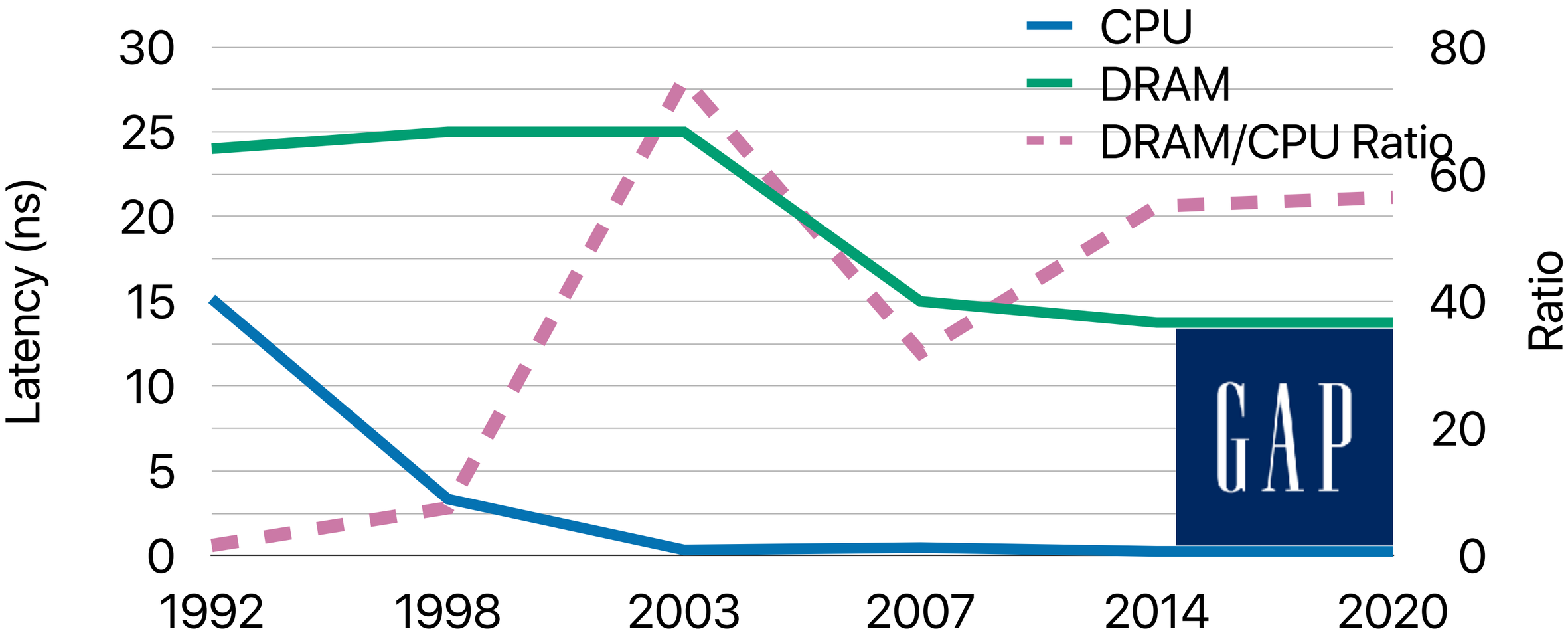
Memory Hierarchy (5): Case study — matrix multiplications

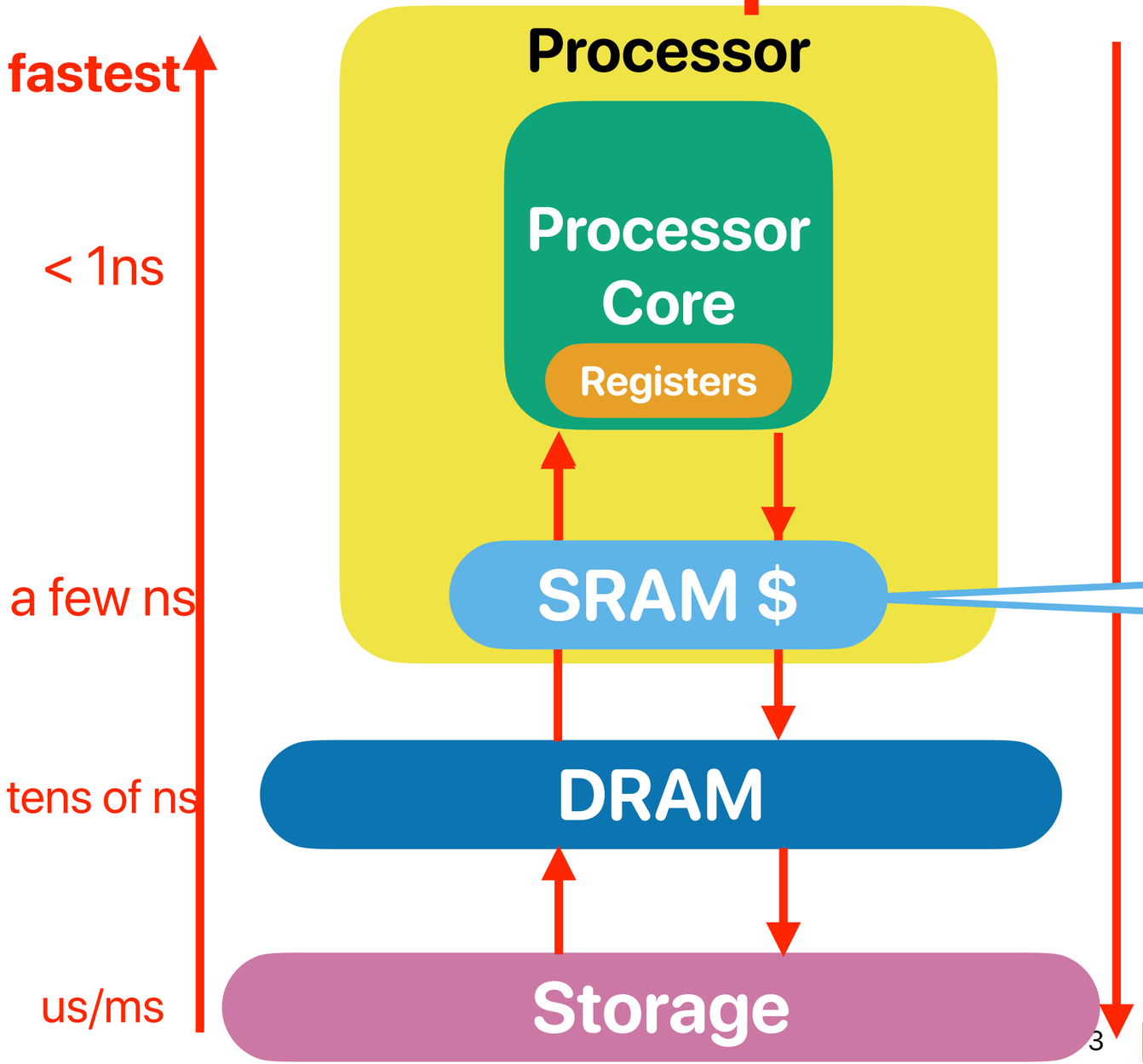
Hung-Wei Tseng

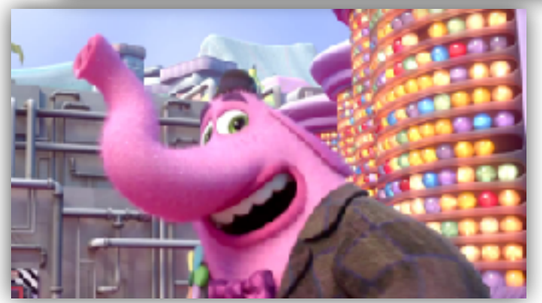
Recap: the "latency" gap between CPU and DRAM



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: 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

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 bits
index = lg(256) = 8 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]
⋮	⋮	⋮	⋮	⋮	⋮	⋮

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]
```

C = ABS
48KB = 12 * 64 * S
S = 64
offset = lg(64) = 6 bits
index = lg(64) = 6 bits
tag = the rest bits

tag index offset

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[0]	0x10000	0b0001000000000000000000	0x10	0x0	Compulsory Miss	
b[0]	0x20000	0b0010000000000000000000	0x20	0x0	Compulsory Miss	
c[0]	0x30000	0b0011000000000000000000	0x30	0x0	Compulsory Miss	
d[0]	0x40000	0b0100000000000000000000	0x40	0x0	Compulsory Miss	
e[0]	0x50000	0b0101000000000000000000	0x50	0x0	Compulsory Miss	
a[1]	0x10008	0b0001000000000000001000	0x10	0x0	Hit	
b[1]	0x20008	0b0010000000000000001000	0x20	0x0	Hit	
c[1]	0x30008	0b0011000000000000001000	0x30	0x0	Hit	
d[1]	0x40008	0b0100000000000000001000	0x40	0x0	Hit	
e[1]	0x50008	0b0101000000000000001000	0x50	0x0	Hit	
⋮	⋮	⋮	⋮	⋮	⋮	⋮

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];
}
```

Outline

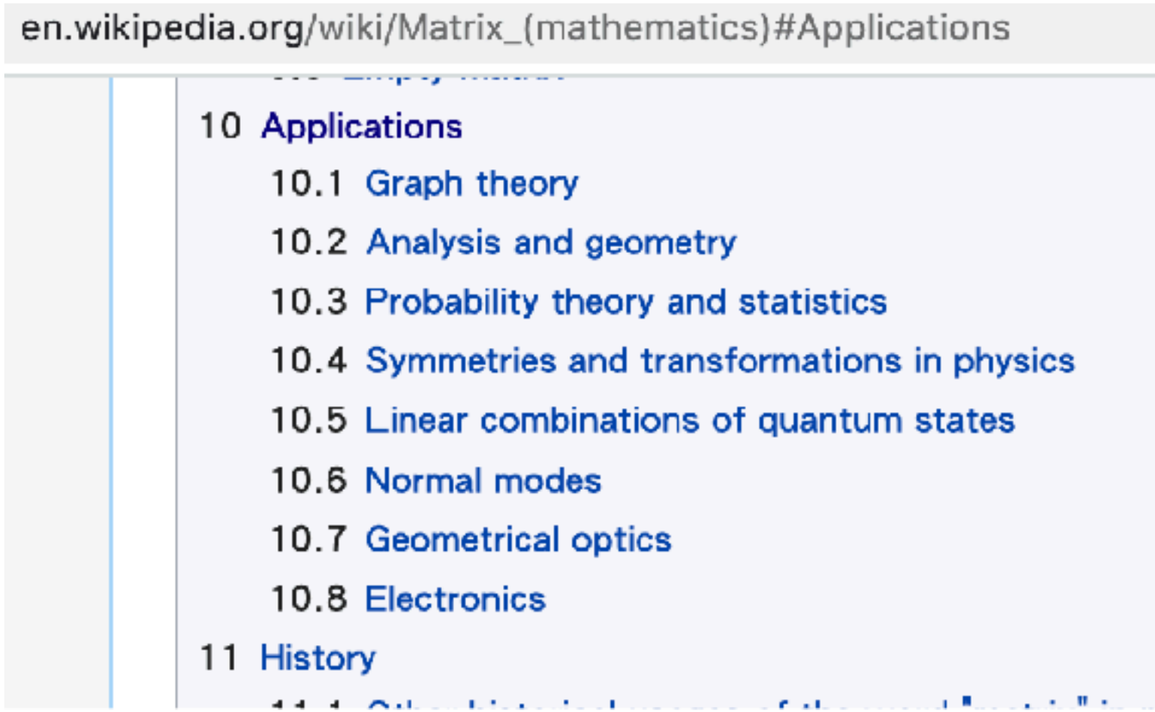
- Case study: matrix multiplications
- Optimizations for matrix multiplications

3:00

Why should we care about matrix multiplications?

What problems are “matrix-based”?

- Graph algorithms
- Dynamic programming
- Relational database operations
- Language recognition
- Or any other thing with matrices…



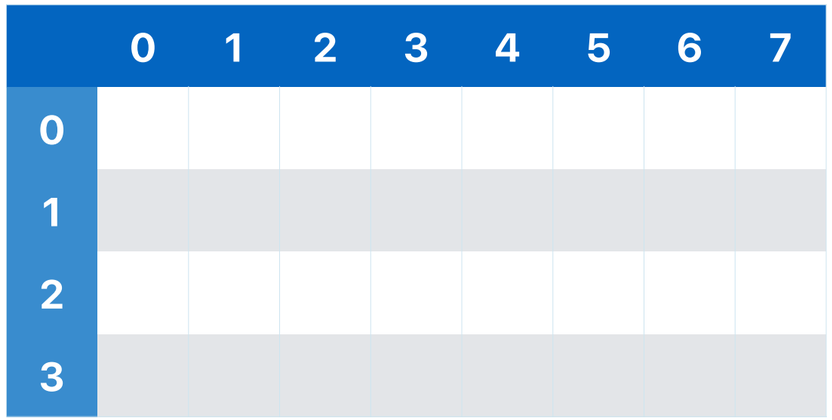
Case study: matrix multiplications

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

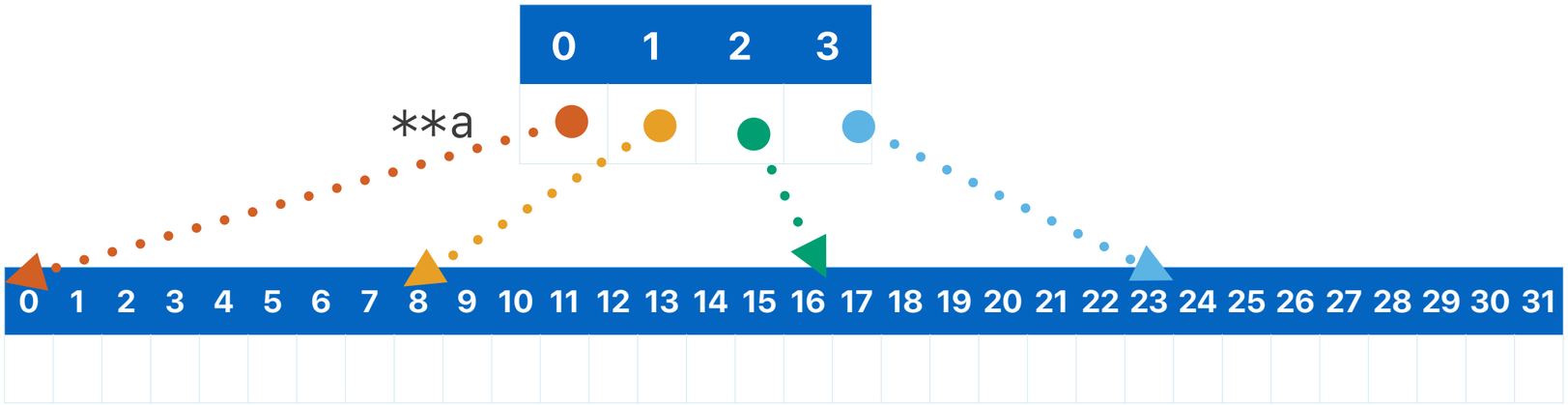
```
a = (double **)malloc(M*sizeof(double *));
for(i = 0; i < N; i++)
{
  a[i] = (double *)malloc(N*sizeof(double));
}
```

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

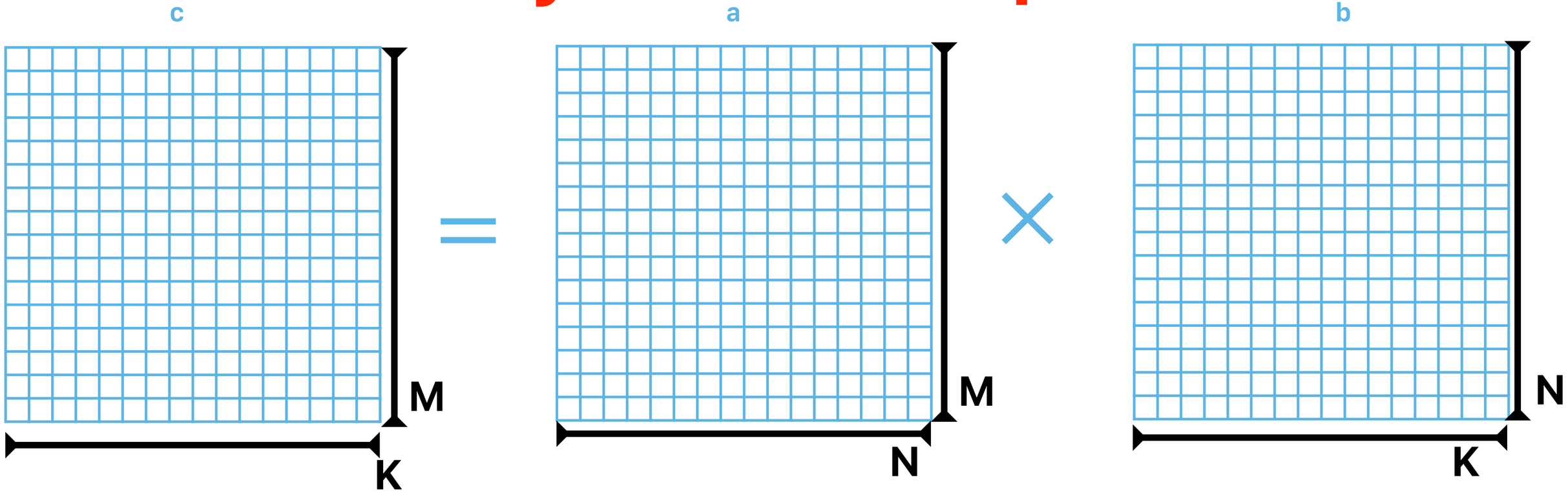
abstraction



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?

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

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

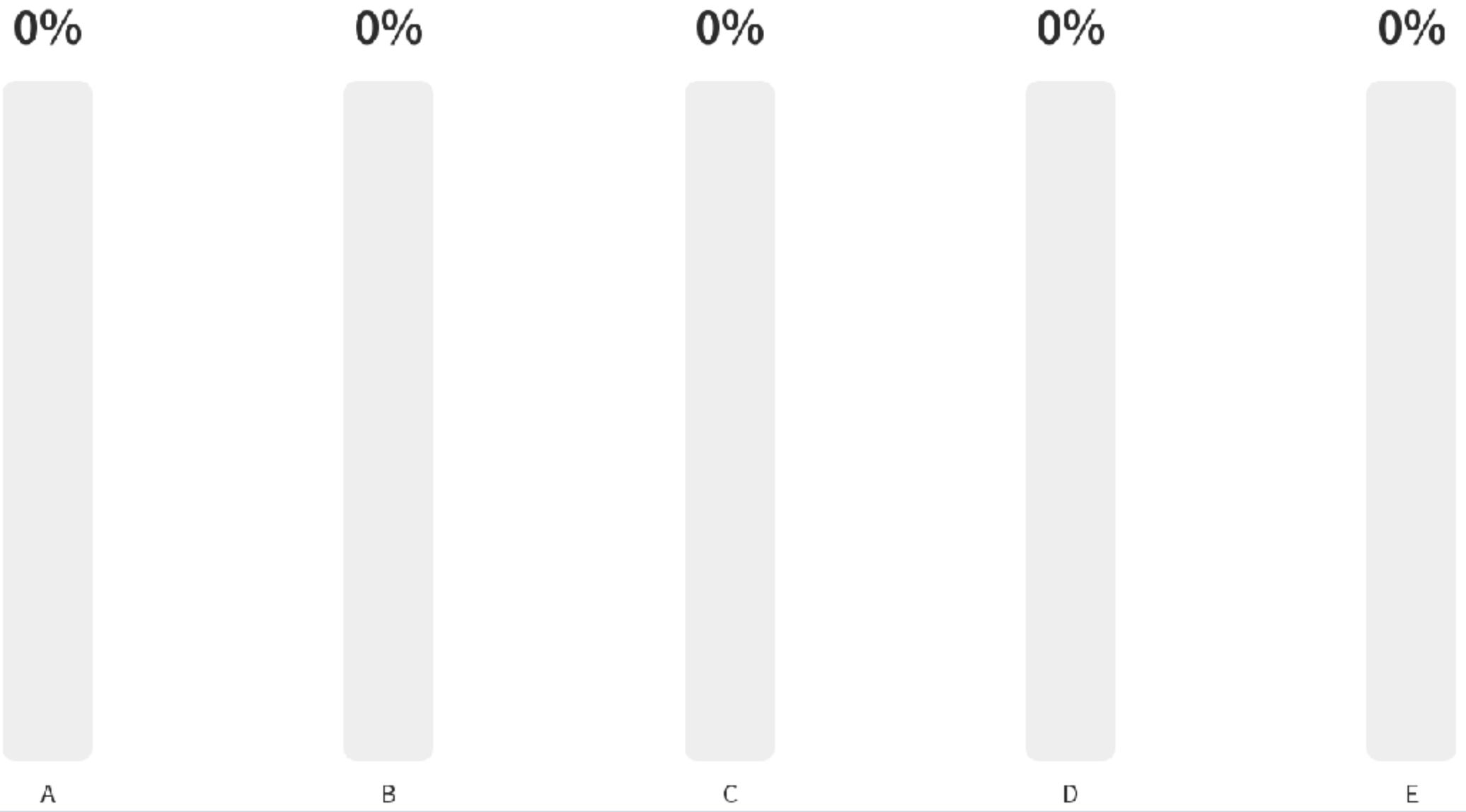
- Considering the case where M=N=K=2048, what do you think the majority type(s) of cache misses are we seeing on an intel processor with intel Core i7 is 48 KB, 12-way, 64-byte blocked L1-$?

```
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



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

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

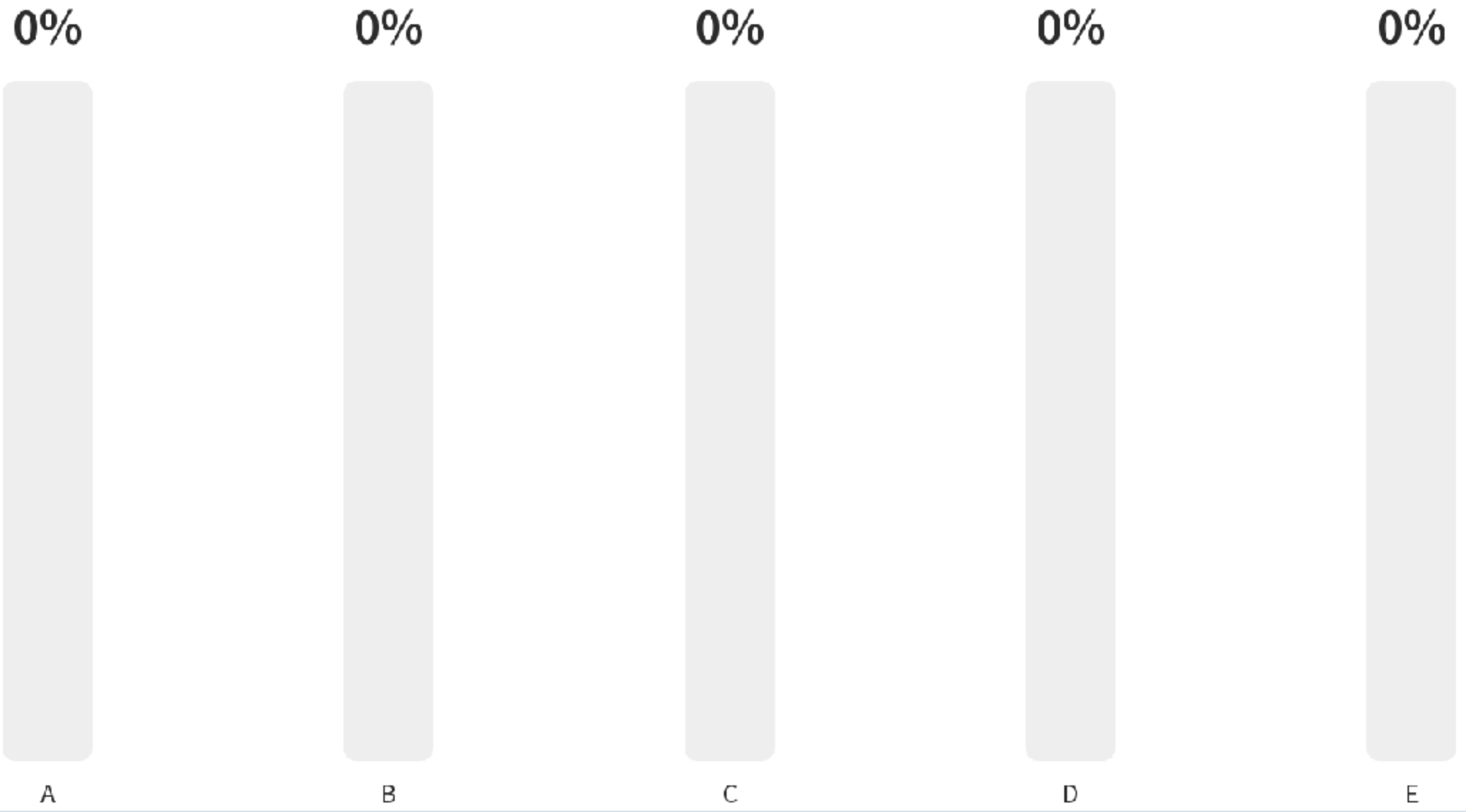
- Considering the case where M=N=K=2048, what do you think the majority type(s) of cache misses are we seeing on an intel processor with intel Core i7 is 48 KB, 12-way, 64-byte blocked L1-$?

```
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



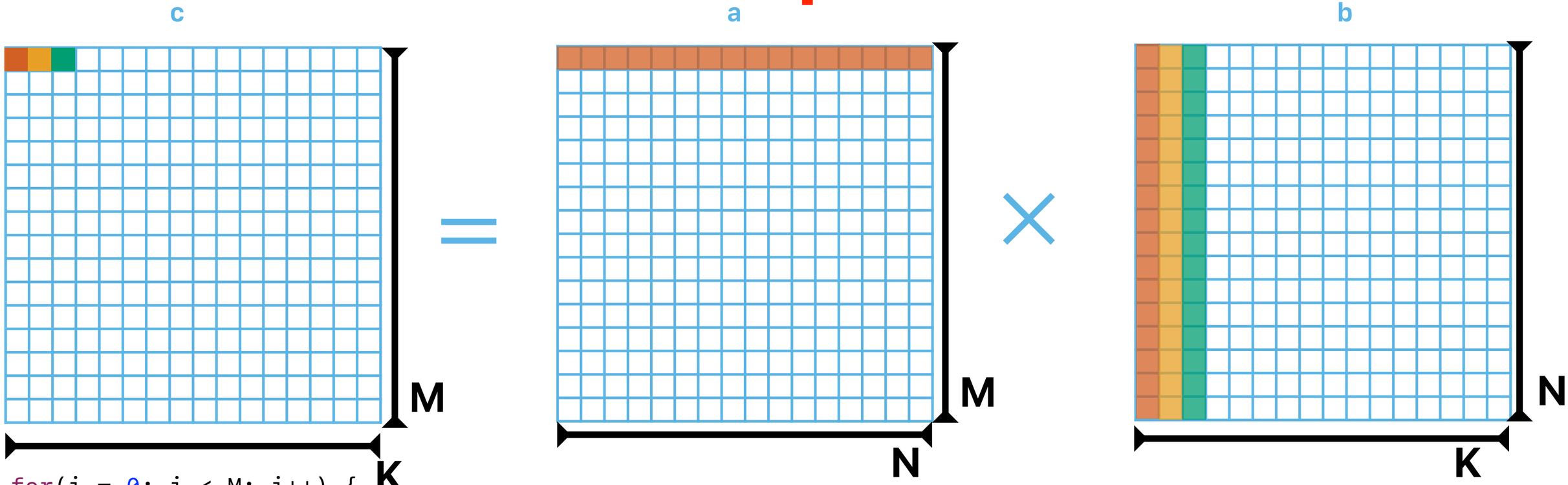
What kind(s) of misses are there in Matrix Multiplications

- 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];
    }
  }
}
```

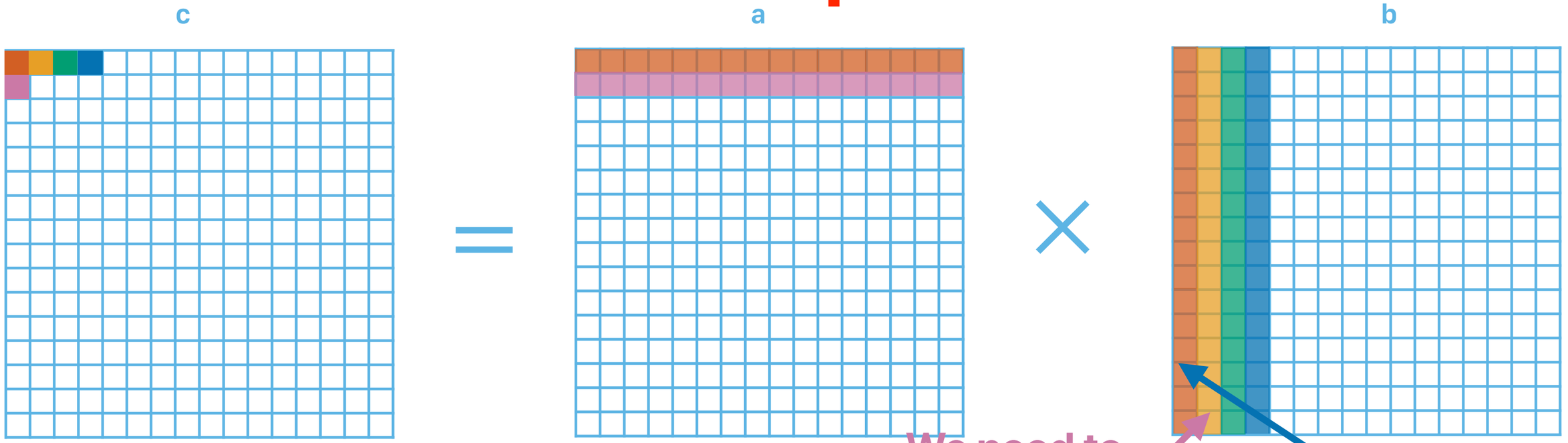
A. Compulsory miss
B. Capacity miss
C. Conflict miss
D. Capacity & conflict miss
E. Compulsory & conflict miss

Matrix Multiplications




```
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 Multiplications



- If each dimension of your matrix is 2048
  - Each row takes 2048×8 Bytes = 16 KB
  - Each column takes 2048×8 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

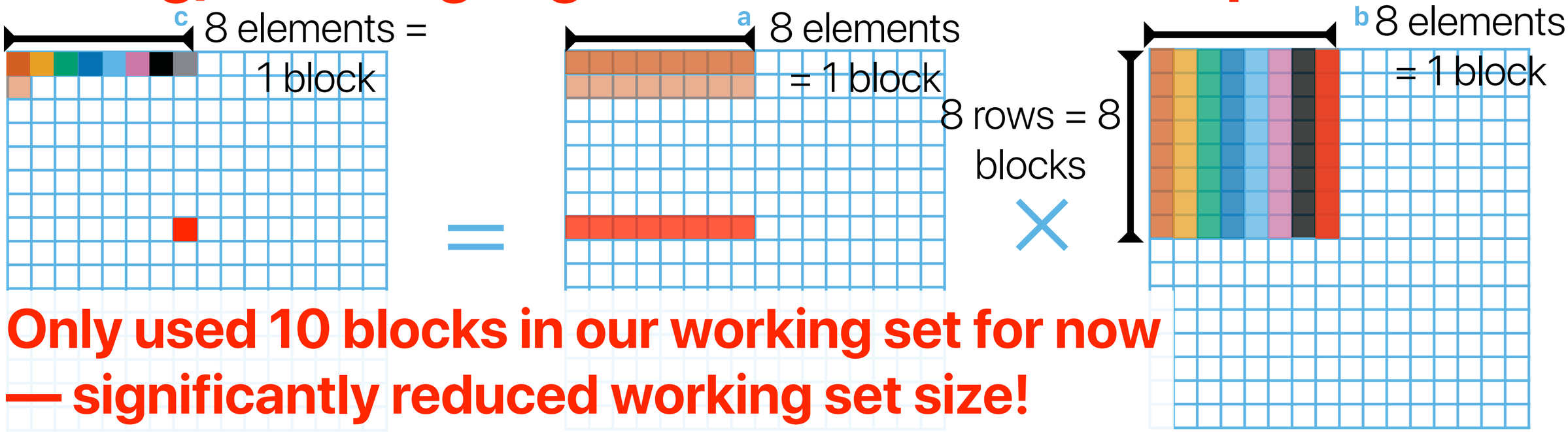
$$c_{i,j} = \sum_{k=0}^{k=N-1} a_{i,k} \times b_{k,j} = \sum_{k=0}^{k=\frac{N}{2}-1} a_{i,k} \times b_{k,j} + \sum_{k=\frac{N}{2}}^{k=N-1} a_{i,k} \times b_{k,j}$$

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

$$\vdots$$

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

Tiling/Blocking Algorithm for Matrix Multiplications



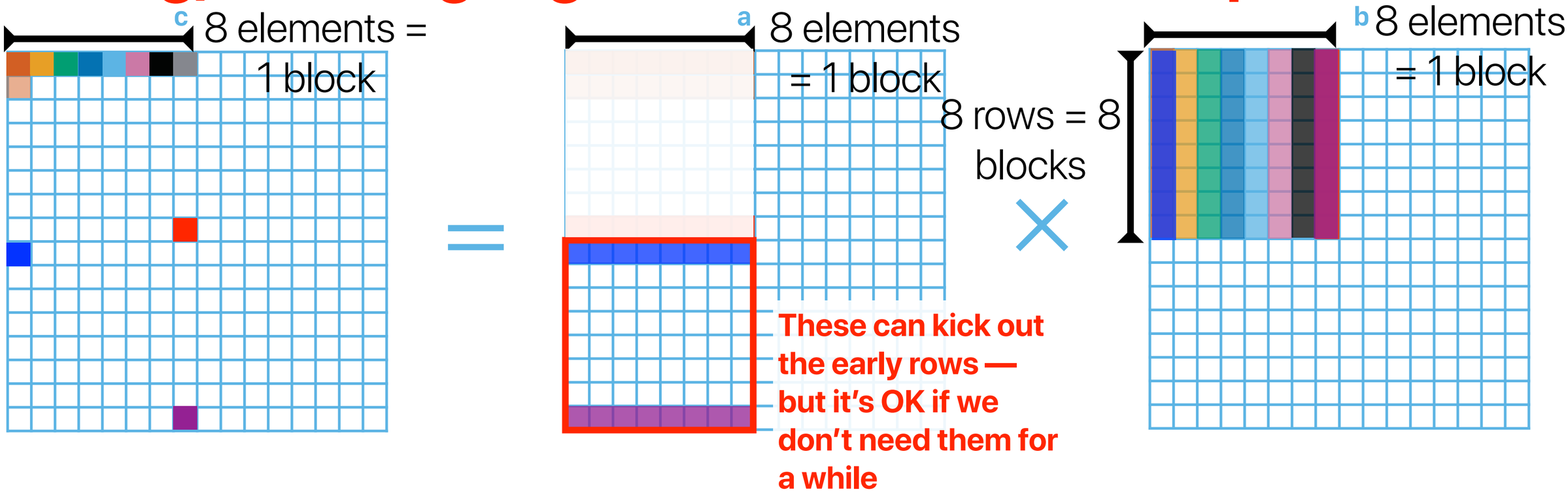
Only used 10 blocks in our working set for now — significantly reduced working set size!

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

Only compulsory misses —

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

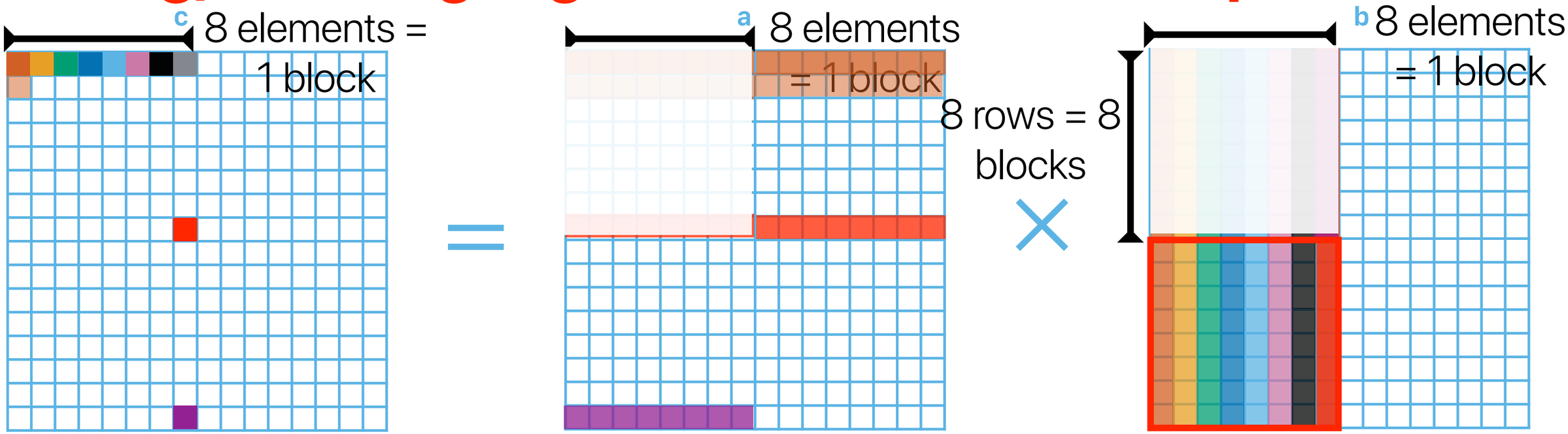
Tiling/Blocking Algorithm for Matrix Multiplications



Bringing miss rate even further lower now —

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

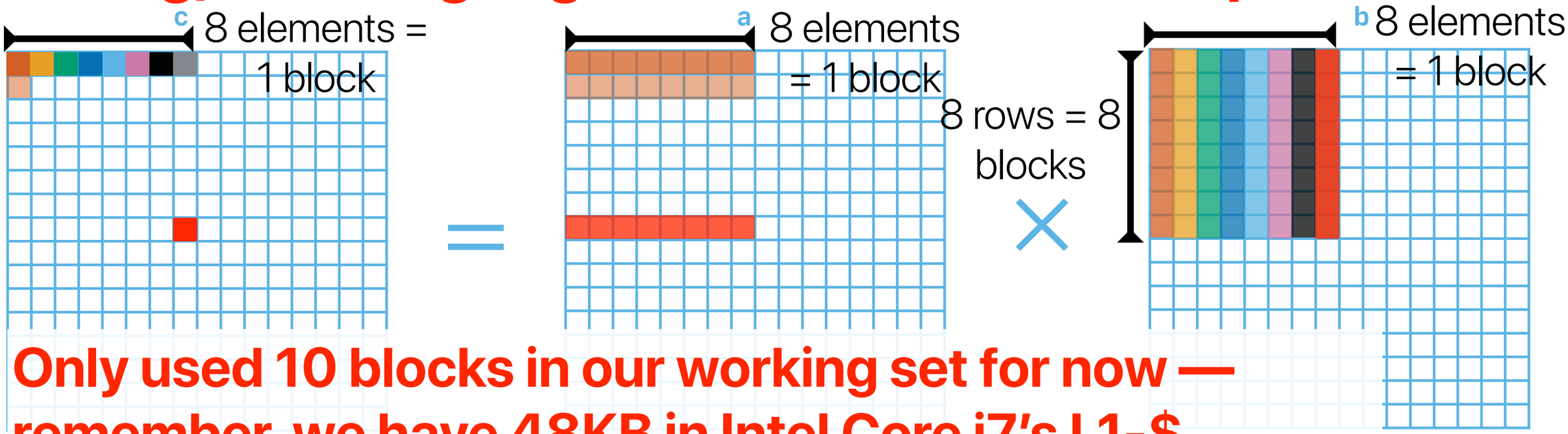
Tiling/Blocking Algorithm for Matrix Multiplications




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

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

What if we partition based on cache capacity?

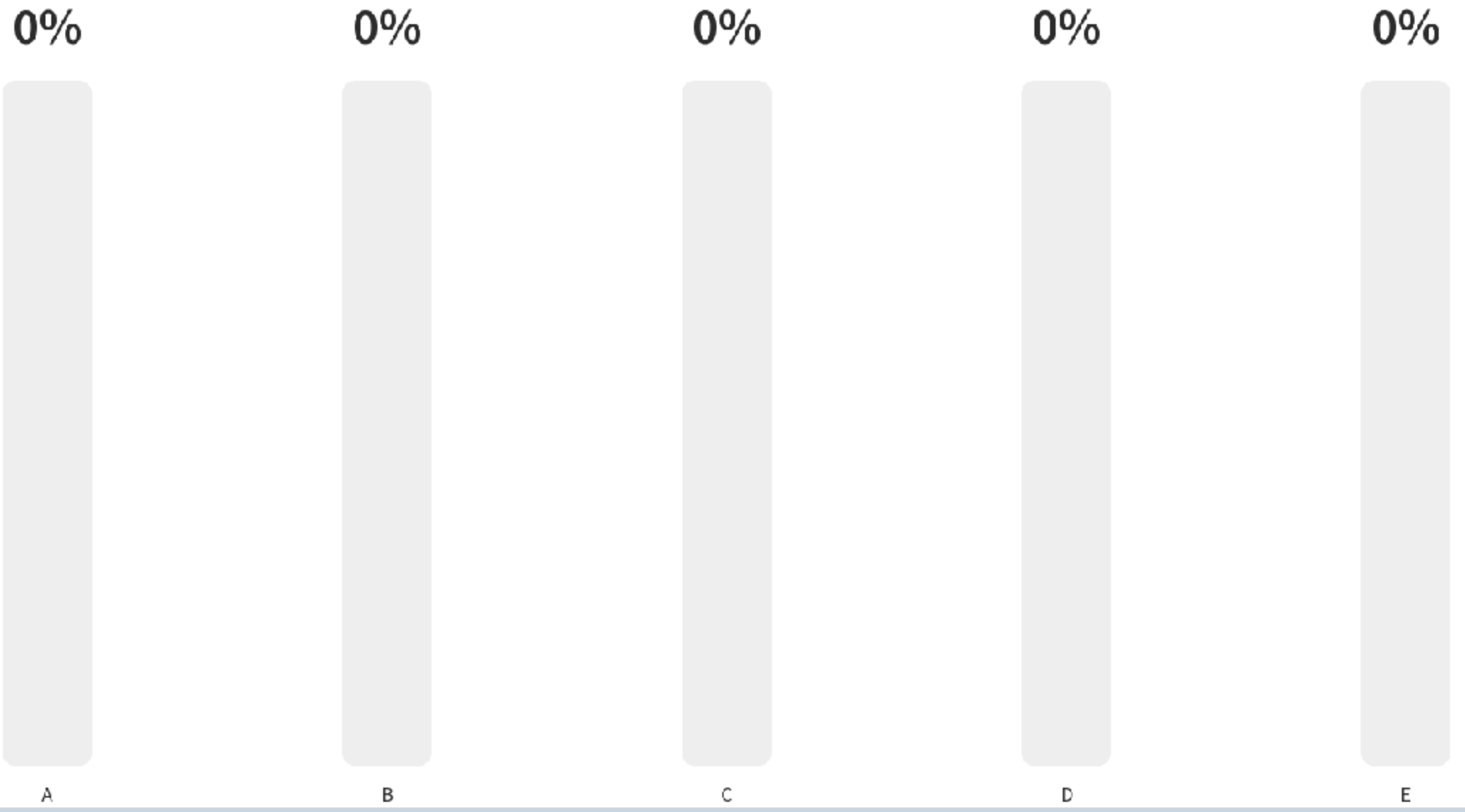
- Considering the case where M=N=K=2048, and a `tile_size=32`, what do you think the majority type(s) of cache misses are we seeing on an intel processor with intel Core i7 is 48 KB, 12-way, 64-byte blocked L1-$?

```
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



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

What if we partition based on cache capacity?

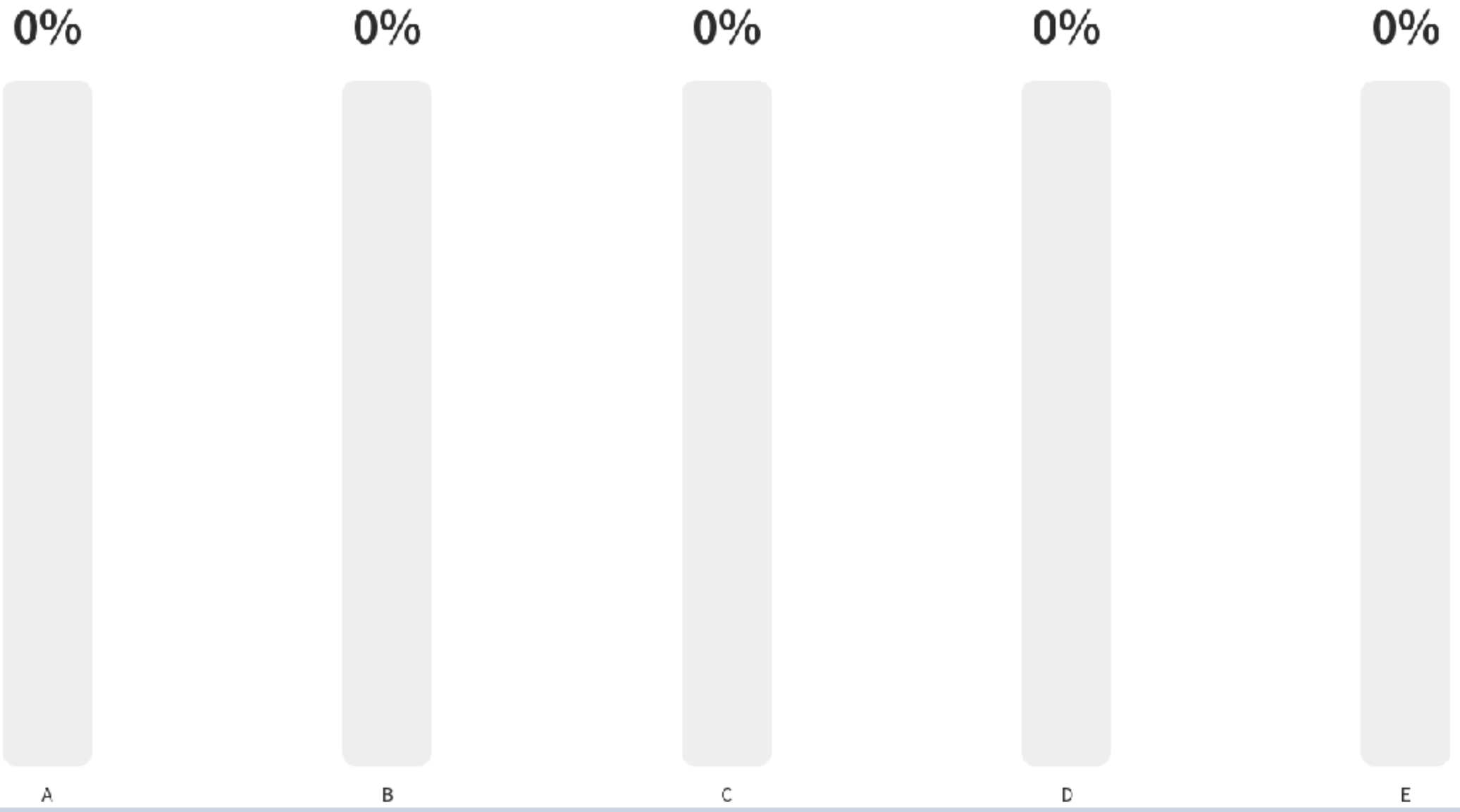
- Considering the case where M=N=K=2048, and a `tile_size=32`, what do you think the majority type(s) of cache misses are we seeing on an intel processor with intel Core i7 is 48 KB, 12-way, 64-byte blocked L1-$?

```
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



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

	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] …

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];
```

	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] ...

Why is “8” not the best performing?

size	tile_size	IC	Cycles	CPI	CT_ns	ET_s	DL1_miss_rate
2048	4	97777239805	29495935940	0.301665	0.179157	5.284412	0.015102
2048	8	81046430772	21228217500	0.261927	0.179061	3.801137	0.010388
2048	16	74473973614	19755427656	0.265266	0.179065	3.537511	0.072680
2048	32	71535221334	27876993012	0.389696	0.179240	4.996672	0.217053
2048	64	70144685635	32761508207	0.467056	0.179120	5.868227	0.242372
2048	128	69475139233	37635077503	0.541706	0.179043	6.738286	0.245563
2048	256	69174855313	56096341072	0.810935	0.179058	10.044526	0.240964
2048	512	69032750224	68635282626	0.994242	0.179063	12.290007	0.236299

More instructions due to more loop control overhead!

Best performing at 16?

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

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
- But be careful about the increasing amount of IC/loop control overhead

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!

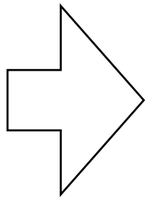
```
for(i = 0; i < M; i+=tile_size)
    for(j = 0; j < K; j+=tile_size)
        for(k = 0; k < N; k+=tile_size)
            for(ii = i; ii < i+tile_size; ii++)
                for(jj = j; jj < j+tile_size; jj++)
                {
                    result = 0;
                    for(kk = k; kk < k+tile_size; 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

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];
    }
  }
}
```

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

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];
    }
  }
}
```

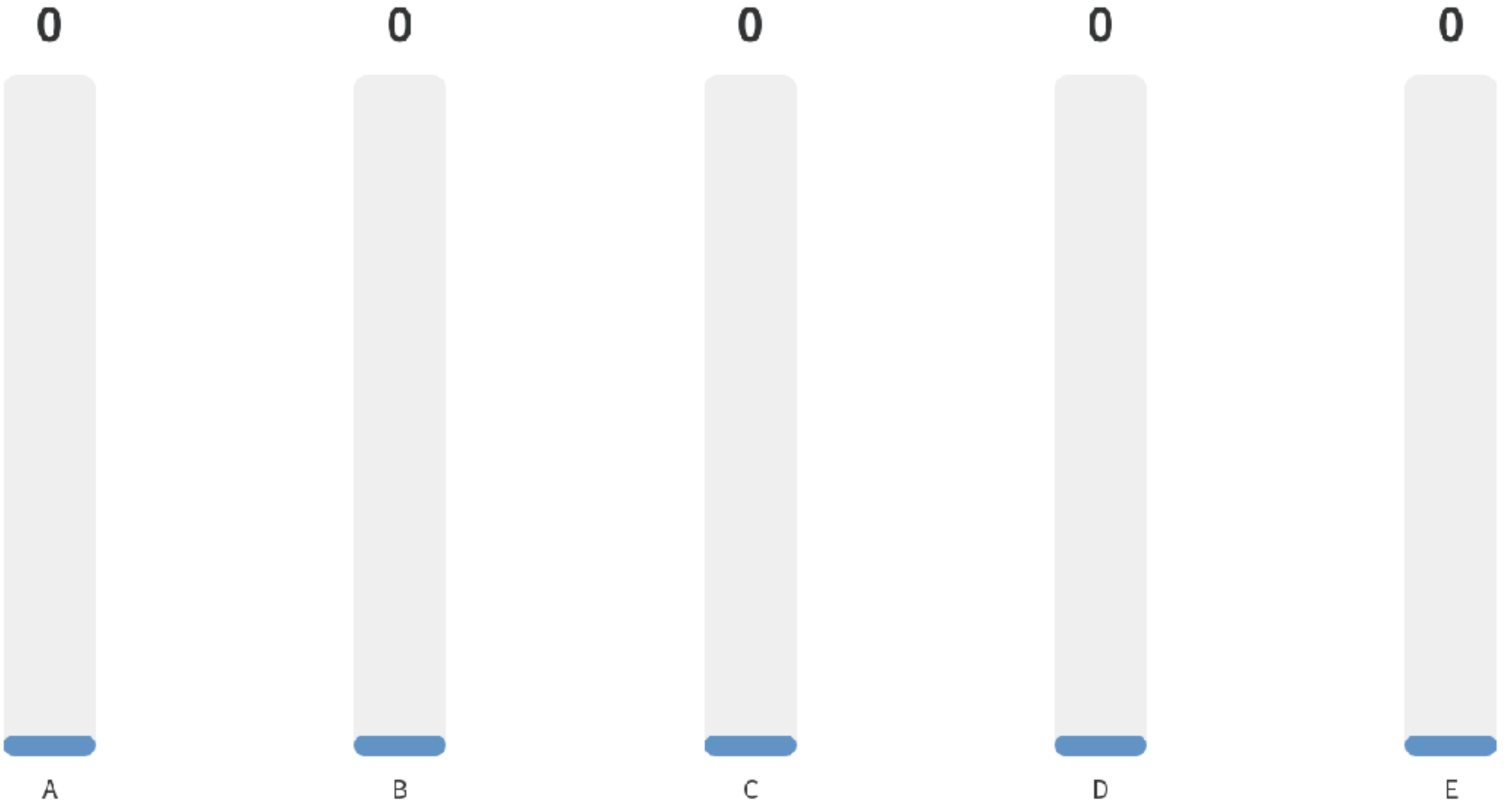
Block + Transpose

```
// 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

0



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];
    }
  }
}
```

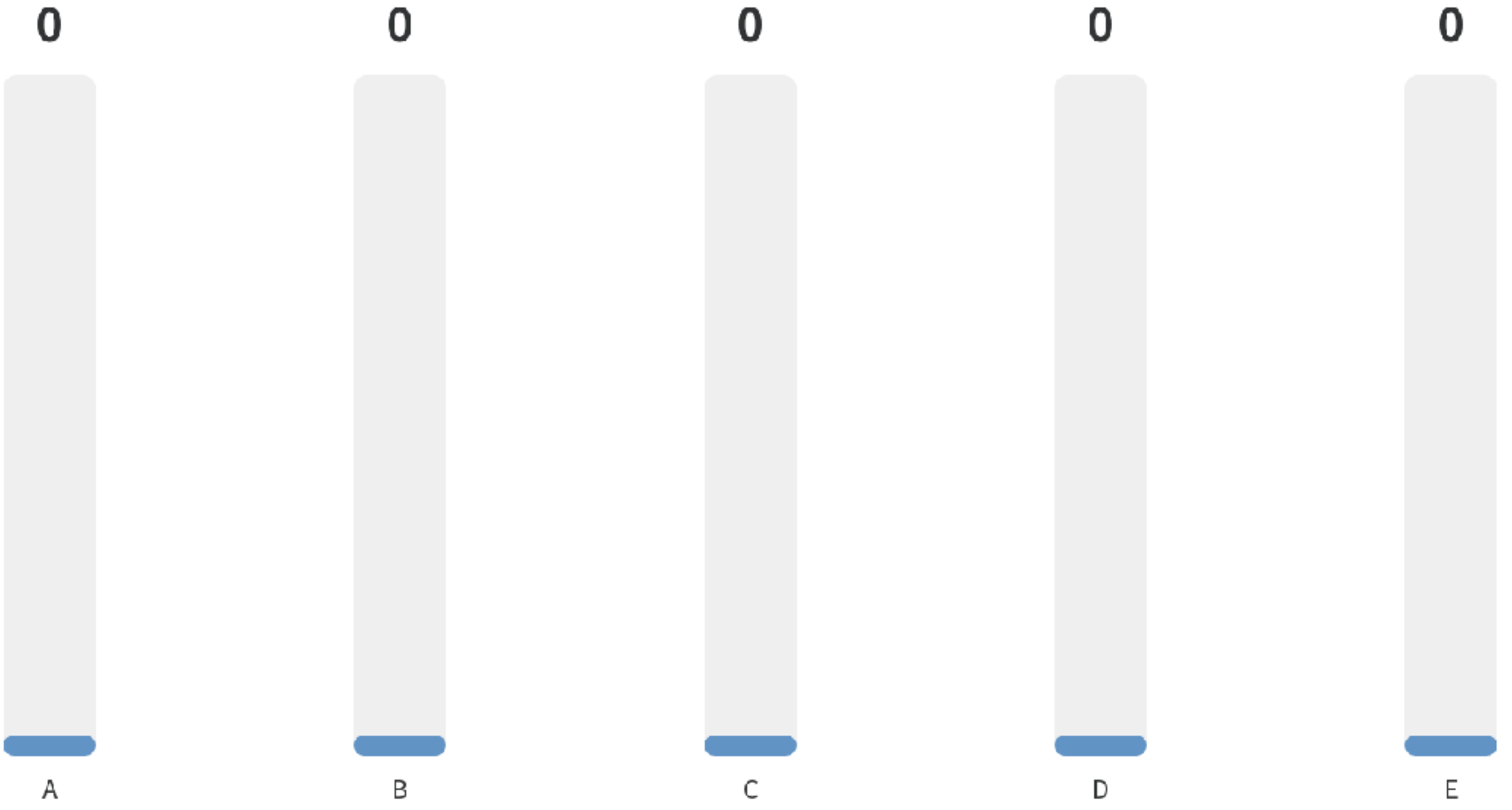
Block + Transpose

```
// 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

0



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];
    }
  }
}
```

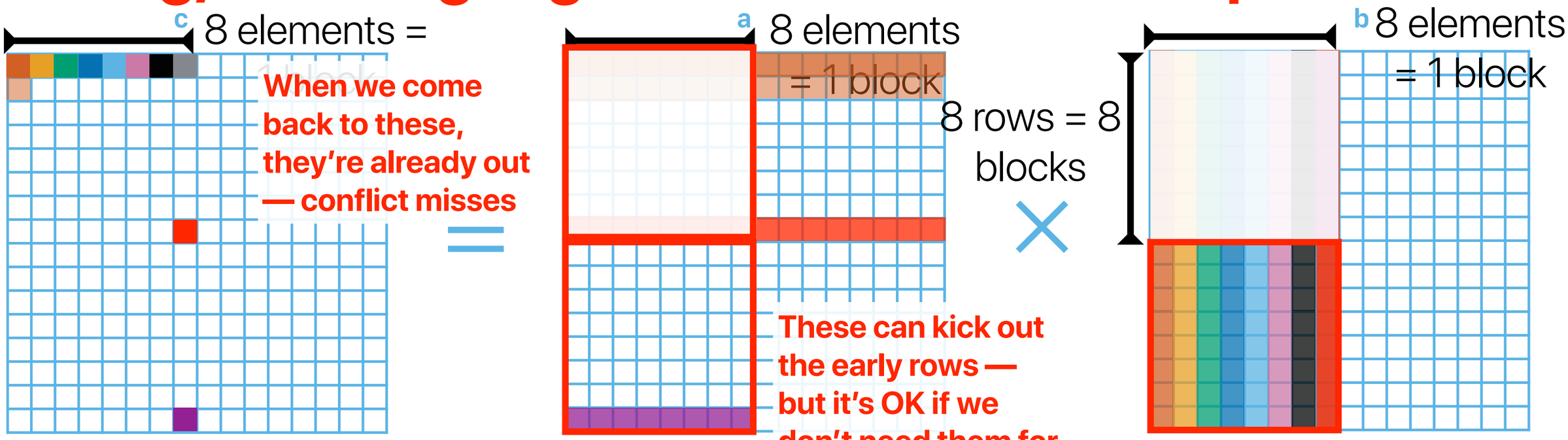
A. Compulsory miss
B. Capacity miss
C. Conflict miss
D. Capacity & conflict miss
E. Compulsory & conflict miss

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];
```

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];
    }
  }
}
```

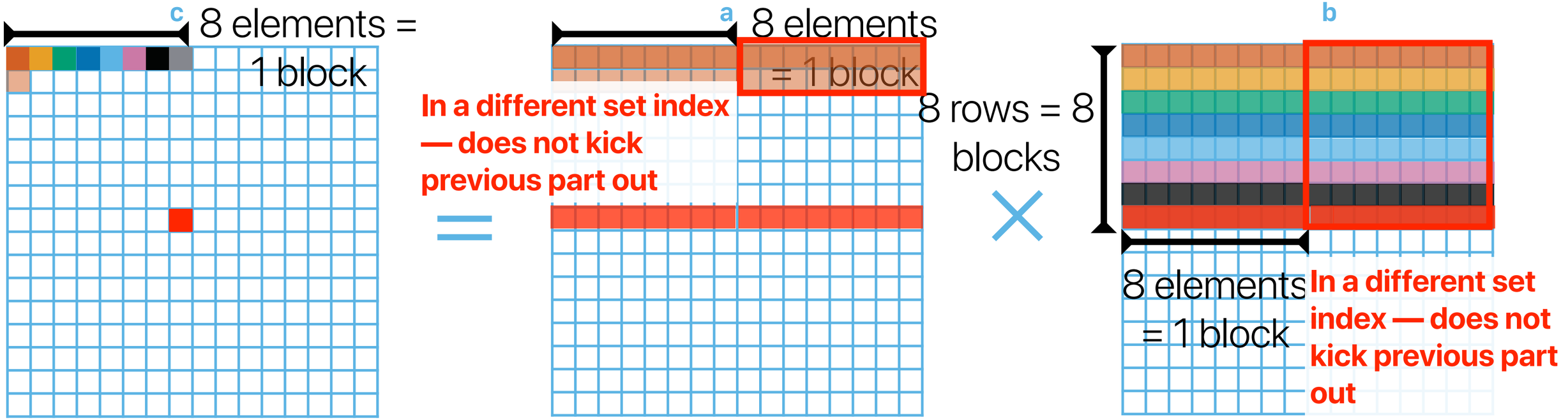
A. Compulsory miss
B. Capacity miss
C. Conflict miss
D. Capacity & conflict miss
E. Compulsory & conflict miss

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 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];
```

The effect of transposition

Block/tile

size	tile_size	IC	Cycles	CPI	CT_ns	ET_s	DL1_miss_rate
2048	4	97777239805	29495935940	0.301665	0.179157	5.284412	0.015394
2048	8	81046430772	21228217500	0.261927	0.179061	3.801137	0.010388
2048	16	74473973614	19755427656	0.265266	0.179065	3.537511	0.072680
2048	32	71535221334	27876993012	0.389696	0.179240	4.996672	0.217053
2048	64	70144685635	32761508207	0.467056	0.179120	5.868227	0.242372
2048	128	69475139233	37635077503	0.541706	0.179043	6.738286	0.245563
2048	256	69174855313	56096341072	0.810935	0.179058	10.044526	0.240964

Best performing from 16 to 64?

Block + Transpose

size	tile_size	IC	Cycles	CPI	CT_ns	ET_s	DL1_miss_rate
2048	8	70408494537	16474898796	0.233990	0.179050	2.949823	0.003063
2048	16	64863961959	14040845180	0.216466	0.179152	2.515449	0.027740
2048	32	62449092729	15013932980	0.240419	0.179006	2.687589	0.041441
2048	64	[illegible]	[illegible]	0.224164	0.179182	2.461678	0.023440
2048	128	[illegible]	[illegible]	0.282928	0.178999	3.077470	0.017444
2048	256	60495059818	17659377139	0.291914	0.179154	3.163753	0.024576

Transpose improves the miss rate at larger tile size!

The effect of transposition + register

Block + Transpose

size	tile_size	IC	Cycles	CPI	CT_ns	ET_s	DL1_miss_rate	DL1_accesses
2048	8	70408494537	16474898796	0.233990	0.179050	2.949823	0.003063	28795120136
2048	16	64863961959	14040845180	0.216466	0.179152	2.515449	0.027740	27069370660
2048	32	62449092729	15013932980	0.240419	0.179006	2.687589	0.041441	26388884006
2048	64	61287300182	13738422737	0.224164	0.179182	2.461678	0.023440	26075713629
2048	128	60766823678	17192655415	0.282928	0.178999	3.077470	0.017444	25946188195
2048	256	60495059818	17659377139	0.291914	0.179154	3.163753	0.024576	25877202822

Significant reduction of memory accesses

Block + Transpose + Reg.

size	tile_size	IC	Cycles	CPI	CT_ns	ET_s	DL1_miss_rate	DL1_accesses
2048	8	60790698108	16485243231	0.271180	0.179310	2.955969	0.003943	15892307771
2048	16	49276148749	10288720619	0.208797	0.179004	1.841719	0.066200	12015253818
2048	32	43911066047	9870555579	0.224785	0.179014	1.766965	0.096022	10272381116
2048	64	41301192477	10081924672	0.244107	0.179124	1.805919	0.073499	9437517387
2048	128	40017005995	11795575680	[illegible]	0.179014	2.111573	0.049084	9029619561
2048	256	39382447982	14335991956	0.364020	0.179105	2.567647	0.019428	8828955663

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;
              }
          }
      }
   }
}
```

If we could have a rectangular tile + transposition

Block + Transpose + Reg.

size	tile_size	IC	Cycles	CPI	CT_ns	ET_s	DL1_miss_rate
2048	8	60790698108	16485243231	0.271180	0.179310	2.955969	0.003943
2048	16	49276148749	10288720619	0.208797	0.179004	1.841719	0.066200
2048	32	43911066047	9870555579	0.224785	0.179014	1.766965	0.096022
2048	64	41301192477	10081924672	0.244107	0.179124	1.805919	0.073499
2048	128	40017005995	11795578680	0.294764	0.179014	2.111573	0.049084
2048	256	393[illegible]	[illegible]	0.364020	0.179105	[illegible]	0.019428

Very low miss rate compared to 32x32

Rect. Block + Transpose + Reg.

size	tile_size_x	tile_size_y	IC	Cycles	CPI	CT_ns	ET_s	DL1_miss_rate
2048	8	8	60695609484	11687337167	0.192557	0.179044	2.092552	0.004755
2048	8	16	59756790511	12537990317	0.209817	0.179048	2.244906	0.023099
2048	16	8	49689298094	11447344529	0.230378	0.179035	2.049476	0.004942
2048	16	16	[illegible]	[illegible]	[illegible]	[illegible]9391	1.801964	0.066462
2048	32	16	[illegible]	[illegible]	[illegible]	[illegible]9039	1.685461	0.005974
2048	32	8	43946648680	9760965906	0.222109	0.179051	1.747714	0.064311
2048	64	8	41431489003	9754293903	0.235432	0.179057	1.746575	0.006423
2048	128	8	40059768822	11729101534	0.292790	0.179035	2.099916	0.007373
2048	256	8	39376341663	13824142364	0.351077	0.179305	2.478741	0.003962

Best performing at 32x16

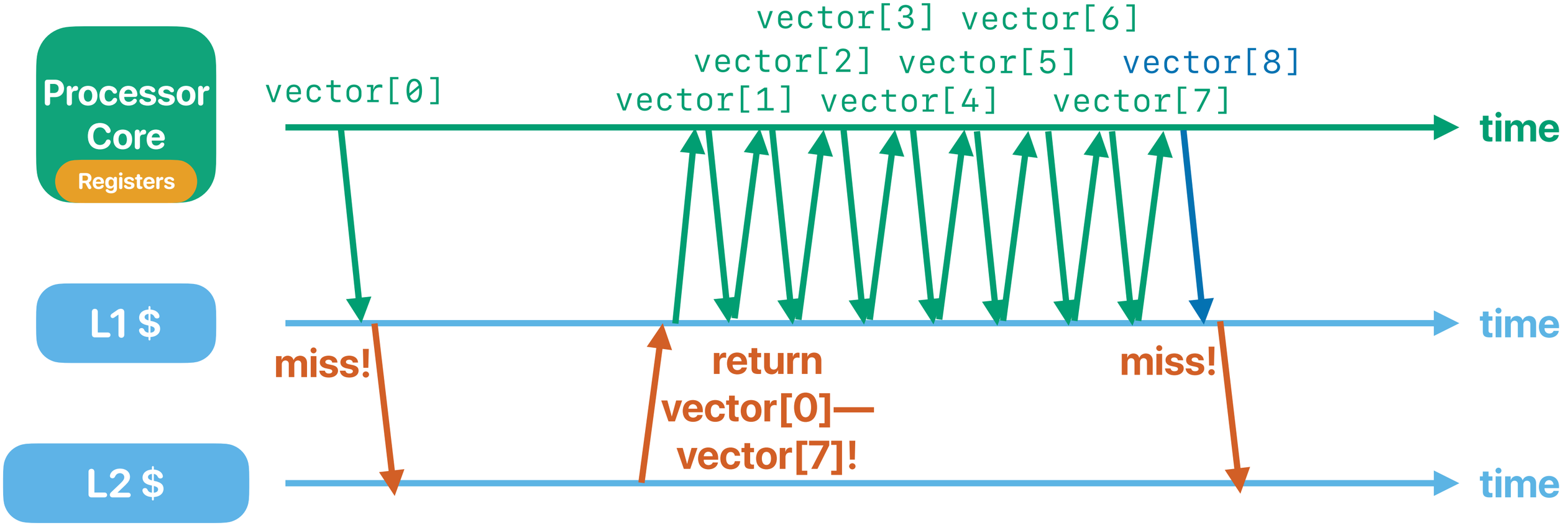
Takeaways: Software Optimizations

- 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!

Prefetching

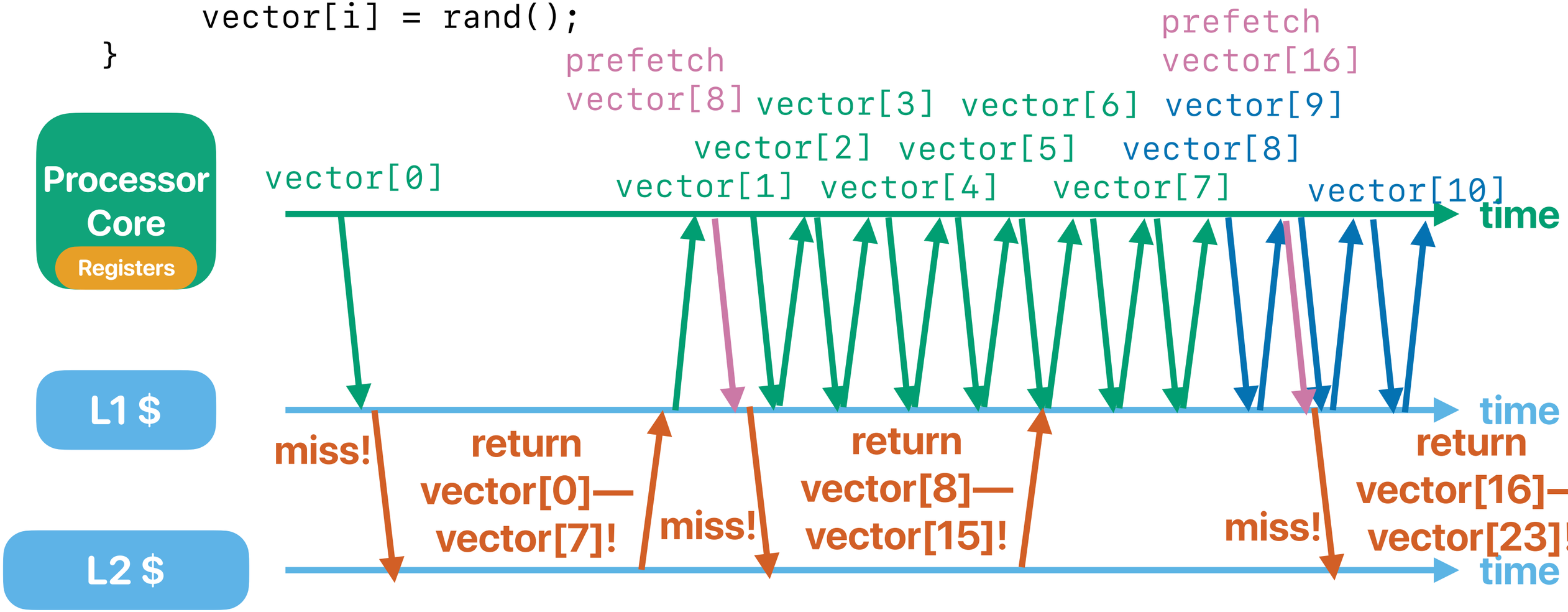
Spatial locality revisited

```
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();
}
```



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
- 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

Sample Midterm

Format

- When and where
  - Schedule with Testing Center through https://us.prairietest.com/pt/
  - Only between 10/27 and 10/31. It's your responsibility to schedule
  - Testing center will handle the testing related thing. You are responsible for being there on time and following the rules. Contact Testing center if you need additional assistance. I don't handle those
  - 80 minutes
  - Closed note, closed book, no cheatsheet or outside materials allowed
- 15 Multiple Choice Questions — 45%
- 3 Assignment Style Free Answer Questions — 55%

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];
  }
}
```

Practicing Amdahl's Law (2)

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

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 reflective is “inference per second”

- Regarding inferences per second (IPS), please identify how many of the following statements are correct
  - ① IPS can change if the application changes the ML model used
  - ② IPS can become worse if the application adds more features to improve the quality of the answer or the precision
  - ③ IPS can improve if the system applies a hardware that offers higher FLOPS
  - ④ IPS remains the same if the system/application answers questions 20x slower but can answer 20x more questions in parallel
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

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

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];
    }
  }
}
```

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

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 in improving conflict misses?
  - ① Missing cache
  - ② Victim cache
  - ③ Prefetch
  - ④ Stream buffer
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

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

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

Practicing Amdahl's Law (2)

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

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

1. How many sets do we have?
2. How many offset bits are there in the address?
3. How many index bits are there in the address?
4. How many tag bits are there in the address?

Simulate a 2-way cache

- A 2-way cache with 256 bytes total capacity, a block size of 16 bytes
- We're accessing the addresses on the right. Can you complete the table with their address partitions and identify its hit or miss? If it's a miss, what type of miss is it?

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

Announcement

- Reserve your slots for midterm ASAP!
- Check your grades on https://www.escalab.org/my_grades
- Assignment #2 due **tonight**
  - 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 Tuesday** before the lecture

Computer
Science &
Engineering

203

つづく