? Install the app or get help at [PollEv.com/app](https://PollEv.com/app)

# **Taxonomy/reasons of cache misses**

# 3Cs of misses

- Compulsory miss
  - Cold start miss. First-time access to a block
- Capacity miss
  - The working set size of an application is bigger than cache size
- Conflict miss
  - Required data replaced by block(s) mapping to the same set
  - Similar collision in hash



# NVIDIA Tegra Orin

- D-L1 Cache configuration of NVIDIA Tegra Orin
  - Size 64KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

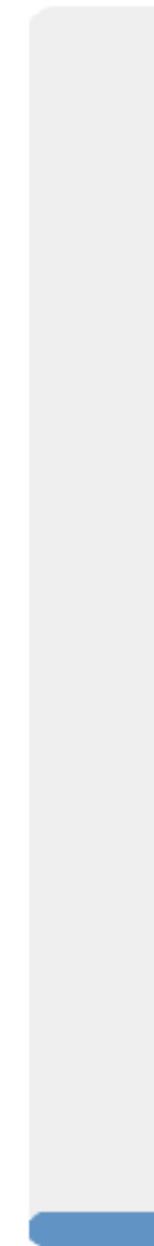
```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

What's the major type of cache misses when running the above code snippet?

- A. Compulsory misses
- B. Conflict misses
- C. Capacity misses
- D. Capacity and conflict misses
- E. They are all equally critical

 0

0



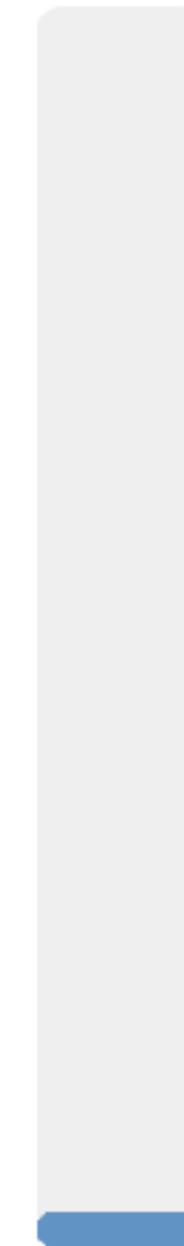
A

0



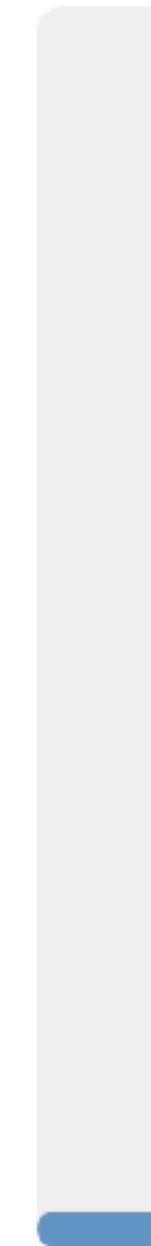
B

0



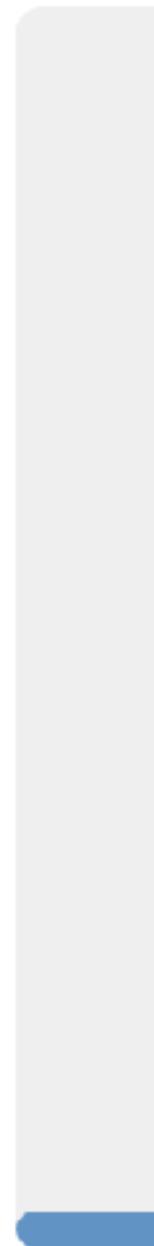
C

0



D

0



E



# NVIDIA Tegra Orin

- D-L1 Cache configuration of NVIDIA Tegra Orin
  - Size 64KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

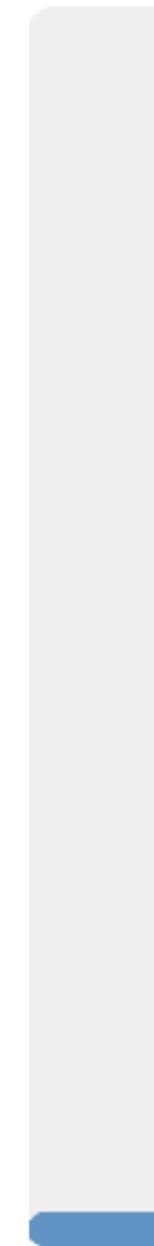
```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

What's the major type of cache misses when running the above code snippet?

- A. Compulsory misses
- B. Conflict misses
- C. Capacity misses
- D. Capacity and conflict misses
- E. They are all equally critical

 0

0



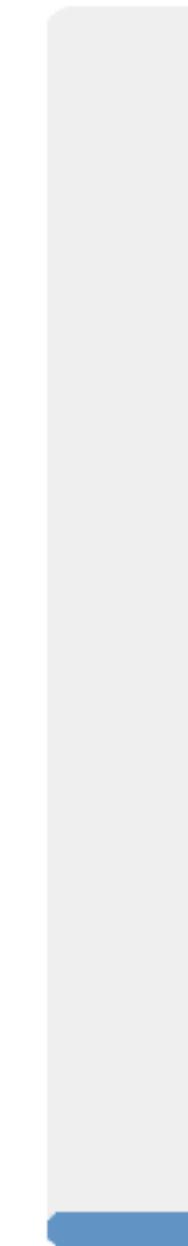
A

0



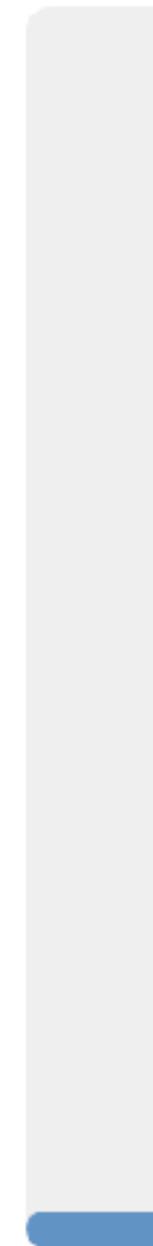
B

0



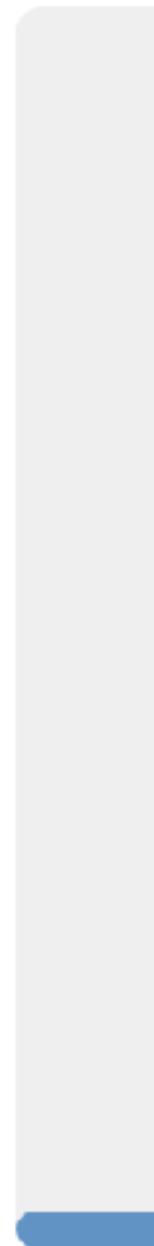
C

0



D

0



E

# NVIDIA Tegra Orin

100% miss rate!

- Size 64KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
    tag index offset
```

$C = ABS$   
 $64KB = 4 * 64 * S$   
 $S = 256$   
 $offset = \lg(64) = 6 \text{ bits}$   
 $index = \lg(256) = 8 \text{ bits}$   
 $tag = \text{the rest bits}$

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[0]	0x10000	0b000100000000000000000000000000	0x8	0x0	Compulsory Miss	
b[0]	0x20000	0b001000000000000000000000000000	0x10	0x0	Compulsory Miss	
c[0]	0x30000	0b001100000000000000000000000000	0x18	0x0	Compulsory Miss	
d[0]	0x40000	0b010000000000000000000000000000	0x20	0x0	Compulsory Miss	
e[0]	0x50000	0b010100000000000000000000000000	0x28	0x0	Compulsory Miss	a[0-7]
a[1]	0x10008	0b000100000000000000100000000000	0x8	0x0	Conflict Miss	b[0-7]
b[1]	0x20008	0b001000000000000000000000001000	0x10	0x0	Conflict Miss	c[0-7]
c[1]	0x30008	0b001100000000000000000000001000	0x18	0x0	Conflict Miss	d[0-7]
d[1]	0x40008	0b010000000000000000000000001000	0x20	0x0	Conflict Miss	e[0-7]
e[1]	0x50008	0b010100000000000000000000001000	0x28	0x0	Conflict Miss	a[0-7]

# NVIDIA Tegra Orin (cont.)

- Size 64KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

$C = \text{ABS}$   
 $64\text{KB} = 4 * 64 * S$   
 $S = 256$   
 $\text{offset} = \lg(64) = 6 \text{ bits}$   
 $\text{index} = \lg(256) = 8 \text{ bits}$   
 $\text{tag} = \text{the rest bits}$

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[7]	0x10038	0b000100000000000111000	0x8	0x0	Conflict Miss	
b[7]	0x20038	0b001000000000000111000	0x10	0x0	Conflict Miss	
c[7]	0x30038	0b001100000000000111000	0x18	0x0	Conflict Miss	
d[7]	0x40038	0b010000000000000111000	0x20	0x0	Conflict Miss	
e[7]	0x50038	0b010100000000000111000	0x28	0x0	Conflict Miss	a[0-7]
a[8]	0x10040	0b0001000000001000000	0x8	0x1	Compulsory Miss	
b[8]	0x20040	0b0010000000001000000	0x10	0x1	Compulsory Miss	
c[8]	0x30040	0b0011000000001000000	0x18	0x1	Compulsory Miss	
d[8]	0x40040	0b0100000000001000000	0x20	0x1	Compulsory Miss	
e[8]	0x50040	0b0101000000001000000	0x28	0x1	Compulsory Miss	a[8-15]

100% miss rate!

# NVIDIA Tegra Orin

- D-L1 Cache configuration of NVIDIA Tegra Orin
  - Size 64KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

What's the major type of cache misses when running the above code snippet?

- A. Compulsory misses
- B. Conflict misses
- C. Capacity misses
- D. Capacity and conflict misses
- E. They are all equally critical



# intel Core i7 14700K (cont.)

- D-L1 Cache configuration of intel Core i7 14700K's P-core
  - Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

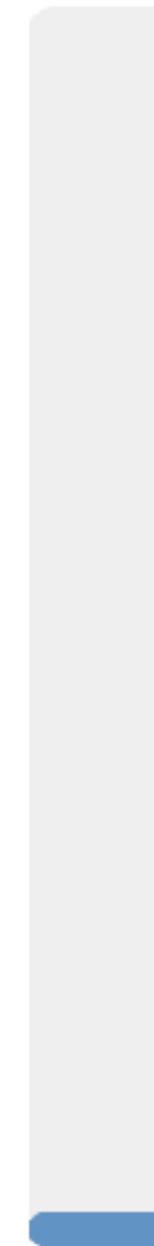
```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

What's the major type of cache misses when running the above code snippet?

- A. Compulsory misses
- B. Conflict misses
- C. Capacity misses
- D. Capacity and conflict misses
- E. They are all equally critical

 0

0



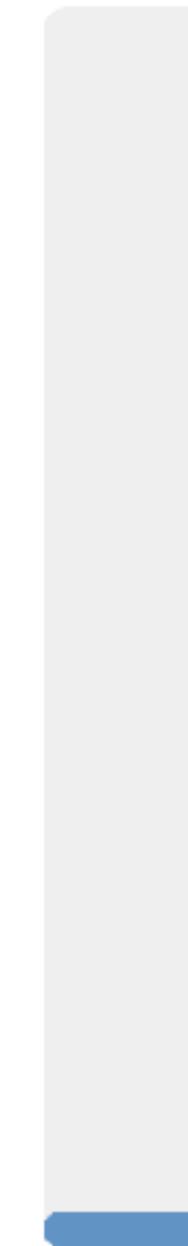
A

0



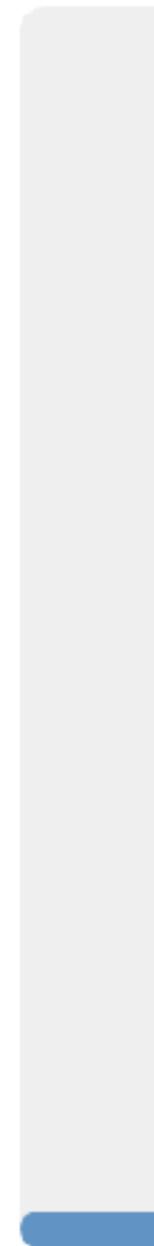
B

0



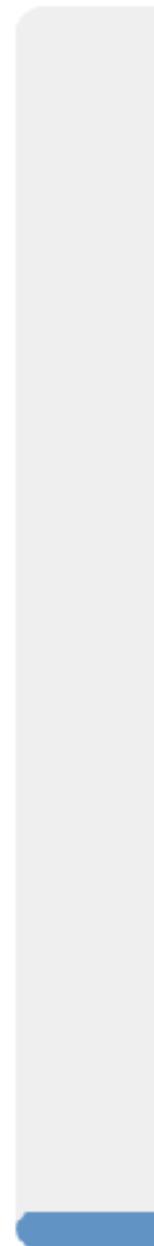
C

0



D

0



E



# intel Core i7 14700K

- D-L1 Cache configuration of intel Core i7 14700K's P-core
  - Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

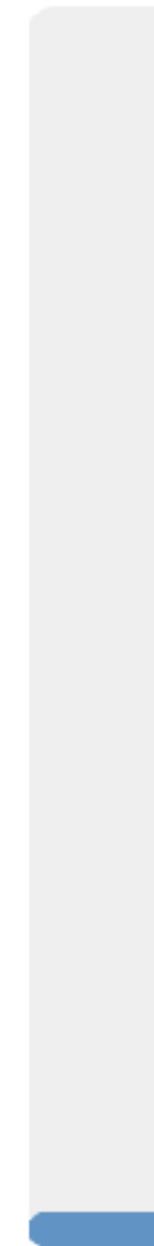
```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

What's the major type of cache misses when running the above code snippet?

- A. Compulsory misses
- B. Conflict misses
- C. Capacity misses
- D. Capacity and conflict misses
- E. They are all equally critical

 0

0



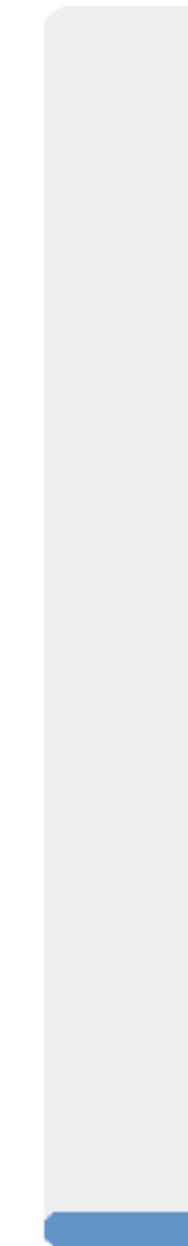
A

0



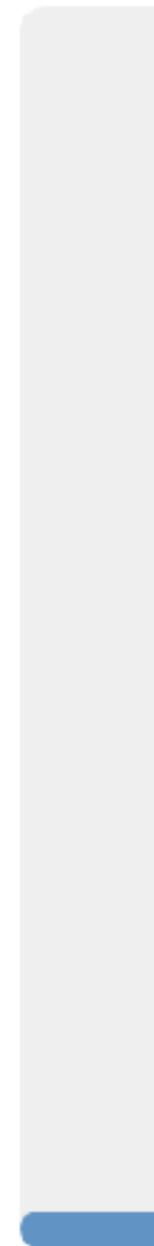
B

0



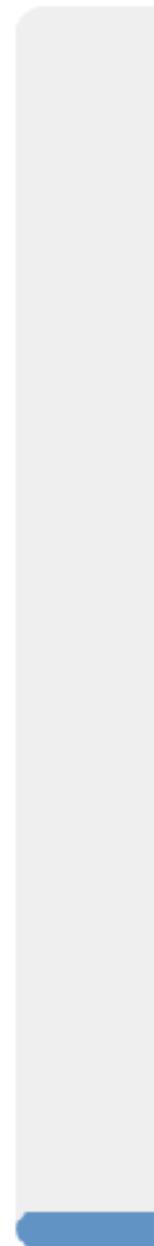
C

0



D

0



E

# intel Core i7 14700K

- Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

$C = \text{ABS}$   
 $48\text{KB} = 12 * 64 * S$   
 $S = 64$   
 $\text{offset} = \lg(64) = 6 \text{ bits}$   
 $\text{index} = \lg(64) = 6 \text{ bits}$   
 $\text{tag} = \text{the rest bits}$

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[0]	0x10000	0b000100000000000000000000000000	0x10	0x0	Compulsory	Miss
b[0]	0x20000	0b001000000000000000000000000000	0x20	0x0	Compulsory	Miss
c[0]	0x30000	0b001100000000000000000000000000	0x30	0x0	Compulsory	Miss
d[0]	0x40000	0b010000000000000000000000000000	0x40	0x0	Compulsory	Miss
e[0]	0x50000	0b010100000000000000000000000000	0x50	0x0	Compulsory	Miss
a[1]	0x10008	0b000100000000000000100000000000	0x10	0x0	Hit	
b[1]	0x20008	0b001000000000000000000000100000	0x20	0x0	Hit	
c[1]	0x30008	0b001100000000000000000000100000	0x30	0x0	Hit	
d[1]	0x40008	0b010000000000000000000000100000	0x40	0x0	Hit	
e[1]	0x50008	0b010100000000000000000000100000	0x50	0x0	Hit	

# intel Core i7 14700K (cont.)

- Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

$C = ABS$   
 $48KB = 12 * 64 * S$   
 $S = 64$   
 $offset = \lg(64) = 6 \text{ bits}$   
 $index = \lg(64) = 6 \text{ bits}$   
 $tag = \text{the rest bits}$

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[7]	0x10038	0b00010000000000111000	0x10	0x0	Hit	
b[7]	0x20038	0b00100000000000111000	0x20	0x0	Hit	
c[7]	0x30038	0b00110000000000111000	0x30	0x0	Hit	
d[7]	0x40038	0b01000000000000111000	0x40	0x0	Hit	
e[7]	0x50038	0b01010000000000111000	0x50	0x0	Hit	
a[8]	0x10040	0b00010000000010000000	0x10	0x1	Compulsory Miss	
b[8]	0x20040	0b00100000000010000000	0x20	0x1	Compulsory Miss	
c[8]	0x30040	0b00110000000010000000	0x30	0x1	Compulsory Miss	
d[8]	0x40040	0b01000000000010000000	0x40	0x1	Compulsory Miss	
e[8]	0x50040	0b01010000000010000000	0x50	0x1	Compulsory Miss	
a[9]	0x10048	0b0001000000001001000	0x10	0x1	Hit	
b[9]	0x20048	0b0010000000001001000	0x20	0x1	Hit	
c[9]	0x30048	0b0011000000001001000	0x30	0x1	Hit	
d[9]	0x40048	0b0100000000001001000	0x40	0x1	Hit	

# intel Core i7 14700K

- D-L1 Cache configuration of intel Core i7 14700K's P-core
  - Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

What's the major type of cache misses when running the above code snippet?

- A. Compulsory misses
- B. Conflict misses
- C. Capacity misses
- D. Capacity and conflict misses
- E. They are all equally critical

# Take-aways: cache misses and their remedies

- Our code behaves differently on different cache configurations
- Cache misses
  - Compulsory misses — the miss due to first time access of a block
  - Conflict misses — the miss due to insufficient blocks of the target set **(associativity)**
  - Capacity misses — the miss due to the working set size surpasses the **capacity**

# **A, B, C, S and cache misses**



# 3Cs and A, B, C

- Regarding 3Cs: compulsory, conflict and capacity misses and  
A, B, C: associativity, block size, capacity

How many of the following are correct?

- ① Without changing B & C, increasing A can reduce conflict misses but make each cache hit slower
- ② Without changing A & C, increasing B can reduce compulsory misses but potentially lead to more conflict misses
- ③ Without changing A & C, increasing B will make each cache miss slower
- ④ Without changing A & B, increasing C can reduce capacity misses but make each cache hit slower

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



## 3C & Misses

0





# 3Cs and A, B, C

- Regarding 3Cs: compulsory, conflict and capacity misses and  
A, B, C: associativity, block size, capacity

How many of the following are correct?

- ① Without changing B & C, increasing A can reduce conflict misses but make each cache hit slower
- ② Without changing A & C, increasing B can reduce compulsory misses but potentially lead to more conflict misses
- ③ Without changing A & C, increasing B will make each cache miss slower
- ④ Without changing A & B, increasing C can reduce capacity misses but make each cache hit slower

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



## 3C & Misses – group

0

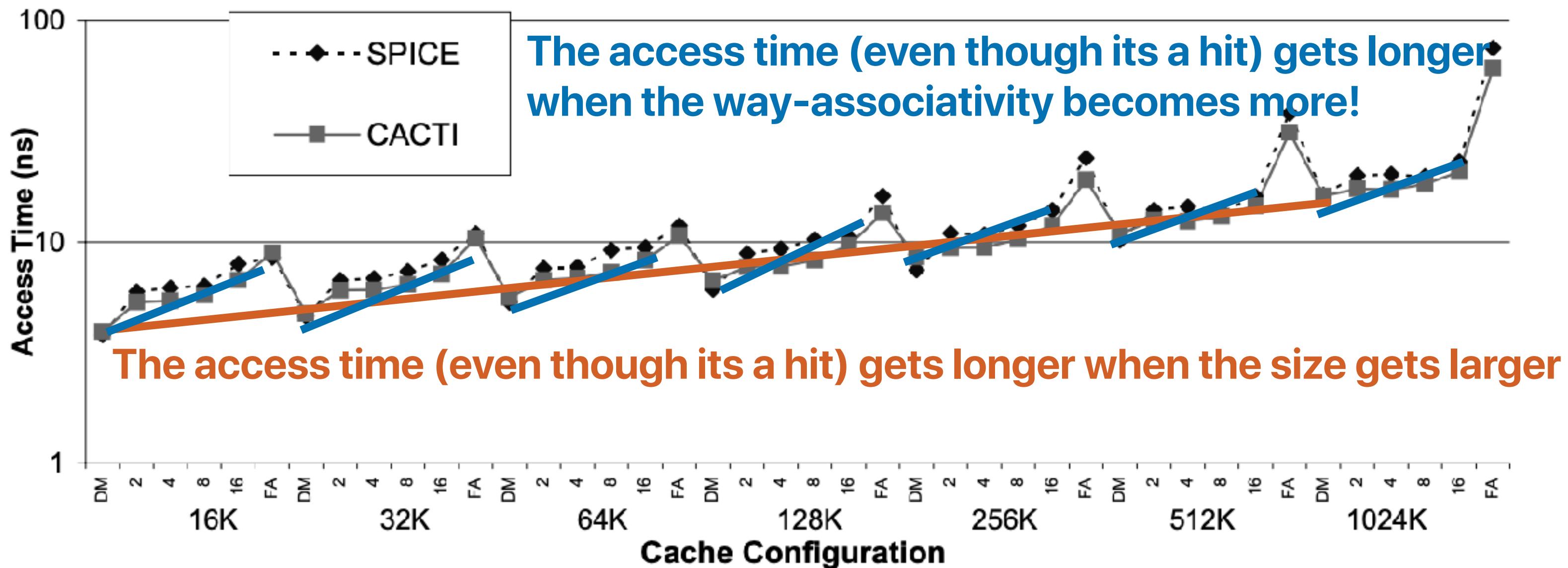


# 3Cs and A, B, C

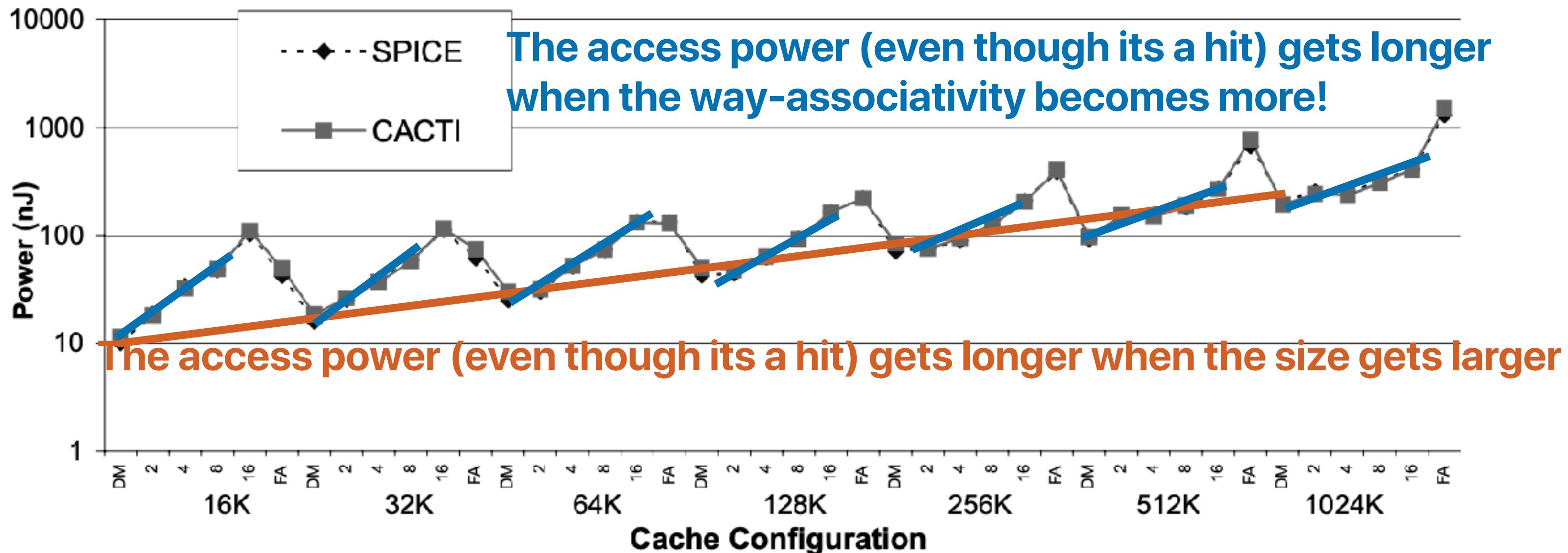
- Regarding 3Cs: compulsory, conflict and capacity misses and A, B, C: associativity, block size, capacity  
How many of the following are correct?
  - ① Without changing B & C, increasing A can reduce conflict misses but make each cache hit slower
  - ② Without changing A & C, increasing B can reduce compulsory misses but potentially lead to more conflict misses
  - ③ Without changing A & C, increasing B will make each cache miss slower
  - ④ Without changing A & B, increasing C can reduce capacity misses but make each cache hit slower

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

# Cache configurations and accessing time



# Cache configurations and accessing power



# 3Cs and A, B, C

- Regarding 3Cs: compulsory, conflict and capacity misses and A, B, C: associativity, block size, capacity

How many of the following are correct?

- ① Without changing B & C, increasing A can reduce conflict misses but make each cache hit slower
- ② Without changing A & C, increasing B can reduce compulsory misses but potentially lead to more conflict misses
- ③ Without changing A & C, increasing B will make each cache miss slower
- ④ Without changing A & B, increasing C can reduce capacity misses but make each cache hit slower

A. 0

You reduce the number of sets now

B. 1

C. 2

D. 3

E. 4

You need to compare more tags

Increases hit time because your data array is larger (longer time to fully charge your bit-lines)

You bring more into the cache when a miss occurs

You need to fetch more data for each miss

# Take-aways: cache misses and their remedies

- Our code behaves differently on different cache configurations
- Cache misses
  - Compulsory misses — the miss due to first time access of a block
  - Conflict misses — the miss due to insufficient blocks of the target set **(associativity)**
  - Capacity misses — the miss due to the working set size surpasses the **capacity**
- There is no optimal cache configurations — trade-offs are everywhere
  - Increasing C — (+): capacity misses; (-): cost, access time, power
  - Increasing A — (+): conflict misses; (-): access time, power
  - Increasing B — (+): compulsory misses; (-): miss penalty

**How can we improve cache  
performance without changing  
ABCs?**

# Recap: NVIDIA Tegra X1

- D-L1 Cache configuration of NVIDIA Tegra X1
  - Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

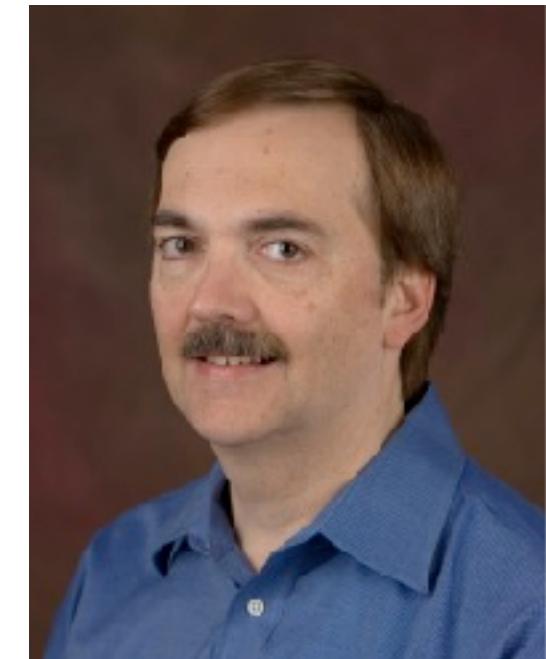
```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

What's the data cache miss rate for this code?

- A. 12.5%
- B. 56.25%
- C. 66.67%
- D. 68.75%
- E. 100%

# **Improving Direct-Mapped Cache Performance by the Addition of a Small Fully- Associative Cache and Prefetch Buffers**

**Norman P. Jouppi**





## Which of the following schemes can help NVIDIA Orin?

- How many of the following schemes mentioned in “improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers” would help NVIDIA’s Tegra for the code in the previous slide?

- ① Missing cache
  - ② Victim cache
  - ③ Prefetch
  - ④ Stream buffer
- A. 0
- B. 1
- C. 2
- D. 3
- E. 4

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 8192; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```



# Small Buffers

0





## Which of the following schemes can help NVIDIA Orin?

- How many of the following schemes mentioned in “improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers” would help NVIDIA’s Tegra for the code in the previous slide?

- ① Missing cache
  - ② Victim cache
  - ③ Prefetch
  - ④ Stream buffer
- A. 0
- B. 1
- C. 2
- D. 3
- E. 4

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 8192; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

## Small Buffers — group

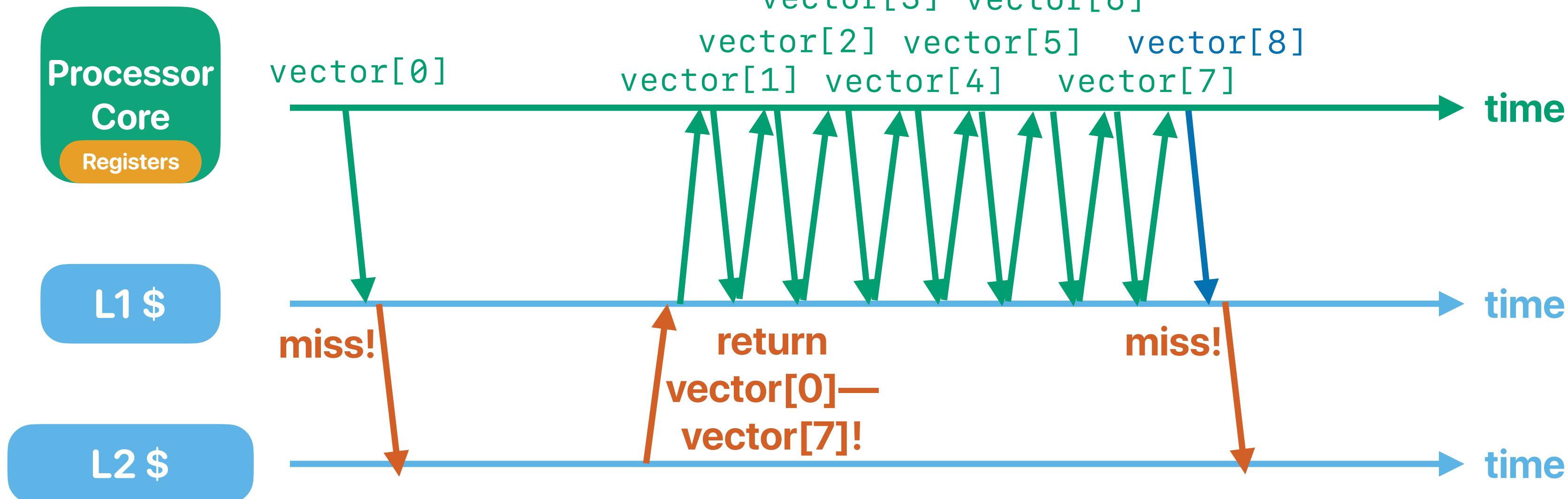
0



# Prefetching

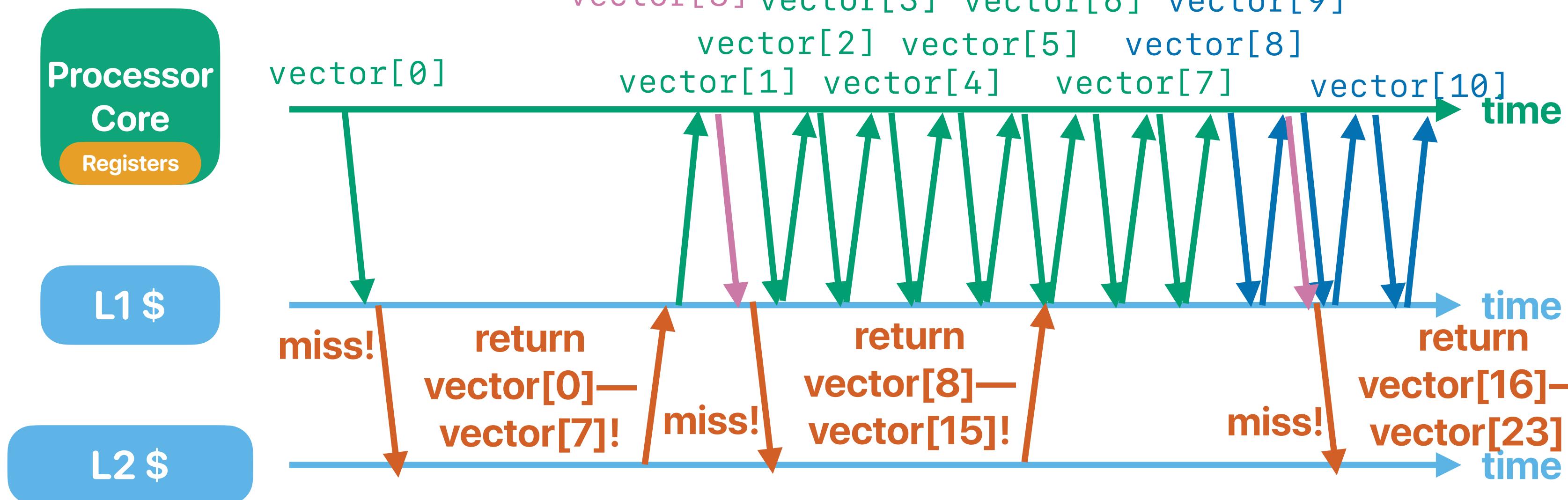
# Spatial locality revisited

```
for(i = 0; i < size; i++) {  
    vector[i] = rand();  
}
```



# What if we “pre-“fetch the next line?

```
for(i = 0; i < size; i++) {  
    vector[i] = rand();  
}
```



# Hardware Prefetching

- The hardware identify the access pattern and proactively fetch data/instruction before the application asks for the data/instruction
- Trigger the cache miss earlier to eliminate the miss when the application needs the data/instruction
- The processor can keep track the distance between misses. If there is a pattern between misses, fetch `miss_data_address + offset` for a miss



# Where can prefetch work effectively?

- How many of the following code snippet can “prefetching” effectively help improving performance?

(1)  
while(node){  
 node = node->next;  
}

(3)  
while (root != NULL){  
 if (key > root->data)  
 root = root->right;  
  
 else if (key < root->data)  
 root = root->left;  
 else  
 return true;  
}

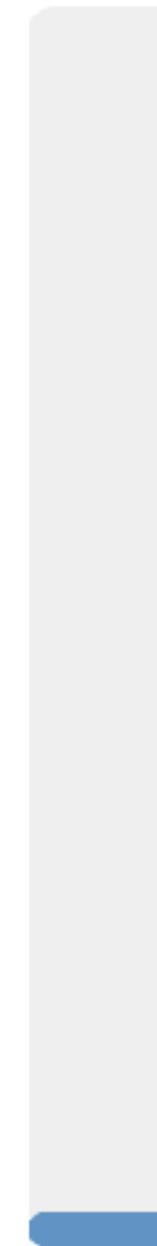
- A. 0
- B. 1
- C. 2
- D. 3
- E. 4

(2)  
while(++i<100000)  
 a[i]=rand();

(4)  
for (i = 0; i < 65536; i++) {  
 mix\_i = ((i \* 167) + 13) & 65536;  
 results[mix\_i]++;  
}

 0

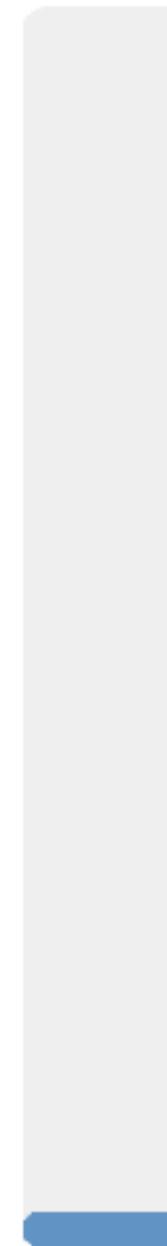
0



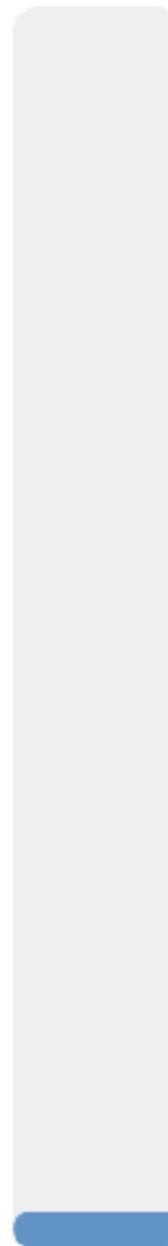
0



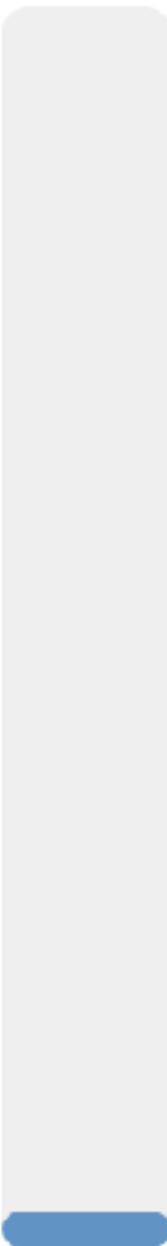
0



0



0



A

B

C

D

E



# Where can prefetch work effectively?

- How many of the following code snippet can “prefetching” effectively help improving performance?

(1)  
while(node){  
 node = node->next;  
}

(3)  
while (root != NULL){  
 if (key > root->data)  
 root = root->right;  
  
 else if (key < root->data)  
 root = root->left;  
 else  
 return true;  
}

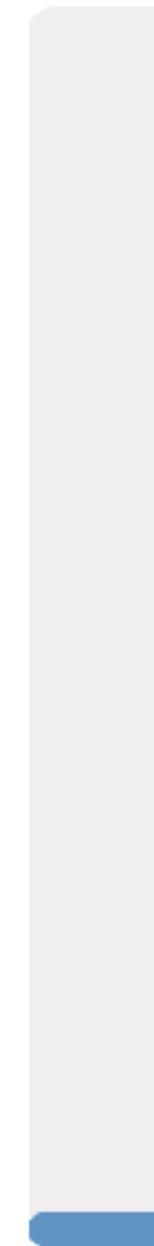
- A. 0
- B. 1
- C. 2
- D. 3
- E. 4

(2)  
while(++i<100000)  
 a[i]=rand();

(4)  
for (i = 0; i < 65536; i++) {  
 mix\_i = ((i \* 167) + 13) & 65536;  
 results[mix\_i]++;  
}

 0

0



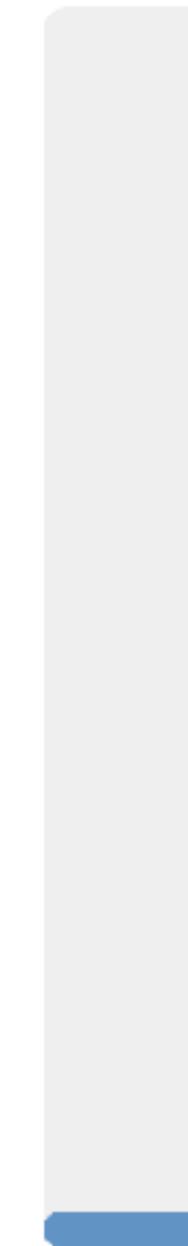
A

0



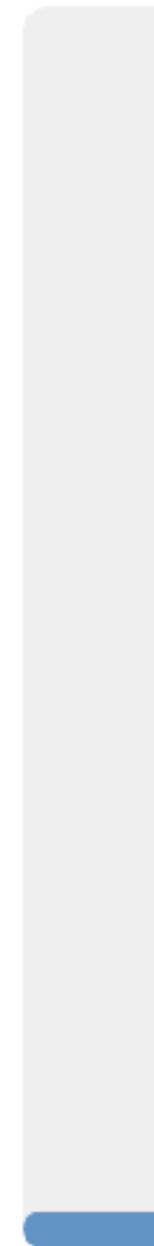
B

0



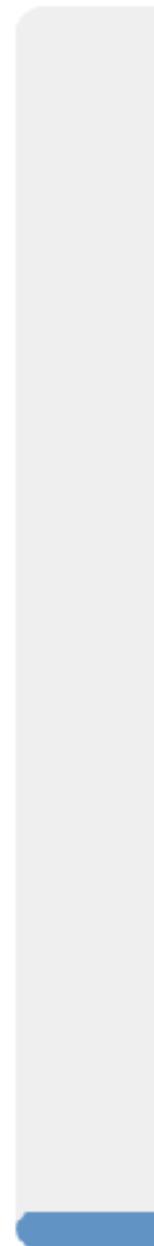
C

0



D

0



E

# Where can prefetch work effectively?

- How many of the following code snippet can “prefetching” effectively help improving performance?

```
(1)
while(node){
    node = node->next;
}
```

— where the next pointing to is hard to predict

```
(3)
while (root != NULL){
    if (key > root->data)
        root = root->right;

    else if (key < root->data)
        root = root->left;
    else
        return true;
}
```

A. 0

B. 1

C. 2

D. 3

E. 4

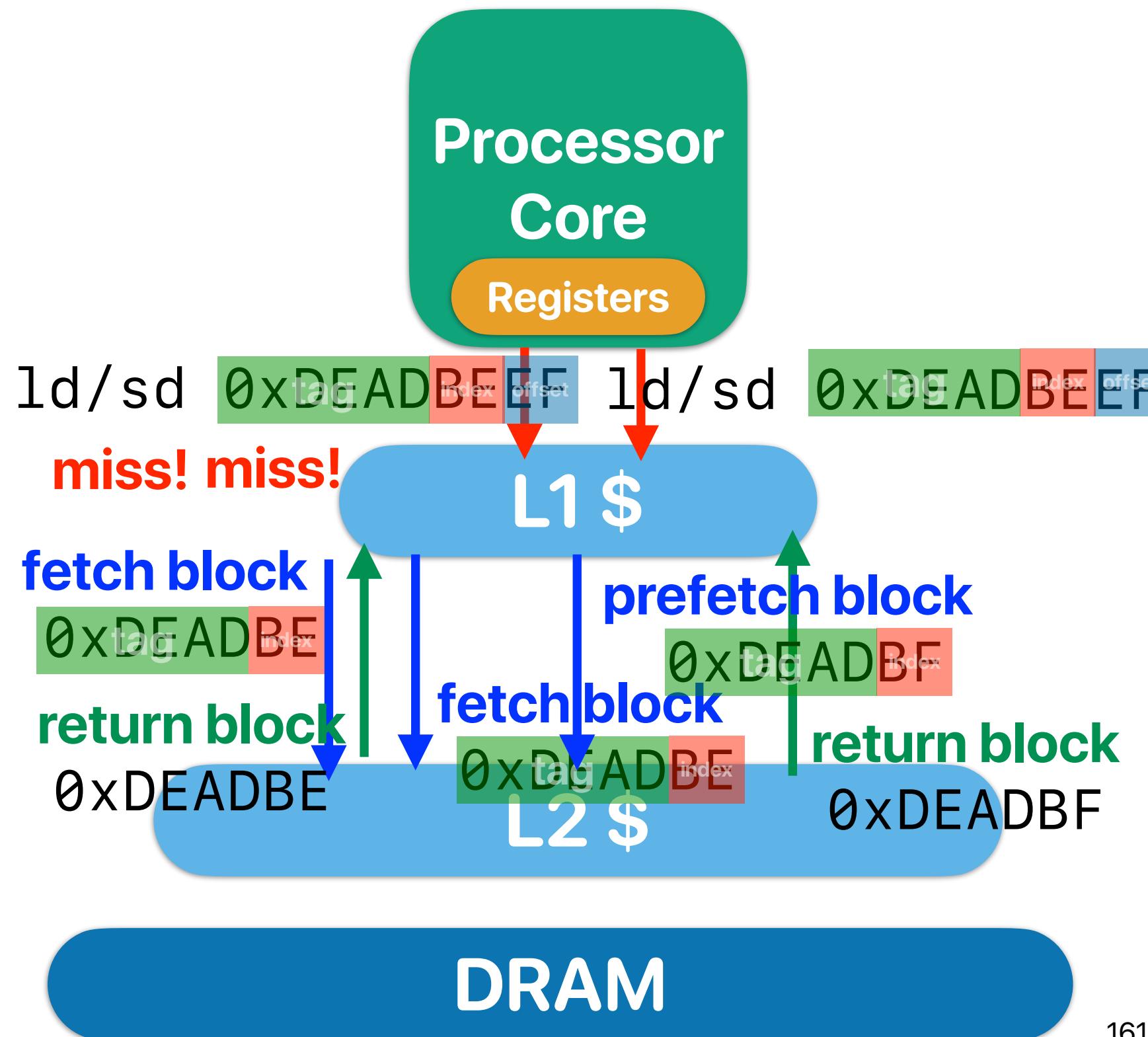
```
(2)
while(++i<100000)
    a[i]=rand();
```

```
(4)
for (i = 0; i < 65536; i++) {
    mix_i = ((i * 167) + 13) & 65536;
    results[mix_i]++;
}
```

— the stride to the next element is hard to predict...

— where the next node is also hard to predict

# What's after prefetching?



# NVIDIA Tegra X1 with prefetch

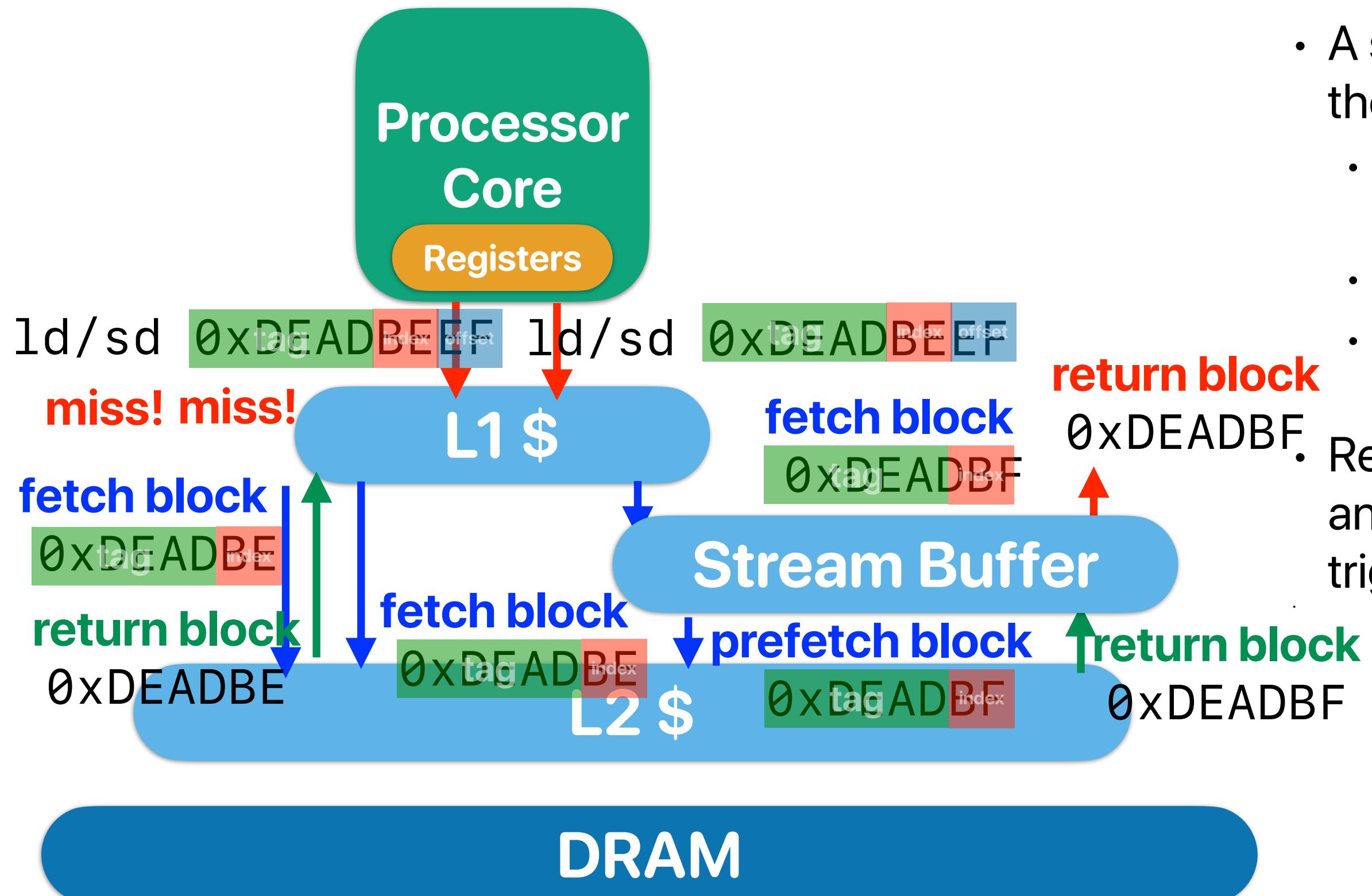
- Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

$C = \text{ABS}$   
 $32\text{KB} = 4 * 64 * S$   
 $S = 128$   
 $\text{offset} = \lg(64) = 6 \text{ bits}$   
 $\text{index} = \lg(128) = 7 \text{ bits}$   
 $\text{tag} = \text{the rest bits}$

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?	Prefetch
a[0]	0x10000	0b000100000000000000000000000000	0x8	0x0	Miss		a[8-15]
b[0]	0x20000	0b001000000000000000000000000000	0x10	0x0	Miss		b[8-15]
c[0]	0x30000	0b001100000000000000000000000000	0x18	0x0	Miss		c[8-15]
d[0]	0x40000	0b010000000000000000000000000000	0x20	0x0	Miss		d[8-15]
e[0]	0x50000	0b010100000000000000000000000000	0x28	0x0	Miss	a[0-7]	e[8-15]
a[1]	0x10008	0b0001000000000000000000001000	0x8	0x0	Miss	b[0-7]	e[8-15] will kick out a[8-15]
b[1]	0x20008	0b0010000000000000000000001000	0x10	0x0	Miss	c[0-7]	
c[1]	0x30008	0b0011000000000000000000001000	0x18	0x0	Miss	d[0-7]	
d[1]	0x40008	0b0100000000000000000000001000	0x20	0x0	Miss	e[0-7]	
e[1]	0x50008	0b0101000000000000000000001000	0x28	0x0	Miss	a[0-7]	

# Stream buffer



- A small cache that captures the prefetched blocks
  - Can be built as fully associative since it's small
  - Consult when there is a miss
  - Retrieve the block if found in the stream buffer
- Reduce compulsory misses and avoid conflict misses triggered by prefetching

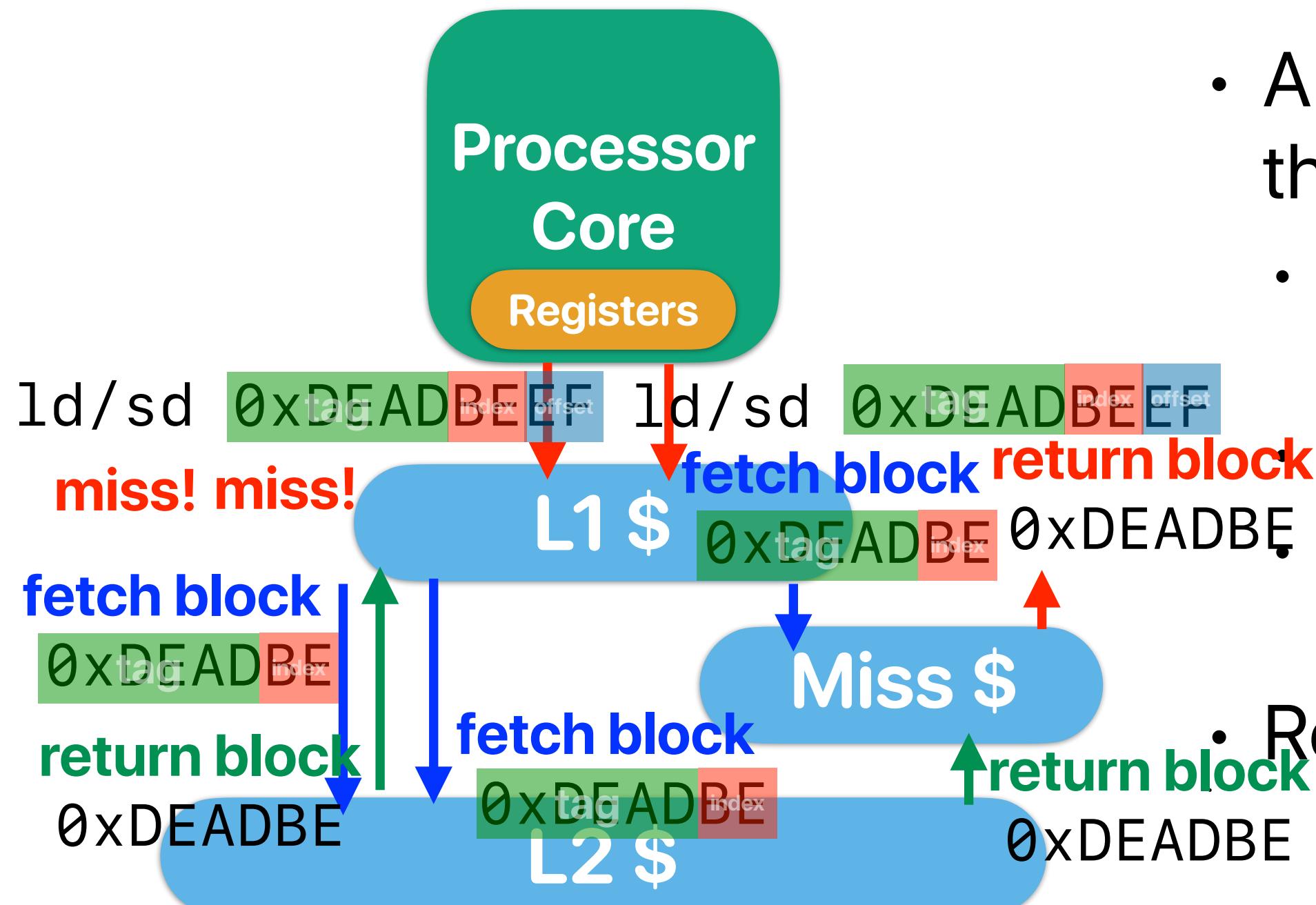
# Software Prefetching — through prefetching instructions

- x86 provide prefetch instructions
- As a programmer, you may insert `_mm_prefetch` in x86 programs to perform software prefetch for your code
- gcc also has a flag “`-fprefetch-loop-arrays`” to automatically insert software prefetch instructions



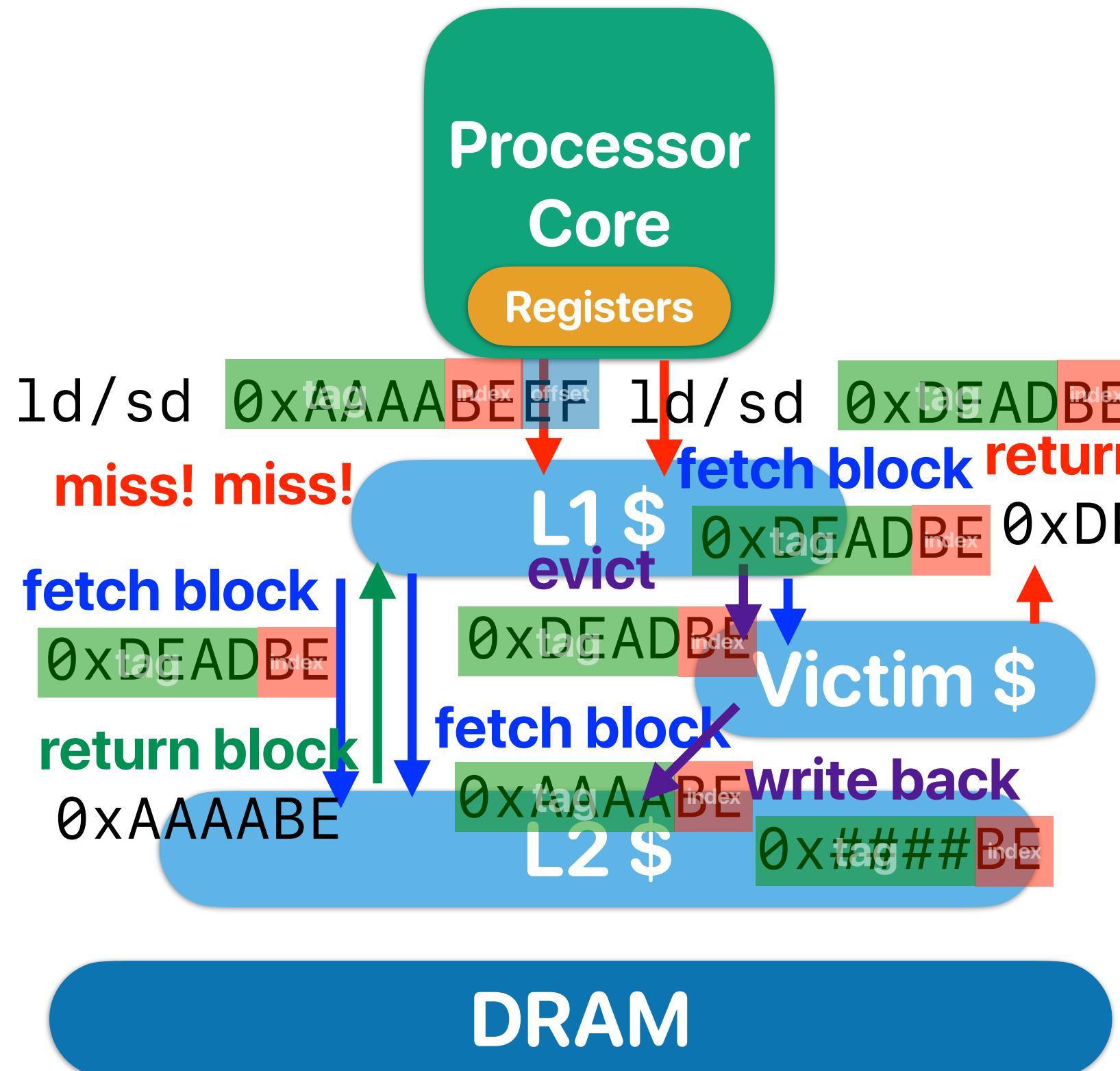
Start the presentation to see live content. Still no live content? Install the app or get help at [PollEv.com/app](https://PollEv.com/app)

# Miss cache



- A small cache that captures the missing blocks
  - Can be built as fully associative since it's small
  - Consult when there is a miss
  - Retrieve the block if found in the missing cache
- Reduce conflict misses

# Victim cache



- A small cache that captures the evicted blocks
  - Can be built as fully associative since it's small
  - Consult when there is a miss
  - Swap the entry if hit in victim cache
- Athlon/Phenom has an 8-entry victim cache
- Reduce conflict misses
- Jouppi [1990]: 4-entry victim cache removed 20% to 95% of conflicts for a 4 KB direct mapped data cache

# Victim cache v.s. miss caching

- Both of them improves conflict misses
- Victim cache can use cache block more efficiently — swaps when miss
  - Miss caching maintains a copy of the missing data — the cache block can both in L1 and miss cache
  - Victim cache only maintains a cache block when the block is kicked out
- Victim cache captures conflict miss better
  - Miss caching captures every missing block

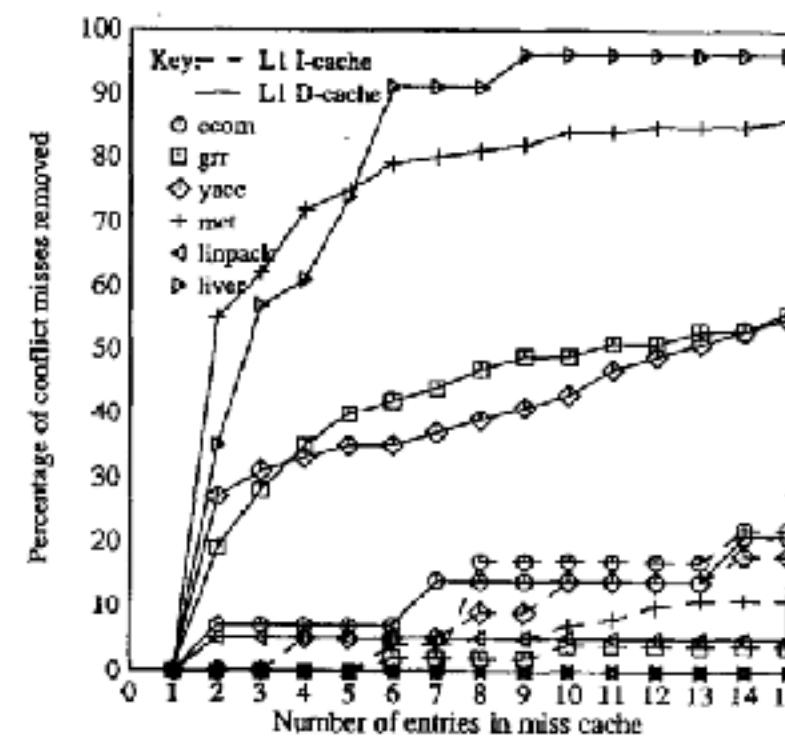


Figure 3-3: Conflict misses removed by miss caching

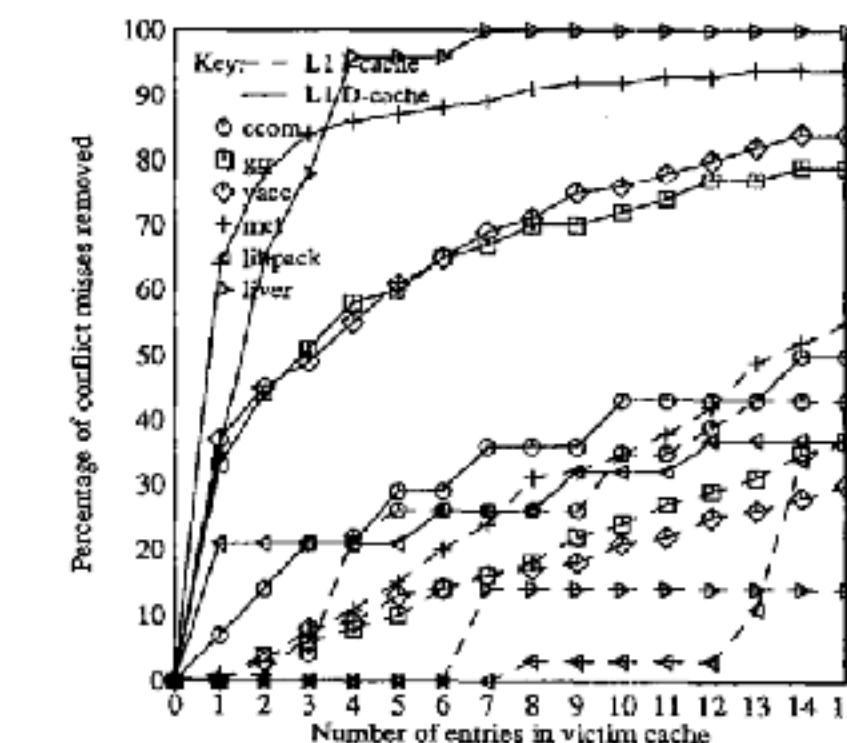


Figure 3-5: Conflict misses removed by victim caching

# Which of the following schemes can help Tegra?

- How many of the following schemes mentioned in “improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers” would help NVIDIA’s Tegra for the code in the previous slide?

① Missing cache — **help improving conflict misses**

② Victim cache — **help improving conflict misses**

③ Prefetch — **improving compulsory misses , but can potentially hurt, if we did not do it right**

④ Stream buffer — **only help improving compulsory misses**

A. 0

B. 1

C. 2

D. 3

E. 4

## Takeaways: Optimizing cache performance through hardware

- There is no optimal cache configurations — trade-offs are everywhere
  - Increasing C — (+): capacity misses; (-): cost, access time, power
  - Increasing A — (+): conflict misses; (-): access time, power
  - Increasing B — (+): compulsory misses; (-): miss penalty
- Adding a small buffer alongside the L1 cache can —
  - Virtually add an associative set to frequently used data structures
  - Prefetched blocks won't cause conflict misses

## Takeaways: Optimizing cache performance through hardware

- There is no optimal cache configurations — trade-offs are everywhere
  - Increasing C — (+): capacity misses; (-): cost, access time, **power**
  - Increasing A — (+): conflict misses; (-): access time, **power**
  - Increasing B — (+): compulsory misses; (-): miss penalty
- Adding a small buffer alongside the L1 cache can —
  - Virtually add an associative set to frequently used data structures
  - Prefetched blocks won't cause conflict misses

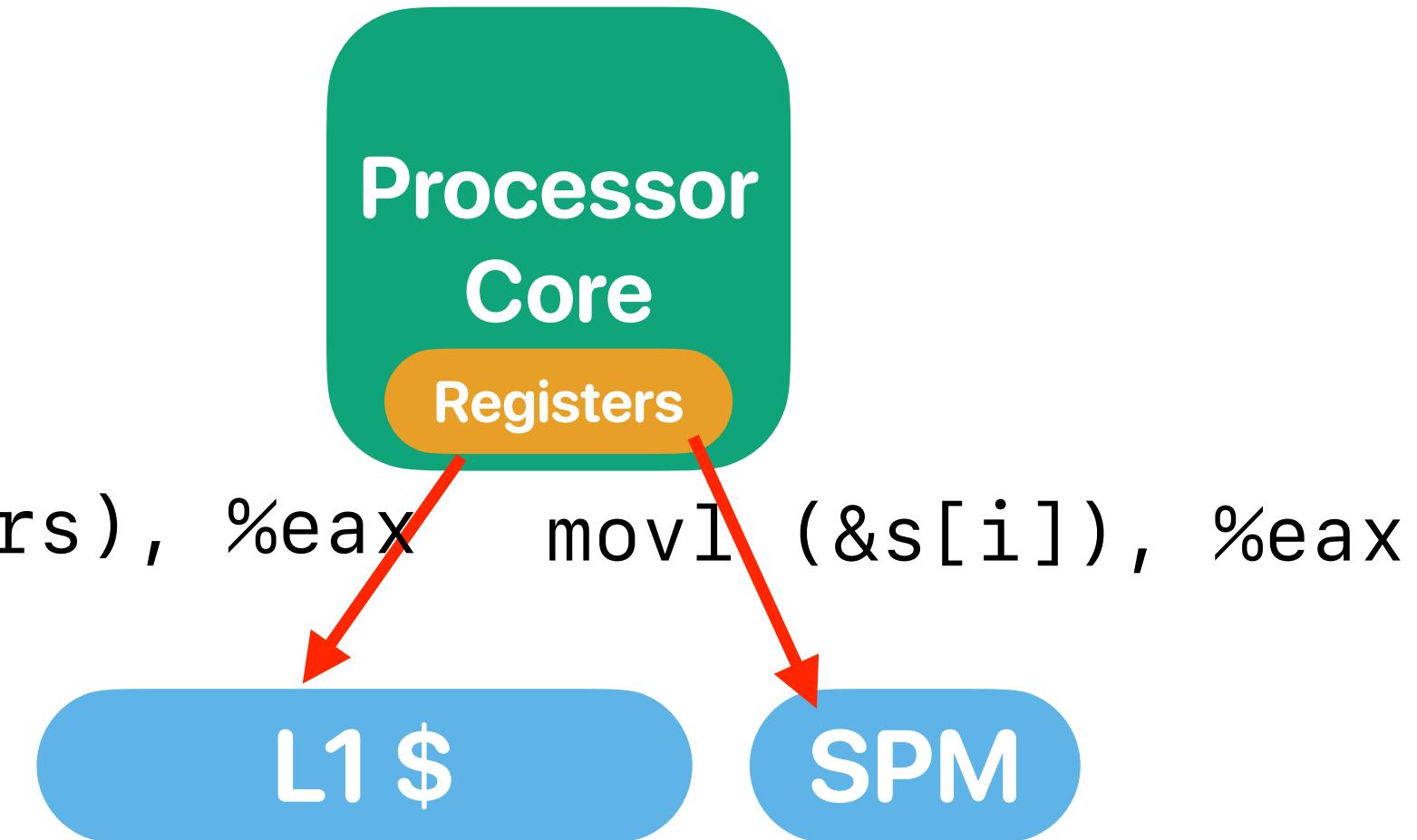
We need additional search time and power

# Implementation of SPM — GPU's shared memory

```
__global__ void staticReverse(int *d, int n)
{
    __shared__ int s[64];
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}

__global__ void dynamicReverse(int *d, int n)
{
    extern __shared__ int s[];
    int t = threadIdx.x;
    int tr = n-t-1;
    s[t] = d[t];
    __syncthreads();
    d[t] = s[tr];
}
```

movl (others), %eax      movl (&s[i]), %eax



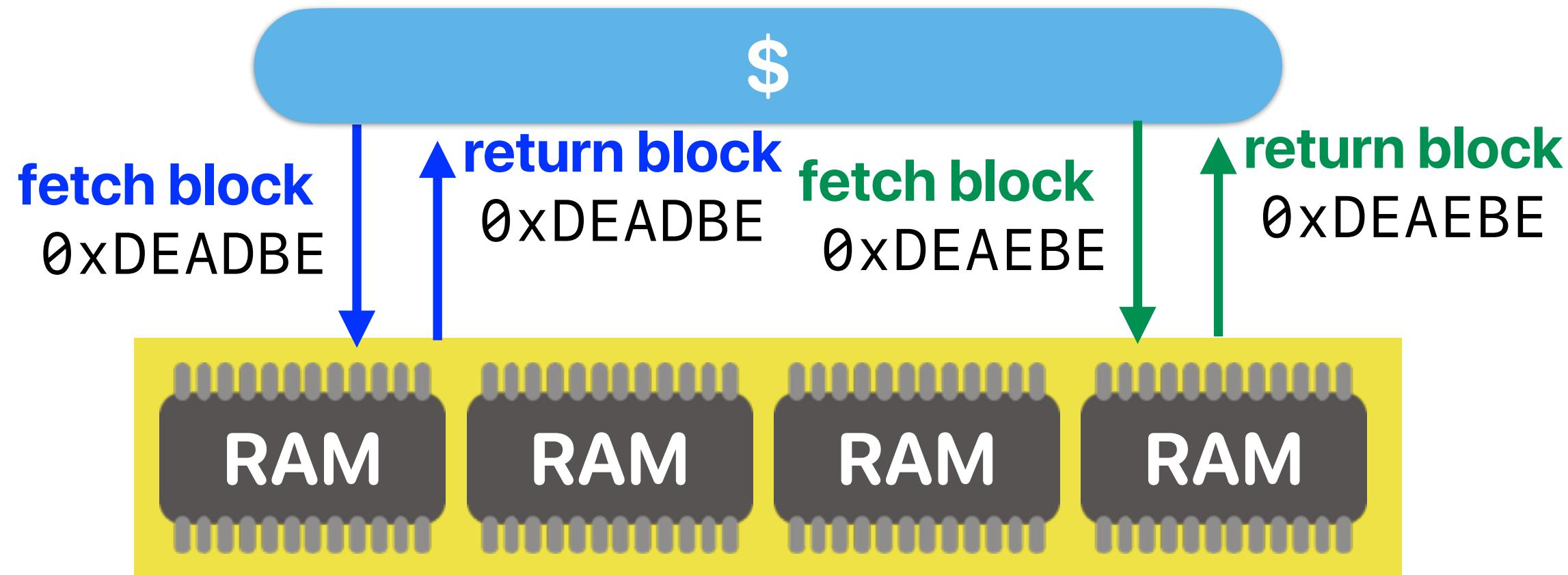
<https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/>

## Takeaways: Optimizing cache performance through hardware

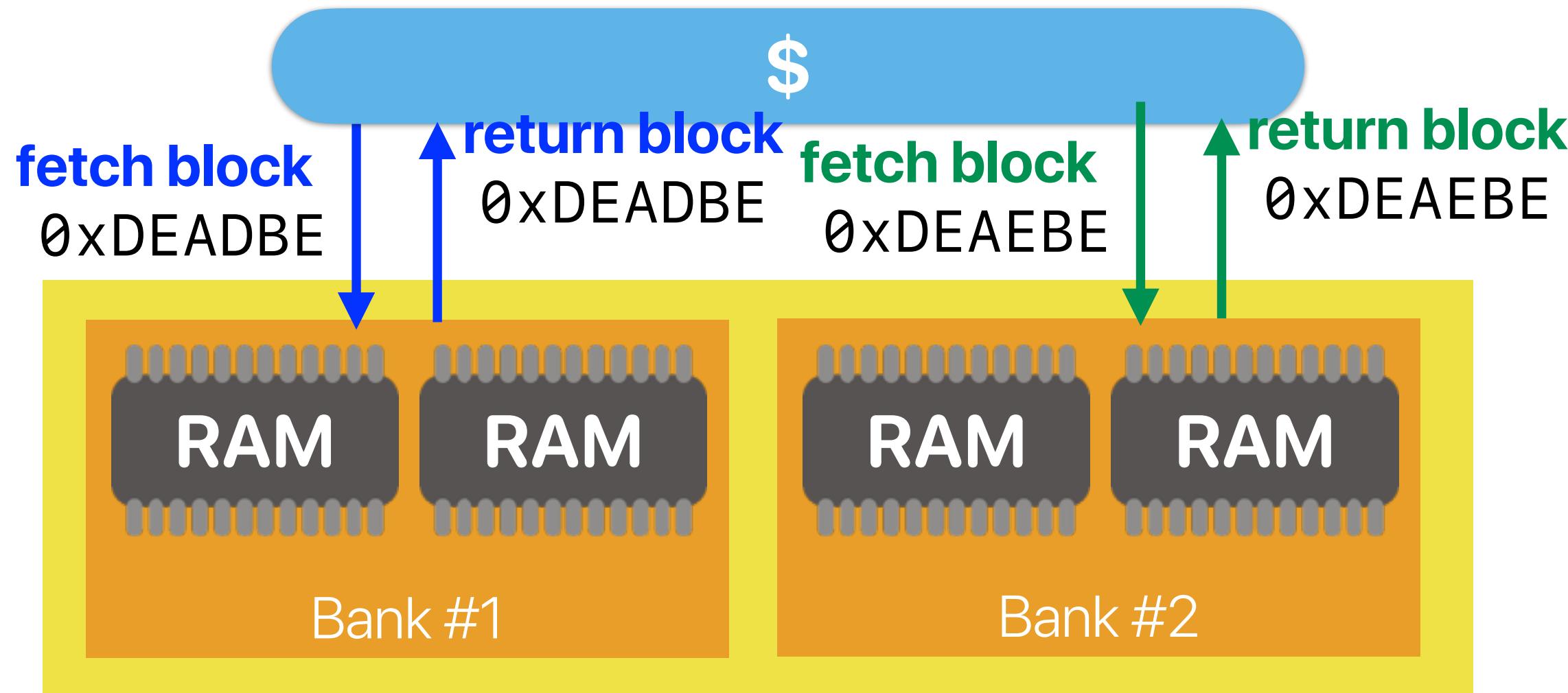
- There is no optimal cache configurations — trade-offs are everywhere
  - Increasing C — (+): capacity misses; (-): cost, access time, power
  - Increasing A — (+): conflict misses; (-): access time, power
  - Increasing B — (+): compulsory misses; (-): miss penalty
- Adding a small buffer alongside the L1 cache can —
  - Virtually add an associative set to frequently used data structures
  - Prefetched blocks won't cause conflict misses
- Adding a tag-less, programmable small buffer alongside the L1 cache can reduce power consumption

# **Advanced Hardware Techniques in Improving Memory Performance**

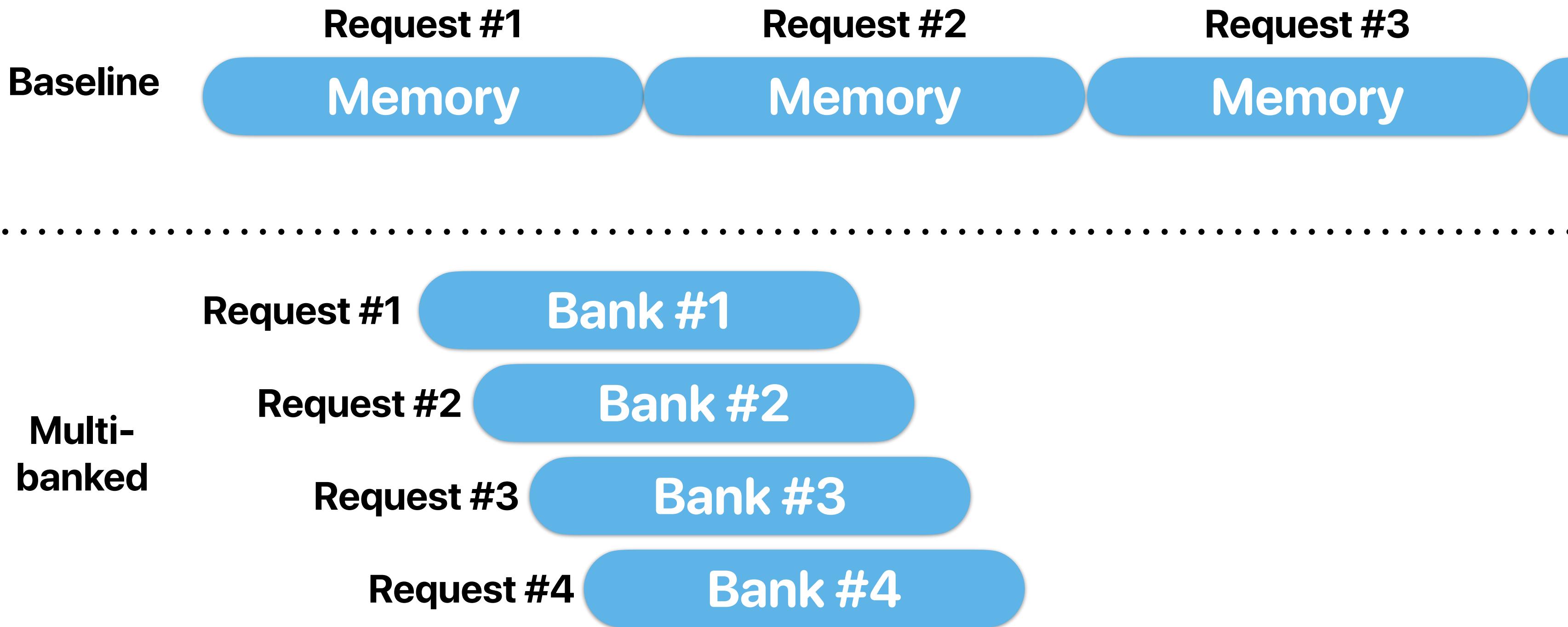
# Blocking cache



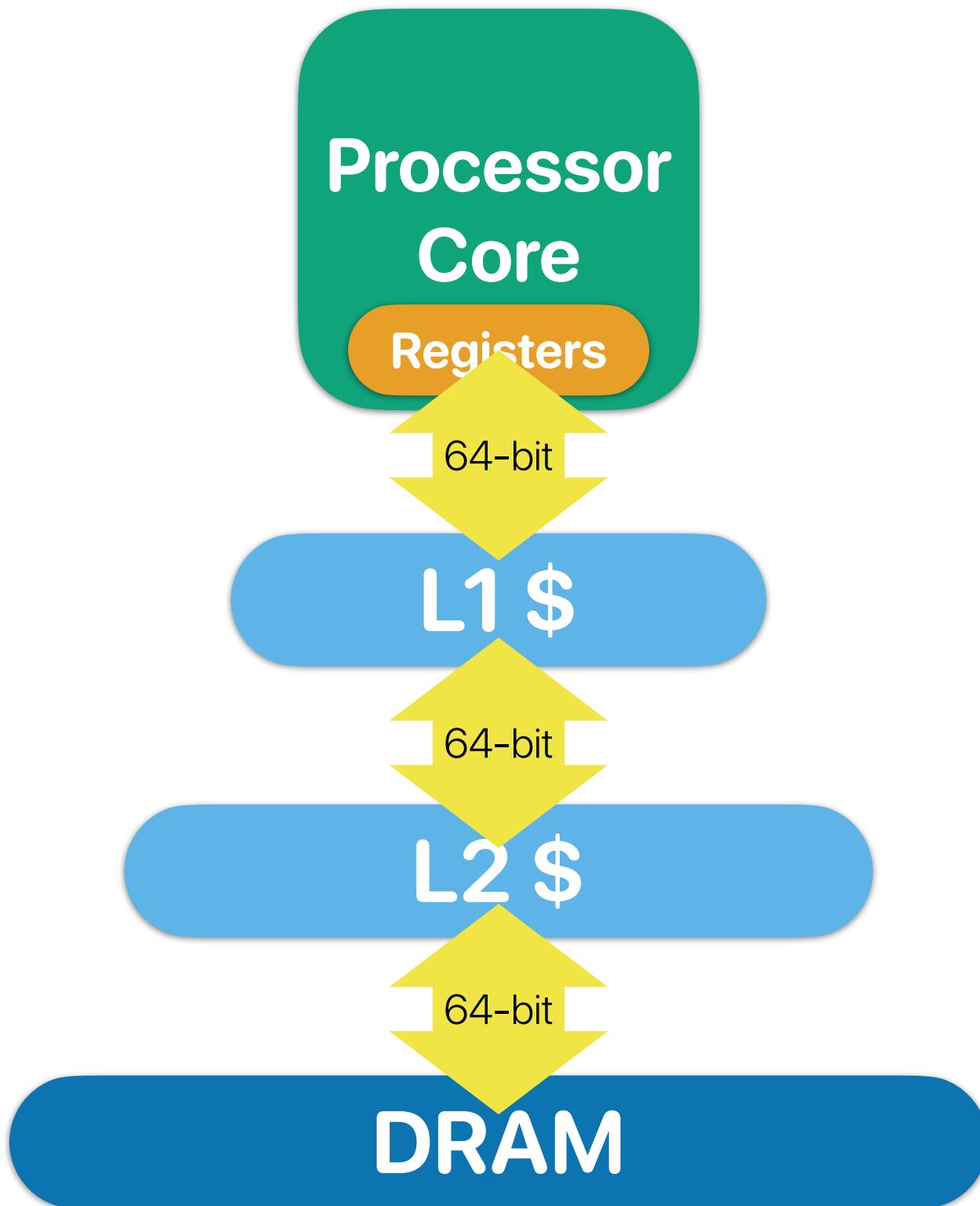
# Multibanks & non-blocking caches



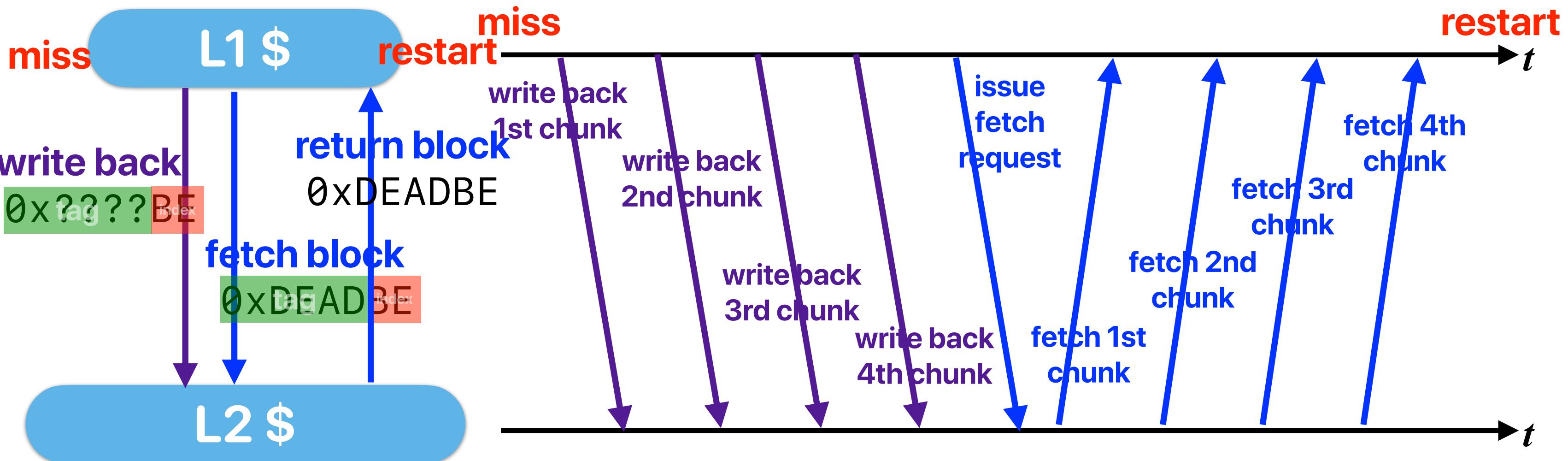
# Pipelined access and multi-banked caches



# The bandwidth between units is limited

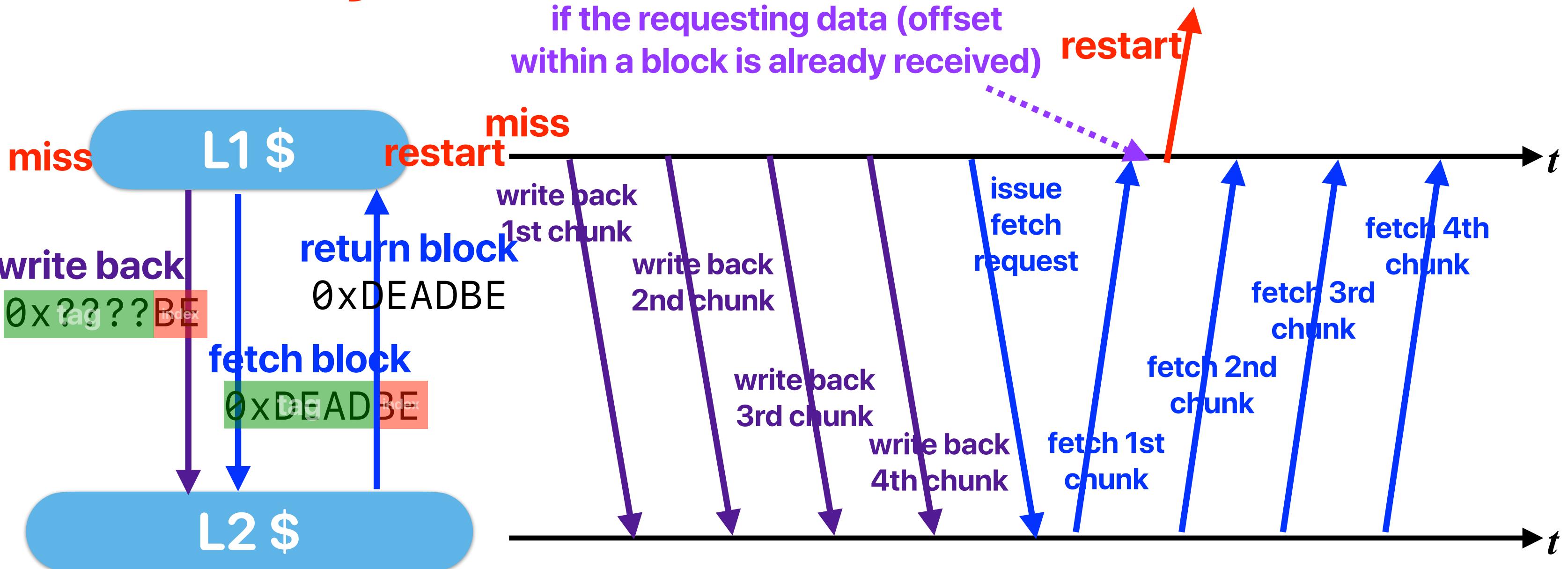


# When we handle a miss



assume the bus between L1/L2 only allows a quarter of the cache block go through it

# Early Restart and Critical Word First

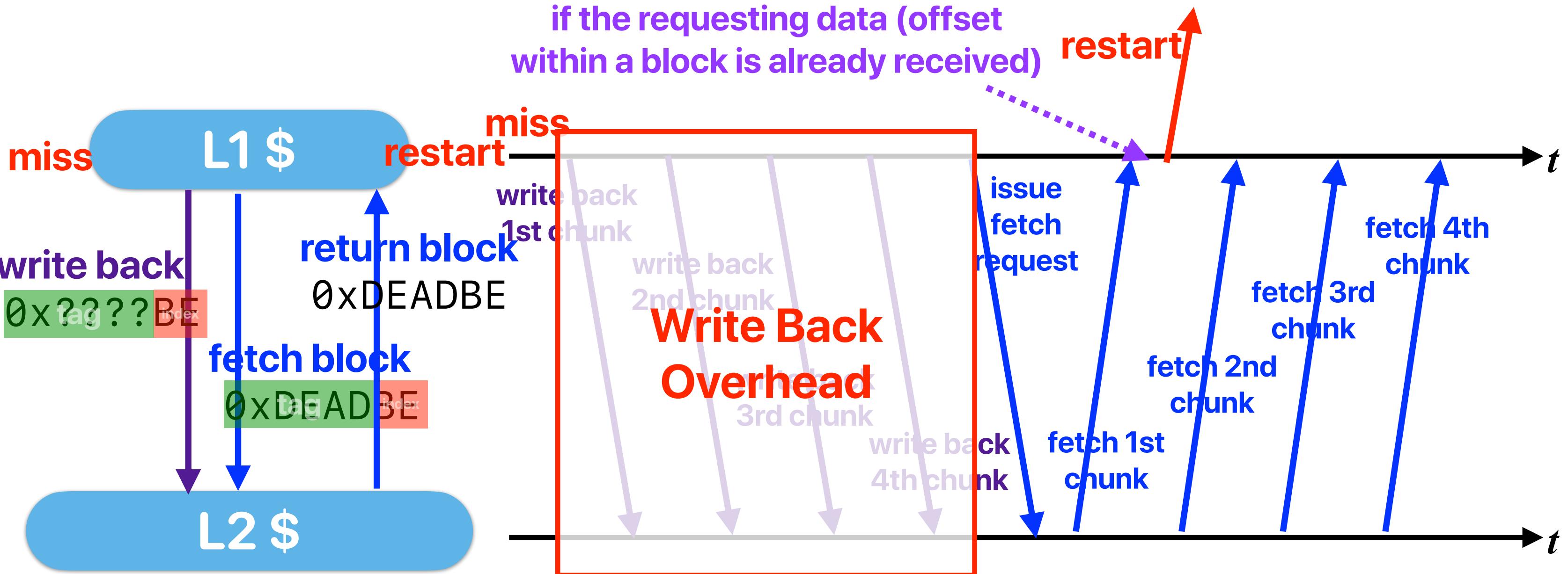


assume the bus between L1/L2 only allows a quarter of the cache block go through it

# Early Restart and Critical Word First

- Don't wait for full block to be loaded before restarting CPU
  - Early restart—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
  - Critical Word First—Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block. Also called wrapped fetch and requested word first
- Most useful with large blocks
- Spatial locality is a problem; often we want the next sequential word soon, so not always a benefit (early restart).

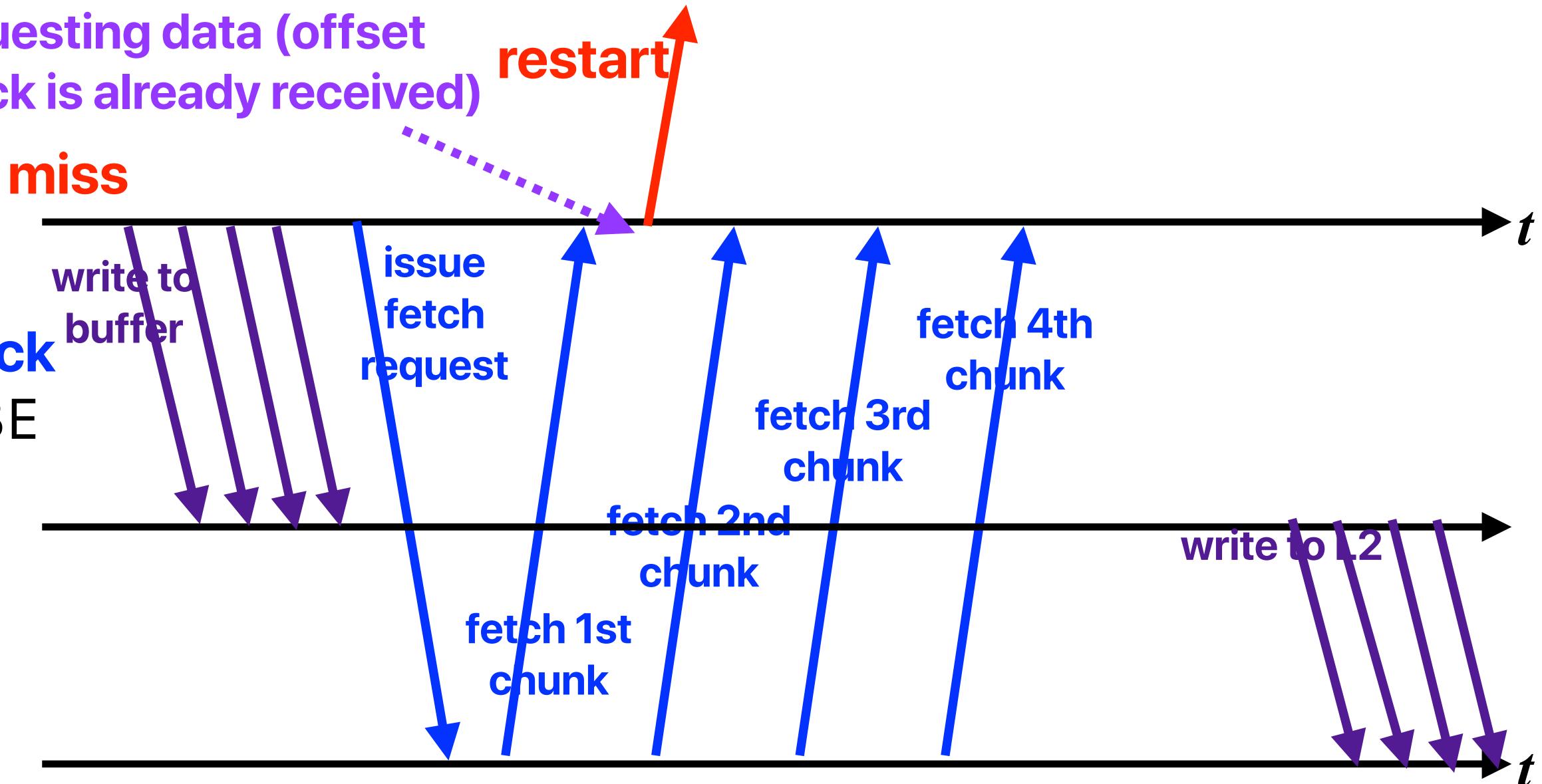
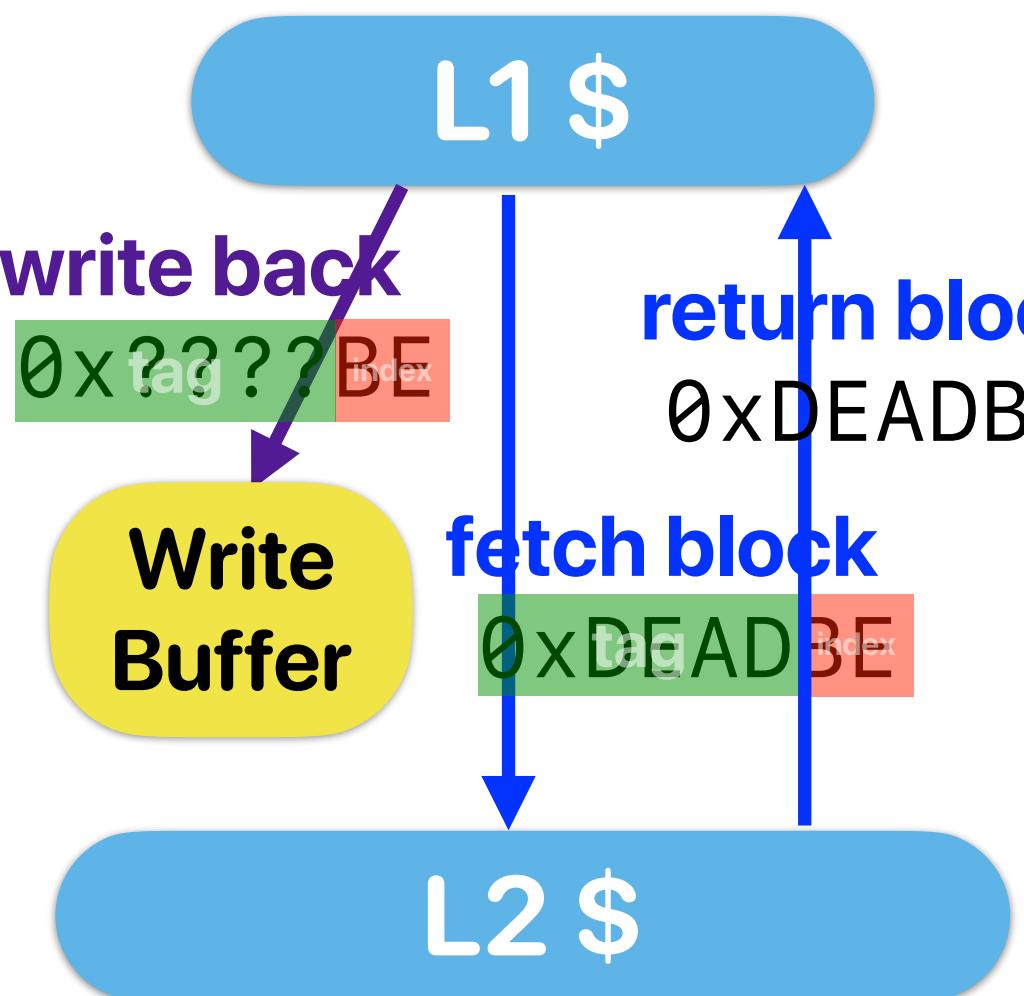
# Can we avoid the overhead of writes?



assume the bus between L1/L2 only allows a quarter of the cache block go through it

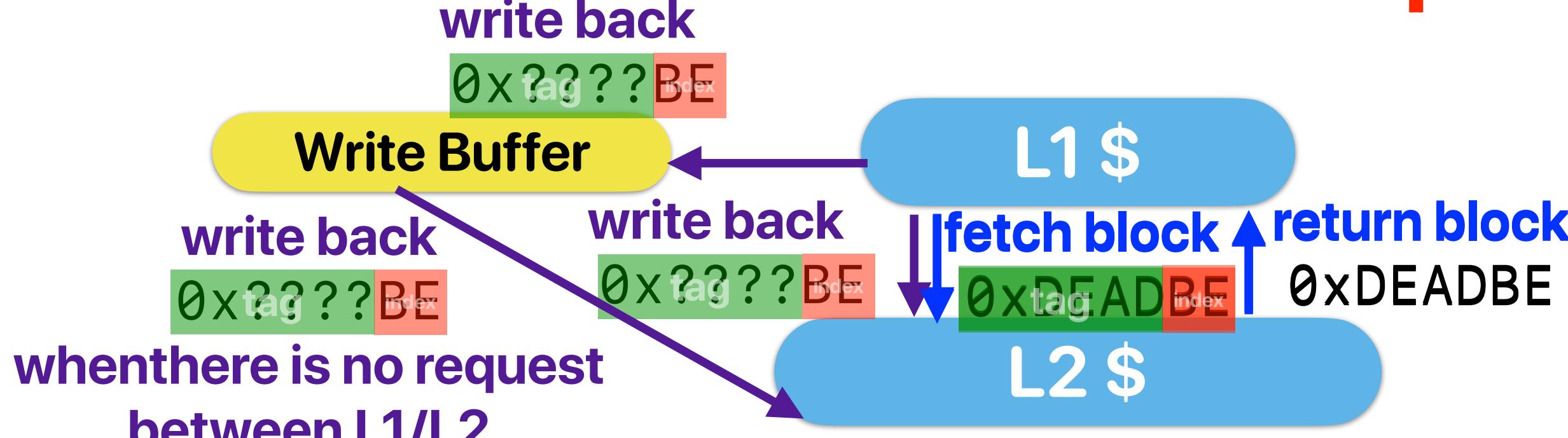
# Write buffer!

if the requesting data (offset within a block is already received)



assume the bus between L1/L2 only allows a quarter of the cache block go through it

# Can we avoid the “double penalty”?



- Every write to lower memory will first write to a small SRAM buffer.
  - store does not incur data hazards, but the pipeline has to stall if the write misses
  - The write buffer will continue writing data to lower-level memory
  - The processor/higher-level memory can respond as soon as the data is written to write buffer.
- Write merge
  - Since application has locality, it's highly possible the evicted data have neighboring addresses. Write buffer delays the writes and allows these neighboring data to be grouped together.

# Can we avoid the “double penalty”?

- Every write to lower memory will first write to a small SRAM buffer.
  - store does not incur data hazards, but the pipeline has to stall if the write misses
  - The write buffer will continue writing data to lower-level memory
  - The processor/higher-level memory can response as soon as the data is written to write buffer.
- Write merge
  - Since application has locality, it's highly possible the evicted data have neighboring addresses. Write buffer delays the writes and allows these neighboring data to be grouped together.



# Summary of Optimizations

- Regarding the following cache optimizations, how many of them would help improve miss rate?
  - ① Non-blocking/pipelined/multibanked cache
  - ② Critical word first and early restart
  - ③ Prefetching
  - ④ Write buffer

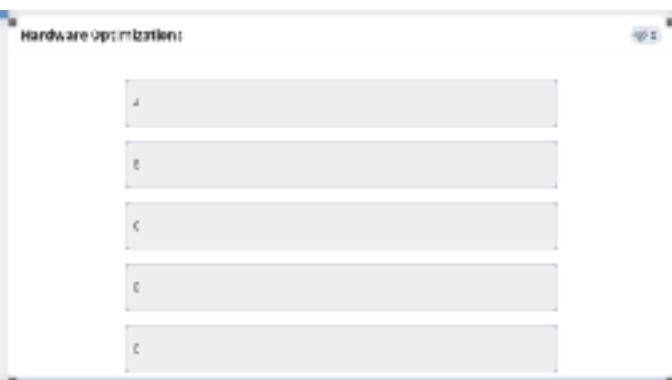
A. 0

B. 1

C. 2

D. 3

E. 4



# Hardware Optimizations

0





# Summary of Optimizations

- Regarding the following cache optimizations, how many of them would help improve miss rate?
  - ① Non-blocking/pipelined/multibanked cache
  - ② Critical word first and early restart
  - ③ Prefetching
  - ④ Write buffer

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



## Hardware Optimizations – group

0



# Summary of Optimizations

- Regarding the following cache optimizations, how many of them would help improve miss rate?
  - ① Non-blocking/pipelined/multibanked cache **Miss penalty/Bandwidth**
  - ② Critical word first and early restart **Miss penalty**
  - ③ Prefetching **Miss rate (compulsory)**
  - ④ Write buffer **Miss penalty**

A. 0

B. 1

C. 2

D. 3

E. 4

# Summary of Architectural Optimizations

- Hardware
  - Prefetch — compulsory miss
  - Write buffer — miss penalty
  - Bank/pipeline — miss penalty
  - Critical word first and early restart — miss panelty

# **How can programmer improve memory performance?**

# Data structures



# Column-store or row-store

- Considering your the most frequently used queries in your database system are similar to

SELECT AVG(assignment\_1) FROM table

Which of the following would be a data structure that better implements the table supporting this type of queries?

Array of objects	object of arrays
<pre>struct grades {     int id;     double *homework;     double average; }; table = (struct grades *) \ malloc(num_of_students*sizeof(struct</pre>	<pre>struct grades {     int *id;     double **homework;     double *average; };  table =(struct grades *)malloc(sizeof(struct grades));</pre>

- A. Array of objects
- B. Object of arrays

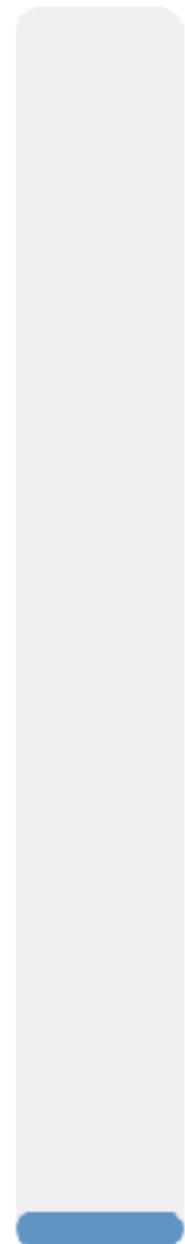
0

0



A

0



B



# Column-store or row-store

- Considering your the most frequently used queries in your database system are similar to

SELECT AVG(assignment\_1) FROM table

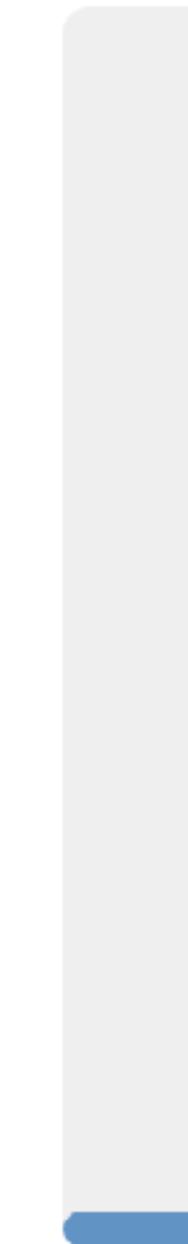
Which of the following would be a data structure that better implements the table supporting this type of queries?

Array of objects	object of arrays
<pre>struct grades {     int id;     double *homework;     double average; }; table = (struct grades *) \ malloc(num_of_students*sizeof(struct</pre>	<pre>struct grades {     int *id;     double **homework;     double *average; }; table =(struct grades *)malloc(sizeof(struct grades));</pre>

- A. Array of objects
- B. Object of arrays

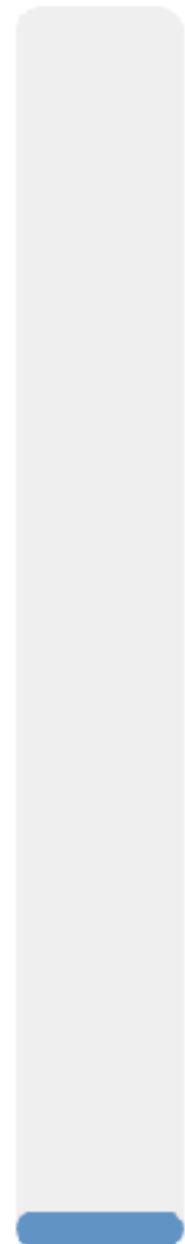
0

0



A

0



B

# Column-store or row-store

- Considering your the most frequently used queries in your database system are similar to

SELECT AVG(assignment\_1) FROM table

Which of the following would be a data structure that better implements the table supporting this type of queries?

Array of objects	object of arrays
<pre>struct grades {     int id;     double *homework;     double average; }; table = (struct grades *) \ malloc(num_of_students*sizeof(struct</pre>	<pre>struct grades {     int *id;     double **homework;     double *average; }; table =(struct grades *)malloc(sizeof(struct grades));</pre>

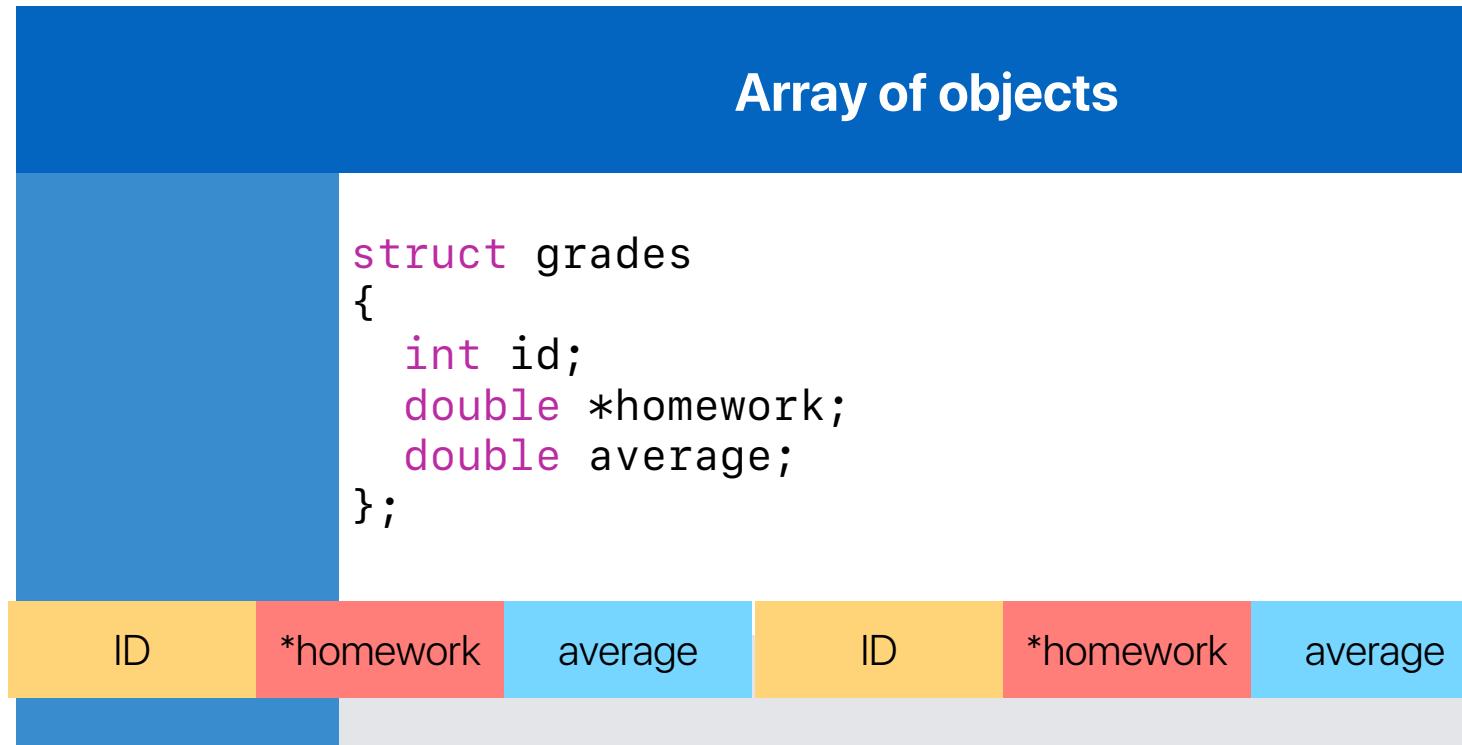
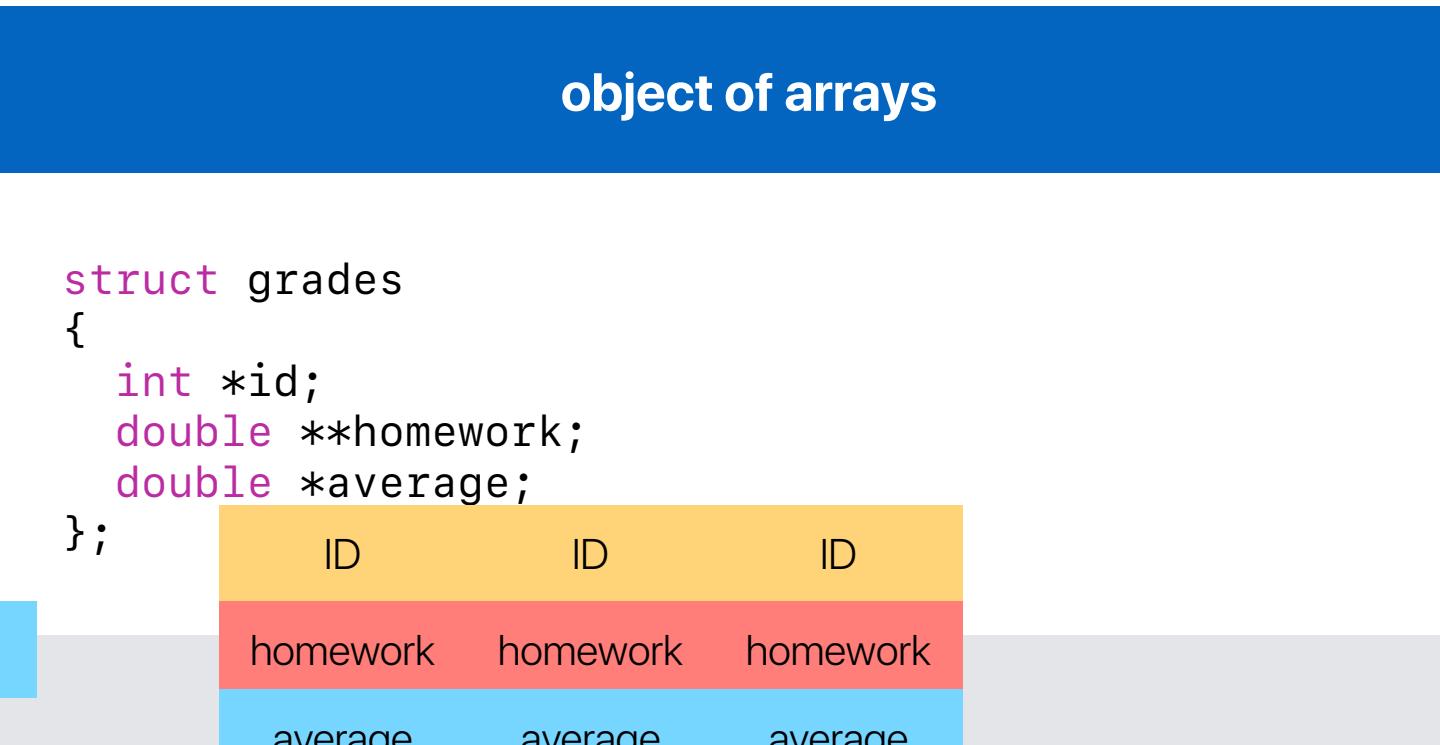
A. Array of objects **What if we want to calculate average scores for each student?**

B. Object of arrays



Start the presentation to see live content. Still no live content? Install the app or get help at [PollEv.com/app](https://PollEv.com/app)

# Array of structures or structure of arrays

	Array of objects	object of arrays
	<pre>struct grades {     int id;     double *homework;     double average; };</pre>  <p>average of each homework</p>	<pre>struct grades {     int *id;     double **homework;     double *average; };</pre> 
	<pre>for(i=0;i&lt;homework_items; i++) {     gradesheet[total_number_students].homework[i] = 0.0;     for(j=0;j&lt;total_number_students;j++)         gradesheet[total_number_students].homework[i]         +=gradesheet[j].homework[i];     gradesheet[total_number_students].homework[i] /= (double)total_number_students; }</pre>	<pre>for(i = 0;i &lt; homework_items; i++) {     gradesheet.homework[i][total_number_students] = 0.0;     for(j = 0; j &lt;total_number_students;j++)     {         gradesheet.homework[i][total_number_students] += gradesheet.homework[i][j];     }     gradesheet.homework[i][total_number_students] /= total_number_students; }</pre>

# Column-store or row-store

- If you're designing an in-memory database system, will you be using

RowId	Empld	Lastname	Firstname	Salary
1	10	Smith	Joe	40000
2	12	Jones	Mary	50000
3	11	Johnson	Cathy	44000
4	22	Jones	Bob	55000

- column-store — stores data tables column by column

10:001,12:002,11:003,22:004;

Smith:001,Jones:002,Johnson:003,Jones:004,

Joe:001,Mary:002,Cathy:003,Bob:004;

40000:001,50000:002,44000:003,55000:004;

**if the most frequently used query looks like –**  
**select Lastname, Firsntname from table**

- row-store — stores data tables row by row

001:10,Smith,Joe,40000;

002:12,Jones,Mary,50000;

003:11,Johnson,Cathy,44000;

004:22,Jones,Bob,55000;

# Takeaways: Software Optimizations

- Data layout — capacity miss, conflict miss, compulsory miss

# **Loop interchange/fission/fusion**

# Demo — programmer & performance

A

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

B

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

$O(n^2)$

Same

Same

Better

Complexity

Instruction Count?

Clock Rate

CPI

$O(n^2)$

Same

Same

Worse

# Loop optimizations

A

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

## Loop interchange



B

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

# Takeaways: Software Optimizations

- Data layout — capacity miss, conflict miss, compulsory miss
- Loop interchange — conflict/capacity miss

# NVIDIA Tegra Orin

- D-L1 Cache configuration of NVIDIA Tegra Orin
  - Size 64KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

What's the data cache miss rate for this code?

- A. 12.5%
- B. 56.25%
- C. 66.67%
- D. 68.75%
- E. 100%



# What if the code look like this?

- D-L1 Cache configuration of NVIDIA Tegra Orin
  - Size 64KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

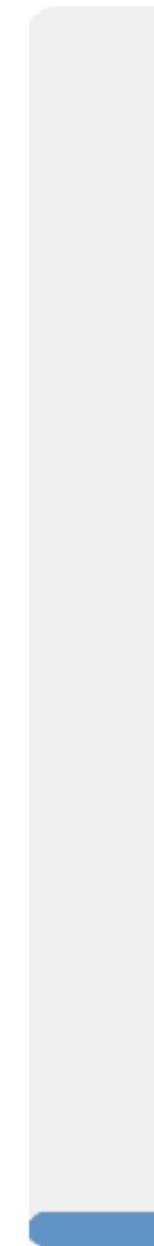
```
double a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i]; //load a, b, c and then store to e
for(i = 0; i < 512; i++)
    e[i] /= d[i]; //load e, load d, and then store to e
```

What's the data cache miss rate for this code?

- A. ~10%
- B. ~20%
- C. ~40%
- D. ~80%
- E. 100%

 0

0



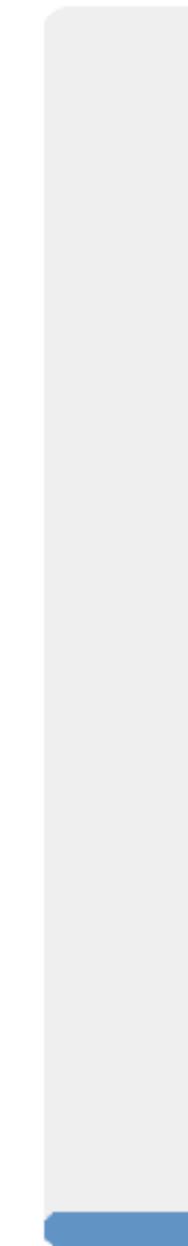
A

0



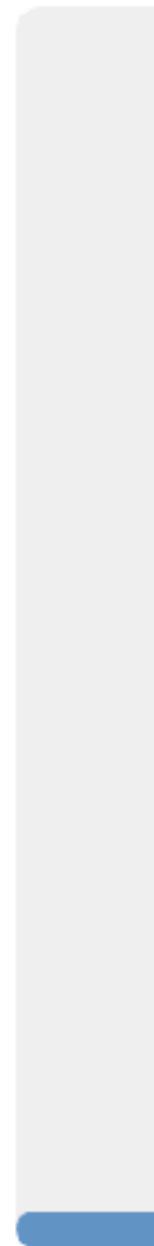
B

0



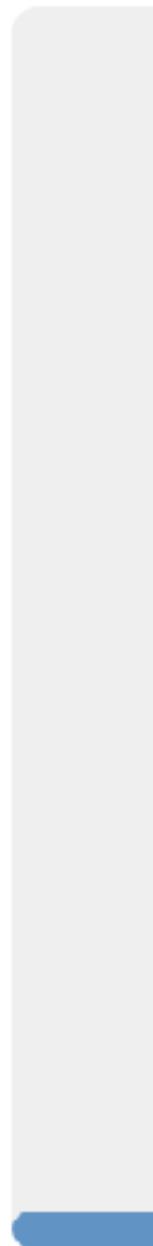
C

0



D

0



E



# What if the code look like this?

- D-L1 Cache configuration of NVIDIA Tegra Orin
  - Size 64KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

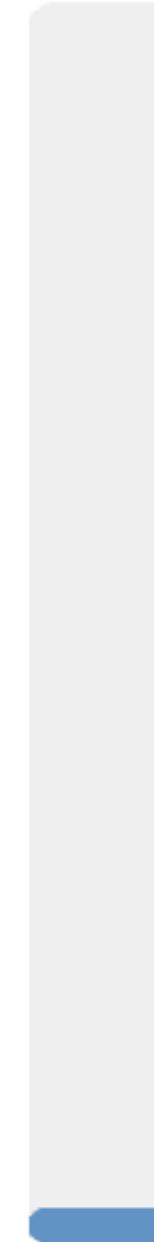
```
double a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i]; //load a, b, c and then store to e
for(i = 0; i < 512; i++)
    e[i] /= d[i]; //load e, load d, and then store to e
```

What's the data cache miss rate for this code?

- A. ~10%
- B. ~20%
- C. ~40%
- D. ~80%
- E. 100%

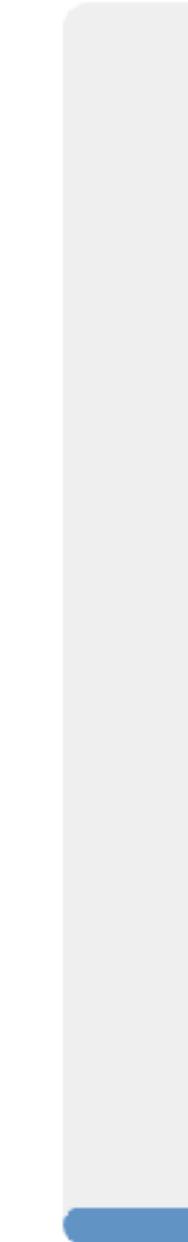


0



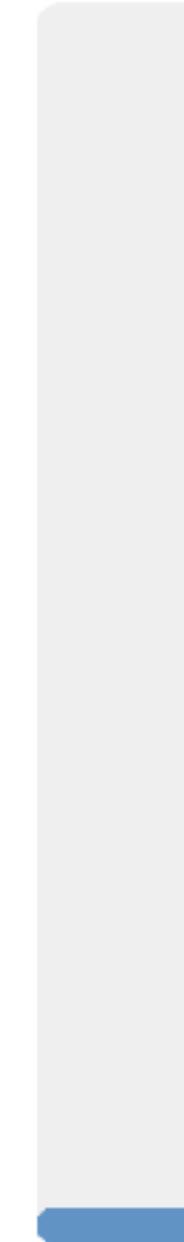
A

0



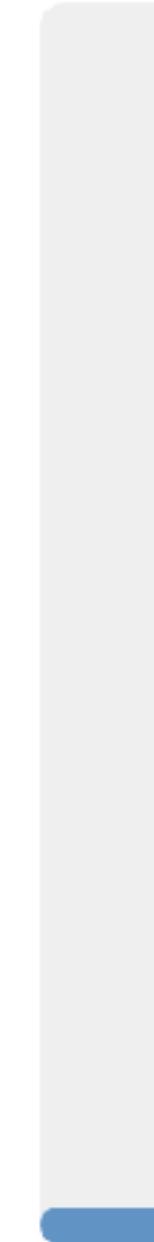
B

0



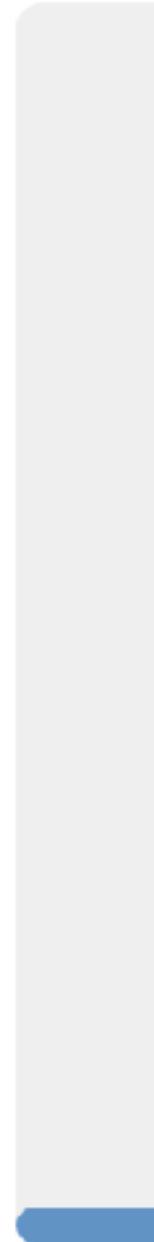
C

0



D

0



E



Start the presentation to see live content. Still no live content? Install the app or get help at [PollEv.com/app](https://PollEv.com/app)

# What if the code look like this?

- D-L1 Cache configuration of NVIDIA Tegra Orin
  - Size 64KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i]; //load a, b, c and then store to e
for(i = 0; i < 512; i++)
    e[i] /= d[i]; //load e, load d, and then store to e
```

What's the data cache miss rate for this code?

- A. ~10%
- B. ~20%
- C. ~40%
- D. ~80%
- E. 100%

# Loop optimizations

A

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

## Loop interchange

B

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

B

```
double a[8192], b[8192], c[8192], \  
       d[8192], e[8192];  
for(i = 0; i < 512; i++) {  
    e[i] = (a[i] * b[i] + c[i])/d[i];  
}
```

## Loop fission

A

```
double a[8192], b[8192], c[8192], \  
       d[8192], e[8192];  
for(i = 0; i < 512; i++)  
    e[i] = a[i] * b[i] + c[i];  
for(i = 0; i < 512; i++)  
    e[i] /= d[i];
```

# Takeaways: Software Optimizations

- Data layout — capacity miss, conflict miss, compulsory miss
- Loop interchange — conflict/capacity miss
- Loop fission — conflict miss — when \$ has limited way associativity



# What if we change the processor?

- If we have an intel processor with a 48KB, 12-way, 64B-blocked L1 cache, which version of code performs better?
  - A. Version A, because the code incurs fewer cache misses
  - B. Version B, because the code incurs fewer cache misses
  - C. Version A, because the code incurs fewer memory references
  - D. Version B, because the code incurs fewer memory references
  - E. They are about the same

**A**

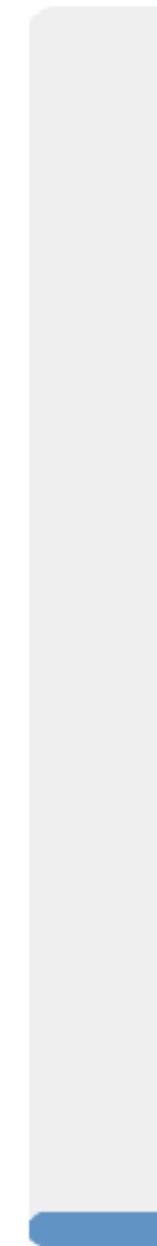
```
double a[8192], b[8192], c[8192], \
        d[8192], e[8192];
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i];
for(i = 0; i < 512; i++)
    e[i] /= d[i];
```

**B**

```
double a[8192], b[8192], c[8192], \
        d[8192], e[8192];
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
}
```

 0

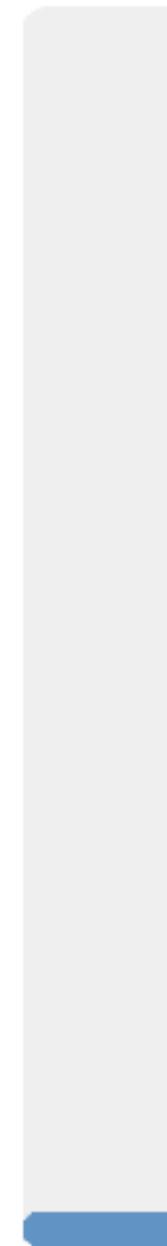
0



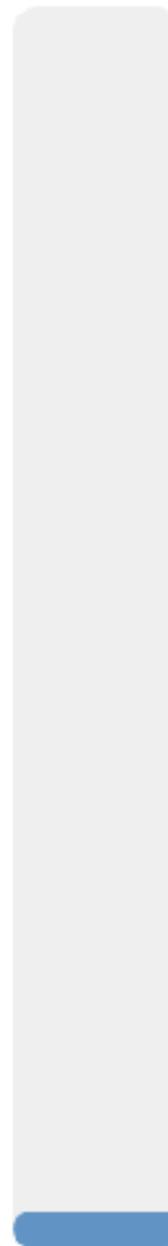
0



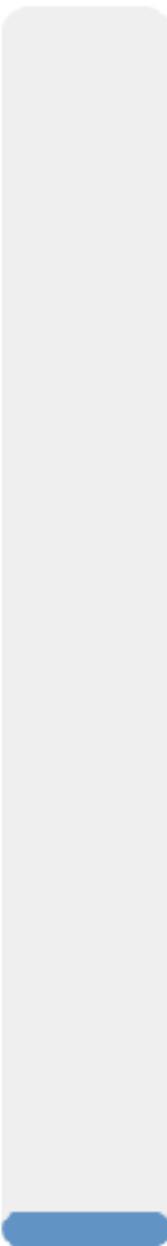
0



0



0



A

B

C

D

E



# What if we change the processor?

- If we have an intel processor with a 48KB, 12-way, 64B-blocked L1 cache, which version of code performs better?
  - Version A, because the code incurs fewer cache misses
  - Version B, because the code incurs fewer cache misses
  - Version A, because the code incurs fewer memory references
  - Version B, because the code incurs fewer memory references
  - They are about the same

A

```
double a[8192], b[8192], c[8192], \
        d[8192], e[8192];
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i];
for(i = 0; i < 512; i++)
    e[i] /= d[i];
```

B

```
double a[8192], b[8192], c[8192], \
        d[8192], e[8192];
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
}
```

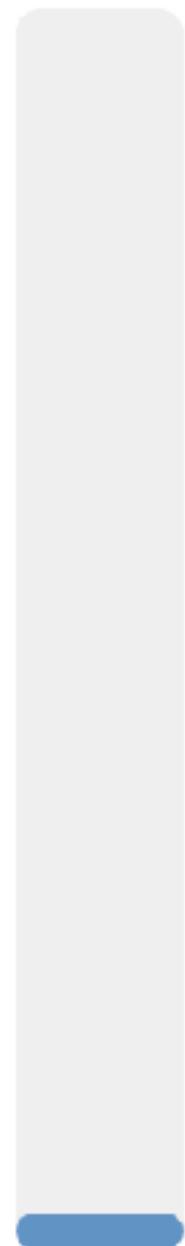
 0

0



A

0



B



Start the presentation to see live content. Still no live content? Install the app or get help at [PollEv.com/app](https://PollEv.com/app)

# What if we change the processor?

- If we have an intel processor with a 32KB, 8-way, 64B-blocked L1 cache, which version of code performs better?
  - A. Version A, because the code incurs fewer cache misses
  - B. Version B, because the code incurs fewer cache misses
  - C. Version A, because the code incurs fewer memory references
  - D. Version B, because the code incurs fewer memory references
  - E. They are about the same

A

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i];
for(i = 0; i < 512; i++)
    e[i] /= d[i];
```

B

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
}
```

# Loop optimizations

A

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

## Loop interchange

B

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

B

```
double a[8192], b[8192], c[8192], \  
       d[8192], e[8192];  
for(i = 0; i < 512; i++) {  
    e[i] = (a[i] * b[i] + c[i])/d[i];  
}
```

## Loop fission

A

```
double a[8192], b[8192], c[8192], \  
       d[8192], e[8192];  
for(i = 0; i < 512; i++)  
    e[i] = a[i] * b[i] + c[i];  
for(i = 0; i < 512; i++)  
    e[i] /= d[i];
```

A

```
double a[8192], b[8192], c[8192], \  
       d[8192], e[8192];  
for(i = 0; i < 512; i++)  
    e[i] = a[i] * b[i] + c[i];  
for(i = 0; i < 512; i++)  
    e[i] /= d[i];
```

## Loop fusion

B

```
double a[8192], b[8192], c[8192], \  
       d[8192], e[8192];  
for(i = 0; i < 512; i++) {  
    e[i] = (a[i] * b[i] + c[i])/d[i];  
}
```

# Takeaways: Software Optimizations

- Data layout — capacity miss, conflict miss, compulsory miss
- Loop interchange — conflict/capacity miss
- Loop fission — conflict miss — when \$ has limited way associativity
- Loop fusion — capacity miss — when \$ has enough way associativity

# Outline

- Case study: matrix multiplications
- Optimizations for matrix multiplications

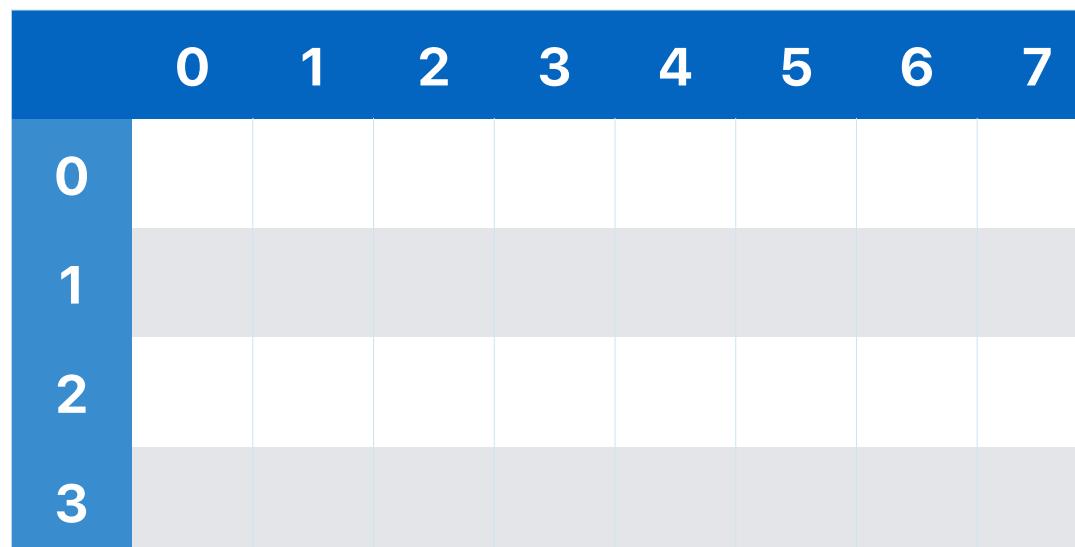
# **Case study: matrix multiplications**

# What is an M by N “2-D” array in C?

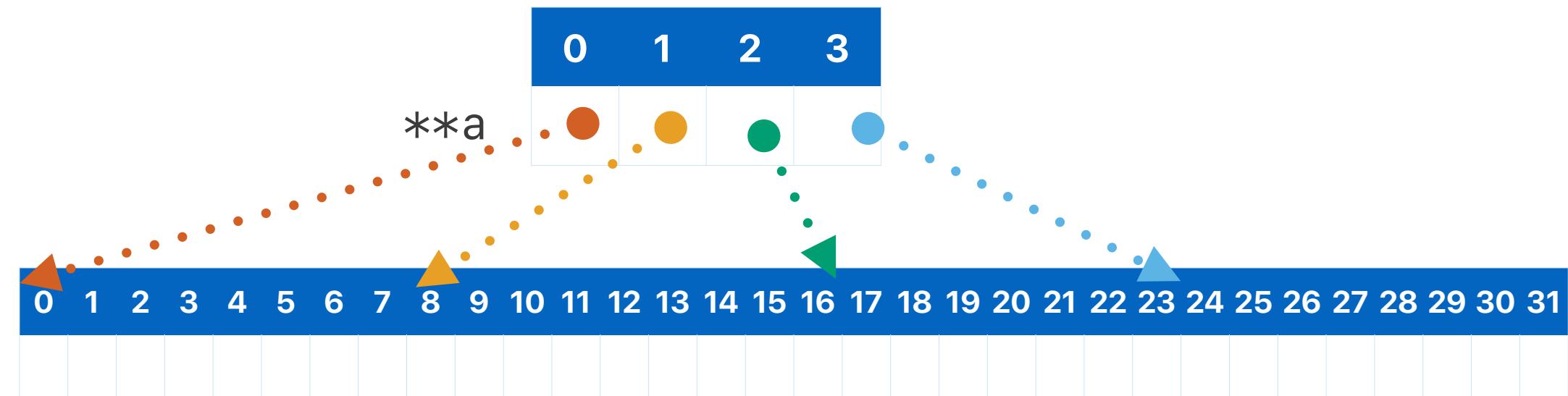
```
a = (double **)malloc(M*sizeof(double *));
for(i = 0; i < N; i++)
{
    a[i] = (double *)malloc(N*sizeof(double));
}
```

**a[i][j] is essentially a[i\*N+j]**

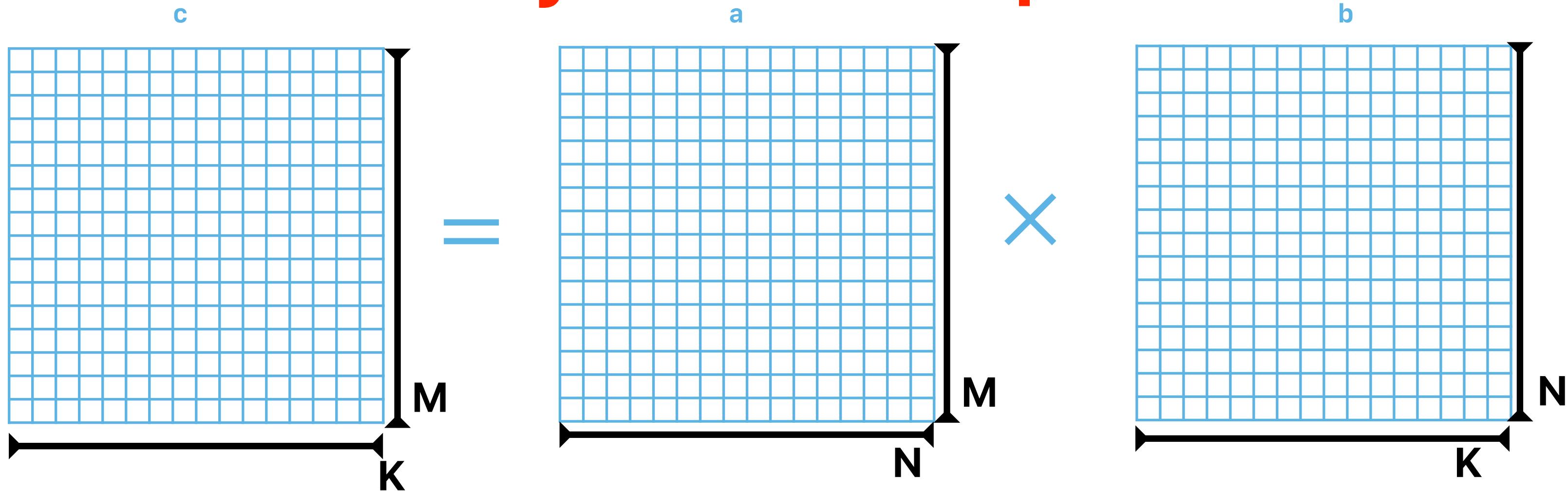
**abstraction**



**physical implementation**



# Case Study: Matrix Multiplications



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

**Algorithm class tells you it's  $O(n^3)$**   
**If  $M=N=K=1024$ , it takes about 2 sec**  
**How long is it take when  $M=N=K=2048$ ?**



Start the presentation to see live content. Still no live content? Install the app or get help at [PollEv.com/app](https://PollEv.com/app)



## What kind(s) of misses are there in Matrix Multiplications

- Considering the case where  $M=N=K=2048$ , what do you think the majority type(s) of cache misses are we seeing on an intel processor with intel Core i7 is 48 KB, 12-way, 64-byte blocked L1-\$?

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

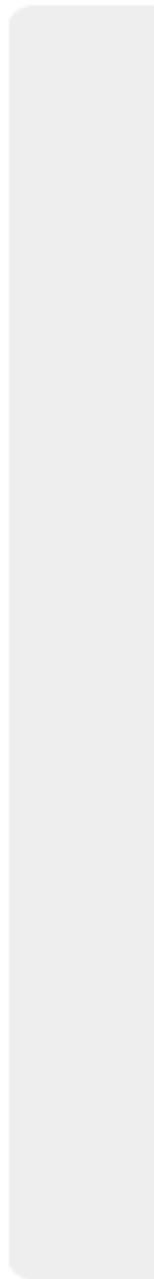
- A. Compulsory miss
- B. Capacity miss
- C. Conflict miss
- D. Capacity & conflict miss
- E. Compulsory & conflict miss



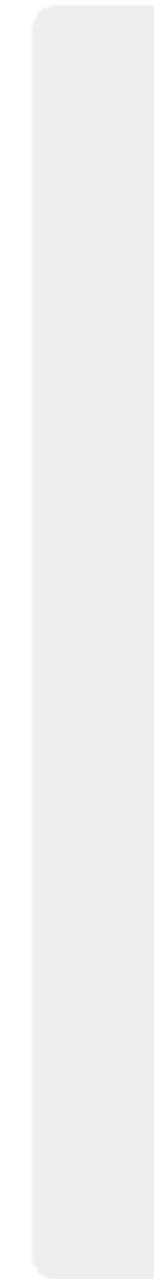
## Type of misses in MM

0

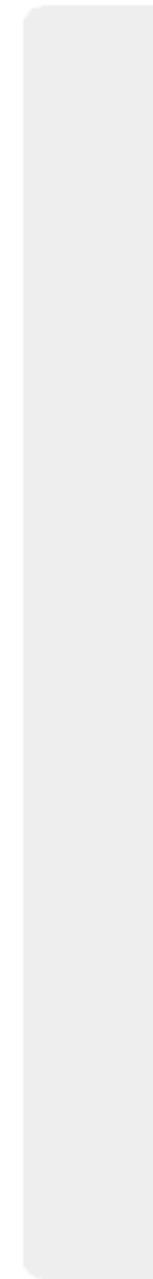
0%



0%



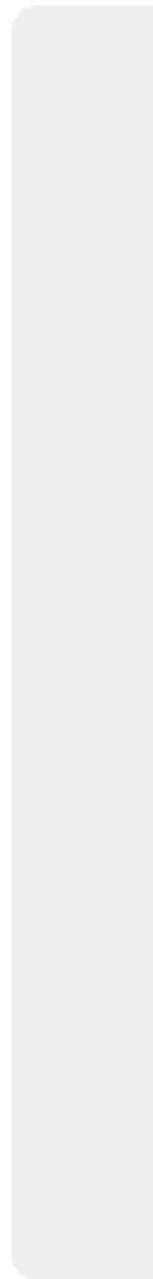
0%



0%



0%



A

B

C

D

E



## What kind(s) of misses are there in Matrix Multiplications

- Considering the case where  $M=N=K=2048$ , what do you think the majority type(s) of cache misses are we seeing on an intel processor with intel Core i7 is 48 KB, 12-way, 64-byte blocked L1-\$?

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

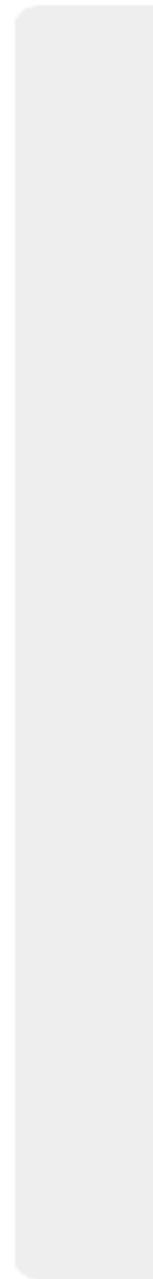
- A. Compulsory miss
- B. Capacity miss
- C. Conflict miss
- D. Capacity & conflict miss
- E. Compulsory & conflict miss



## Type of misses in MM – group

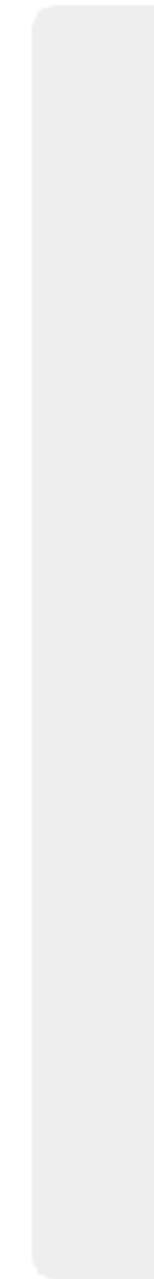
0

0%



A

0%



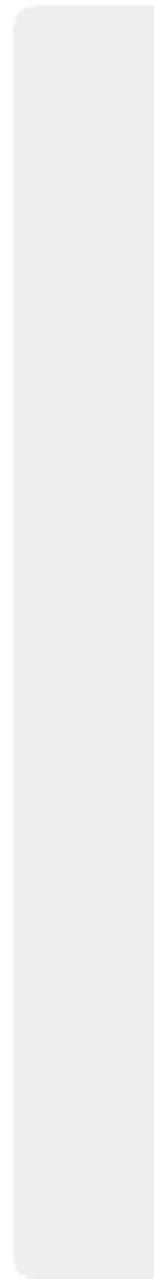
B

0%



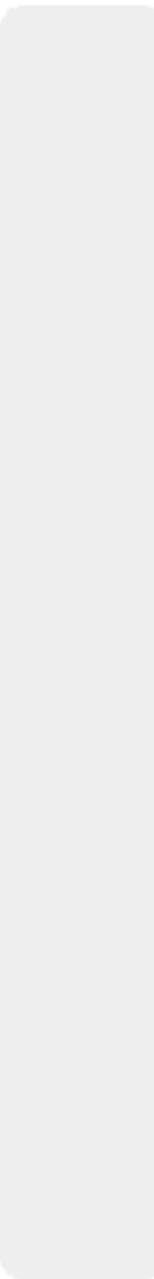
C

0%



D

0%



E

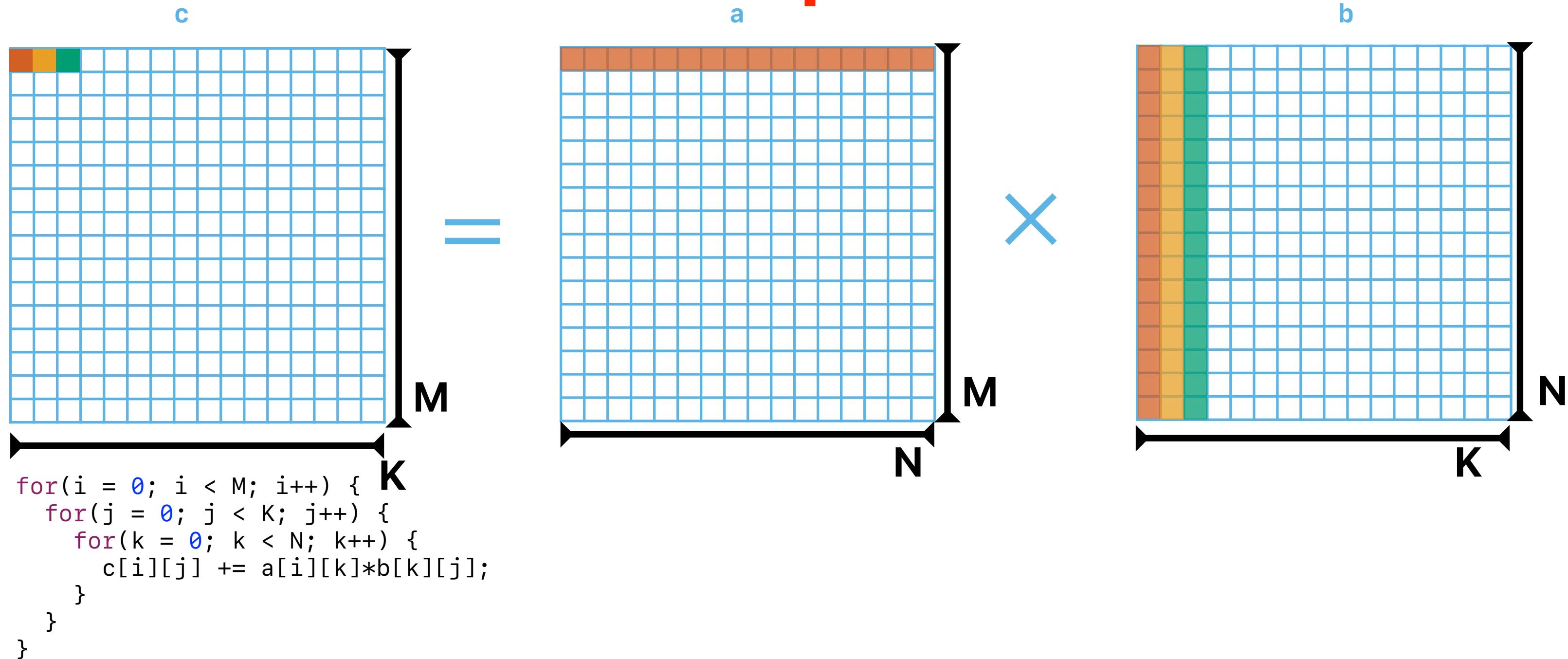
# What kind(s) of misses are there in Matrix Multiplications

- Considering the case where  $M=N=K=2048$ , what do you think the majority type(s) of cache misses are we seeing on an intel processor with intel Core i7 is 48 KB, 12-way, 64-byte blocked L1-\$?

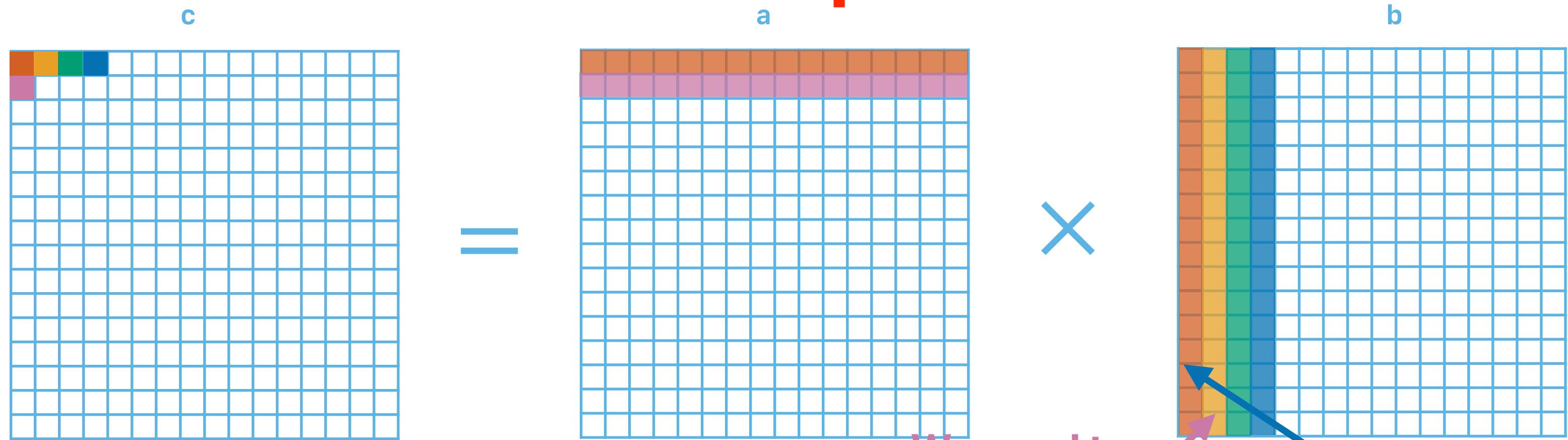
```
for(i = 0; i < M; i++) {  
    for(j = 0; j < K; j++) {  
        for(k = 0; k < N; k++) {  
            c[i][j] += a[i][k]*b[k][j];  
        }  
    }  
}
```

- A. Compulsory miss
- B. Capacity miss
- C. Conflict miss
- D. Capacity & conflict miss
- E. Compulsory & conflict miss

# Matrix Multiplications



# Matrix Multiplications



- If each dimension of your matrix is 2048
  - Each row takes  $2048 \times 8$  Bytes = 16 KB
  - Each column takes  $2048 \times 8$  Bytes = 16 KB
  - The L1-\$ of intel Core i7 is 48 KB
  - You can only hold at most 3 rows or 3 columns at most of each matrix!

We need to  
fetch  
everything  
again —  
capacity miss!

Unlikely to be  
kept in the  
cache

# Ideas regarding reducing misses in matrix multiplications

- Reducing capacity misses — we need to reduce the length of a row that we visit within a period of time

# What kind(s) of misses are there in Matrix Multiplications

- Considering the case where  $M=N=K=2048$ , what do you think the majority type(s) of cache misses are we seeing on an intel processor with intel Core i7 is 48 KB, 12-way, 64-byte blocked L1-\$?

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

- A. Compulsory miss
- B. Capacity miss
- C. Conflict miss
- D. Capacity & conflict miss
- E. Compulsory & conflict miss

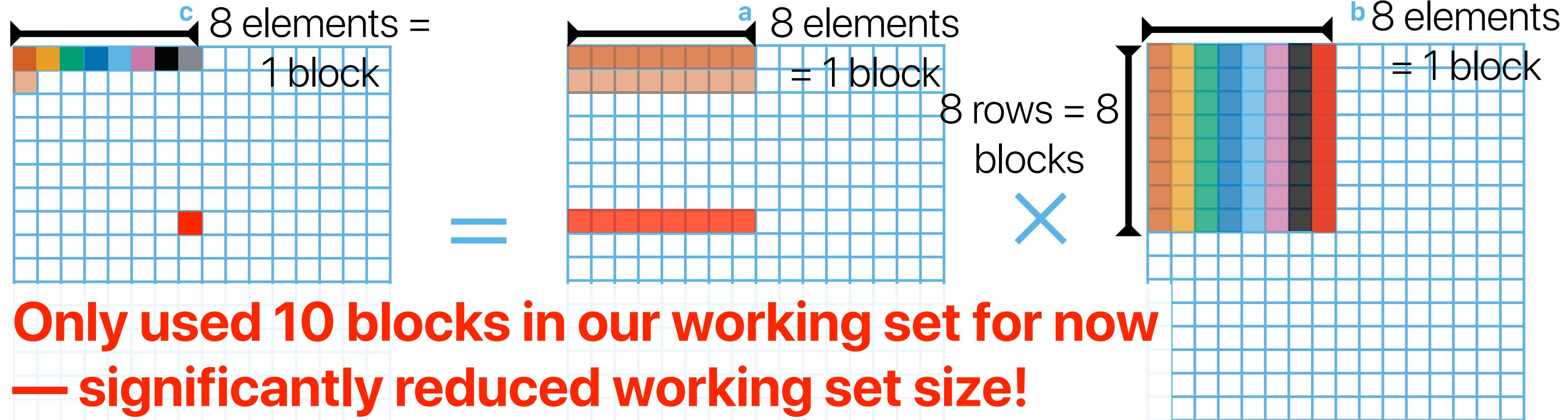
# Mathematical view of MM

$$\begin{aligned} c_{i,j} &= \sum_{k=0}^{k=N-1} a_{i,k} \times b_{k,j} = \sum_{k=0}^{k=\frac{N}{2}-1} a_{i,k} \times b_{k,j} + \sum_{k=\frac{N}{2}}^{k=N-1} a_{i,k} \times b_{k,j} \\ &= \sum_{k=0}^{k=\frac{N}{4}-1} a_{i,k} \times b_{k,j} + \sum_{k=\frac{N}{4}}^{k=\frac{N}{2}-1} a_{i,k} \times b_{k,j} + \sum_{k=\frac{N}{2}}^{k=\frac{3N}{4}-1} a_{i,k} \times b_{k,j} + \sum_{k=3N4-1}^{k=N-1} a_{i,k} \times b_{k,j} \end{aligned}$$

⋮

**Let's break up the multiplications and accumulations  
into something fits in the cache well**

# Tiling/Blocking Algorithm for Matrix Multiplications

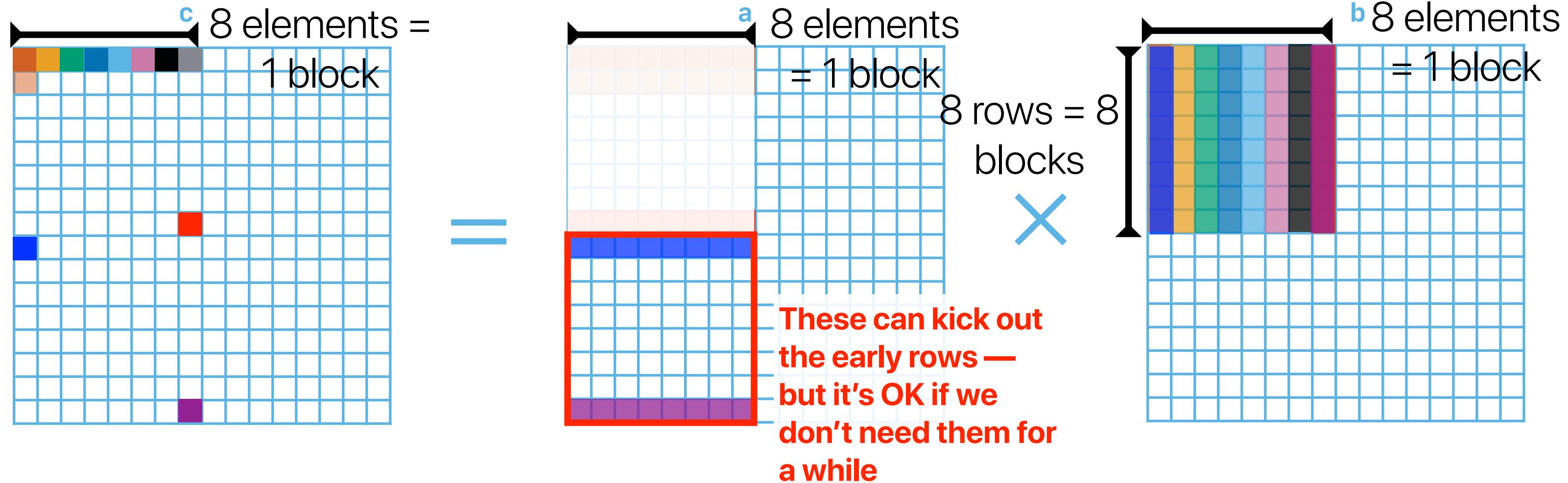


**Only compulsory misses —**

$$\text{miss\_rate} = \frac{\text{total misses}}{\text{total accesses}} = \frac{8 + 8 + 8}{3 \times 8 \times 8 \times 8} = 0.015625$$

**These are still around when we move to the next row in the “tile”**

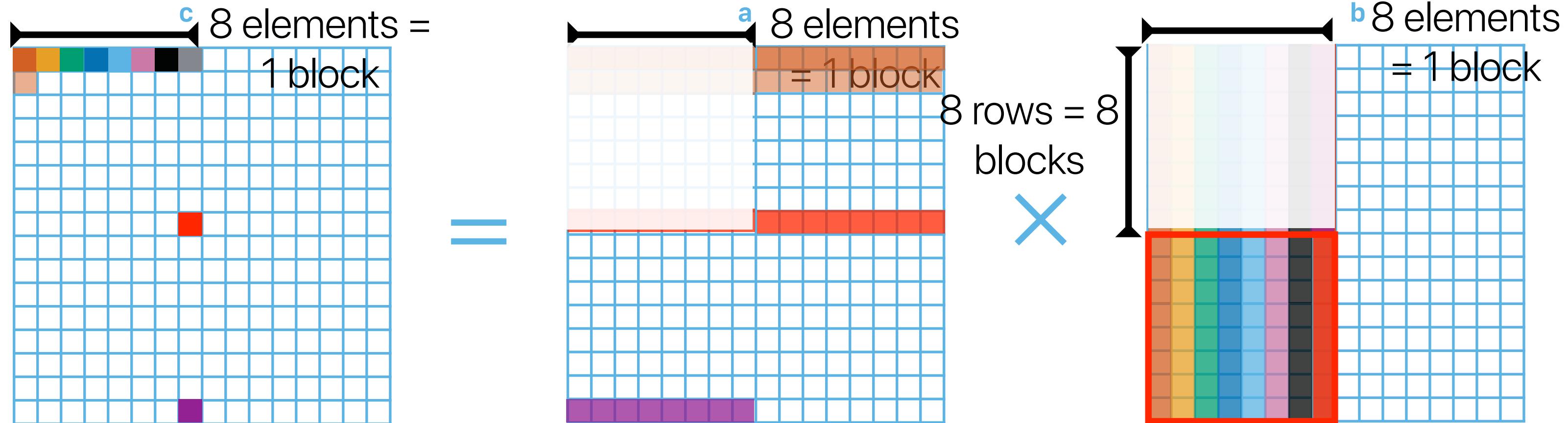
# Tiling/Blocking Algorithm for Matrix Multiplications



**Bringing miss rate even further  
lower now —**

$$\text{miss\_rate} = \frac{\text{total misses}}{\text{total accesses}} = \frac{8 + 2 \times 8 + 8}{2 \times 3 \times 8 \times 8 \times 8} = 0.0104$$

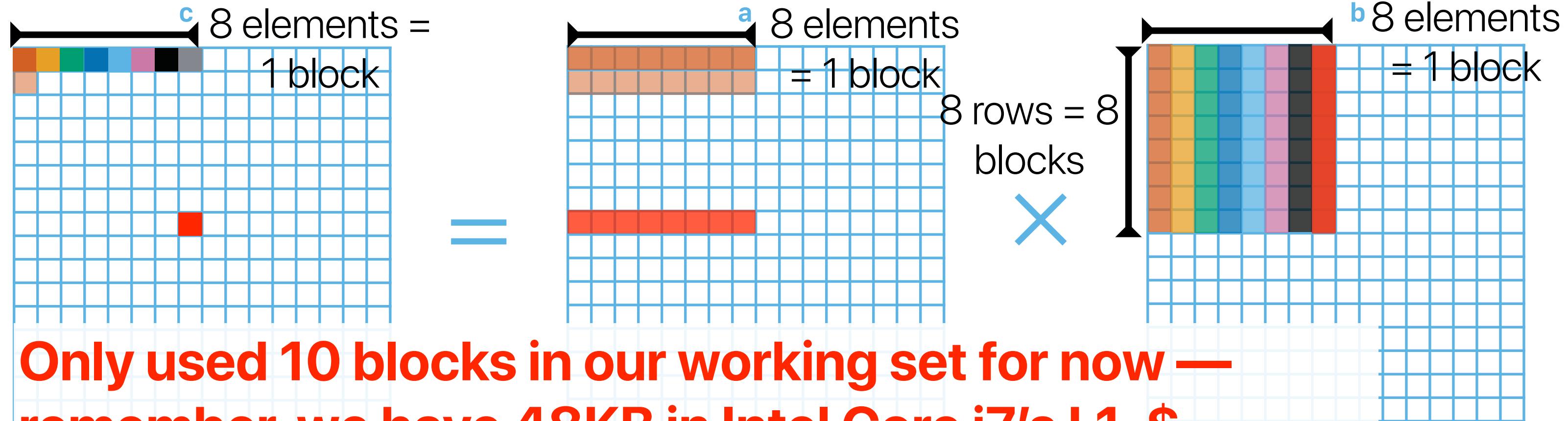
# Tiling/Blocking Algorithm for Matrix Multiplications



```
for(i = 0; i < M; i+=tile_size)
    for(j = 0; j < K; j+=tile_size)
        for(k = 0; k < N; k+=tile_size)
            for(ii = i; ii < i+tile_size; ii++)
                for(jj = j; jj < j+tile_size; jj++)
                    for(kk = k; kk < k+tile_size; kk++)
                        c[ii][jj] += a[ii][kk]*b[kk][jj];
```

These can kick out  
the upper portion  
of the columns—  
but it's OK if we  
don't need them for  
a while

# Tiling/Blocking Algorithm for Matrix Multiplications



Only used 10 blocks in our working set for now —  
remember, we have 48KB in Intel Core i7's L1-\$

We can have at most  $\frac{48KB}{3} = 16KB$  for each matrix  
“tile”

So, let's pick tile size to be 32 as

$$\sqrt{\frac{48KB}{3 \times 8bytes}} \approx 45$$



Start the presentation to see live content. Still no live content? Install the app or get help at [PollEv.com/app](https://PollEv.com/app)



# What if we partition based on cache capacity?

- Considering the case where  $M=N=K=2048$ , and a `tile_size=32`, what do you think the majority type(s) of cache misses are we seeing on an intel processor with intel Core i7 is 48 KB, 12-way, 64-byte blocked L1-\$?

```
for(i = 0; i < M; i+=tile_size)
    for(j = 0; j < K; j+=tile_size)
        for(k = 0; k < N; k+=tile_size)
            for(ii = i; ii < i+tile_size; ii++)
                for(jj = j; jj < j+tile_size; jj++)
                    for(kk = k; kk < k+tile_size; kk++)
                        c[ii][jj] += a[ii][kk]*b[kk][jj];
```

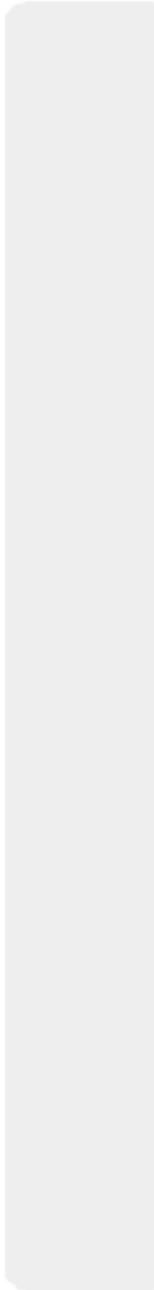
- A. Compulsory miss
- B. Capacity miss
- C. Conflict miss
- D. Capacity & conflict miss
- E. Compulsory & conflict miss



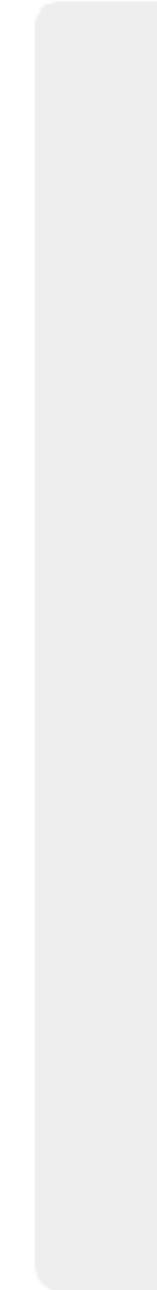
## Tilesizes in MM

0

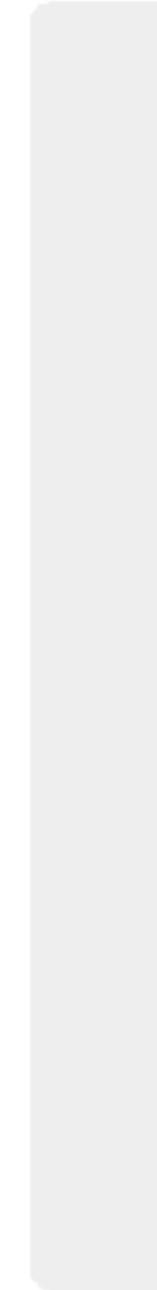
0%



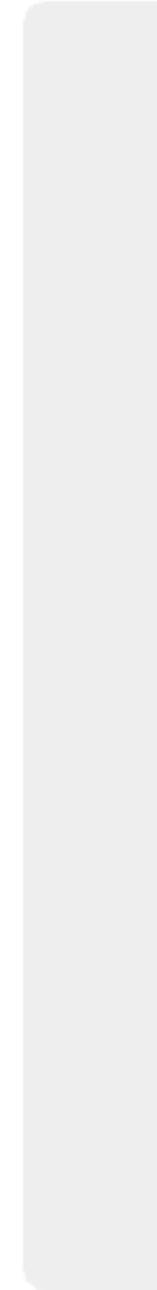
0%



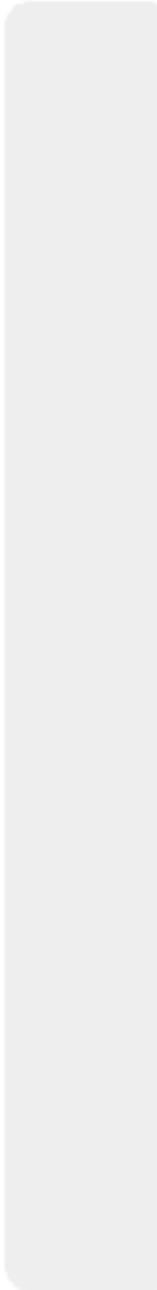
0%



0%



0%



A

B

C

D

E



# What if we partition based on cache capacity?

- Considering the case where  $M=N=K=2048$ , and a `tile_size=32`, what do you think the majority type(s) of cache misses are we seeing on an intel processor with intel Core i7 is 48 KB, 12-way, 64-byte blocked L1-\$?

```
for(i = 0; i < M; i+=tile_size)
    for(j = 0; j < K; j+=tile_size)
        for(k = 0; k < N; k+=tile_size)
            for(ii = i; ii < i+tile_size; ii++)
                for(jj = j; jj < j+tile_size; jj++)
                    for(kk = k; kk < k+tile_size; kk++)
                        c[ii][jj] += a[ii][kk]*b[kk][jj];
```

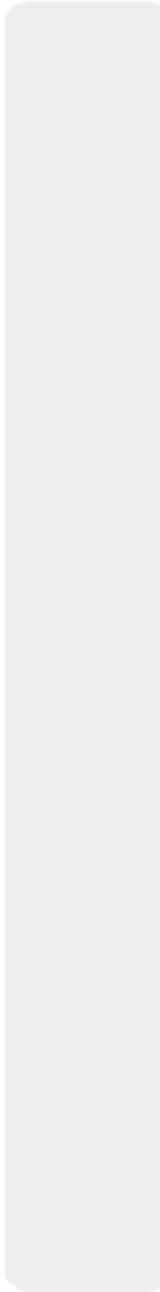
- A. Compulsory miss
- B. Capacity miss
- C. Conflict miss
- D. Capacity & conflict miss
- E. Compulsory & conflict miss



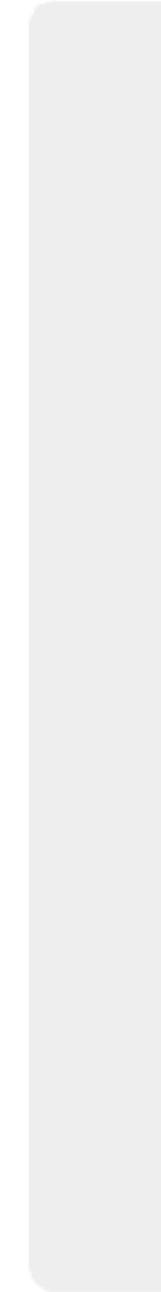
## Tile sizes in MM – group

0

0%



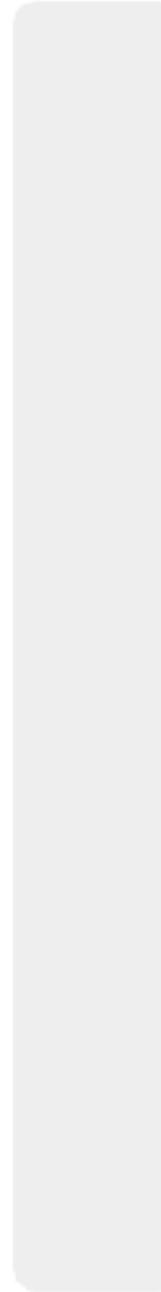
0%



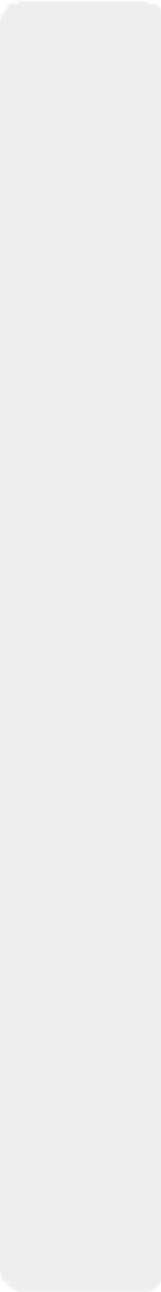
0%



0%



0%



A

B

C

D

E

# What if we partition based on cache capacity?

- Considering the case where  $M=N=K=2048$ , and a `tile_size=32`, what do you think the majority type(s) of cache misses are we seeing on an intel processor with intel Core i7 is 48 KB, 12-way, 64-byte blocked L1-\$?

```
for(i = 0; i < M; i+=tile_size)
    for(j = 0; j < K; j+=tile_size)
        for(k = 0; k < N; k+=tile_size)
            for(ii = i; ii < i+tile_size; ii++)
                for(jj = j; jj < j+tile_size; jj++)
                    for(kk = k; kk < k+tile_size; kk++)
                        c[ii][jj] += a[ii][kk]*b[kk][jj];
```

- A. Compulsory miss
- B. Capacity miss
- C. Conflict miss
- D. Capacity & conflict miss
- E. Compulsory & conflict miss

# Matrix Multiplication — let's consider "b"

```
for(ii = i; ii < i+tile_size; ii++)  
    for(jj = j; jj < j+tile_size; jj++)  
        for(kk = k; kk < k+tile_size; kk++)  
            c[ii][jj] += a[ii][kk]*b[kk][jj];
```

	Address	Tag	Index
b[0][0]	0x20000	0x20	0x0
b[1][0]	0x24000	0x24	0x0
b[2][0]	0x28000	0x28	0x0
b[3][0]	0x2C000	0x2C	0x0
b[4][0]	0x30000	0x30	0x0
b[5][0]	0x34000	0x34	0x0
b[6][0]	0x38000	0x38	0x0
b[7][0]	0x3C000	0x3C	0x0
b[8][0]	0x40000	0x40	0x0
b[9][0]	0x44000	0x44	0x0
b[10][0]	0x48000	0x48	0x0
b[11][0]	0x4C000	0x4C	0x0
b[12][0]	0x50000	0x50	0x0
b[13][0]	0x54000	0x54	0x0
b[14][0]	0x58000	0x58	0x0
b[15][0]	0x5C000	0x5C	0x0
b[16][0]	0x60000	0x60	0x0

- If the row dimension (N) of your matrix is 2048, each row element with the same column index is

$2048 \times 8 = 16384 = 0x4000$  away from each other

Each set can store only 12 blocks! So we will start to kick out b[0][0-7], b[1][0-7] ...

# Now, when we work on c[0][1]

	Address	Tag	Index			
b[0][0]	0x20000	0x20	0x0			
b[1][0]	0x24000	0x24	0x0			
b[2][0]	0x28000	0x28	0x0			
b[3][0]	0x2C000	0x2C	0x0			
b[4][0]	0x30000	0x30	0x0			
b[5][0]	0x34000	0x34	0x0			
b[6][0]	0x38000	0x38	0x0			
b[7][0]	0x3C000	0x3C	0x0			
b[8][0]	0x40000	0x40	0x0			
b[9][0]	0x44000	0x44	0x0			
b[10][0]	0x48000	0x48	0x0			
b[11][0]	0x4C000	0x4C	0x0			
b[12][0]	0x50000	0x50	0x0			
b[13][0]	0x54000	0x54	0x0			
b[14][0]	0x58000	0x58	0x0			
b[15][0]	0x5C000	0x5C	0x0			
b[16][0]	0x60000	0x60	0x0			
b[0][1]	0x20008	0x20	0x0	Conflict	Miss	
b[1][1]	0x24008	0x24	0x0	Conflict	Miss	
b[2][1]	0x28008	0x28	0x0	Conflict	Miss	
b[3][1]	0x2C008	0x2C	0x0	Conflict	Miss	
b[4][1]	0x30008	0x30	0x0	Conflict	Miss	
b[5][1]	0x34008	0x34	0x0	Conflict	Miss	
b[6][1]	0x38008	0x38	0x0	Conflict	Miss	
b[7][1]	0x3C008	0x3C	0x0	Conflict	Miss	
b[8][1]	0x40008	0x40	0x0	Conflict	Miss	
b[9][1]	0x44008	0x44	0x0	Conflict	Miss	
b[10][1]	0x48008	0x48	0x0	Conflict	Miss	
b[11][1]	0x4C008	0x4C	0x0	Conflict	Miss	
b[12][1]	0x50008	0x50	0x0	Conflict	Miss	
b[13][1]	0x54008	0x54	0x0	Conflict	Miss	
b[14][1]	0x58008	0x58	0x0	Conflict	Miss	
b[15][1]	0x5C008	0x5C	0x0	Conflict	Miss	
b[16][1]	0x60008	0x60	0x0	Conflict	Miss	

Each set can store only 12 blocks! So  
we will start to kick out b[0][0-7], b[1][0-7] ...

# How large a tile should be?

- Considering the case where  $M=N=K=2048$ , and a `tile_size=16`, what do you think the majority type(s) of cache misses are we seeing on an intel processor with intel Core i7 is 48 KB, 12-way, 64-byte blocked L1-\$?

```
for(i = 0; i < M; i+=tile_size)
    for(j = 0; j < K; j+=tile_size)
        for(k = 0; k < N; k+=tile_size)
            for(ii = i; ii < i+tile_size; ii++)
                for(jj = j; jj < j+tile_size; jj++)
                    for(kk = k; kk < k+tile_size; kk++)
                        c[ii][jj] += a[ii][kk]*b[kk][jj];
```

- A. Compulsory miss
- B. Capacity miss
- C. Conflict miss
- D. Capacity & conflict miss
- E. Compulsory & conflict miss

# Matrix Multiplication — let's consider "b"

```
for(ii = i; ii < i+tile_size; ii++)  
    for(jj = j; jj < j+tile_size; jj++)  
        for(kk = k; kk < k+tile_size; kk++)  
            c[ii][jj] += a[ii][kk]*b[kk][jj];
```

- If the row dimension of your matrix is 2048, each row element with the same column index is

$$2048 \times 8 = 16384 = 0x4000$$

away from each other

	Address	Tag	Index
b[0][0]	0x20000	0x20	0x0
b[1][0]	0x24000	0x24	0x0
b[2][0]	0x28000	0x28	0x0
b[3][0]	0x2C000	0x2C	0x0
b[4][0]	0x30000	0x30	
b[5][0]	0x34000	0x34	
b[6][0]	0x38000	0x38	
b[7][0]	0x3C000	0x3C	
b[8][0]	0x40000	0x40	
b[9][0]	0x44000	0x44	
b[10][0]	0x48000	0x48	
b[11][0]	0x4C000	0x4C	
b[12][0]	0x50000	0x50	
b[13][0]	0x54000	0x54	
b[14][0]	0x58000	0x58	
b[15][0]	0x5C000	0x5C	
b[16][0]	0x60000	0x60	0x0

If we stop at somewhere before 12 blocks,  
we should be fine!

Since each block has 8 elements, let's break  
up in 8 for now

— 8 elements from a[i]

— 8 columns each covers 8 rows



Start the presentation to see live content. Still no live content? Install the app or get help at [PollEv.com/app](https://PollEv.com/app)

## Ideas regarding reducing misses in matrix multiplications

- Reducing capacity misses — we need to reduce the length of a row that we visit within a period of time
- Reducing conflict misses — we need to ensure an appropriate tile size would not lead to conflict in sets

# Use registers wisely

```
for(i = 0; i < M; i+=tile_size)
    for(j = 0; j < K; j+=tile_size)
        for(k = 0; k < N; k+=tile_size)
            for(ii = i; ii < i+tile_size; ii++)
                for(jj = j; jj < j+tile_size; jj++)
                    for(kk = k; kk < k+tile_size; kk++)
                        c[ii][jj] += a[ii][kk]*b_t[jj][kk];
```

This will create a memory access!

```
for(i = 0; i < M; i+=tile_size)
    for(j = 0; j < K; j+=tile_size)
        for(k = 0; k < N; k+=tile_size)
            for(ii = i; ii < i+tile_size; ii++)
                for(jj = j; jj < j+tile_size; jj++)
                {
                    result = 0;
                    for(kk = k; kk < k+tile_size; kk++)
                        result += a[ii][kk]*b_t[jj][kk];
                    c[ii][jj] += result;
```

The compiler will try to make result in a register

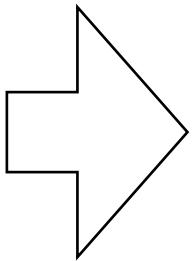
— without writing code in this way, compiler may not optimize



Start the presentation to see live content. Still no live content? Install the app or get help at [PollEv.com/app](https://PollEv.com/app)

# Matrix Transpose

```
for(i = 0; i < M; i+=tile_size) {  
    for(j = 0; j < N; j+=tile_size) {  
        for(k = 0; k < K; k+=tile_size) {  
            for(ii = i; ii < i+tile_size; ii++)  
                for(jj = j; jj < j+tile_size; jj++)  
                    for(kk = k; kk < k+tile_size; kk++)  
                        c[ii][jj] += a[ii][kk]*b[kk][jj];  
    }  
}
```



```
// Transpose matrix b into b_t  
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {  
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {  
        b_t[i][j] += b[j][i];  
    }  
}  
  
for(i = 0; i < M; i+=tile_size) {  
    for(j = 0; j < N; j+=tile_size) {  
        for(k = 0; k < K; k+=tile_size) {  
            for(ii = i; ii < i+tile_size; ii++)  
                for(jj = j; jj < j+tile_size; jj++)  
                    for(kk = k; kk < k+tile_size; kk++)  
                        // Compute on b_t  
                        c[ii][jj] += a[ii][kk]*b_t[jj][kk];  
    }  
}
```



Start the presentation to see live content. Still no live content? Install the app or get help at [PollEv.com/app](https://PollEv.com/app)



# What kind(s) of misses can matrix transpose remove?

- By transposing a matrix, the performance of matrix multiplication can be further improved. What kind(s) of cache misses does matrix transpose help to remove?

```
for(i = 0; i < M; i+=tile_size) {  
    for(j = 0; j < K; j+=tile_size) {  
        for(k = 0; k < N; k+=tile_size) {  
            for(ii = i; ii < i+tile_size; ii++)  
                for(jj = j; jj < j+tile_size; jj++)  
                    for(kk = k; kk < k+tile_size; kk++)  
                        c[ii][jj] += a[ii][kk]*b[kk][jj];  
        }  
    }  
}
```

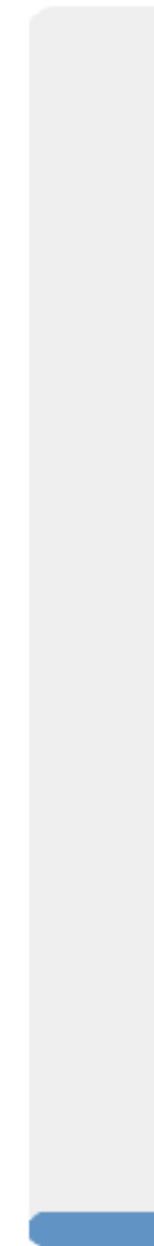
- A. Compulsory miss
- B. Capacity miss
- C. Conflict miss
- D. Capacity & conflict miss
- E. Compulsory & conflict miss

```
// Transpose matrix b into b_t  
for(i = 0; i < ARRAY_SIZE; i++) {  
    for(j = 0; j < ARRAY_SIZE; j++) {  
        b_t[i][j] += b[j][i];  
    }  
}  
  
for(i = 0; i < M; i+=tile_size) {  
    for(j = 0; j < K; j+=tile_size) {  
        for(k = 0; k < N; k+=tile_size) {  
            for(ii = i; ii < i+tile_size; ii++)  
                for(jj = j; jj < j+tile_size; jj++)  
                    for(kk = k; kk < k+tile_size; kk++)  
                        // Compute on b_t  
                        c[ii][jj] += a[ii][kk]*b_t[jj][kk];  
        }  
    }  
}
```

Block + Transpose

 0

0



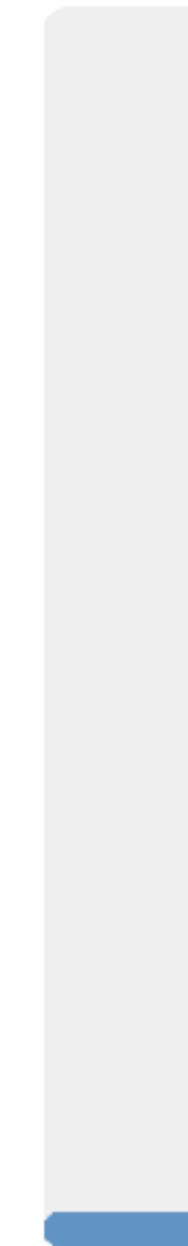
A

0



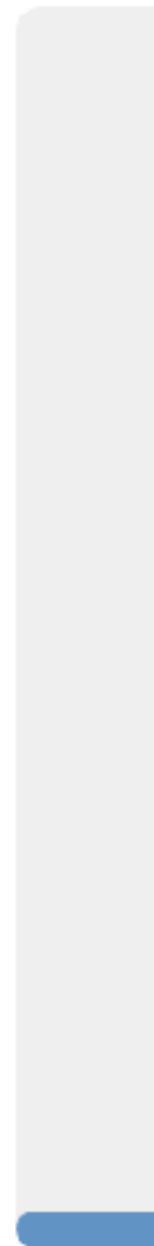
B

0



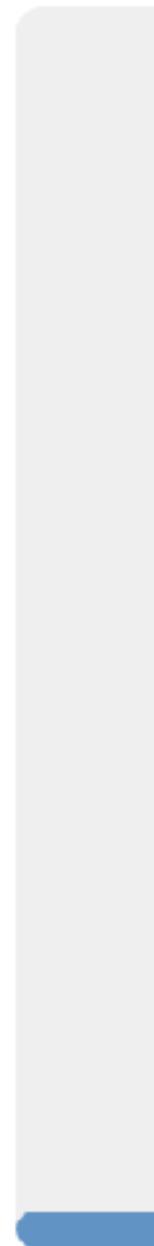
C

0



D

0



E



# What kind(s) of misses can matrix transpose remove?

- By transposing a matrix, the performance of matrix multiplication can be further improved. What kind(s) of cache misses does matrix transpose help to remove?

```
for(i = 0; i < M; i+=tile_size) {  
    for(j = 0; j < K; j+=tile_size) {  
        for(k = 0; k < N; k+=tile_size) {  
            for(ii = i; ii < i+tile_size; ii++)  
                for(jj = j; jj < j+tile_size; jj++)  
                    for(kk = k; kk < k+tile_size; kk++)  
                        c[ii][jj] += a[ii][kk]*b[kk][jj];  
        }  
    }  
}
```

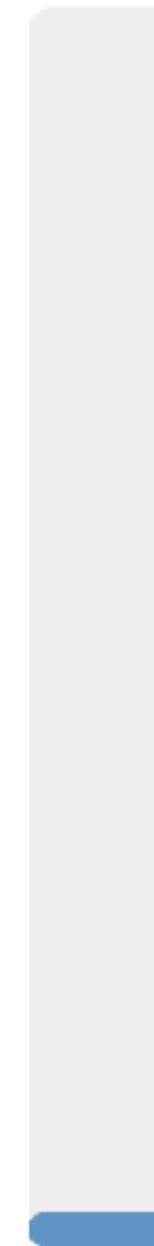
- A. Compulsory miss
- B. Capacity miss
- C. Conflict miss
- D. Capacity & conflict miss
- E. Compulsory & conflict miss

Block + Transpose

```
// Transpose matrix b into b_t  
for(i = 0; i < ARRAY_SIZE; i++) {  
    for(j = 0; j < ARRAY_SIZE; j++) {  
        b_t[i][j] += b[j][i];  
    }  
}  
  
for(i = 0; i < M; i+=tile_size) {  
    for(j = 0; j < K; j+=tile_size) {  
        for(k = 0; k < N; k+=tile_size) {  
            for(ii = i; ii < i+tile_size; ii++)  
                for(jj = j; jj < j+tile_size; jj++)  
                    for(kk = k; kk < k+tile_size; kk++)  
                        // Compute on b_t  
                        c[ii][jj] += a[ii][kk]*b_t[jj][kk];  
        }  
    }  
}
```

 0

0



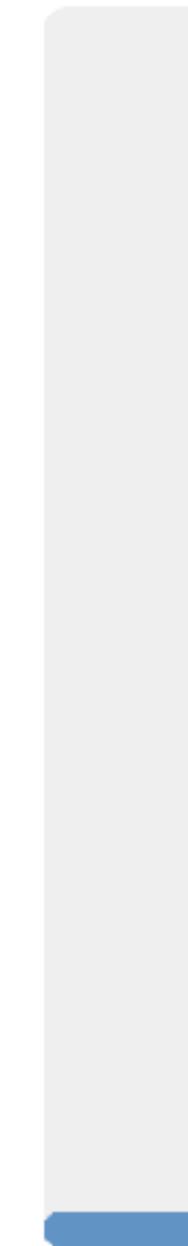
A

0



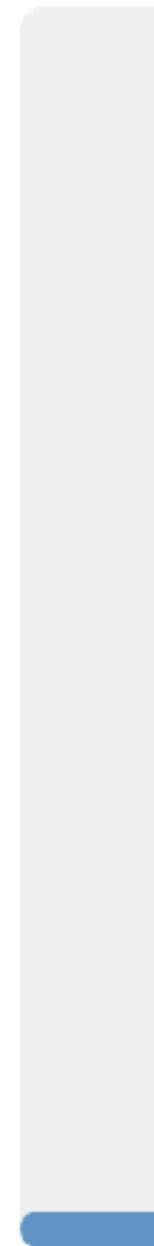
B

0



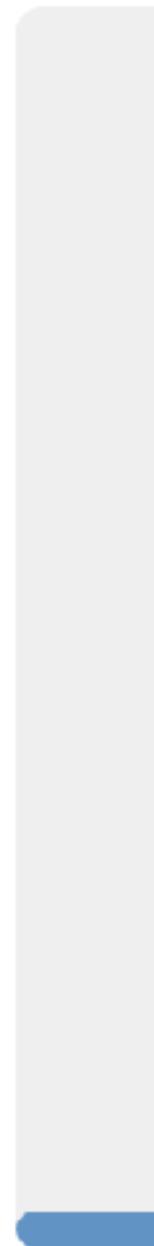
C

0



D

0



E

# What kind(s) of misses can matrix transpose remove?

- By transposing a matrix, the performance of matrix multiplication can be further improved. What kind(s) of cache misses does matrix transpose help to remove?

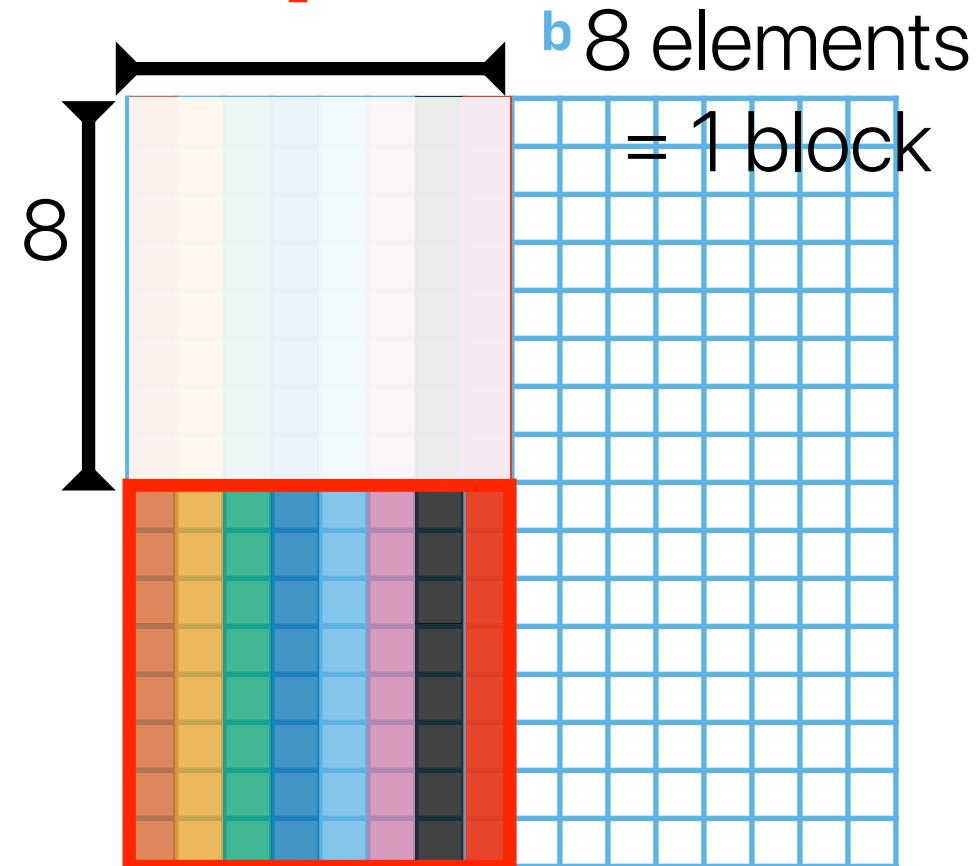
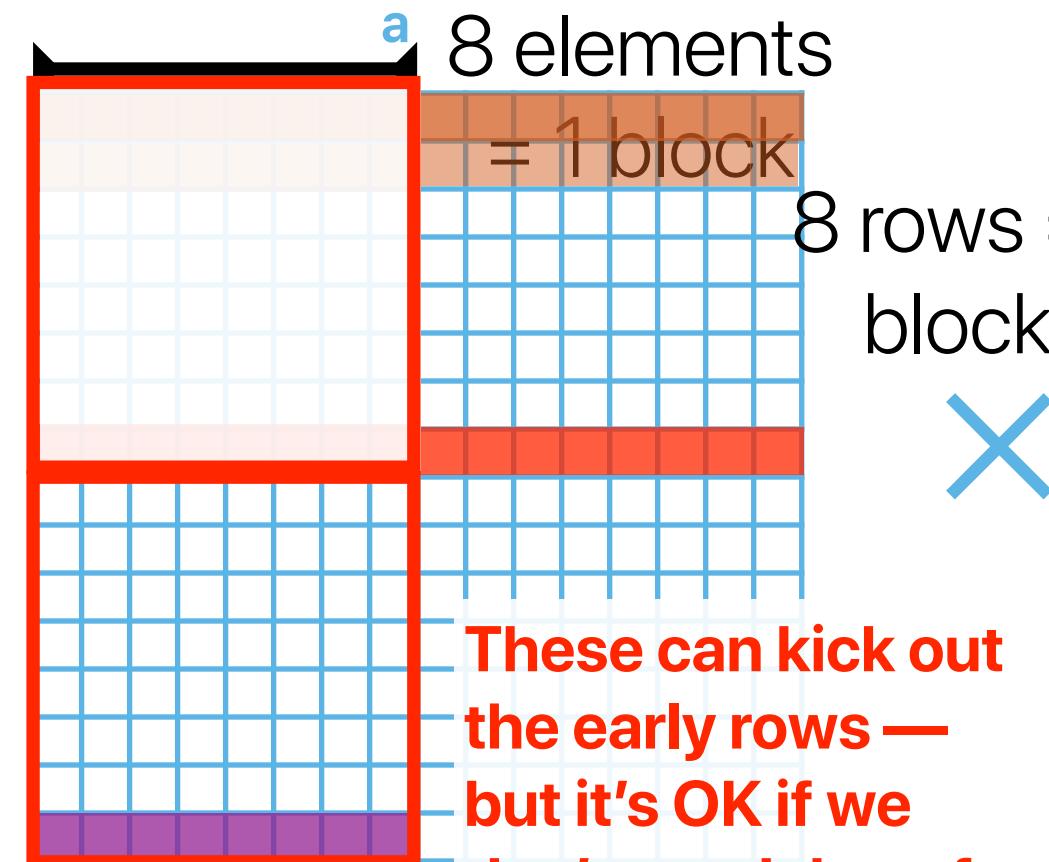
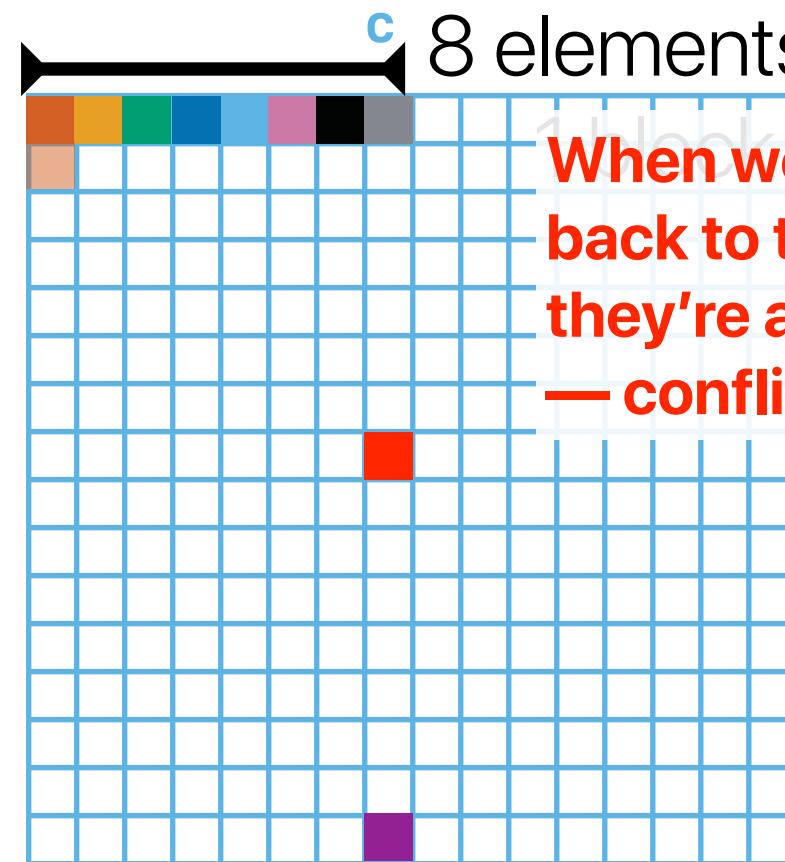
```
for(i = 0; i < M; i+=tile_size) {  
    for(j = 0; j < K; j+=tile_size) {  
        for(k = 0; k < N; k+=tile_size) {  
            for(ii = i; ii < i+tile_size; ii++)  
                for(jj = j; jj < j+tile_size; jj++)  
                    for(kk = k; kk < k+tile_size; kk++)  
                        c[ii][jj] += a[ii][kk]*b[kk][jj];  
        }  
    }  
}
```

- A. Compulsory miss
- B. Capacity miss
- C. Conflict miss
- D. Capacity & conflict miss
- E. Compulsory & conflict miss

```
// Transpose matrix b into b_t  
for(i = 0; i < ARRAY_SIZE; i++) {  
    for(j = 0; j < ARRAY_SIZE; j++) {  
        b_t[i][j] += b[j][i];  
    }  
}  
  
for(i = 0; i < M; i+=tile_size) {  
    for(j = 0; j < K; j+=tile_size) {  
        for(k = 0; k < N; k+=tile_size) {  
            for(ii = i; ii < i+tile_size; ii++)  
                for(jj = j; jj < j+tile_size; jj++)  
                    for(kk = k; kk < k+tile_size; kk++)  
                        // Compute on b_t  
                        c[ii][jj] += a[ii][kk]*b_t[jj][kk];  
        }  
    }  
}
```

Block + Transpose

# Tiling/Blocking Algorithm for Matrix Multiplications



```
for(i = 0; i < M; i+=tile_size)
    for(j = 0; j < K; j+=tile_size)
        for(k = 0; k < N; k+=tile_size)
            for(ii = i; ii < i+tile_size; ii++)
                for(jj = j; jj < j+tile_size; jj++)
                    for(kk = k; kk < k+tile_size; kk++)
                        c[ii][jj] += a[ii][kk]*b[kk][jj];
```

# What kind(s) of misses can matrix transpose remove?

- By transposing a matrix, the performance of matrix multiplication can be further improved. What kind(s) of cache misses does matrix transpose help to remove?

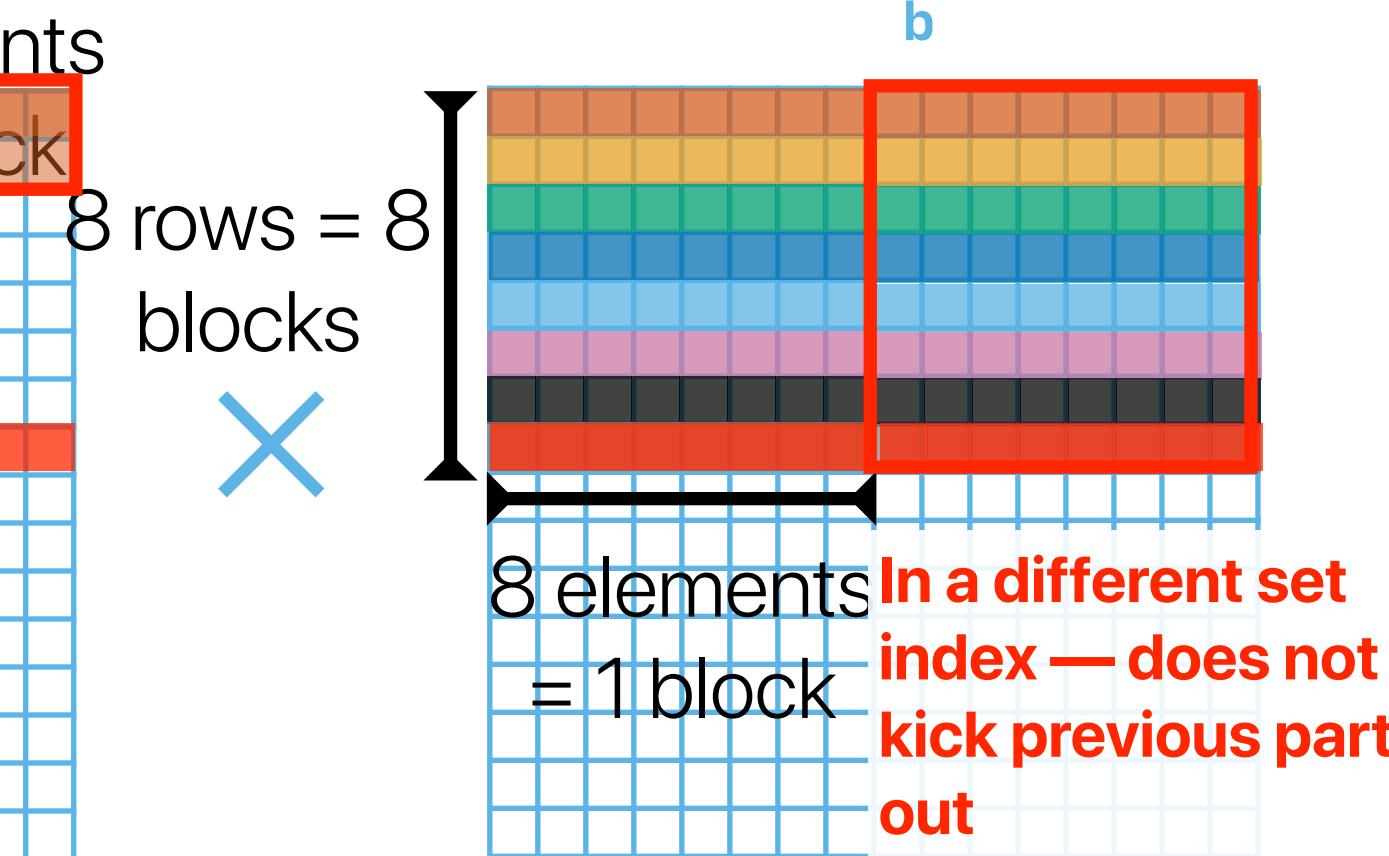
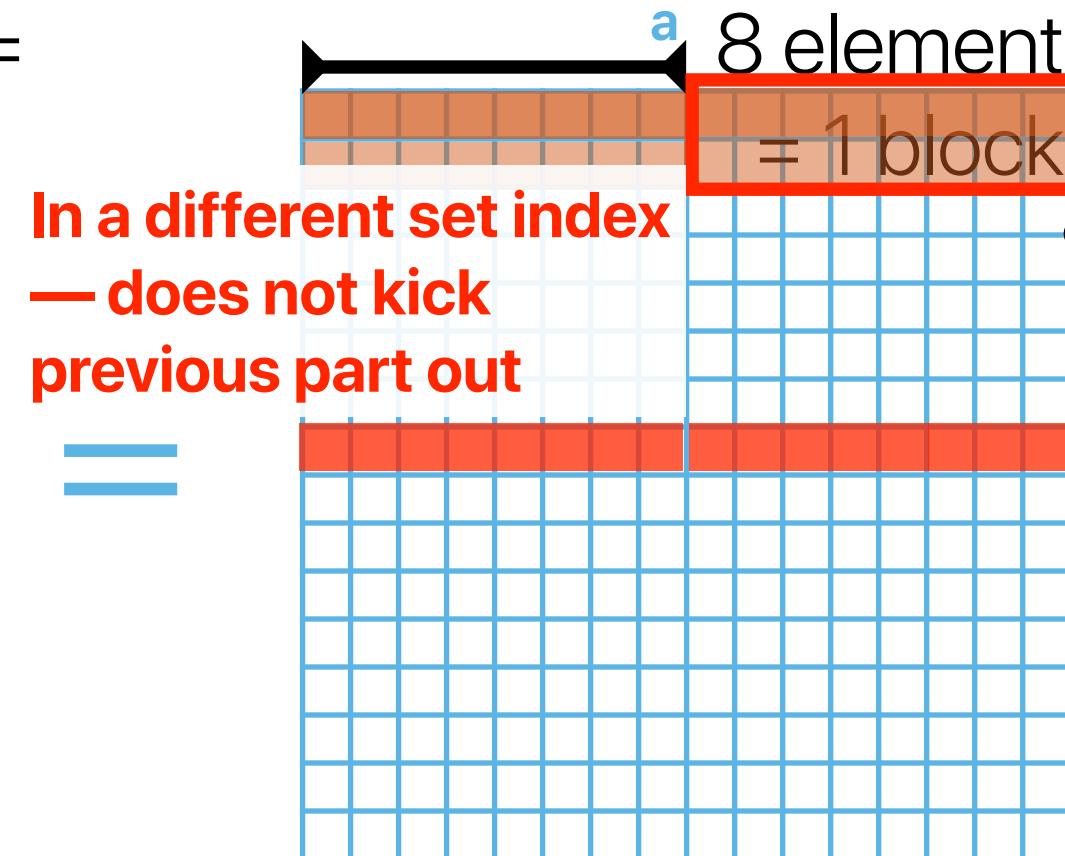
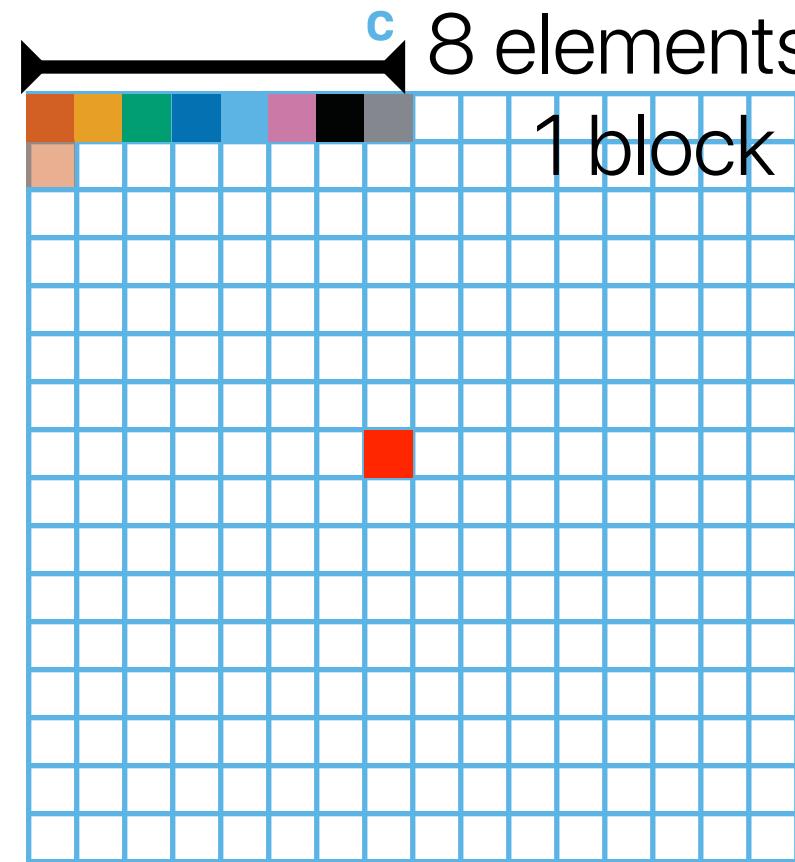
```
for(i = 0; i < M; i+=tile_size) {  
    for(j = 0; j < K; j+=tile_size) {  
        for(k = 0; k < N; k+=tile_size) {  
            for(ii = i; ii < i+tile_size; ii++)  
                for(jj = j; jj < j+tile_size; jj++)  
                    for(kk = k; kk < k+tile_size; kk++)  
                        c[ii][jj] += a[ii][kk]*b[kk][jj];  
        }  
    }  
}
```

- A. Compulsory miss
- B. Capacity miss
- C. Conflict miss
- D. Capacity & conflict miss
- E. Compulsory & conflict miss

```
// Transpose matrix b into b_t  
for(i = 0; i < ARRAY_SIZE; i++) {  
    for(j = 0; j < ARRAY_SIZE; j++) {  
        b_t[i][j] += b[j][i];  
    }  
}  
  
for(i = 0; i < M; i+=tile_size) {  
    for(j = 0; j < K; j+=tile_size) {  
        for(k = 0; k < N; k+=tile_size) {  
            for(ii = i; ii < i+tile_size; ii++)  
                for(jj = j; jj < j+tile_size; jj++)  
                    for(kk = k; kk < k+tile_size; kk++)  
                        // Compute on b_t  
                        c[ii][jj] += a[ii][kk]*b_t[jj][kk];  
        }  
    }  
}
```

Block + Transpose

# Tiling/Blocking Algorithm for Transposed Matrix Multiplications



We can make the "tile\_size" larger without interfacing conflict misses

```
for(i = 0; i < M; i+=tile_size) conflict misses
    for(j = 0; j < K; j+=tile_size)
        for(k = 0; k < N; k+=tile_size)
            for(ii = i; ii < i+tile_size; ii++)
                for(jj = j; jj < j+tile_size; jj++)
                    for(kk = k; kk < k+tile_size; kk++)
                        c[ii][jj] += a[ii][kk]*b_t[jj][kk];
```

# The effect of transposition

Block/tile	size	tile_size	IC	Cycles	CPI	CT_ns	ET_s	DL1_miss_rate
2048	4	97766571061	23972375695	0.245200	0.193290	4.633619	0.015394	
	8	81047436195	21583826614	0.266311	0.193122	4.168303	0.010260	
	16	74472586117	19268082018	0.258727	0.193325	3.724997	0.071558	
	32	71543547491	27661109860	0.386633	0.193218	5.344616	0.217189	
	64	70151970961	32605985592	0.464791	0.193248	6.301039	0.242202	
	128	69470212062	34530336995	0.490235	0.193235	6.672484	0.246013	
	256	69131368754	35151111975	0.508468	0.193311	6.795085	0.246800	

Best performing  
from 16 to 64?

Block + Transpose	size	tile_size	IC	Cycles	CPI	CT_ns	ET_s	DL1_miss_rate
2048	8	70351352368	16009523067	0.227565	0.193097	3.091384	0.001897	
	16	64810353582	15145593176	0.233691	0.193199	2.926121	0.026059	
	32	62397963236	14854143892	0.238055	0.193161	2.869243	0.040979	
	64	60712004318	14724348174	0.223012	0.193260	2.640043	0.023464	
	128	60438882203	17003851823	0.275478	0.193123	3.229842	0.018013	
	256	60438882203	17003851823	0.281340	0.193330	3.287351	0.012972	

Transpose improves  
the miss rate at larger  
tile size!

# The effect of transposition + register

	size	tile_size	IC	Cycles	CPI	CT_ns	ET_s	DL1_miss_rate	DL1_accesses
Block + Transpose	2048	8	70351352368	16009523067	0.227565	0.193097	3.091384	0.001897	28769906635
	2048	16	64810353582	15145593176	0.233691	0.193199	2.926121	0.026059	27045466371
	2048	32	62397963236	14854143892	0.238055	0.193161	2.869243	0.040979	26365975027
	2048	64	61255133491	13660607640	0.223012	0.193260	2.640043	0.023464	26061062917
	2048	128	60710004318	16724248174	0.275478	0.193123	3.229842	0.018013	25921112408
	2048	256	60438882203	17003851823	0.281340	0.193330	3.287351	0.012972	25852375287

Significant reduction

	size	tile_size	IC	Cycles	CPI	CT_ns	ET_s	DL1_miss_rate	DL1_accesses
Block + Transpose + Reg.	2048	8	60667686406	11604601609	0.191281	0.193985	2.251117	0.003363	15849319789
	2048	16	49205927530	10032628049	0.203891	0.193373	1.940040	0.066542	11986874817
	2048	32	43854347828	9736190510	0.222012	0.193008	1.879164	0.096360	10247265803
	2048	64	41246516681	10038942768	0.243389	0.193141	1.938936	0.069853	9413044702
	2048	128	39962465083	10372638803	0.293836	0.193141	2.299423	0.045847	9005357520
	2048	256	39326412762	14051616706	0.357307	0.193021	2.712263	0.032001	8804127506

Best performing at 32

# Tiles do not have to be “squares”

```
for(i = 0; i < M; i+=tile_size_y) {  
    for(j = 0; j < K; j+=tile_size_y) {  
        for(k = 0; k < N; k+=tile_size_x) {  
            for(ii = i; ii < i+tile_size_y; ii++)  
                for(jj = j; jj < j+tile_size_y; jj++) {  
                    result = 0;  
                    for(kk = k; kk < k+tile_size_x; kk++)  
                        result += a[ii][kk]*b[jj][kk];  
                    c[ii][jj] += result;  
                }  
        }  
    }  
}
```



Start the presentation to see live content. Still no live content? Install the app or get help at [PollEv.com/app](https://PollEv.com/app)

# If we could have a rectangular tile + transposition

size	tile_size	IC	Cycles	CPI	CT_ns	ET_s	DL1_miss_rate
Block + Transpose + Reg.	8	60667686406	11604601609	0.191281	0.193985	2.251117	0.003363
	16	49205927530	10032628049	0.203891	0.193373	1.940040	0.066542
	32	43854347828	9736190510	0.222012	0.193008	1.879164	0.096360
	64	41246516681	10038942768	0.243389	0.193141	1.938936	0.069853
	128	39962465083	11887280358	0.297461	0.193436	2.299423	0.045847
	256	3932612762	14051616706	0.357307	0.193021	2.712263	0.032001

Very low miss rate compared to 32x32

size	tile_size_x	tile_size_y	IC	Cycles	CPI	CT_ns	ET_s	DL1_miss_rate
Rect. Block + Transpose + Reg.	8	8	60696024519	11624419280	0.191519	0.193479	2.249077	0.003664
	8	16	59757245622	12523769669	0.209577	0.193096	2.418293	0.022092
	16	8	49689499294	11415401958	0.229735	0.193200	2.205459	0.003957
	16	16	43215435235	10018627798	0.203567	0.193202	1.935619	0.067519
	32	16	43947034522	9718597643	0.221143	0.193352	1.879111	0.066730
	32	8	44182255983	9353772552	0.211709	0.193233	1.807453	0.005696
	64	8	41431394002	9636426904	0.232588	0.193230	1.862043	0.005849
	128	8	40059698391	11599727168	0.289561	0.193133	2.240289	0.006533
	256	8	39376445298	13785740954	0.350101	0.193180	2.663130	0.004190

Best performing at 32x8

# Why is “8” not the best performing?

size	tile_size	IC	Cycles	CPI	CT_ns	ET_s	DL1_miss_rate	DL1_accesses
2048	4	97765686275	24149510064	0.247014	0.193189	4.665430	0.015102	43641501149
2048	8	80996985555	21043742544	0.259809	0.193444	4.070776	0.010135	38128531445
2048	16	74473114435	19369857501	0.260092	0.193204	3.742332	0.071790	36105122733
2048	32	71543334296	27812871208	0.388756	0.193112	5.371009	0.217011	35214370198

**More instructions  
due to more loop  
control overhead!**

**Why 1% and 7% miss rate  
do not make significant  
difference?**

# Why is “8” not the best performing?

size	tile_size	IC	Cycles	CPI	CT_ns	ET_s	DL1_miss_rate
2048	4	97766571061	23972375695	0.245200	0.193290	4.633619	0.015394
2048	8	81047436195	21583826614	0.266311	0.193122	4.168303	0.010260
2048	16	74472586117	19268082018	0.258727	0.193325	3.724997	0.071558
2048	32	71543547491	27661109860	0.386633	0.193218	5.344616	0.217189
2048	64	70151970961	32605985592	0.464791	0.193248	6.301039	0.242202
2048	128	69470212062	34530336995	0.497052	0.193235	6.672484	0.246013
2048	256	69131368754	35151111975	0.508468	0.193311	6.795085	0.246800
2048	512	68985162572	47048159619	0.682004	0.193298	9.094299	0.239775

More instructions  
due to more loop  
control overhead!

Best  
performing  
at 16?

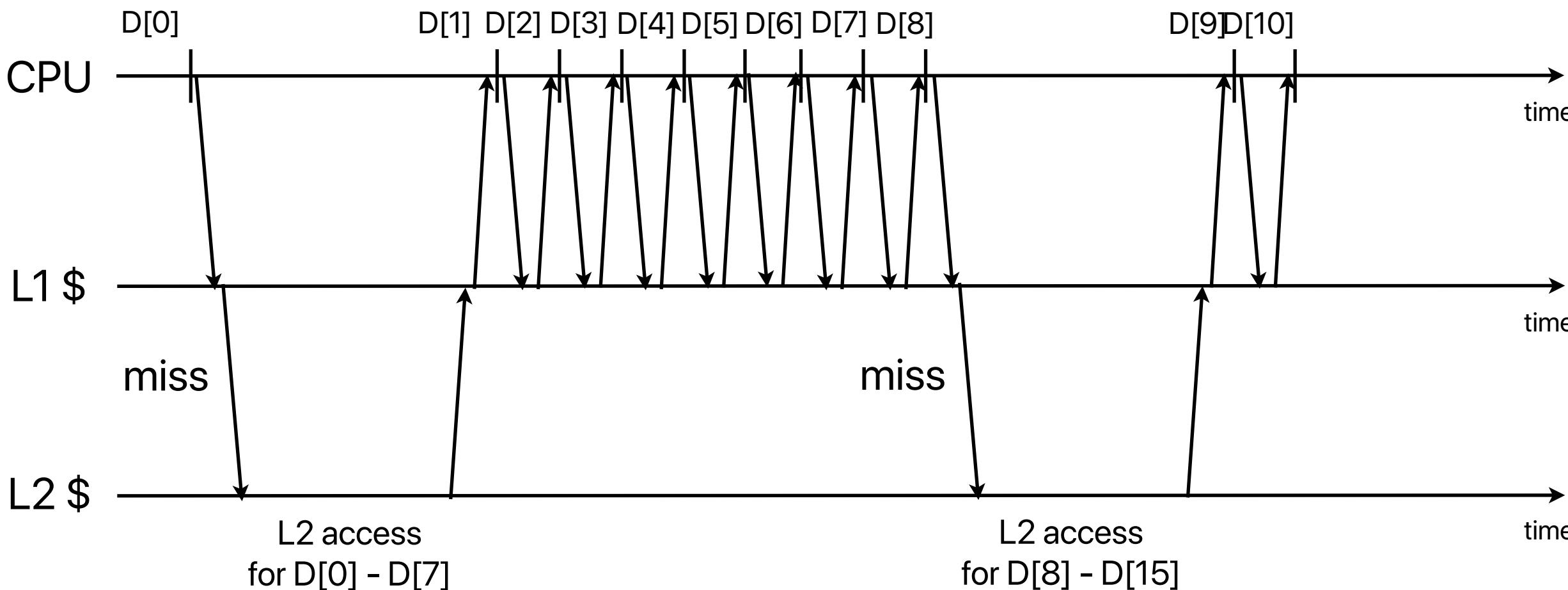
“8” indeed has the best  
miss rate — and matches  
our predictions!

# Takeaways: Software Optimizations

- Data layout — capacity miss, conflict miss, compulsory miss
- Loop interchange — conflict/capacity miss
- Loop fission — conflict miss — when \$ has limited way associativity
- Loop fusion — capacity miss — when \$ has enough way associativity
- Blocking/tiling — capacity miss, conflict miss
- Matrix transpose (a technique changes layout) — conflict misses
- Using registers whenever possible — reduce memory accesses!

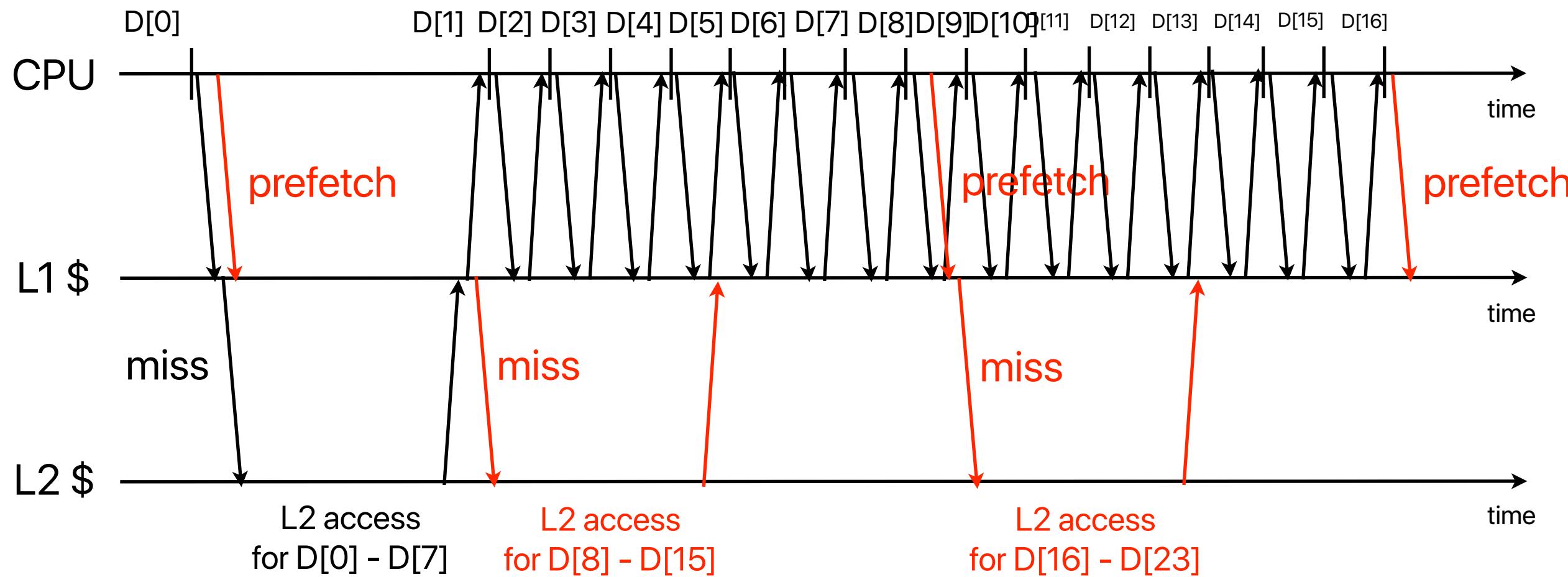
# Characteristic of memory accesses

```
for(i = 0; i < 1000000; i++) {  
    D[i] = rand();  
}
```



# Prefetching

```
for(i = 0; i < 1000000; i++) {  
    D[i] = rand();  
    // prefetch D[i+8] if i % 8 == 0  
}
```



# Software Prefetching — through prefetching instructions

- x86 provide prefetch instructions
- As a programmer, you may insert `_mm_prefetch` in x86 programs to perform software prefetch for your code
- gcc also has a flag “`-fprefetch-loop-arrays`” to automatically insert software prefetch instructions



Start the presentation to see live content. Still no live content? Install the app or get help at [PollEv.com/app](https://PollEv.com/app)

# Takeaways: Software Optimizations

- Data layout — capacity miss, conflict miss, compulsory miss
- Loop interchange — conflict/capacity miss
- Loop fission — conflict miss — when \$ has limited way associativity
- Loop fusion — capacity miss — when \$ has enough way associativity
- Blocking/tiling — capacity miss, conflict miss
- Matrix transpose (a technique changes layout) — conflict misses
- Using registers whenever possible — reduce memory accesses!

# Software Prefetching — through prefetching instructions

- x86 provide prefetch instructions
- As a programmer, you may insert `_mm_prefetch` in x86 programs to perform software prefetch for your code
- gcc also has a flag “`-fprefetch-loop-arrays`” to automatically insert software prefetch instructions

# Takeaways: Software Optimizations

- Data layout — capacity miss, conflict miss, compulsory miss
- Loop interchange — conflict/capacity miss
- Loop fission — conflict miss — when \$ has limited way associativity
- Loop fusion — capacity miss — when \$ has enough way associativity
- Blocking/tiling — capacity miss, conflict miss

# Announcement

- Make a reservation for practice examine at the testing center
  - Please schedule through  
<https://testcenter.cs.ucr.edu/Make-Change-Delete-Reservations>
  - You may start scheduling today
    - Instructor/TA does not handle the scheduling
    - First come first serve — the same policy for midterm/final
  - This will help you get familiar with the environment for midterm and final
  - It's your responsibility if you miss a question due to any unfamiliarity of the testing environment
- Reading quiz #3 due next **Tuesday** before the lecture
- Assignment #1 due **tonight**
  - You should run the performance measurement yourself and calculate results based on that — everyone should have a different answer
  - We do not support people on the last minute. Please carefully plan your time and respect the instructor/TA
- Assignment 2 and Programming Assignment 2 are both released
  - You should start working on them as soon as possible
  - Make sure you submit to the right place (we won't open them until the assignment 1 due)



# Computer Science & Engineering

203

つづく

