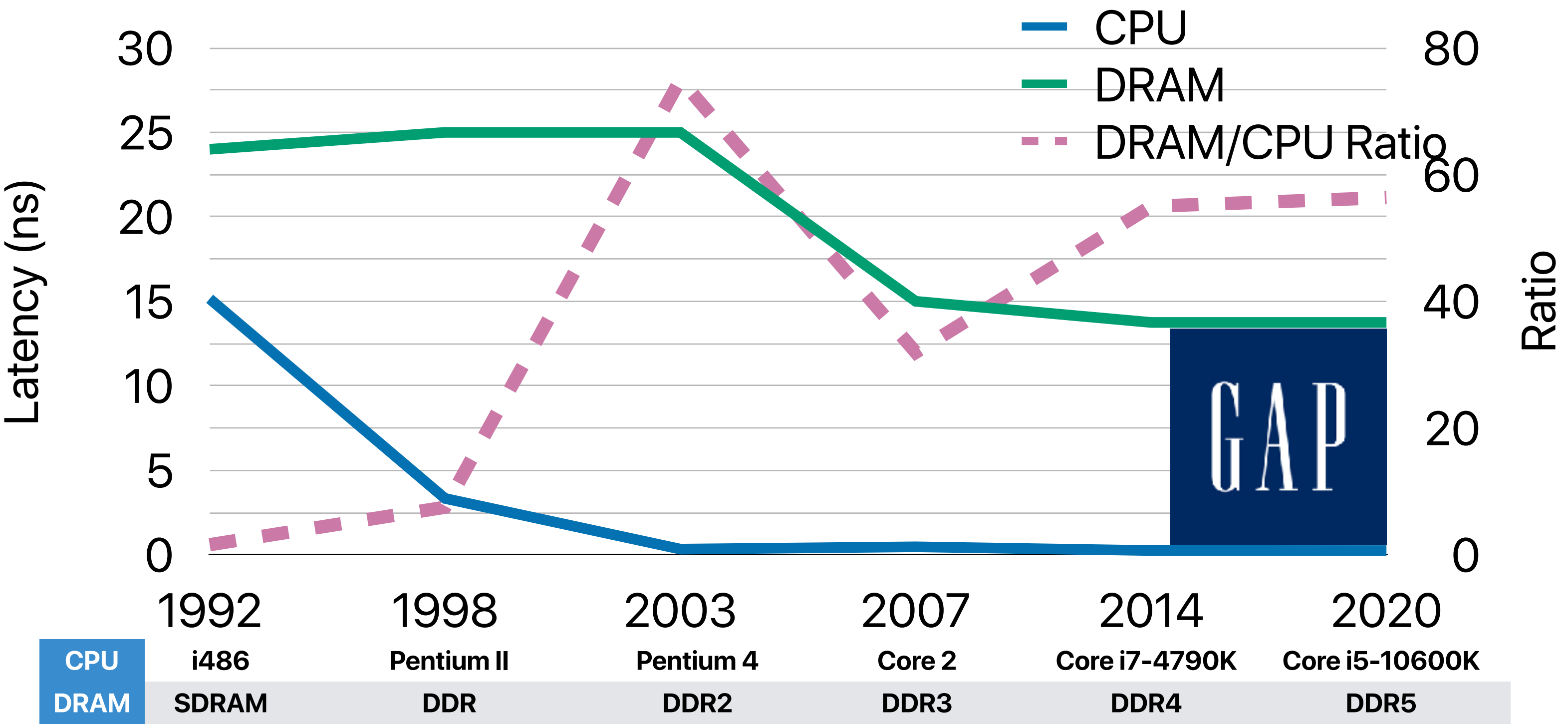


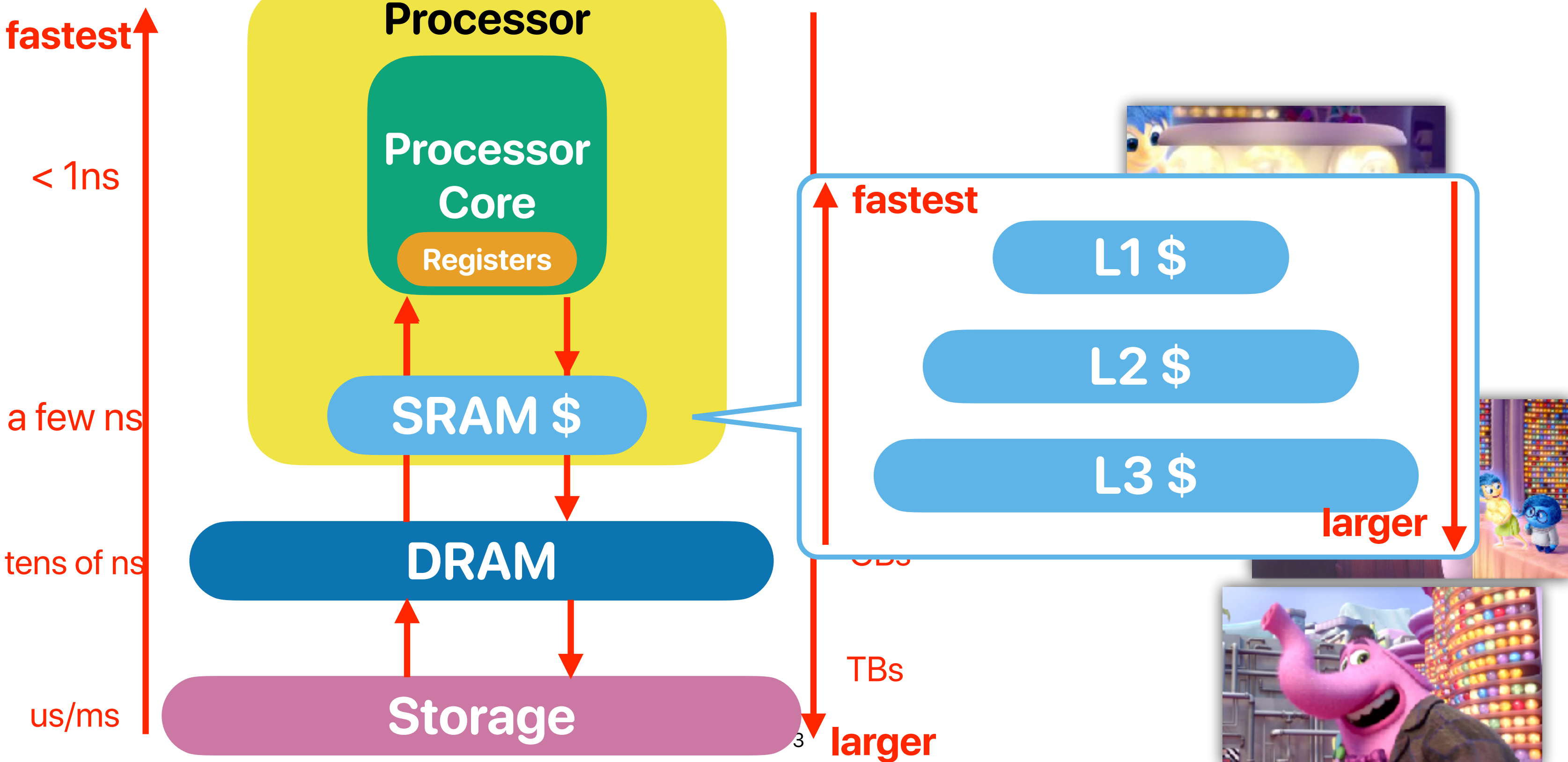
Memory Hierarchy (5): Cache Misses and How to Address Them — A Case Study

Hung-Wei Tseng

Recap: The "latency" gap between CPU and DRAM



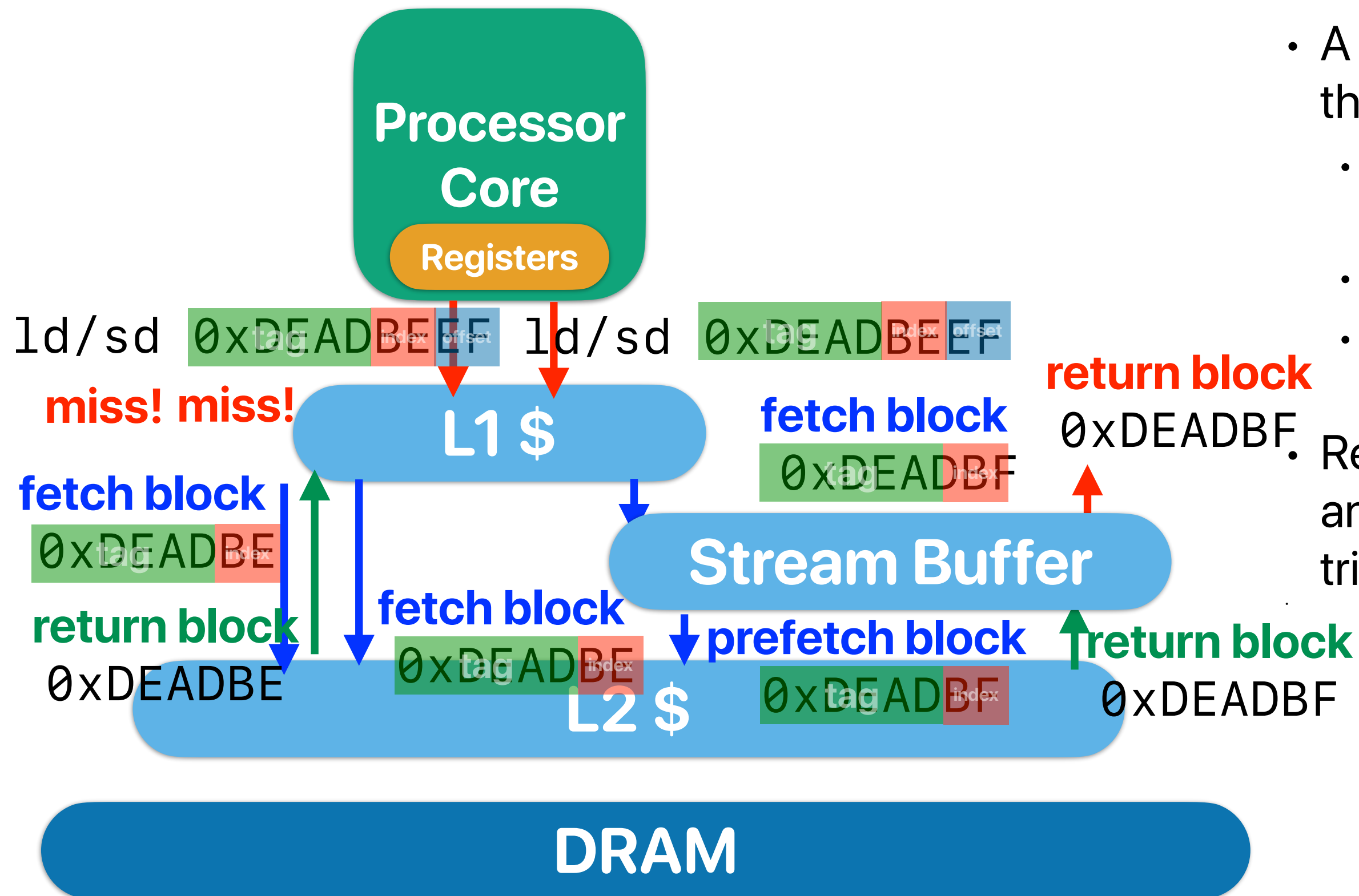
Recap: Memory Hierarchy



Recap: 3Cs of misses

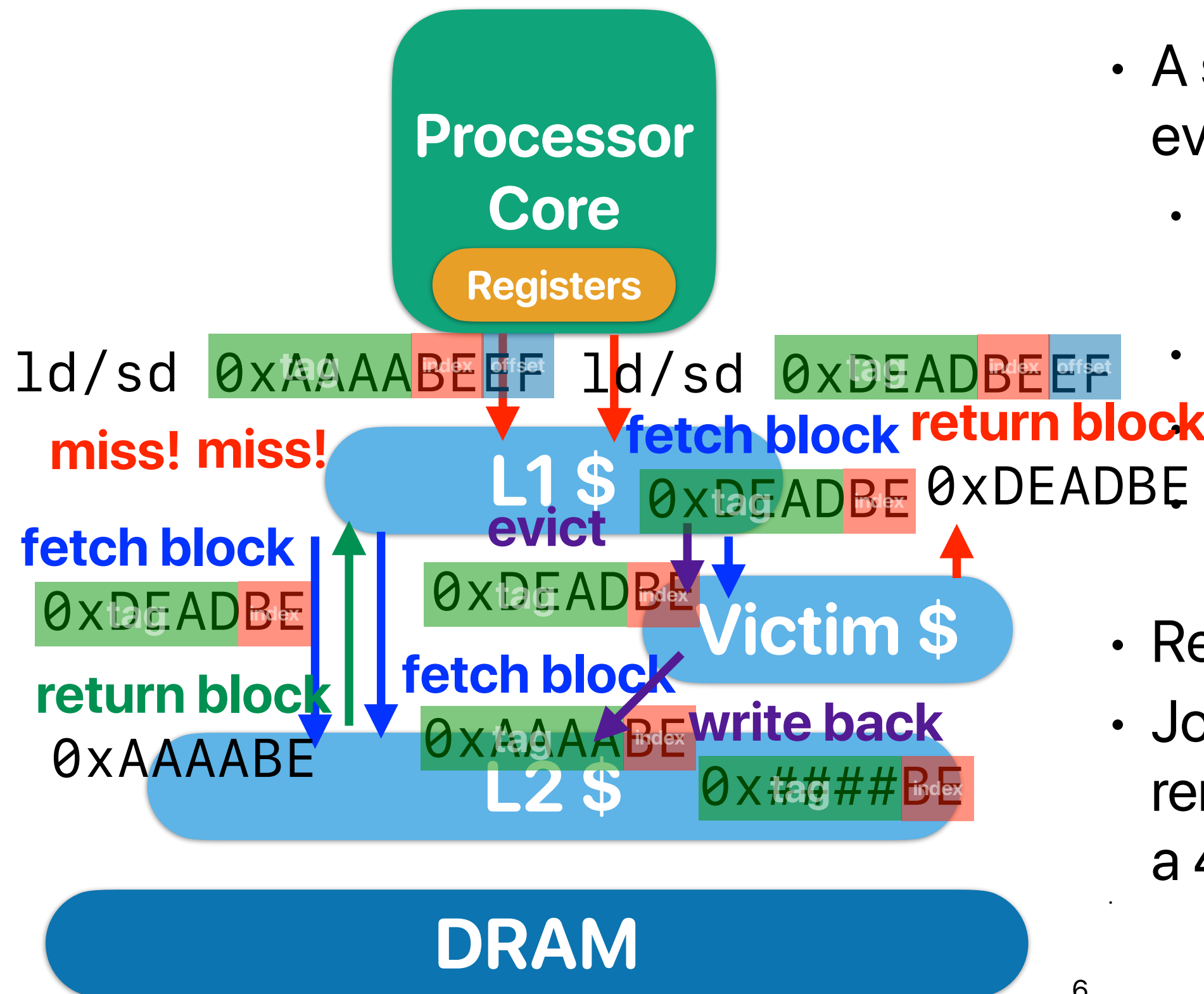
- Compulsory miss
 - Cold start miss. **First-time access to a block**
- Capacity miss
 - The working set size of an application is bigger than cache size
 - Working set size means the total size of **cache blocks** we visit between the last access and the current access of the current block
- Conflict miss
 - Required data replaced by block(s) mapping to the same set
 - Similar collision in hash
 - The working set size is still smaller than the capacity

Recap: Stream buffer



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

Recap: Victim cache



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

Array of structures or structure of arrays

Array of objects					object of arrays					
<pre>struct grades { int id; double assignment_1, assignment_2, assignment 3, ...; };</pre>					<pre>struct grades { int *id; double *assignment_1, *assignment_2, *assignment_3, ...; };</pre>					
ID	assignment_1	assignment_2	assignment_3	...	ID	assignment_1	assignment_2	assignment_3	...	
average of each homework	<pre>for(i=0;i<homework_items; i++) { gradesheet[total_number_students].homework[i] = 0.0; for(j=0;j<total_number_students;j++) gradesheet[total_number_students].homework[i] +=gradesheet[j].homework[i]; gradesheet[total_number_students].homework[i] /= (double)total_number_students; }</pre>					<pre>for(i = 0;i < homework_items; i++) { gradesheet.homework[i][total_number_students] = 0.0; for(j = 0; j <total_number_students;j++) { gradesheet.homework[i][total_number_students] += gradesheet.homework[i][j]; } gradesheet.homework[i][total_number_students] /= total_number_students; }</pre>				
	ID	ID	ID	assignment_1	assignment_1	assignment_1	assignment_2	assignment_2	assignment_2	assignment_3

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];
    }
}
```



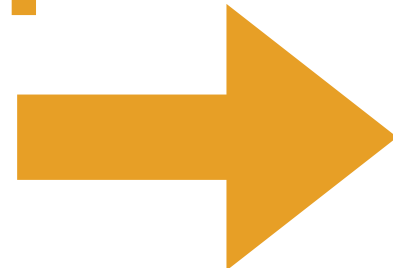
B

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

Loop fission

A

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



A

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

Loop fusion

B

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 8192; 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?

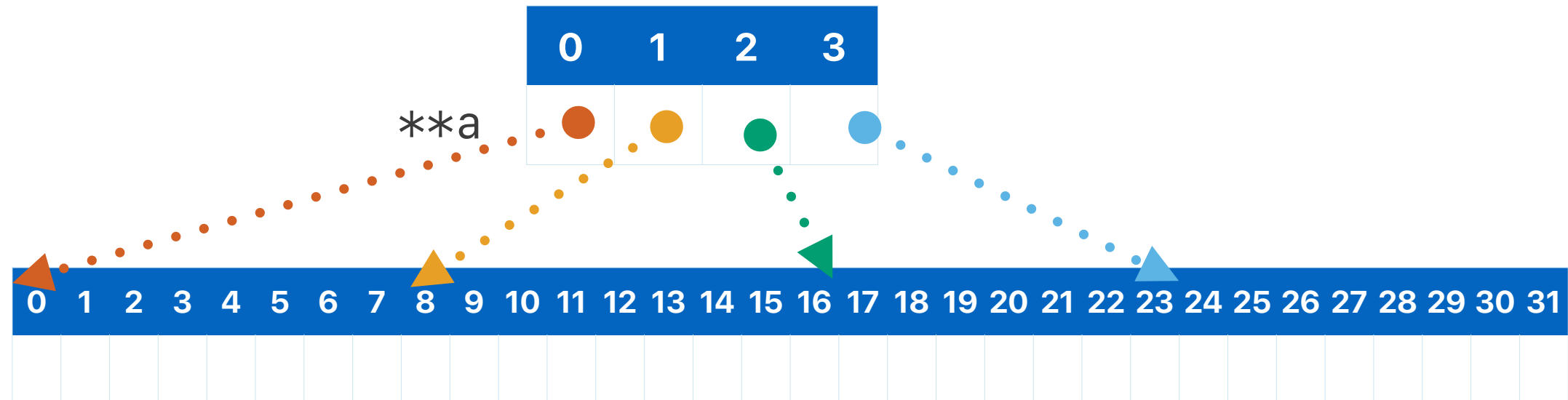
```
a = (double **)malloc(M*sizeof(double *));  
for(i = 0; i < N; i++)  
{  
    a[i] = (double *)malloc(N*sizeof(double));  
}
```

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

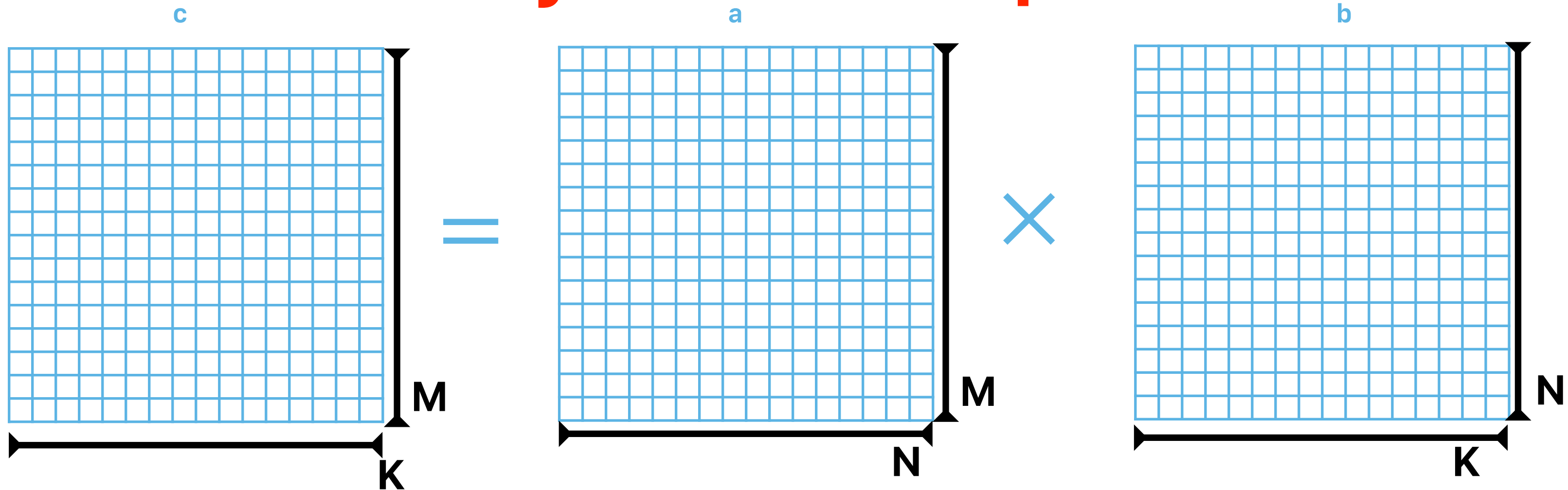
abstraction

	0	1	2	3	4	5	6	7
0								
1								
2								
3								

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 $M=N=K=2048$?



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



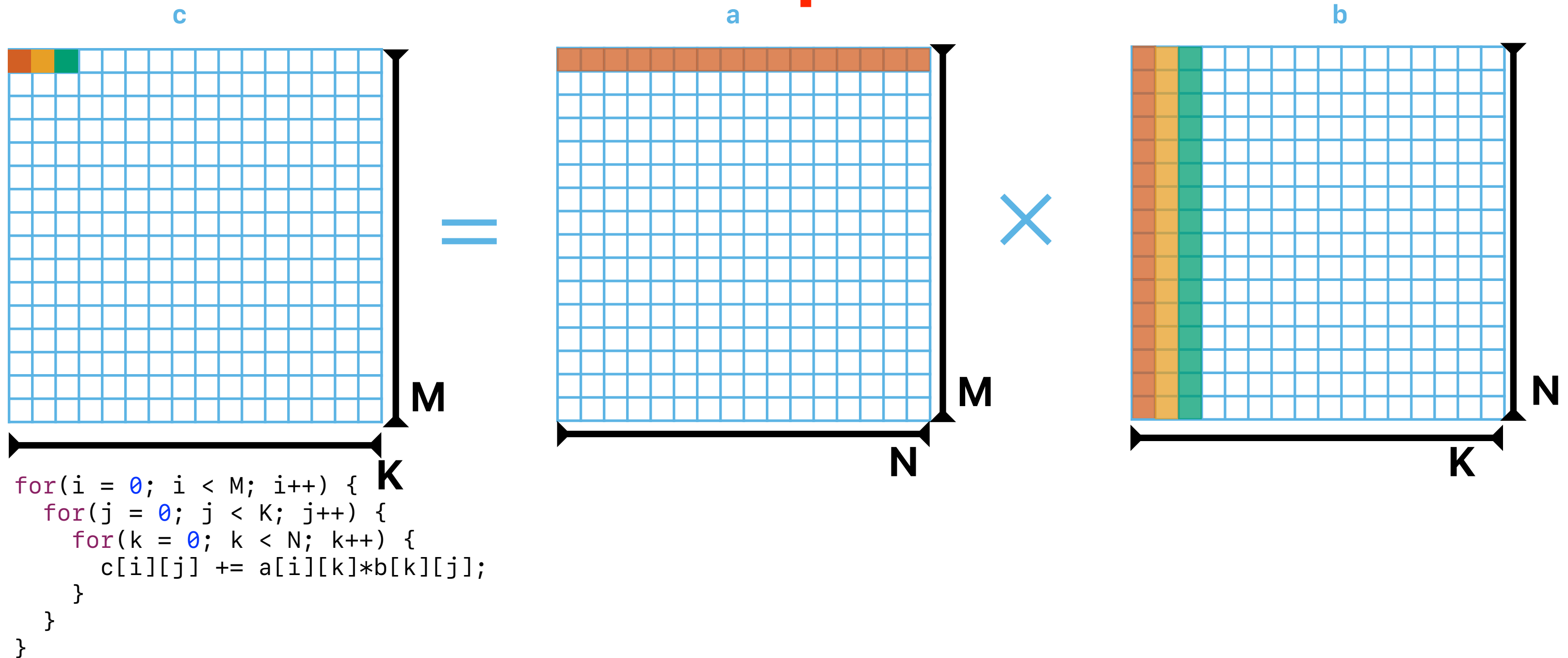
What kind(s) of misses are there in Matrix Multiplications

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

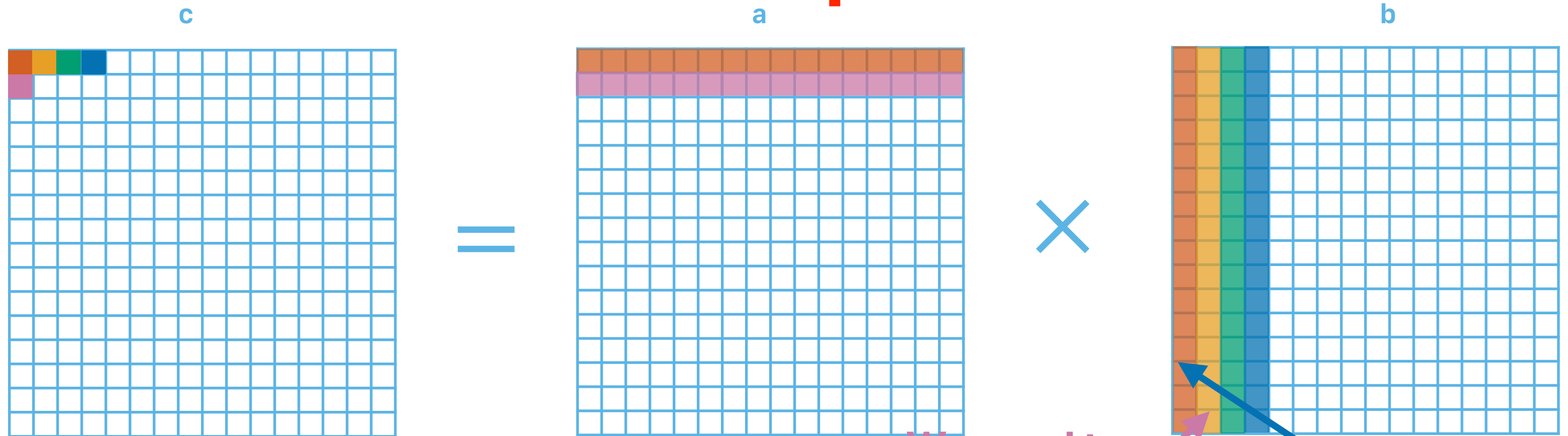
```
for(i = 0; i < M; i++) {  
    for(j = 0; j < K; j++) {  
        for(k = 0; k < N; k++) {  
            c[i][j] += a[i][k]*b[k][j];  
        }  
    }  
}
```

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

Matrix Multiplications



Matrix Multiplications



- If each dimension of your matrix is 2048
 - Each row takes $2048 \times 8 \text{ Bytes} = 16 \text{ KB}$
 - Each column takes $\frac{2048 \times 8B}{8B} = 2048 \text{ blocks}$
 - The L1-\$ of intel Core i7 is 48 KB, 12-way, 64-byte blocked, we only have $\frac{48 \times 1024B}{64B} = 768 \text{ blocks}$
 - You can only hold at most 3 rows or 0.25 of a column of each matrix!

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

Ideas regarding reducing misses in matrix multiplications

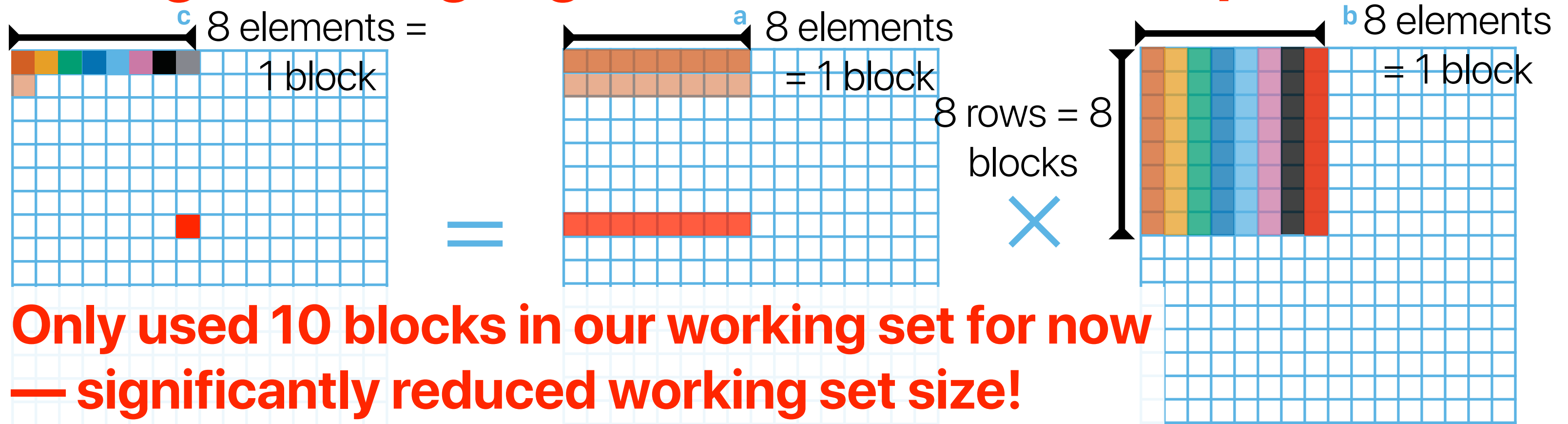
- Reducing capacity misses — we need to reduce the length of a row that we visit within a period of time

Mathematical view of MM

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

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

Tiling/Blocking Algorithm for Matrix Multiplications

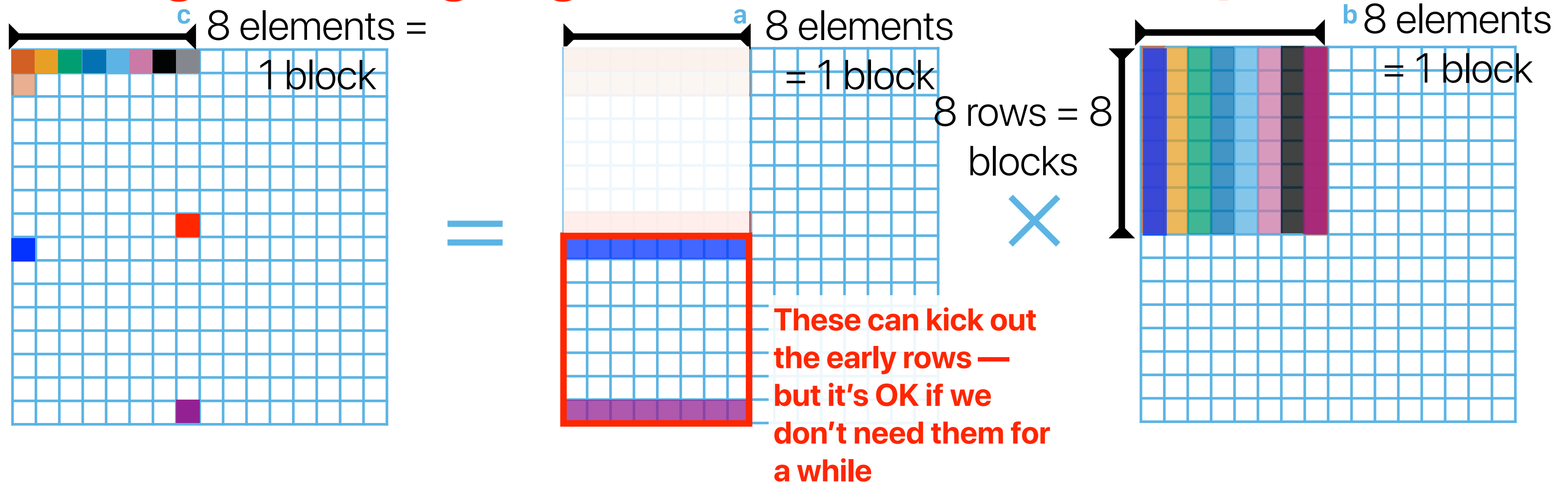


These are still around when we move to the next row in the "tile"

Only compulsory misses —

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

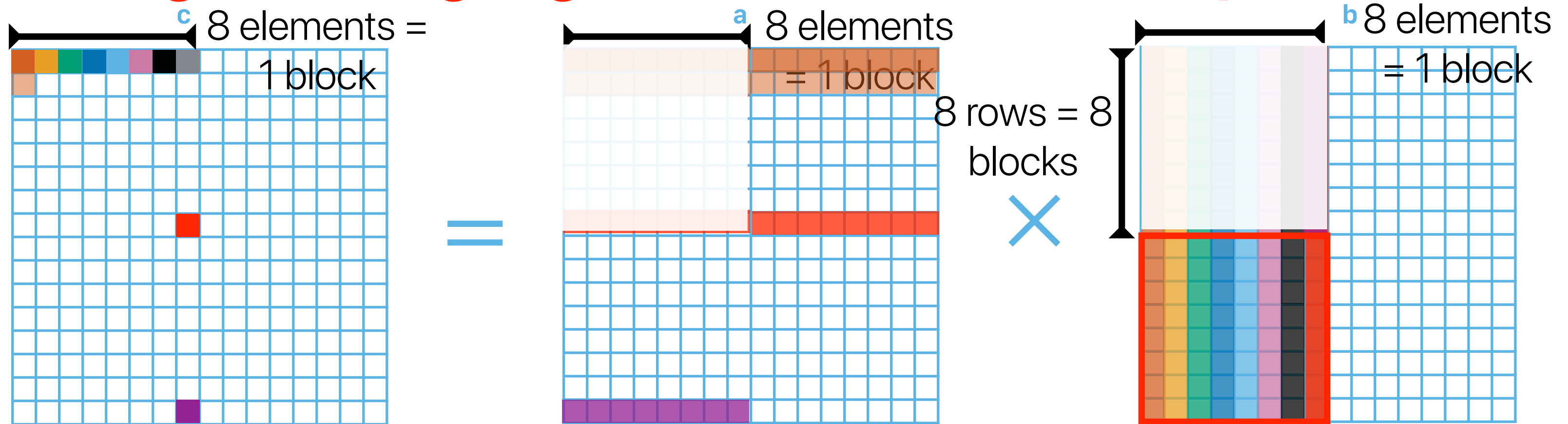
Tiling/Blocking Algorithm for Matrix Multiplications



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

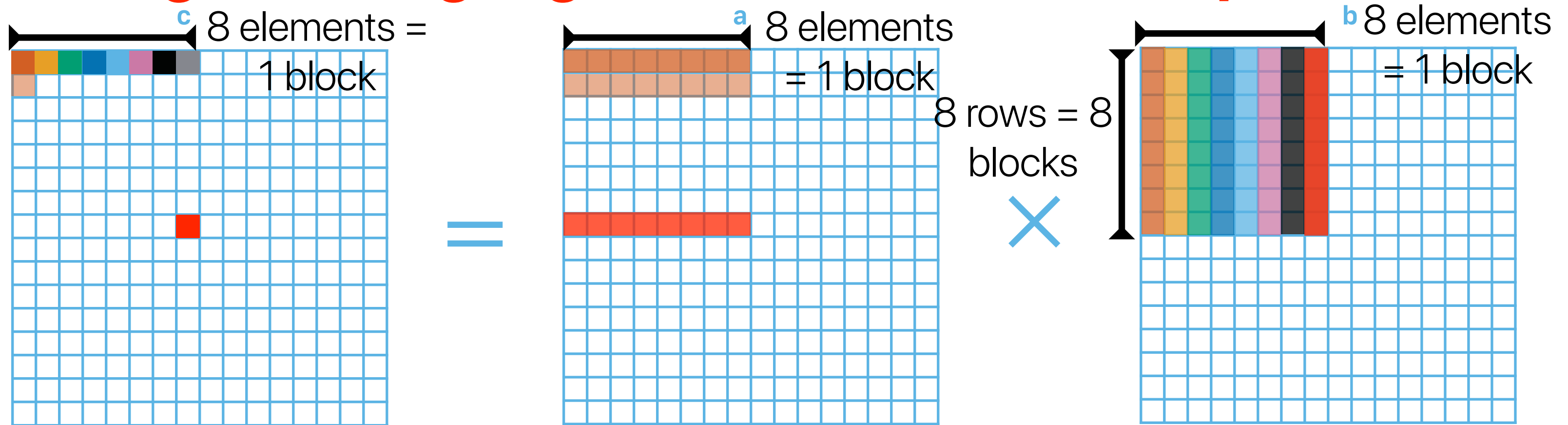


```

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 768 blocks in Intel Core i7's L1-\$**

What if we have larger tiles?



How large a tile should be?

- 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



How large a tile should be?

- 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];
```

- 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

	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

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];
```

- 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

	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

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

Ideas regarding reducing misses in matrix multiplications

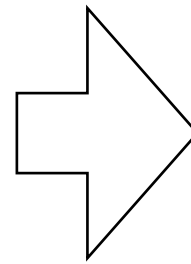
- Reducing capacity misses — we need to reduce the length of a row that we visit within a period of time
- Reducing conflict misses — we need to ensure an appropriate tile size would not lead to conflict in sets
- Balancing the trade-offs — increased instruction count can demolishing the improvement of cache misses
 - Cache miss rates affect the CPI
 - Tiling increases the IC
 - Remember, $ET = IC \times CPI \times CT$

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];
        }
    }
}
```

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];
        }
    }
}
```

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];
        }
    }
}
```

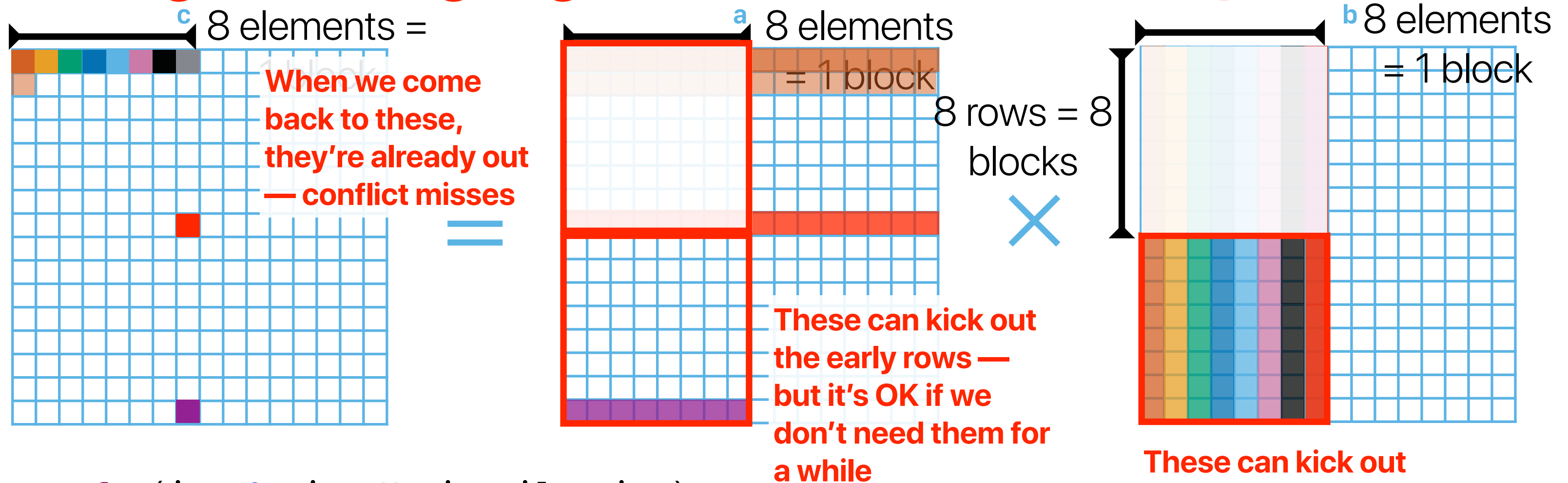
- A. Compulsory miss
- B. Capacity miss
- C. Conflict miss
- D. Capacity & conflict miss
- E. Compulsory & conflict miss

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];
        }
    }
}
```

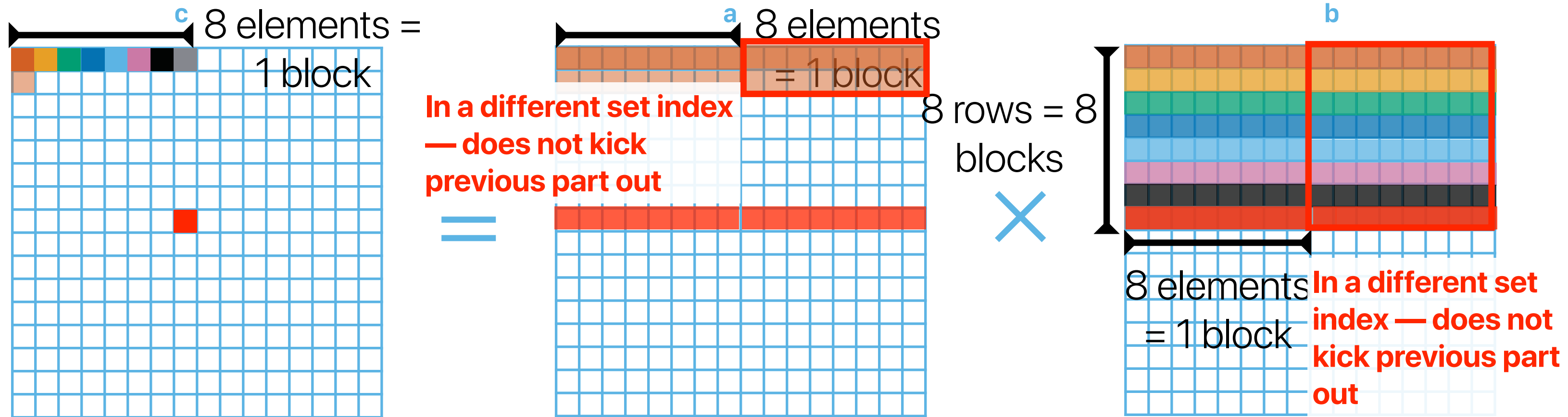
Tiling/Blocking Algorithm for Matrix Multiplications



```

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

```

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

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];
        }
    }
}
```

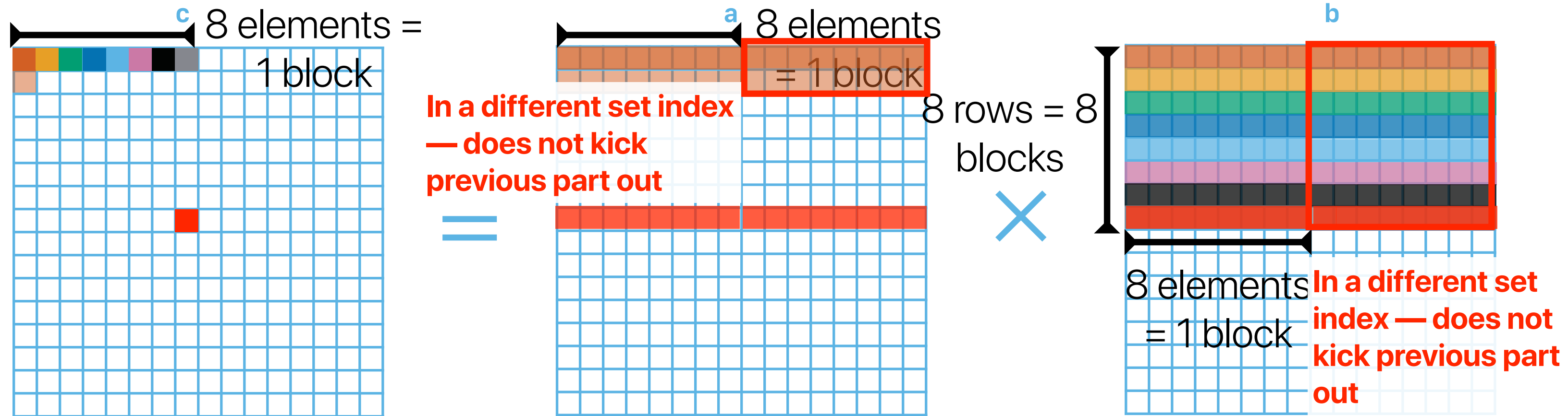
- A. Compulsory miss
- B. Capacity miss
- C. Conflict miss
- D. Capacity & conflict miss
- E. Compulsory & conflict miss

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

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


The effect of transposition

Block/tile	size	tile_size	IC	Cycles	CPI	CT_ns	ET_s	DL1_miss_rate
	2048	4	97766571061	23972375695	0.245200	0.193290	4.633619	0.015394
	2048	8	81047436195	21583826614	0.266311	0.193122	4.168303	0.010260
	2048	16	74472586117	19268082018	0.258727	0.193325	3.724997	0.071558
	2048	32	71543547491	27661109860	0.386633	0.193218	5.344616	0.217189
	2048	64	70151970961	32605985592	0.464791	0.193245	6.301039	0.242202
	2048	128	69470212062	34530336995	0.497832	0.193235	6.672484	0.246013
	2048	256	69131368754	35151111975	0.508468	0.193311	6.795085	0.246800

Best performing
from 16 to 64?

Block + Transpose	size	tile_size	IC	Cycles	CPI	CT_ns	ET_s	DL1_miss_rate
	2048	8	70351352368	16009523067	0.227565	0.193097	3.091384	0.001897
	2048	16	64810353582	15145593176	0.233691	0.193199	2.926121	0.026059
	2048	32	62397963236	14854143892	0.238055	0.193161	2.869243	0.040979
	2048	64	60508160004	14500000000	0.223012	0.193260	2.640043	0.023464
	2048	128	60712004318	15724248174	0.275478	0.193123	3.229842	0.018013
	2048	256	60438882203	17003851823	0.281340	0.193330	3.287351	0.012972

Transpose improves
the miss rate at larger
tile size!

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

The effect of transposition + register

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

Significant reduction
of memory accesses

Block + Transpose + Reg.	size	tile_size	IC	Cycles	CPI	CT_ns	ET_s	DL1_miss_rate	DL1_accesses
	2048	8	60667686406	11604601609	0.191281	0.193985	2.251117	0.003363	15849319789
	2048	16	49205927530	10032628049	0.203891	0.193373	1.940040	0.066542	11986874817
	2048	32	43854347828	9736190510	0.222012	0.193008	1.879164	0.096360	10247265803
	2048	64	41246516681	10038942768	0.243389	0.193141	1.938936	0.069853	9413044702
	2048	128	39962465083	12527203181	0.311944	0.193136	2.299423	0.045847	9005357520
	2048	256	39326412762	14051616706	0.357307	0.193021	2.712263	0.032001	8804127506

Best performing at 32

Tiles do not have to be "squares"

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

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	60667686406	11604601609	0.191281	0.193985	2.251117	0.003363
	2048	16	49205927530	10032628049	0.203891	0.193373	1.940040	0.066542
	2048	32	43854347828	9736190510	0.222012	0.193008	1.879164	0.096360
	2048	64	41246516681	10038942768	0.243389	0.193141	1.938936	0.069853
	2048	128	39962465083	11887280358	0.297461	0.193436	2.299423	0.045847
	2048	256	39326412762	14051616706	0.357307	0.193021	2.712243	0.032001

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	60696024519	11624419280	0.191519	0.193479	2.249077	0.003664
	2048	8	16	59757245622	12523769669	0.209577	0.193096	2.418293	0.022092
	2048	16	8	49689499294	11415401958	0.229735	0.193200	2.205459	0.003957
	2048	16	16	40215435235	10018627798	0.203567	0.193202	1.935619	0.067519
	2048	32	16	43947034522	9718597645	0.221143	0.193352	1.879111	0.066730
	2048	32	8	44182255983	9353772552	0.211709	0.193233	1.807453	0.005696
	2048	64	8	41431394002	9636426904	0.232588	0.193230	1.862043	0.005849
	2048	128	8	40059698391	11599727168	0.289561	0.193133	2.240289	0.006533
	2048	256	8	39376445298	13785740954	0.350101	0.193180	2.663130	0.004190

Best performing at 32x8

Why is "8" not the best performing?

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

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

**Best
performing
at 16?**

60

**"8" indeed has the best
miss rate — and matches
our predictions!**

Takeaways: Software Optimizations

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

Software Prefetching — through prefetching instructions

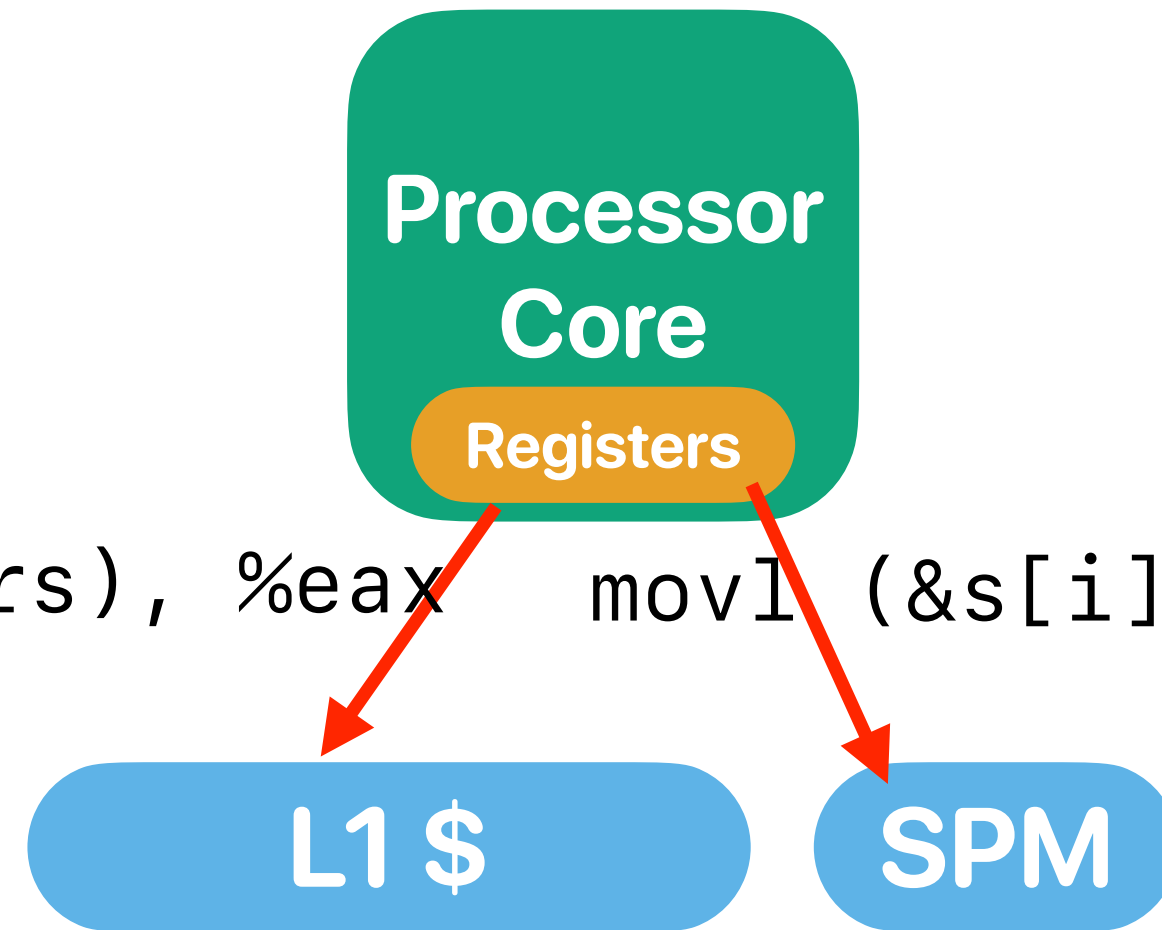
- x86 provide prefetch instructions
- As a programmer, you may insert `_mm_prefetch` in x86 programs to perform software prefetch for your code
- gcc also has a flag `"-fprefetch-loop-arrays"` to automatically insert software prefetch instructions

Implementation of SPM — GPU's shared memory

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

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

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



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

Takeaways: Optimizing cache performance through hardware

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

Let's dig into this code

```
int main(int argc, char *argv[])
{
    int i,j;
    double **a;
    double sum=0, average;
    int dim=32768;
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s dimension\n",argv[0]);
        exit(1);
    }
    dim = atoi(argv[1]);
    a = (double **)malloc(sizeof(double *)*dim);
    for(i = 0 ; i < dim; i++)
        a[i] = (double *)malloc(sizeof(double)*dim);
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            a[i][j] = rand();
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            sum+=a[i][j];
    average = sum/(dim*dim);
    fprintf(stderr,"average: %lf\n",average);
    for(i = 0 ; i < dim; i++)
        free(a[i]);
    free(a);
    return 0;
}
```



What will happen?

- If we execute the code on the right-hand side code on a machine with only 8 GB of physical memory installed and the dim is **33000** (requires $33000 \times 33000 \times 8$ bytes ~ **8.12 GB** memory at least), What will happen?
 - A. The program will crash in one of the malloc function call
 - B. The program will crash due to a "segmentation fault" that caused by accessing NULL pointer
 - C. The program will be killed automatically by the OS as it uses more than installed physical main memory
 - D. The program will finish without any issue

```
int main(int argc, char *argv[])
{
    int i,j;
    double **a;
    double sum=0, average;
    int dim=32768;
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s dimension\n",argv[0]);
        exit(1);
    }
    dim = atoi(argv[1]);
    a = (double **)malloc(sizeof(double *)*dim);
    for(i = 0 ; i < dim; i++)
        a[i] = (double *)malloc(sizeof(double)*dim);
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            a[i][j] = rand();
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            sum+=a[i][j];
    average = sum/(dim*dim);
    fprintf(stderr,"average: %lf\n",average);
    for(i = 0 ; i < dim; i++)
        free(a[i]);
    free(a);
    return 0;
}
```



What will happen?

- If we execute the code on the right-hand side code on a machine with only 8 GB of physical memory installed and the dim is **33000** (requires $33000 \times 33000 \times 8$ bytes ~ **8.12 GB** memory at least), What will happen?
 - A. The program will crash in one of the malloc function call
 - B. The program will crash due to a "segmentation fault" that caused by accessing NULL pointer
 - C. The program will be killed automatically by the OS as it uses more than installed physical main memory
 - D. The program will finish without any issue

```
int main(int argc, char *argv[])
{
    int i,j;
    double **a;
    double sum=0, average;
    int dim=32768;
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s dimension\n",argv[0]);
        exit(1);
    }
    dim = atoi(argv[1]);
    a = (double **)malloc(sizeof(double *)*dim);
    for(i = 0 ; i < dim; i++)
        a[i] = (double *)malloc(sizeof(double)*dim);
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            a[i][j] = rand();
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            sum+=a[i][j];
    average = sum/(dim*dim);
    fprintf(stderr,"average: %lf\n",average);
    for(i = 0 ; i < dim; i++)
        free(a[i]);
    free(a);
    return 0;
}
```

What will happen?

- If we execute the code on the right-hand side code on a machine with only 8 GB of physical memory installed and the dim is **33000** (requires $33000 \times 33000 \times 8$ bytes ~ **8.12 GB** memory at least), What will happen?
 - A. The program will crash in one of the malloc function call
 - B. The program will crash due to a "segmentation fault" that caused by accessing NULL pointer
 - C. The program will be killed automatically by the OS as it uses more than installed physical main memory
 - D. The program will finish without any issue**

```
int main(int argc, char *argv[])
{
    int i,j;
    double **a;
    double sum=0, average;
    int dim=32768;
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s dimension\n",argv[0]);
        exit(1);
    }
    dim = atoi(argv[1]);
    a = (double **)malloc(sizeof(double *)*dim);
    for(i = 0 ; i < dim; i++)
        a[i] = (double *)malloc(sizeof(double)*dim);
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            a[i][j] = rand();
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            sum+=a[i][j];
    average = sum/(dim*dim);
    fprintf(stderr,"average: %lf\n",average);
    for(i = 0 ; i < dim; i++)
        free(a[i]);
    free(a);
    return 0;
}
```

Let's dig into this code

```
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>

double a;

int main(int argc, char *argv[])
{
    int i, number_of_total_processes=4;
    number_of_total_processes = atoi(argv[1]);
    // Create processes
    for(i = 0; i< number_of_total_processes-1 && fork(); i++);
    // Generate rand seed
    srand((int)time(NULL)+(int)getpid());
    a = rand();
    fprintf(stderr, "\nProcess %d. Value of a is %lf and address of a is %p\n",getpid(), a, &a);
    sleep(10);
    fprintf(stderr, "\nProcess %d. Value of a is %lf and address of a is %p\n",getpid(), a, &a);
    return 0;
}
```



Consider the following code ...

- Consider the case when we run 4 instances of the given program at the same time on modern machines, how many statements correct?

- ① The printed "address of a" is the same for every running instances
- ② The printed "address of a" is different for each instance
- ③ All running instances will print the same value of a for the first printf
- ④ Each instance will print a different value of a for the first printf
- ⑤ For the fprintf of 10 Seconds later, each running instance will print the same value from it's last run
- ⑥ For the fprintf of 10 Seconds later, each running instance will print a different value from it's last run

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

```
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>
#include <stdint.h>

double a = 0;

int main(int argc, char *argv[])
{
    uint64_t i, number_of_total_processes=4, p;
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s number_of_processes\n", argv[0]);
        exit(1);
    }
    number_of_total_processes = atoi(argv[1]);
    for(i = 0; i < number_of_total_processes-1 && fork(); i++);
    srand((int)time(NULL)+(int)getpid());
    a = rand();
    fprintf(stderr, "\nProcess %d: Value of a is %lf and address of a is %p\n", (int)getpid(), a, &a);
    sleep(10);
    fprintf(stderr, "\n10 Seconds Later -- Process %d: Value of a is %lf and address of a is %p\n", (int)getpid(), a, &a);
    return 0;
}
```



Consider the following code ...

- Consider the case when we run 4 instances of the given program at the same time on modern machines, how many statements correct?

- ① The printed "address of a" is the same for every running instances
- ② The printed "address of a" is different for each instance
- ③ All running instances will print the same value of a for the first printf
- ④ Each instance will print a different value of a for the first printf
- ⑤ For the fprintf of 10 Seconds later, each running instance will print the same value from it's last run
- ⑥ For the fprintf of 10 Seconds later, each running instance will print a different value from it's last run

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

```
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>
#include <stdint.h>

double a = 0;

int main(int argc, char *argv[])
{
    uint64_t i, number_of_total_processes=4, p;
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s number_of_processes\n", argv[0]);
        exit(1);
    }
    number_of_total_processes = atoi(argv[1]);
    for(i = 0; i < number_of_total_processes-1 && fork(); i++);
    srand((int)time(NULL)+(int)getpid());
    a = rand();
    fprintf(stderr, "\nProcess %d: Value of a is %lf and address of a is %p\n", (int)getpid(), a, &a);
    sleep(10);
    fprintf(stderr, "\n10 Seconds Later -- Process %d: Value of a is %lf and address of a is %p\n", (int)getpid(), a, &a);
    return 0;
}
```


Consider the following code ...

- Consider the case when we run 4 instances of the given program at the same time on modern machines, how many statements correct?

- ① The printed "address of a" is the same for every running instances
- ② The printed "address of a" is different for each instance
- ③ All running instances will print the same value of a for the first printf
- ④ Each instance will print a different value of a for the first printf
- ⑤ For the fprintf of 10 Seconds later, each running instance will print the same value from it's last run
- ⑥ For the fprintf of 10 Seconds later, each running instance will print a different value from it's last run

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

```
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>
#include <stdint.h>

double a = 0;

int main(int argc, char *argv[])
{
    uint64_t i, number_of_total_processes=4, p;
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s number_of_processes\n", argv[0]);
        exit(1);
    }
    number_of_total_processes = atoi(argv[1]);
    for(i = 0; i < number_of_total_processes-1 && fork(); i++);
    srand((int)time(NULL)+(int)getpid());
    a = rand();
    fprintf(stderr, "\nProcess %d: Value of a is %lf and address of a is %p\n", (int)getpid(), a, &a);
    sleep(10);
    fprintf(stderr, "\n10 Seconds Later -- Process %d: Value of a is %lf and address of a is %p\n", (int)getpid(), a, &a);
    return 0;
}
```

Announcement

- Assignment #2 due **this Thursday**
- Reading quiz #5 due **next Tuesday** before the lecture
- Midterm and final exam will be online — **details TBD**

Computer Science & Engineering

203

つづく

