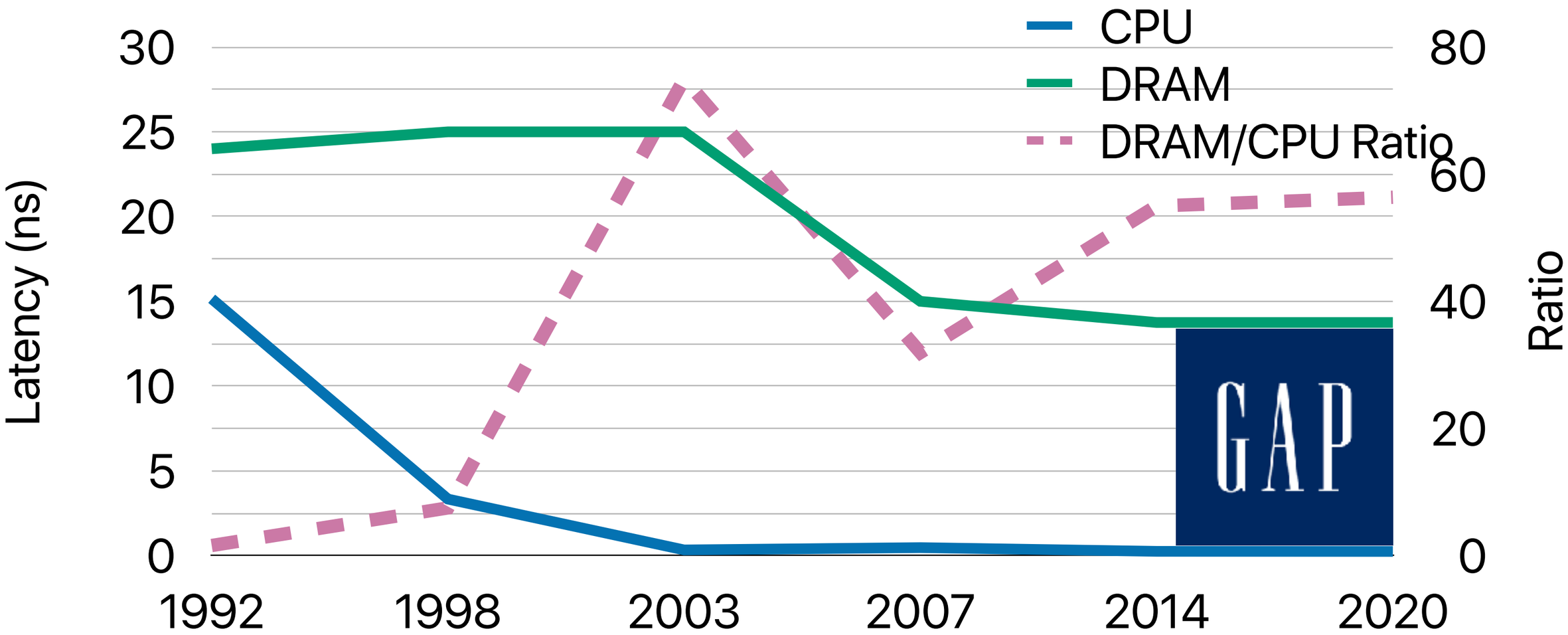
Memory Hierarchy (4): Cache Misses and How to Address Them (cont.)

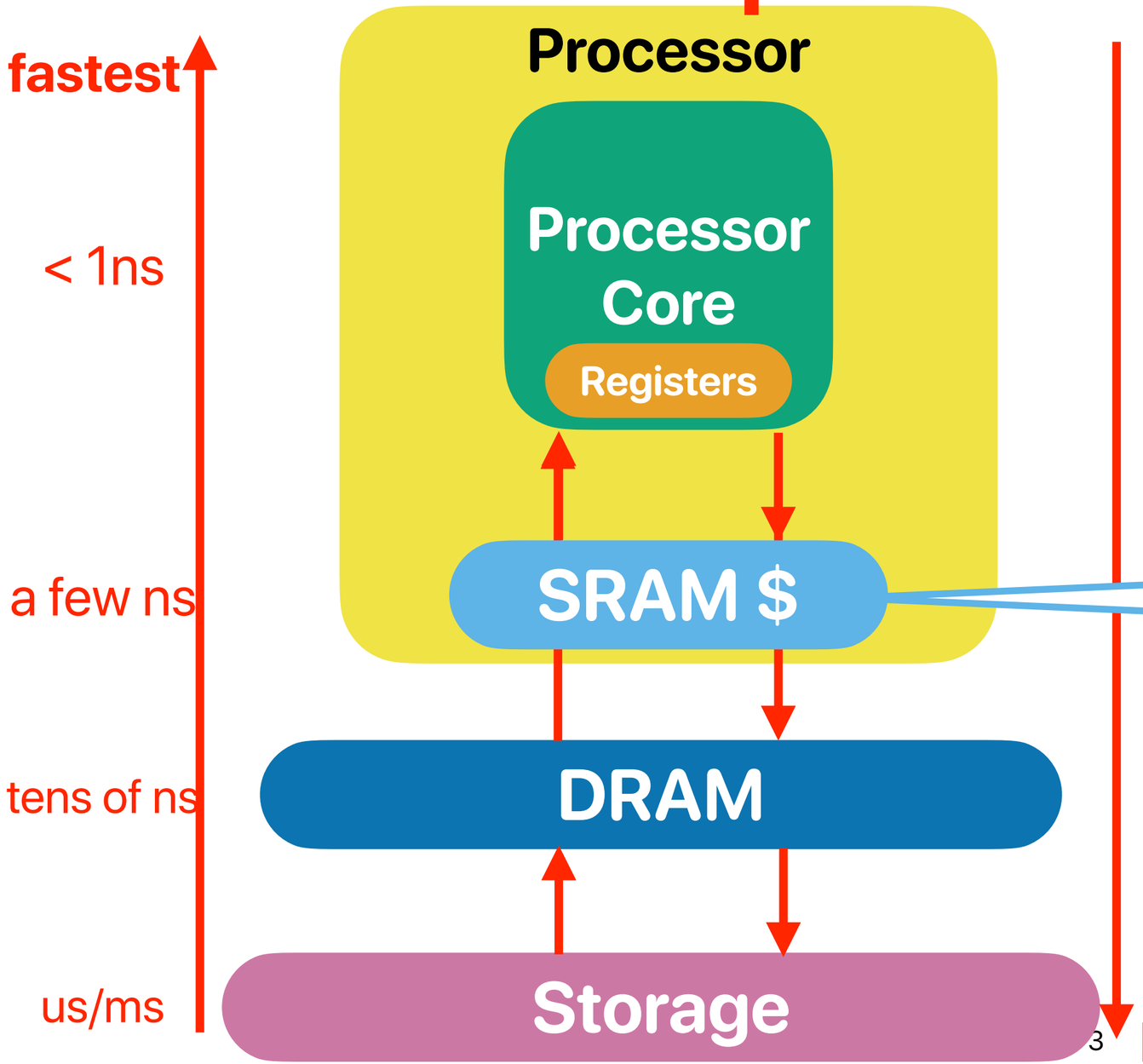
Hung-Wei Tseng

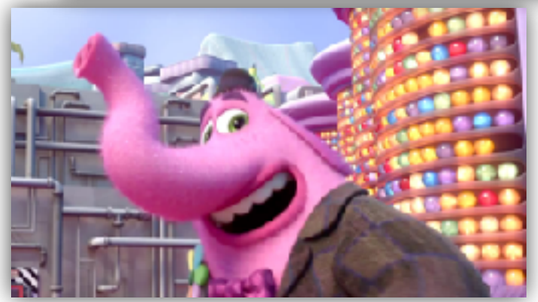
Recap: the "latency" gap between CPU and DRAM



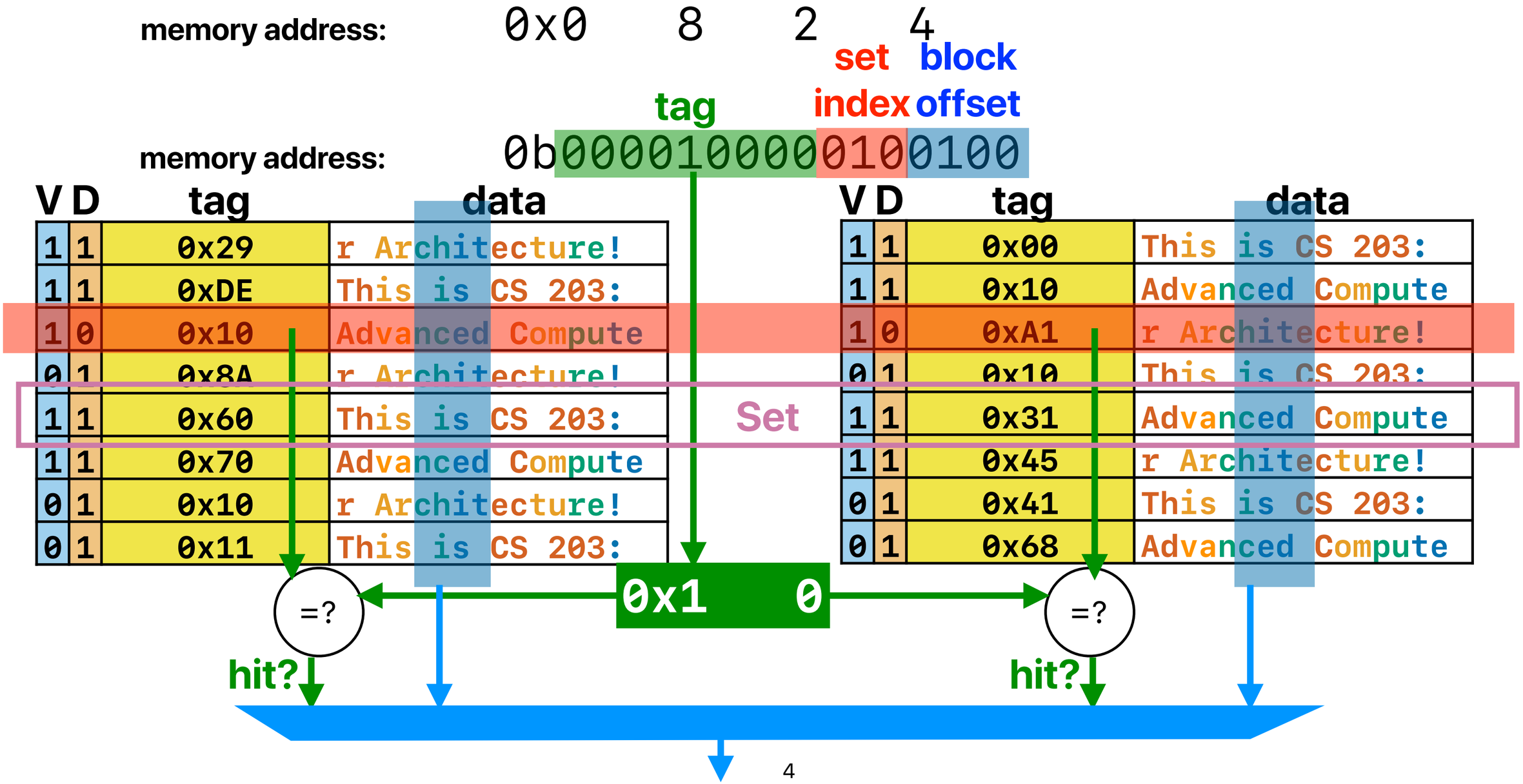
CPU	i486	Pentium II	Pentium 4	Core 2	Core i7-4790K	Core i5-10600K
DRAM	SDRAM	DDR	DDR2	DDR3	DDR4	DDR5

Recap: Memory Hierarchy





Recap: Way-associative cache



Recap: Knowing where and why do I have cache misses

- Simulating code's cache behavior
  - Figure out where in the code will access memory
  - Generate the memory addresses that the code will access
  - Partition the addresses using C=ABS
  - Emulate the cache management
- Three types of misses
  - Compulsory misses
  - Conflict misses
  - Capacity misses

Recap: NVIDIA Tegra X1

100% miss rate!

- Size 32KB, 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
32KB = 4 * 64 * S
S = 128
offset = lg(64) = 6 bits
index = lg(128) = 7 bits
tag = the rest bits

tag index offset

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[0]	0x10000	0b0001000000000000000000	0x8	0x0	Compulsory Miss	
b[0]	0x20000	0b0010000000000000000000	0x10	0x0	Compulsory Miss	
c[0]	0x30000	0b0011000000000000000000	0x18	0x0	Compulsory Miss	
d[0]	0x40000	0b0100000000000000000000	0x20	0x0	Compulsory Miss	
e[0]	0x50000	0b0101000000000000000000	0x28	0x0	Compulsory Miss	a[0–7]
a[1]	0x10008	0b0001000000000000001000	0x8	0x0	**Conflict Miss**	b[0–7]
b[1]	0x20008	0b0010000000000000001000	0x10	0x0	**Conflict Miss**	c[0–7]
c[1]	0x30008	0b0011000000000000001000	0x18	0x0	**Conflict Miss**	d[0–7]
d[1]	0x40008	0b0100000000000000001000	0x20	0x0	**Conflict Miss**	e[0–7]
e[1]	0x50008	0b0101000000000000001000	0x28	0x0	**Conflict Miss**	a[0–7]
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Recap: NVIDIA Tegra X1 (cont.)

- Size 32KB, 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
32KB = 4 * 64 * S
S = 128
offset = lg(64) = 6 bits
index = lg(128) = 7 bits
tag = the rest bits

tag index offset

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[7]	0x10038	0b0001000000000111000	0x8	0x0	**Conflict Miss**	
b[7]	0x20038	0b0010000000000111000	0x10	0x0	**Conflict Miss**	
c[7]	0x30038	0b0011000000000111000	0x18	0x0	**Conflict Miss**	
d[7]	0x40038	0b0100000000000111000	0x20	0x0	**Conflict Miss**	
e[7]	0x50038	0b0101000000000111000	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]
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Recap: NVIDIA Tegra X1 — Loop Fission

- Size 32KB, 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]);
    //load a[i], b[i], c[i], and then store to e[i]
```

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

tag index offset

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[0]	0x10000	0b0001000000000000000000	0x8	0x0	Compulsory Miss	
b[0]	0x20000	0b0010000000000000000000	0x10	0x0	Compulsory Miss	
c[0]	0x30000	0b0011000000000000000000	0x18	0x0	Compulsory Miss	
e[0]	0x50000	0b0101000000000000000000	0x28	0x0	Compulsory Miss	
a[1]	0x10008	0b0001000000000000001000	0x8	0x0	**Hit**	
b[1]	0x20008	0b0010000000000000001000	0x10	0x0	**Hit**	
c[1]	0x30008	0b0011000000000000001000	0x18	0x0	**Hit**	
e[1]	0x50008	0b0101000000000000001000	0x28	0x0	**Hit**	

Recap: NVIDIA Tegra X1 — Loop Fission

- Size 32KB, 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]);
    //load a[i], b[i], c[i], and then store to e[i]
```

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

No capacity misses!

tag index offset

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[7]	0x10038	0b0001000000000111000	0x8	0x0	**Hit**	
b[7]	0x20038	0b0010000000000111000	0x10	0x0	**Hit**	
c[7]	0x30038	0b0011000000000111000	0x18	0x0	**Hit**	
e[7]	0x50038	0b0101000000000111000	0x28	0x0	**Hit**	
a[8]	0x10040	0b0001000000001000000	0x8	0x1	Compulsory Miss	
b[8]	0x20040	0b0010000000001000000	0x10	0x1	Compulsory Miss	
c[8]	0x30040	0b0011000000001000000	0x18	0x1	Compulsory Miss	
e[8]	0x50040	0b0101000000001000000	0x28	0x1	Compulsory Miss	

$$\frac{512}{8} = 64$$

We need only 4 blocks in 64 sets to accommodate everything

Recap: NVIDIA Tegra X1 — Loop Fission (cont.)

- Size 32KB, 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] = e[i] / d[i];
    //load a[i], b[i], c[i], and then store to e[i]
```

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

tag index offset

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
e[0]	0x50000	0b0101000000000000000000	0x28	0x0	**Hit**	
d[0]	0x40000	0b0100000000000000000000	0x20	0x0	Compulsory Miss	
e[0]	0x50000	0b0101000000000000000000	0x28	0x0	**Hit**	
e[1]	0x50008	0b0101000000000001000	0x28	0x0	**Hit**	
d[1]	0x40008	0b0100000000000001000	0x20	0x0	**Hit**	
e[1]	0x50008	0b0101000000000001000	0x28	0x0	**Hit**	
⋮	⋮	⋮	⋮	⋮	⋮	⋮
e[8]	0x50040	0b0101000000001000000	0x28	0x1	**Hit**	
d[8]	0x40040	0b0100000000001000000	0x20	0x1	Compulsory Miss	
e[8]	0x50040	0b0101000000001000000	0x28	0x1	**Hit**	

Details of Loop Fission on Tegra X1

```
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
```

Misses	Memory Accesses	Miss Rate
$4 \times \frac{512}{8}$	4×512	
$\frac{512}{8}$	3×512	
$5 \times \frac{512}{8}$	7×512	**0.0893**

Loop optimizations

Loop interchange

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];
  }
}
```

Loop fission

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];
}
```

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

Outline

- Writing cache-friendly code
- Virtual memory
- Sample Midterm

How can programmer improve memory performance? (cont.)

Data structures

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];
}
```

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

Loop interchange

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];
  }
}
```

Loop fission

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];
}
```

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

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];
}
```

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

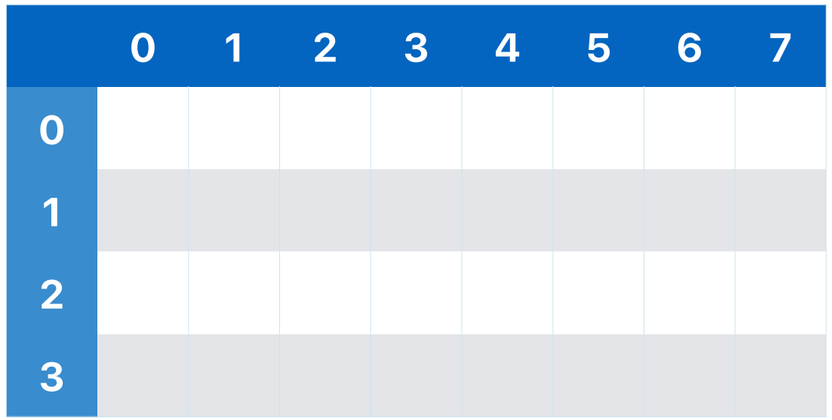
Tiling/Blocking Algorithm

What is an M by N "2-D" array in C?

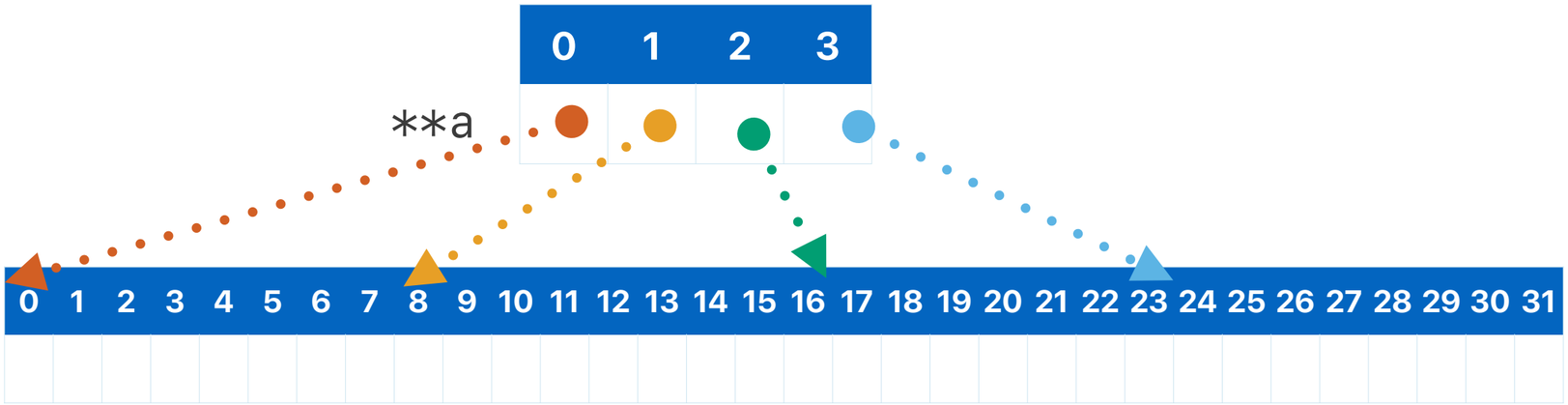
```
a = (double **)malloc(M*sizeof(double *));
for(i = 0; i < N; i++)
{
  a[i] = (double *)malloc(N*sizeof(double));
}
```

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

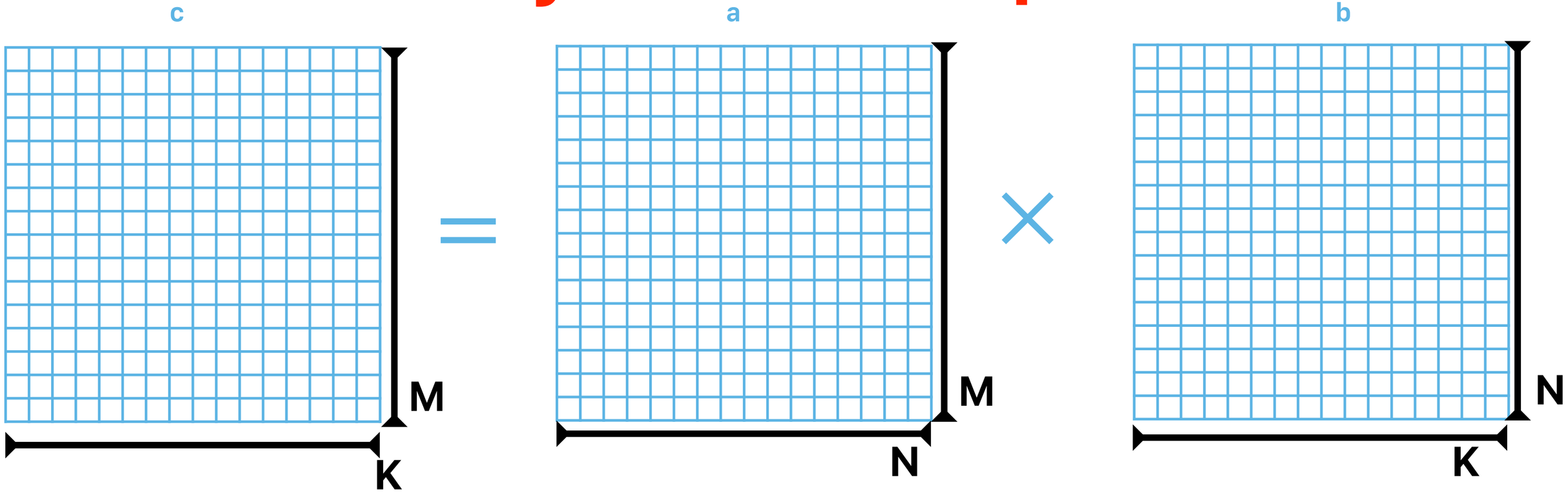
abstraction



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?

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

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?

```
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

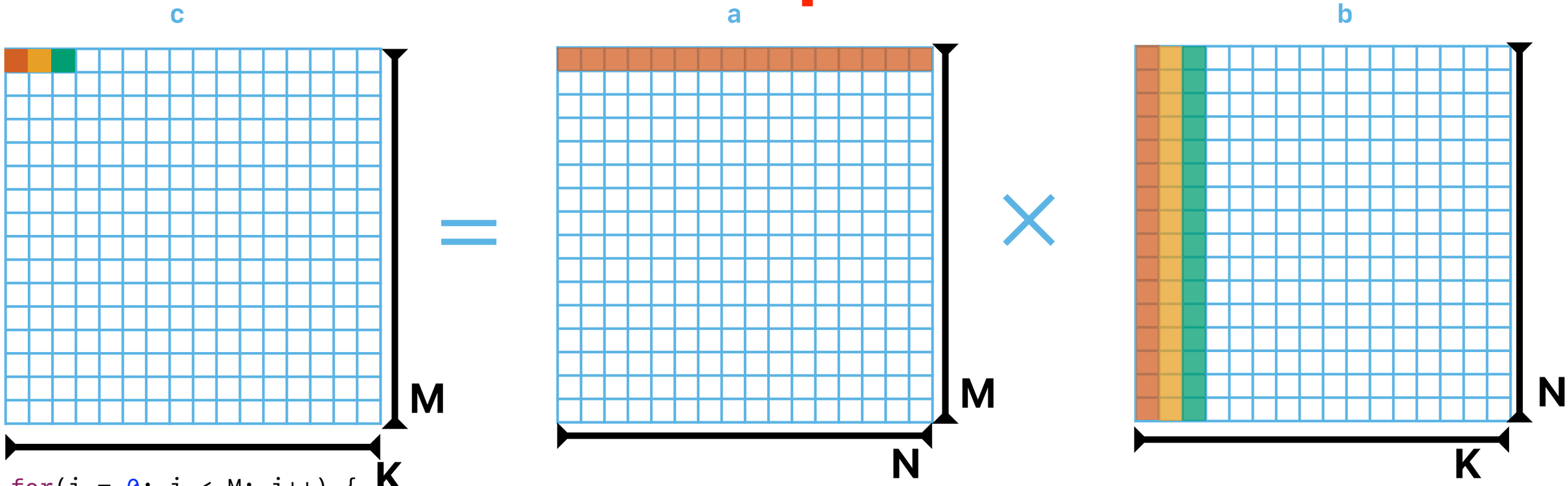
What kind(s) of misses are there in Matrix Multiplications

- Considering the case where M=N=K=2048, what do you think the majority type(s) of cache misses are we seeing?

```
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




```
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];
    }
  }
}
```

Matrix Multiplication — let's consider "b"

```
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];
    }
  }
}
```

	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] …

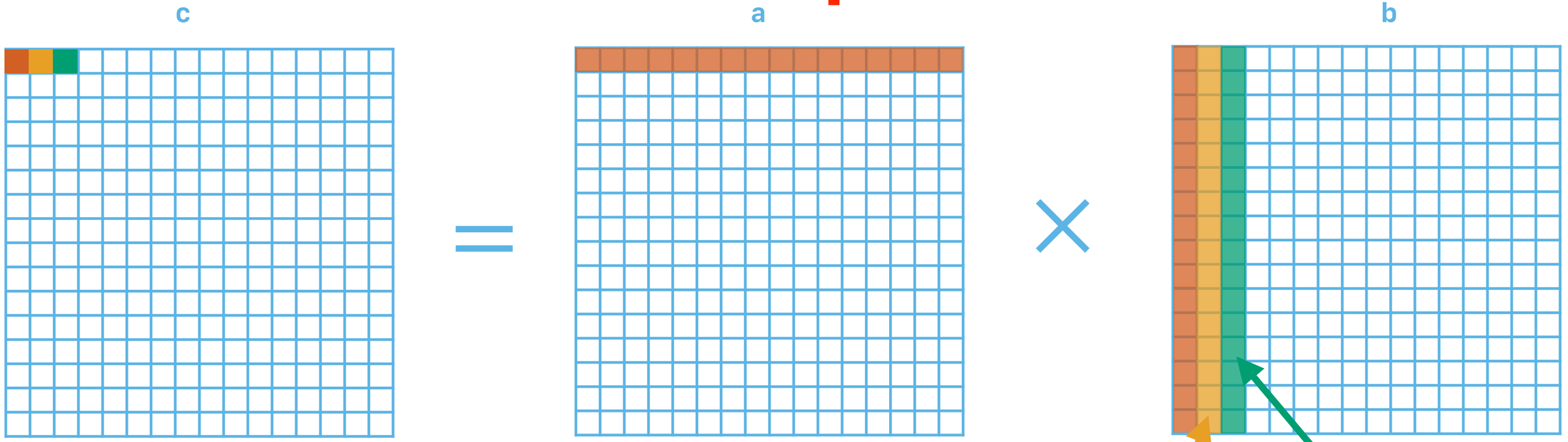
Now, when we work on c[0][1]

	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

	Address	Tag	Index	
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] …

Matrix Multiplications

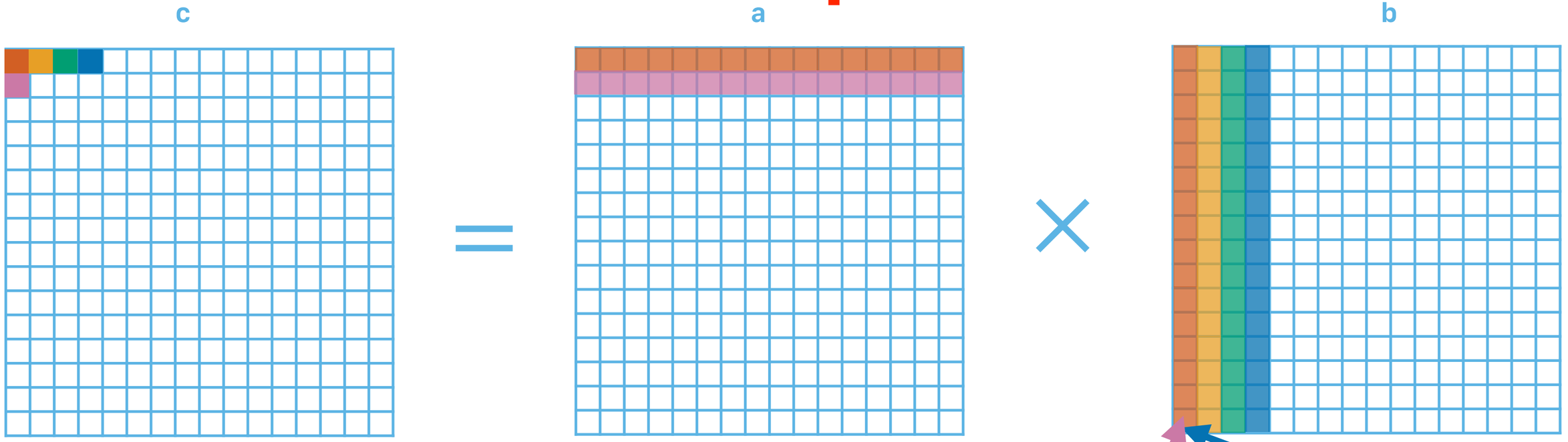



```
for(i = 0; i < M; i++) {
  for(j = 0; j < K; j++) {
    for(k = 0; k < N; k++) {
      c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

These are conflict misses as we have cached them before

These are conflict misses as we have cached them before

Matrix Multiplications



- If each dimension of your matrix is 2048
  - Each row or column takes 2048×8 Bytes = 16 KB
  - The L1-$ of intel Core i7 is 48 KB, 12-way, 64-byte blocked
  - You can only hold at most 3 rows or columns of each matrix!
  - You need the more columns when `j` increase!
- We will have capacity misses when we work on a new `i`

We need to fetch everything again — capacity miss!

Unlikely to be kept in the cache

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?

```
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

Ideas regarding reducing misses in matrix multiplications

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

Mathematical view of MM

$$c_{i,j} = \sum_{k=0}^{k=N-1} a_{i,k} \times b_{k,j} = \sum_{k=0}^{k=\frac{N}{2}-1} a_{i,k} \times b_{k,j} + \sum_{k=\frac{N}{2}}^{k=N-1} a_{i,k} \times b_{k,j}$$

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

$$\vdots$$

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

Matrix Multiplication — let's consider "b"

```
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];
    }
  }
}
```

	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 of your matrix is 2048, each row element with the same column index is $2048 \times 8 = 16384 = 0x4000$ away from each other

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

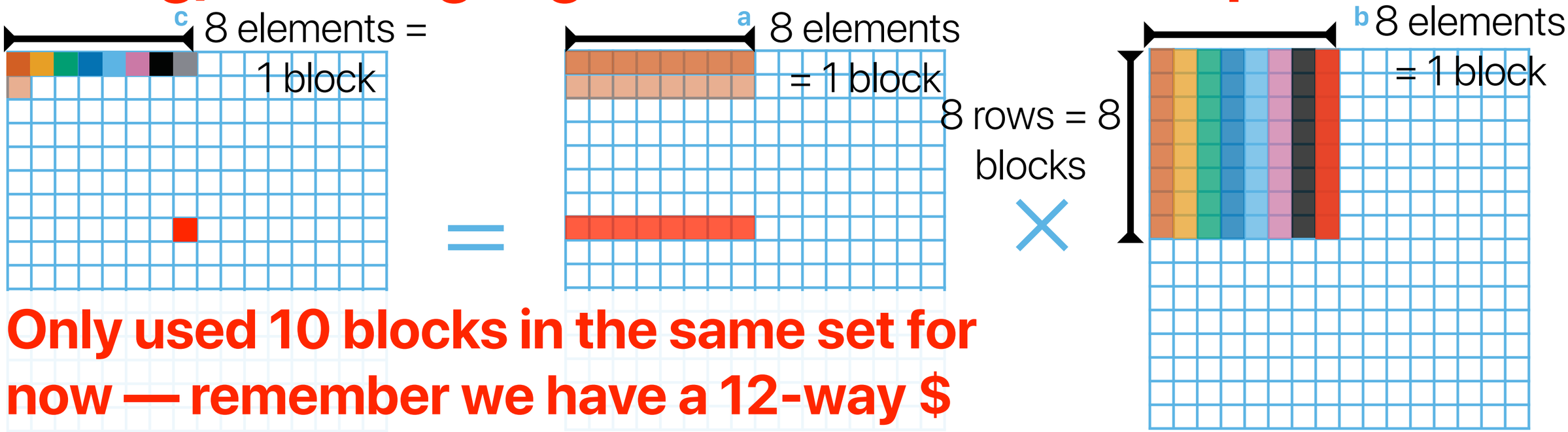
Since each block has 8 elements, let's break up in 8 for now

— 8 elements from a[i]

— 8 columns each covers 8 rows

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

Tiling/Blocking Algorithm for Matrix Multiplications



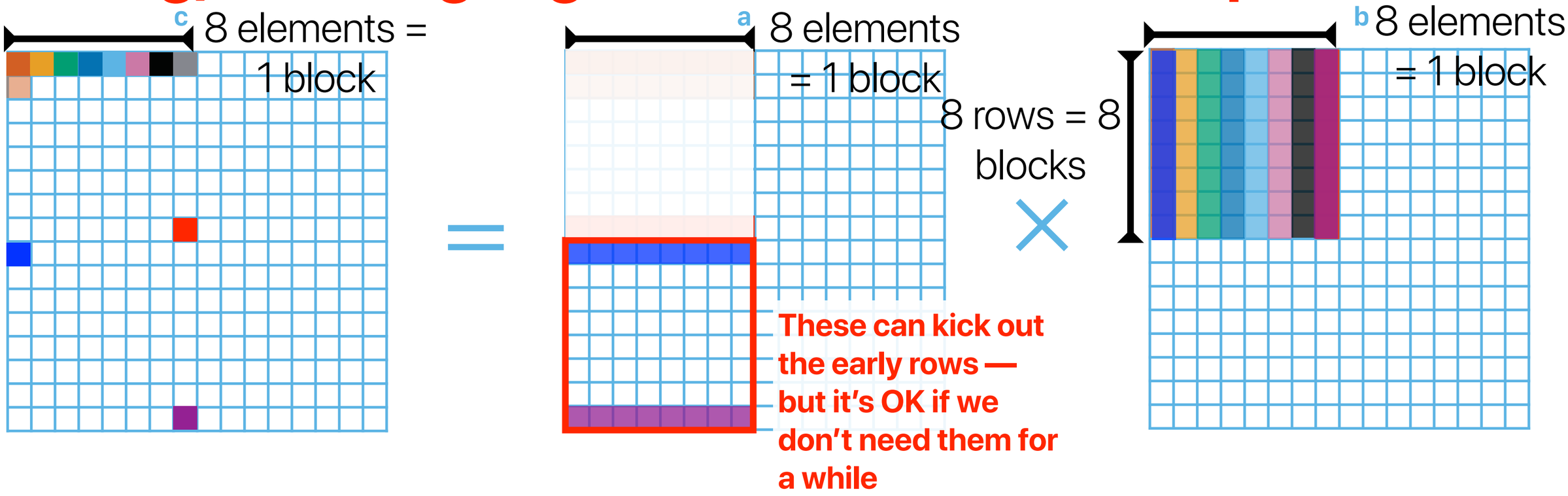
Only used 10 blocks in the same set for now — remember we have a 12-way $

These are still around when we move to the next row in the "tile"

Only compulsory misses —

$$miss\_rate = \frac{total\ misses}{total\ accesses} = \frac{8+8}{3\times 8\times 8} = 0.083$$

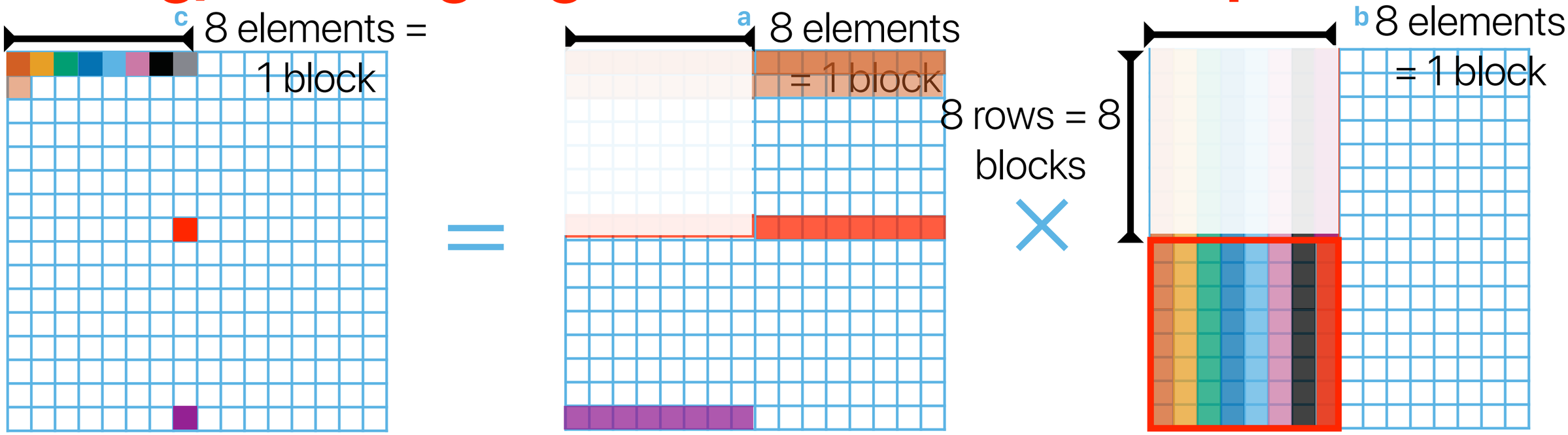
Tiling/Blocking Algorithm for Matrix Multiplications



Bringing miss rate even further lower now —

$$miss\_rate = \frac{total\ misses}{total\ accesses} = \frac{8+8+8}{3\times 8\times 8+3\times 8\times 8} = 0.042$$

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

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!

“8” indeed has the best miss rate

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

What kind(s) of misses can tiling algorithm remove?

- Comparing the naive algorithm and tiling algorithm on matrix multiplication, what kind of misses does tiling algorithm help to remove? (assuming an intel Core i7)

Naive

```
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];
    }
  }
}
```

Block/tile

```
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

What kind(s) of misses can tiling algorithm remove?

- Comparing the naive algorithm and tiling algorithm on matrix multiplication, what kind of misses does tiling algorithm help to remove? (assuming an intel Core i7)

Naive

```
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];
    }
  }
}
```

Block/tile

```
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];
    }
  }
}
```

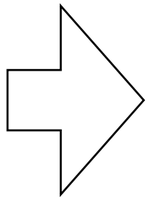
A. Compulsory miss
B. Capacity miss
C. Conflict miss
D. Capacity & conflict miss
E. Compulsory & conflict miss

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

```
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];
    }
  }
}
```



```
// 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 < 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];
    }
  }
}
```

The effect of transposition

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
2048	16	64810073062	15221864848	0.234869	0.193117	2.939604	0.024597	27045326530

Transpose improves the miss rate!

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?

Block/tile

```
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];
    }
  }
}
```

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];
    }
  }
}
```

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

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?

Block/tile

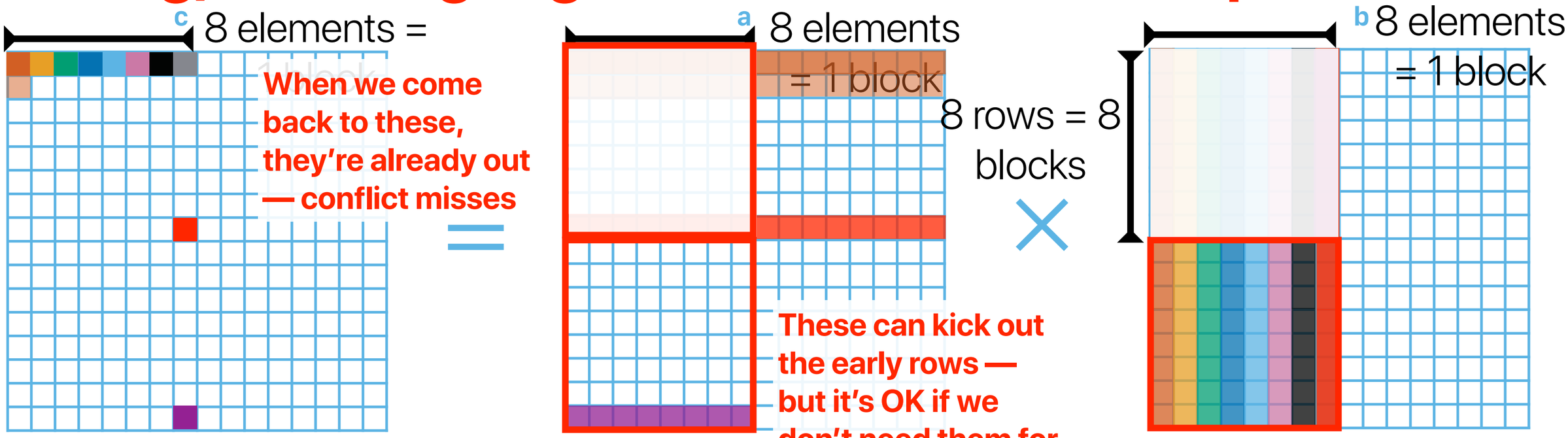
```
for(i = 0; i < M; i+=tile_size) {
  for(j = 0; j < K; j+=tile_size) {
    for(k = 0; k < N; k+=tile_size) {
      for(ii = i; ii < i+tile_size; ii++)
        for(jj = j; jj < j+tile_size; jj++)
          for(kk = k; kk < k+tile_size; kk++)
            c[ii][jj] += a[ii][kk]*b[kk][jj];
    }
  }
}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

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

Block + Transpose

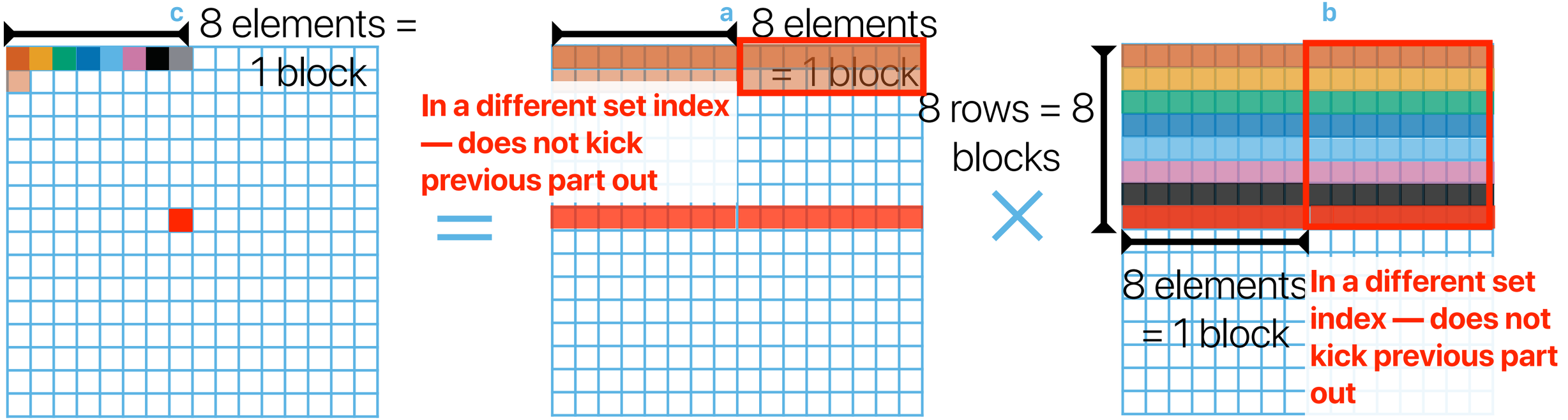
```
// Transpose matrix b into b_t
for(i = 0; i < ARRAY_SIZE; i++) {
  for(j = 0; j < ARRAY_SIZE; j++) {
      b_t[i][j] += b[j][i];
  }
}


  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];
    }
  }
}
```

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];
```

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)
    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];
```

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?

Block/tile

```
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];
    }
  }
}
```

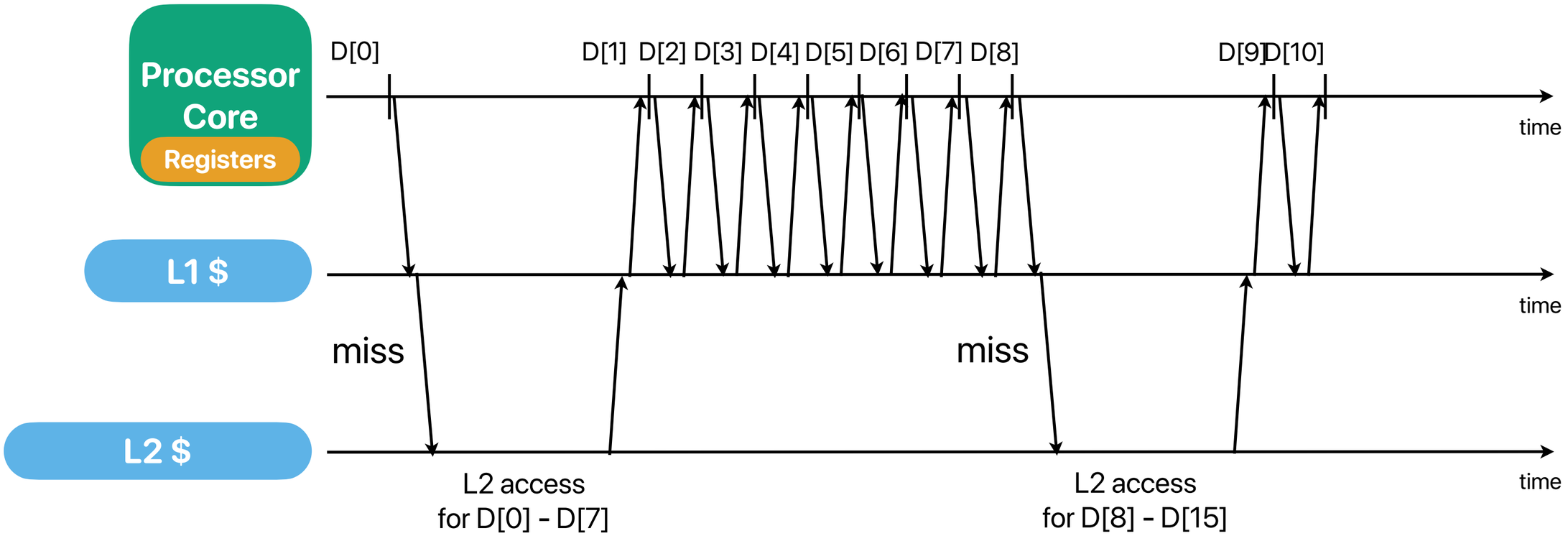
Takeaways: Software Optimizations

- 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

Prefetching

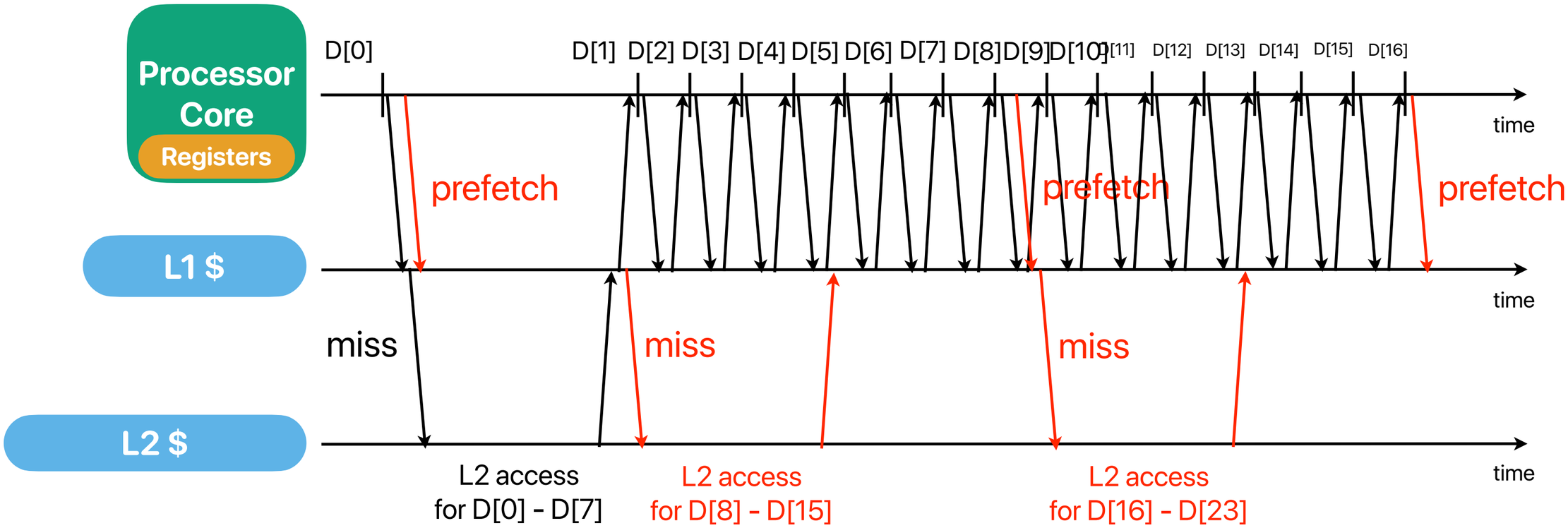
Characteristic of memory accesses

```
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
}
```



Prefetching

- 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
- Hardware prefetch
  - The processor can keep track the distance between misses. If there is a pattern, fetch miss_data_address+distance for a miss
- Software prefetch
  - Load data into some register
  - Using prefetch instructions

Demo: make matrix transpose faster!

- 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
- Matrix transpose (a technique changes layout) — conflict misses
- Prefetching — compulsory misses

Sample Midterm

Format

- 15 Multiple Choice Questions — 30%
- 3 Assignment Style Free Answer Questions — 70%
- Closed note, closed book, no outside materials

Programmer's impact

- By adding the "sort" in the following code snippet, what the programmer changes in the performance equation to achieve **better** performance?

```
std::sort(data, data + arraySize);

for (unsigned c = 0; c < arraySize*1000; ++c) {
        if (data[c%arraySize] >= INT_MAX/2)
            sum ++;
    }
}
```

A. CPI
B. IC
C. CT
D. IC & CPI
E. CPI & CT

How programming languages affect performance

- Performance equation consists of the following three factors
  ① IC
  ② CPI
  ③ CT

  How many can the **programming language** affect?

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

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];
  }
}
```

- How many of the following make(s) the performance of A better than B?
  - ① IC
  - ② CPI
  - ③ CT
  - A. 0
  - B. 1
  - C. 2
  - D. 3

How compilers affect performance

- If we apply compiler optimizations for both code snippets **A** and **B**, how many of the following can we expect?
  - ① Compiler optimizations can reduce IC for both
  - ② Compiler optimizations can make the CPI lower for both
  - ③ Compiler optimizations can make the ET lower for both
  - ④ Compiler optimizations can transform code B into code A
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

A

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

B

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

Amdahl's Law on Multicore Architectures

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

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

How reflective is FLOPS?

- If you're given the FLOPS of an underlying GPU, how many situations below can the FLOPS be representative to the real performance?
  - ① The FLOPS remains the same on the same GPU even if we change the data size
  - ② The FLOPS remains the same on the same GPU even if we change the data type to double
  - ③ The FLOPS remains the same on the same GPU if we change the algorithm implementation
  - ④ The ratio of FLOPS on two different GPUs reflects the ratio execution on these two GPUs when executing floating point applications

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

How programmer affects performance?

- Performance equation consists of the following three factors
  ① IC
  ② CPI
  ③ CT

  How many can a **programmer** affect?

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

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;
}
```

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

intel Core i7

- L1 data (D-L1) cache configuration of Core i7
  - Size 32KB, 8-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

intel Core i7

- D-L1 Cache configuration of intel Core i7
  - 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]
}
```

How many of the cache misses are **compulsory** misses?

A. 12.5%
B. 66.67%
C. 68.75%
D. 87.5%
E. 100%

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];
}
```

What kind(s) of misses can tiling algorithm remove?

- Comparing the naive algorithm and tiling algorithm on matrix multiplication, what kind of misses does tiling algorithm help to remove? (assuming an intel Core i7)

Naive

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

Block

```
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
  for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
    for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
      for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
        for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
          for(kk = k; kk < k+(ARRAY_SIZE/n); 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

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?

Block

```
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
  for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
    for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
      for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
        for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
          for(kk = k; kk < k+(ARRAY_SIZE/n); 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+=(ARRAY_SIZE/n)) {
  for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
      b_t[i][j] += b[j][i];
  }
}


for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
  for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
    for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
        for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
          for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
            for(kk = k; kk < k+(ARRAY_SIZE/n); kk++
              // Compute on b_t
              c[ii][jj] += a[ii][kk]*b_t[jj][kk];
    }
  }
}
```

Tag in the cache

- Regarding the "tag" in cache, please identify the **correct** statement
  A. The tag helps us to identify if the requesting address residing in the cache.
  B. The tag comes from the least significant bits of the memory address.
  C. Each memory address has a unique tag.
  D. The tag gives the location of a word within a block

Conflict misses

- Which of the following statement best describes conflict misses
  A. The cache is large enough to hold the referenced data objects, but because they map to the same cache block, the cache keeps missing.
  B. When the size of the working set exceeds the size of the cache, the cache is just too small to handle this particular working set.
  C. When the code access a data structure for the first time, the cache has a miss of the object.

Remind yourself

- What are the limitations of compiler optimizations? Can you list two?
- Please define Amdahl's Law and explain each term in it
- Please define the CPU performance equation and explain each term.
- Can you list two things affecting each term in the performance equation?
- What's the difference between latency and throughput? When should you use latency or throughput to judge performance?
- What's "benchmark" suite? Why is it important?
- Why TFLOPS or inferences per second is not a good metrics?

Performance equation

- Consider the following c code snippet and x86 instructions implement the code snippet

C	x86

```
for(i = 0; i < count; i++) {
    s += a[i];
}
```

```
.L3:
movslq  (%rdi), %rdx
addq    $4, %rdi
addq    %rdx, %tax
cmpq    %rcx, %rdi
jne     .L3
```

If (1) count is set to 1,000,000,000, (2) a memory instruction takes 4 cycles, (3) a branch/jump instruction takes 3 cycles, (4) other instructions takes 1 cycle on average, and (5) the processor runs at 4 GHz, how much time is it take to finish executing the code snippet?

Speedup of Y over X

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

	Clock Rate	Instructions	Percentage of Type-A	CPI of Type-A	Percentage of Type-B	CPI of Type-B	Percentage of Type-C	CPI of Type-C
Machine X	3 GHz	5000000000	20%	8	20%	4	60%	1
Machine Y	5 GHz	5000000000	20%	13	20%	4	60%	1

Q5 and Q8: cache performance on the following code

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

```
double data[16384]; /* &data[0] = 0x20000 */
uint64_t sum = 0;
for(uint i = 0; i < arg1; i++) {
    for(uint x = 0; x < size; x+=arg1) {
      sum += data[x];
    }
}
return sum;
```

What's the data cache miss rate for this code when **arg1=1** and **size = 16384**?

Q5 and Q8: cache performance on the following code

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

```
double data[16384]; /* &data[0] = 0x20000 */
uint64_t sum = 0;
for(uint i = 0; i < arg1; i++) {
    for(uint x = 0; x < size; x+=arg1) {
      sum += data[x];
    }
}
return sum;
```

What's the data cache miss rate for this code when **arg1=16** and **size = 16384**?

Practicing Amdahl's Law

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

Practicing Amdahl's Law (2)

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

Announcement

- Midterm next Monday in person during the normal lecture time
  - 80 minute
  - In-person, closed book, closed note
  - You can bring a calculator, but not the mobile calculator app
  - Multiple choices and the assignment-style free answer questions
  - Will release a sample midterm tomorrow
  - Hung-Wei will host office hours today 3:30p-6p
- Update your Assignment #2!
  - Execute the script in the notebook's Section 2.4 and refresh the "browser", not your notebook
  - Q8's stride should be "8"
  - Due **this Saturday**
  - You should run the performance measurement yourself and calculate results based on that — everyone should have a different answer
  - All questions this time require **correct** estimations in cache performance to help you better prepare the examines
- Reading Quiz #5 due next Wednesday before the lecture

Computer
Science &
Engineering

142

つづく