CS683: Advanced Computer Architecture

Lecture-2: Cache-friendly code

https://www.cse.iitb.ac.in/~biswa/courses/CS683/main.html

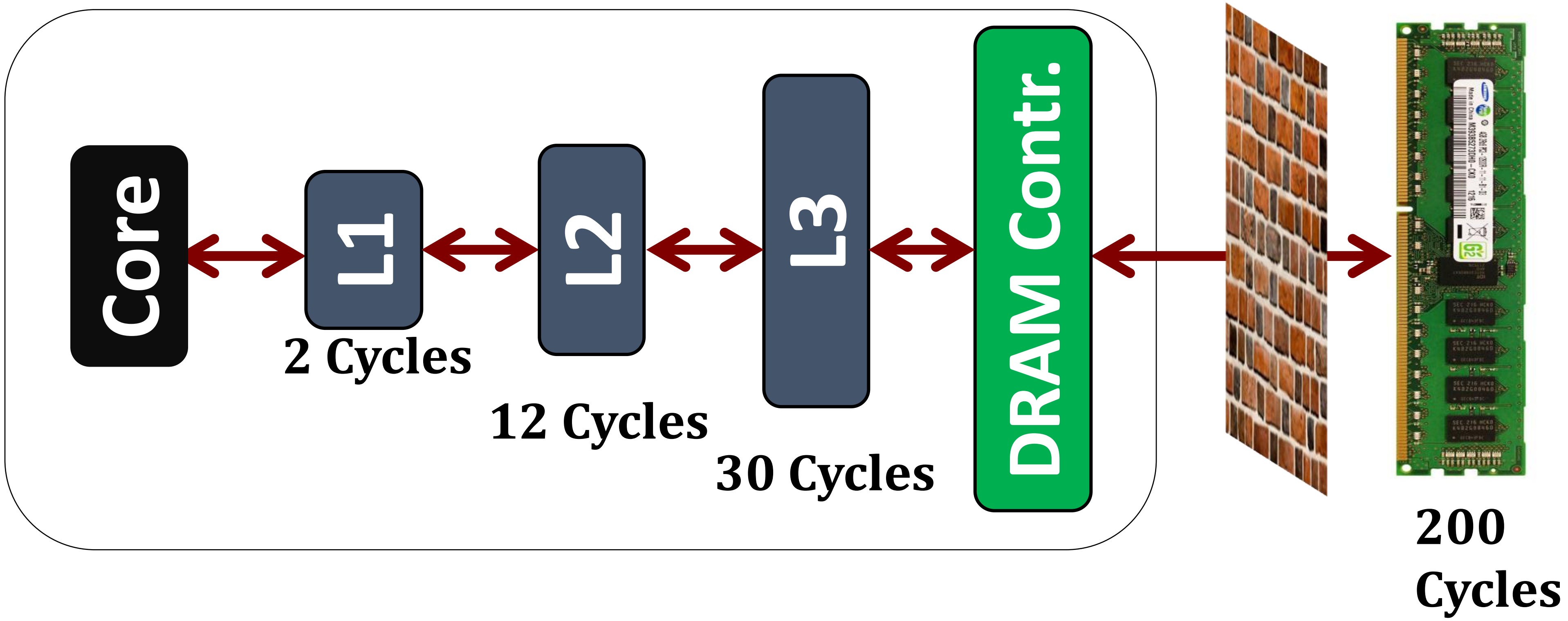
https://www.cse.iitb.ac.in/~biswa/

Recap of last lecture

- Why, what , and how of Computer Architecture
- Performance: What is it?
- The impact of optimizations and why we all should care?

Phones (smart/non-smart) on silence plz, Thanks

From where does these zeros come from?



Memory stores CODE and DATA

Processor accesses through LOADs (reads) and STOREs(writes)

Memory Wall

Hang on!! I got the Mantra!!

Reduction in DRAM accesses ~
Reduction in CPI (cycles per instruction)

WRONG!

- First Law of Performance:

  Make the common case fast

- Second Law of Performance:

  Make the fast case common

  Amdahl's law

What if your program is not memory intensive

Do not ignore the uncommon too

- Give me an example, coffee/chai point +1

Let’s Pause and understand:

The game of “common” and “uncommon”

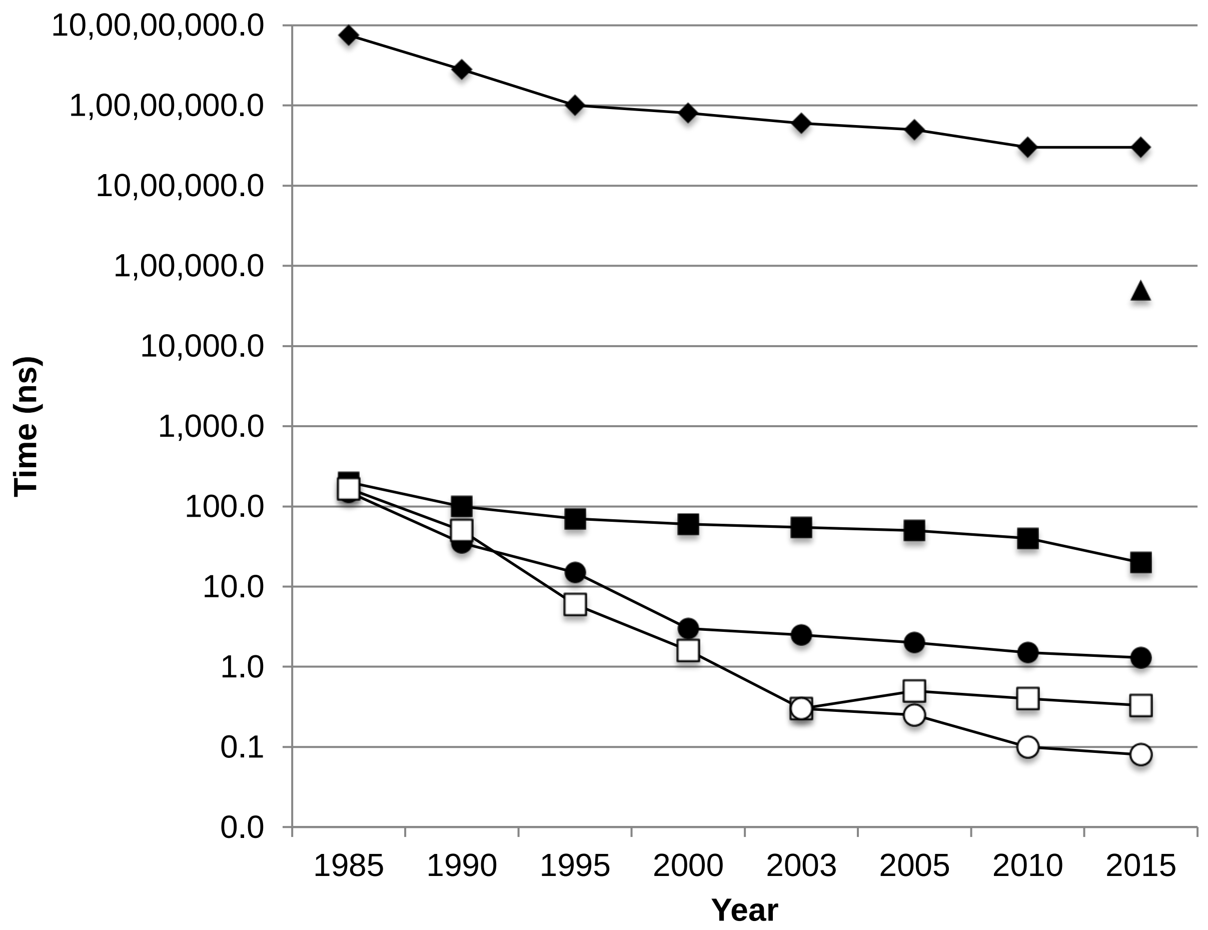
Do not

Over-engineer

Over-think

Over ……………………

10,000 Feet View on Caches



$$: Cache
Speculation technique

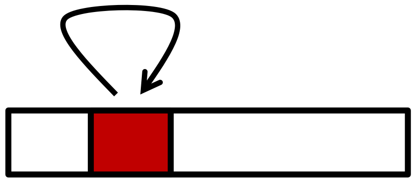
Speculation works because of locality

Caches

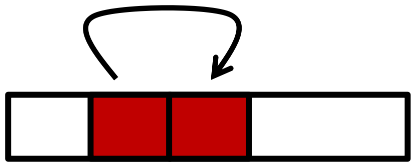
Hardware hash tables ☺

Locality (why does it exist?)

- **Temporal locality:**
  - Recently referenced items are likely to be referenced again



- **Spatial locality:**
  - Items with nearby addresses tend to be referenced again



Locality: Example

```
sum = 0;
for (i = 0; i < n; i++)
  sum += a[i];
return sum;
```

Spatial/Temporal Locality?

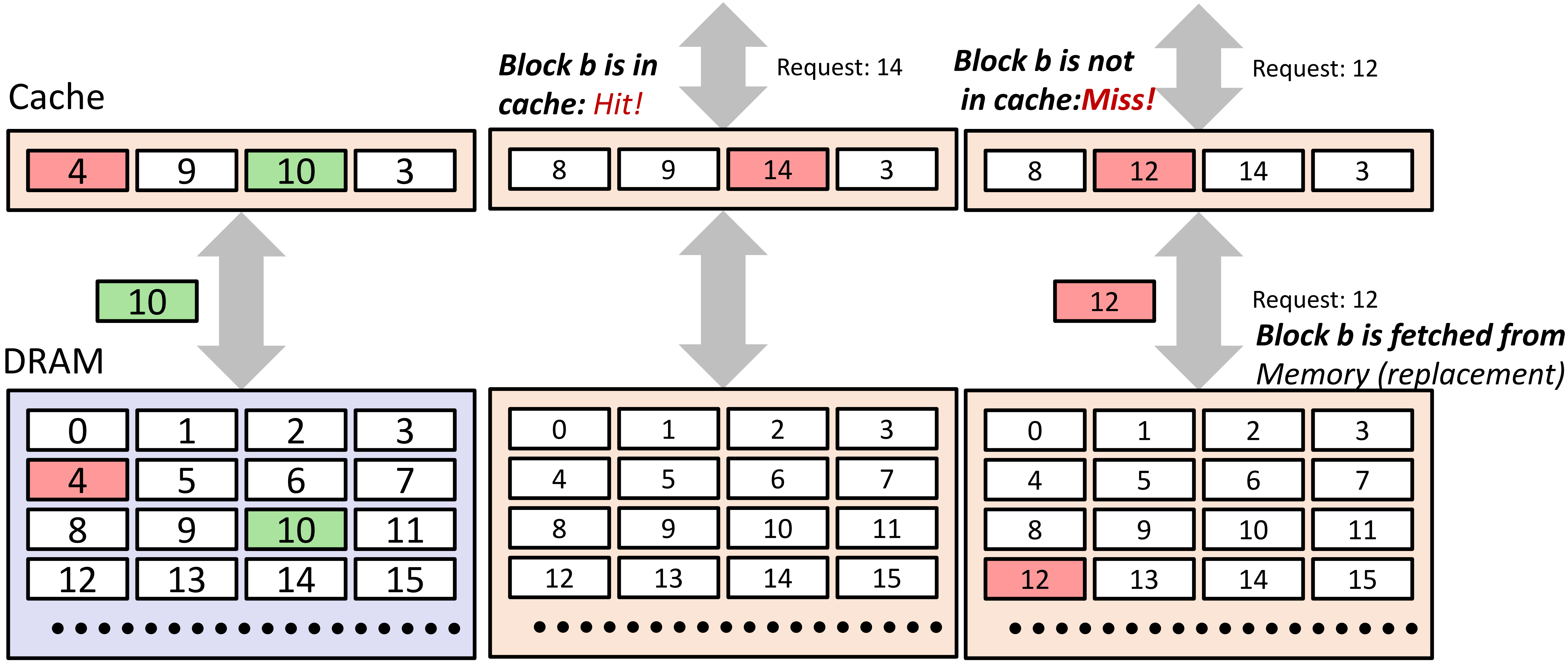
- Data references
  - Reference array elements in succession (stride-1 reference pattern). spatial
  - Reference variable **`sum`** each iteration. temporal

Wake-up Test: Improve Spatial Locality

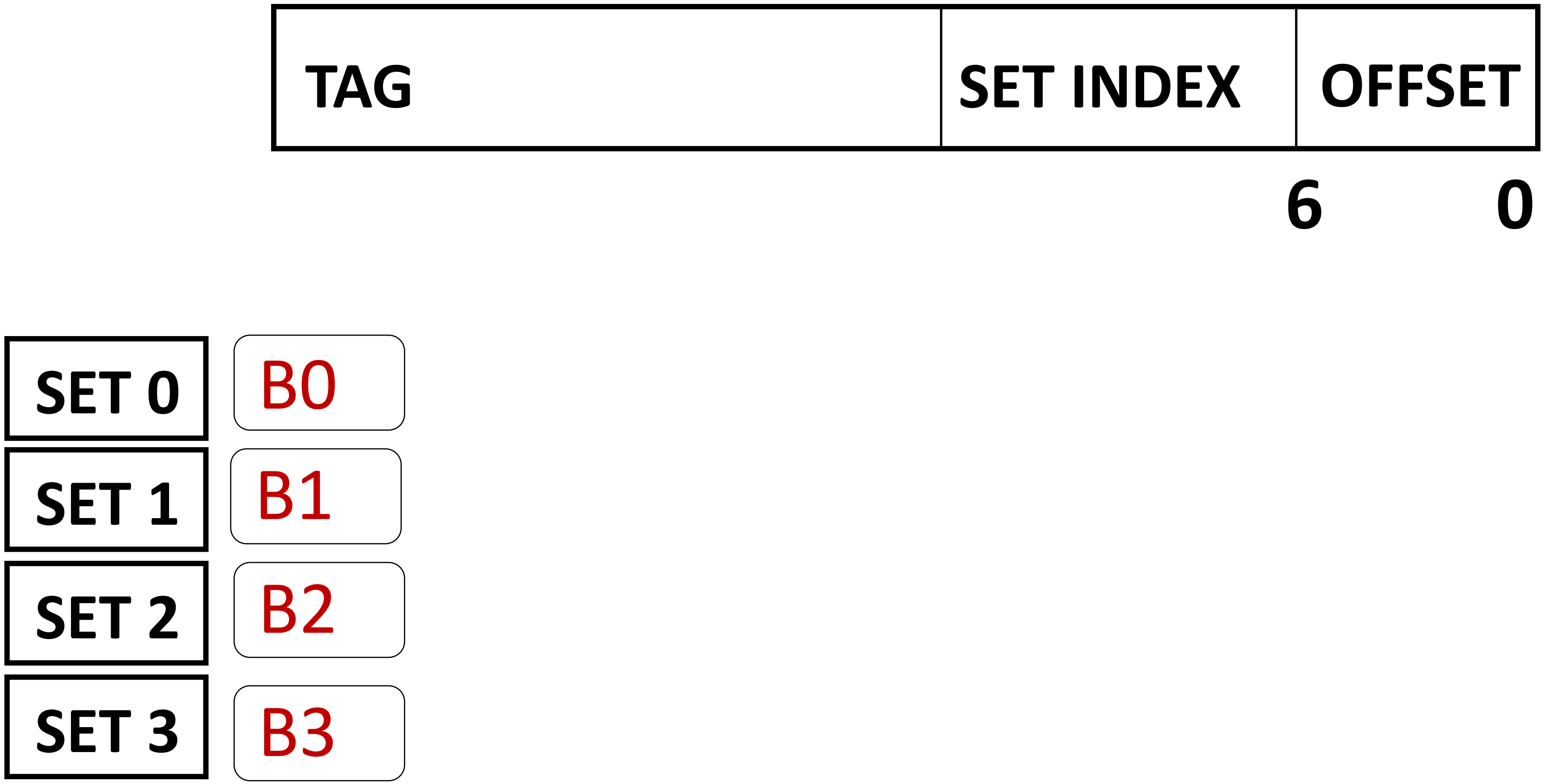
```
int sum_array_3d(int a[M][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < M; k++)
                sum += a[k][i][j];
    return sum;
}
```

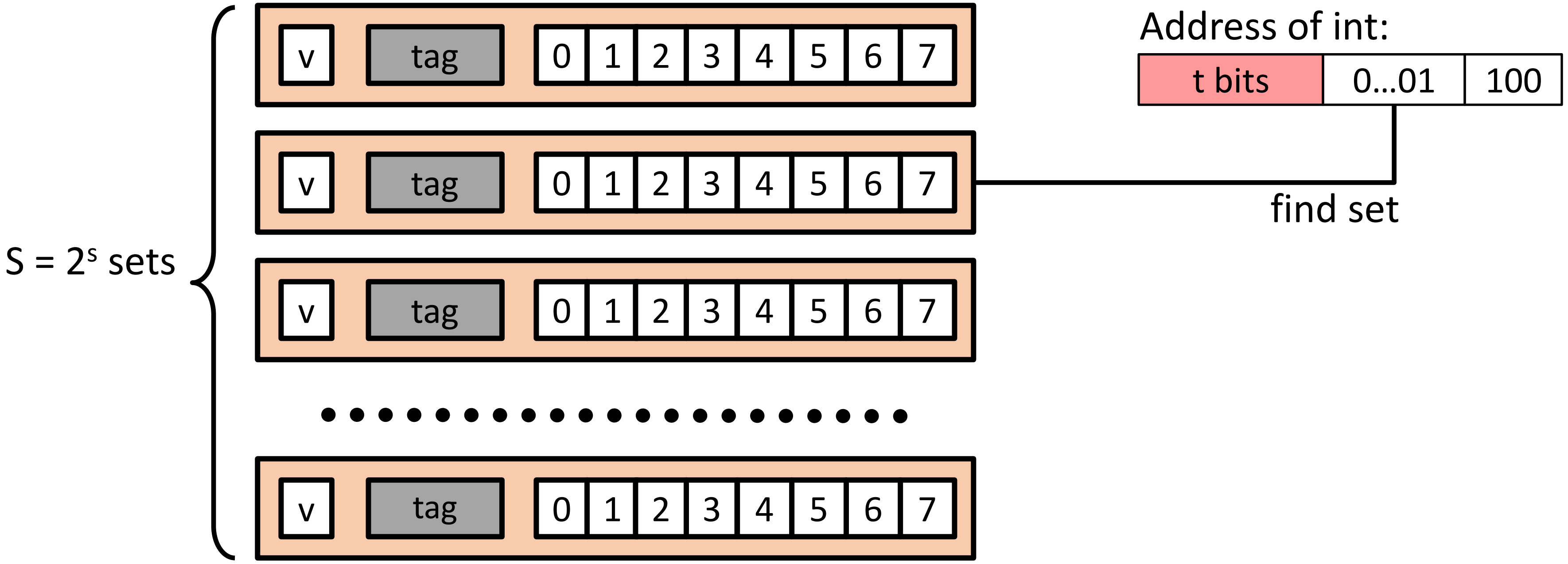
Cache and DRAM



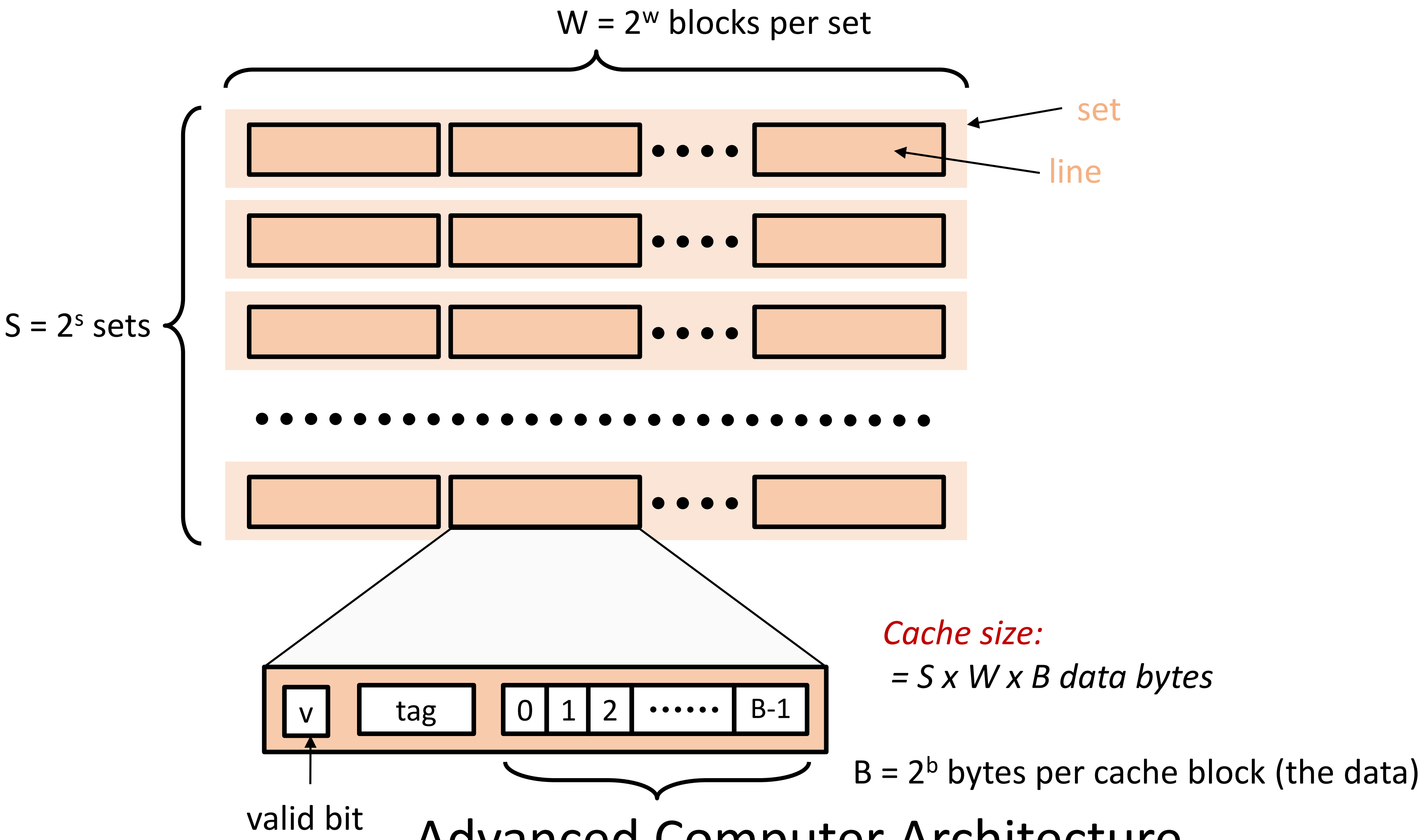
Cache Mapping



Direct Mapped: One block=One set



Set Associative

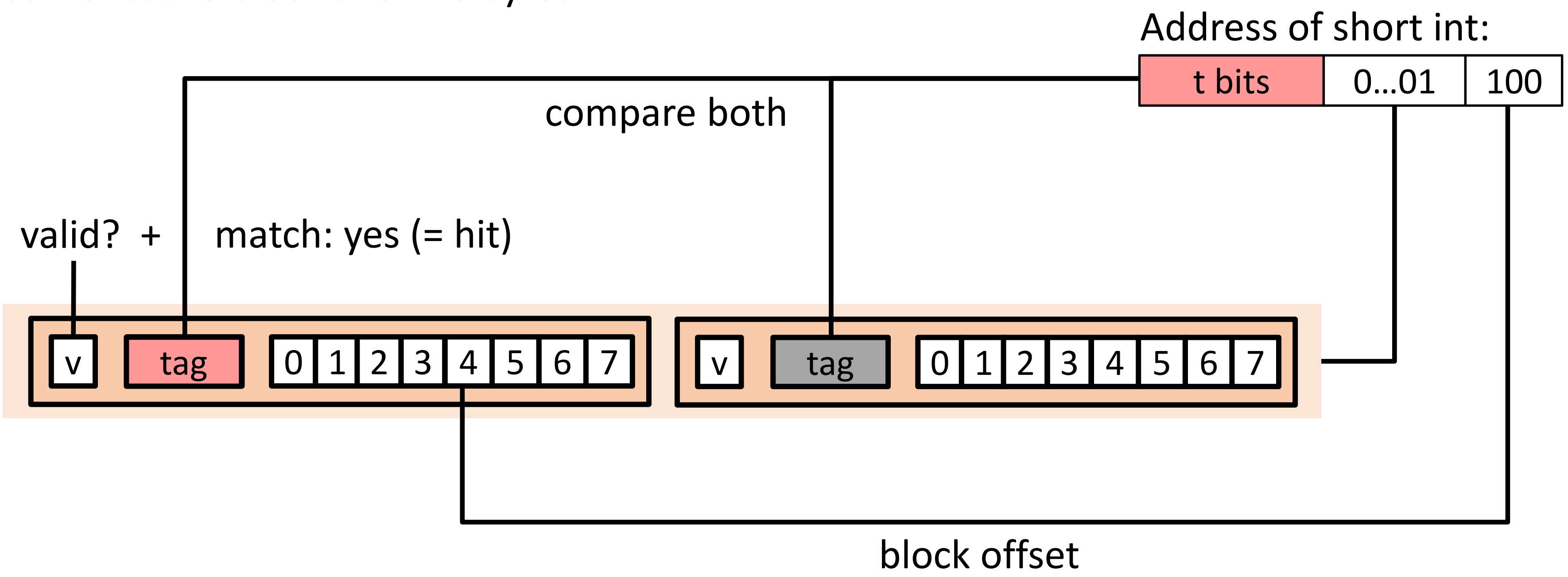


Cache size:

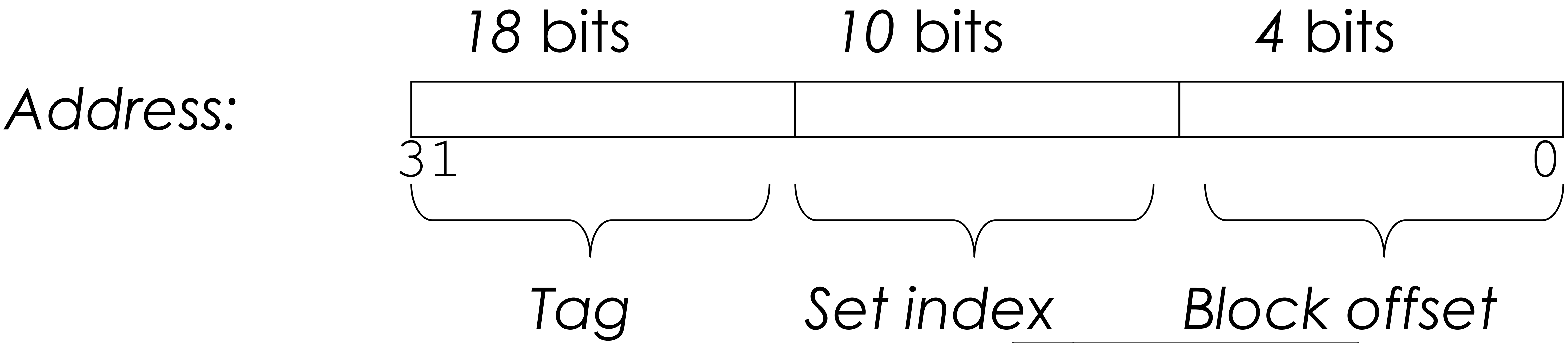
= S x W x B data bytes

Set Associative in Action:

Way = 2: Two lines per set
Assume: cache block size B=8 bytes



Wake-up test again: #ints inside a block?



	# of int in block
A.	0
B.	1
C.	2
D.	4
E.	Unknown: We need more info

Wake-up test again:

If N = 16, how many bytes does the loop access of a?

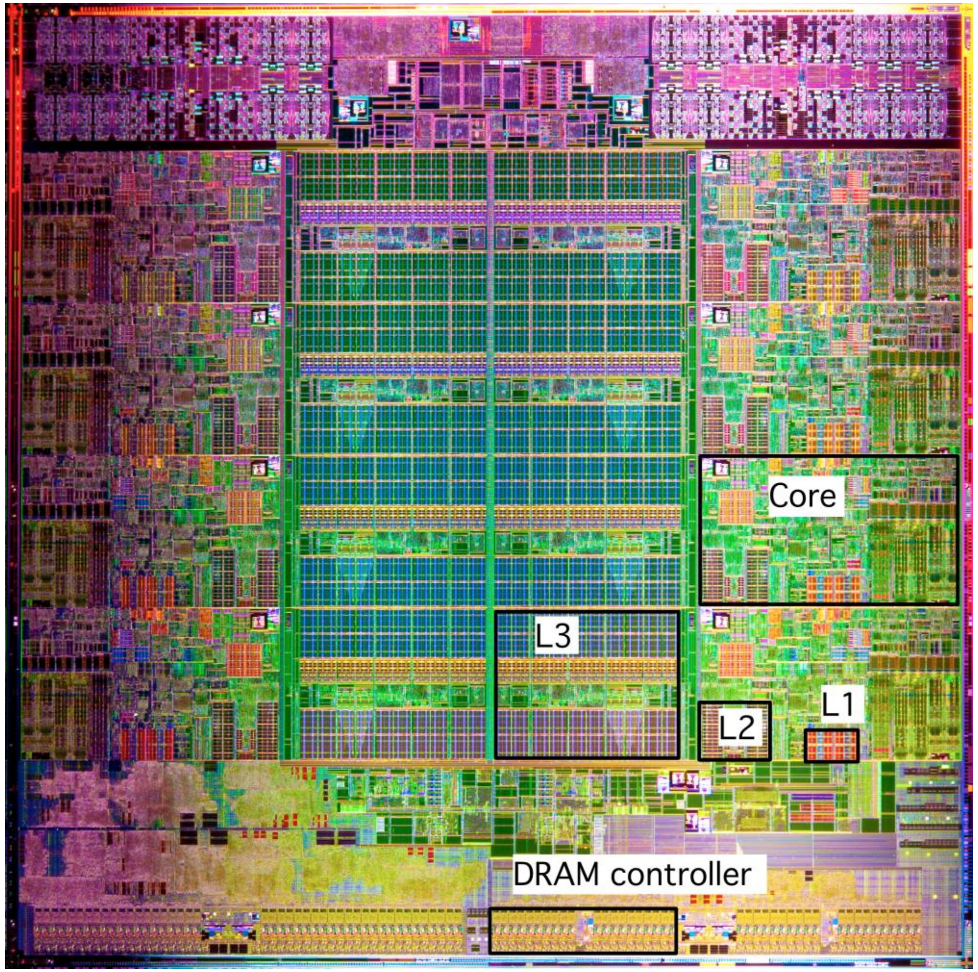
```
int CS683(int* a, int N)
{
    int i;
    int sum = 0;
    for(i = 0; i < N; i++)
    {
        sum += a[i];
    }
    return sum;
}
```

	Accessed Bytes
A	4
B	16
C	64
D	256

The 3Cs

- Cold (compulsory) miss
  - Cold misses occur because the cache starts empty and this is the first reference
- Capacity miss
  - Occurs when the set of active cache blocks (working set) is larger than the cache.
- Conflict miss ☺ conflicting addresses into one set

Looks Like This



Intel Sandy Bridge Processor Die

L1: 32KB Instruction + 32KB Data
L2: 256KB
L3: 3–20MB

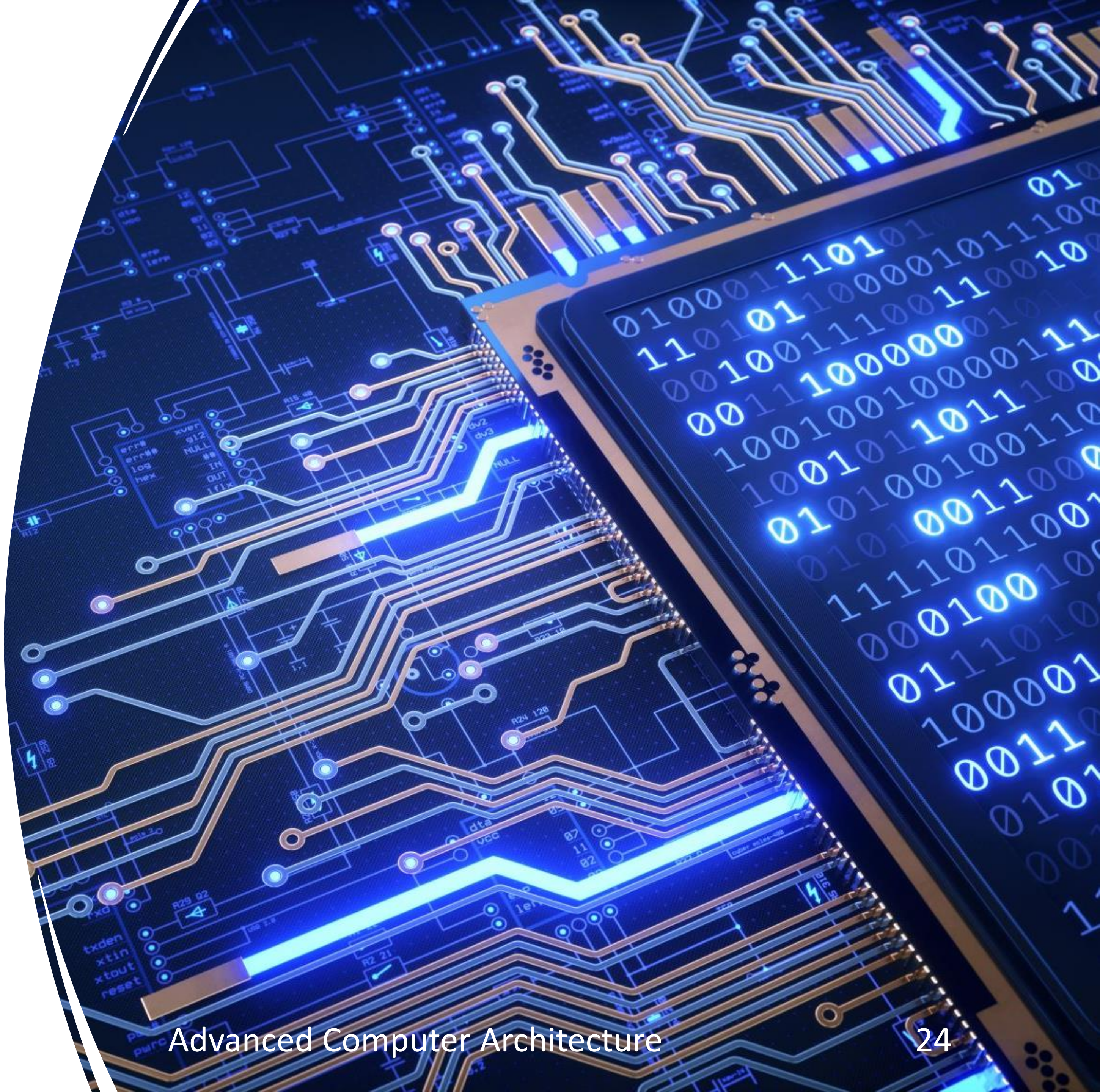
Matrix Multiplication

- Description:
  - Multiply *N* x *N* matrices
  - Matrix elements are doubles (8 bytes)
  - O(N^3) total operations

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

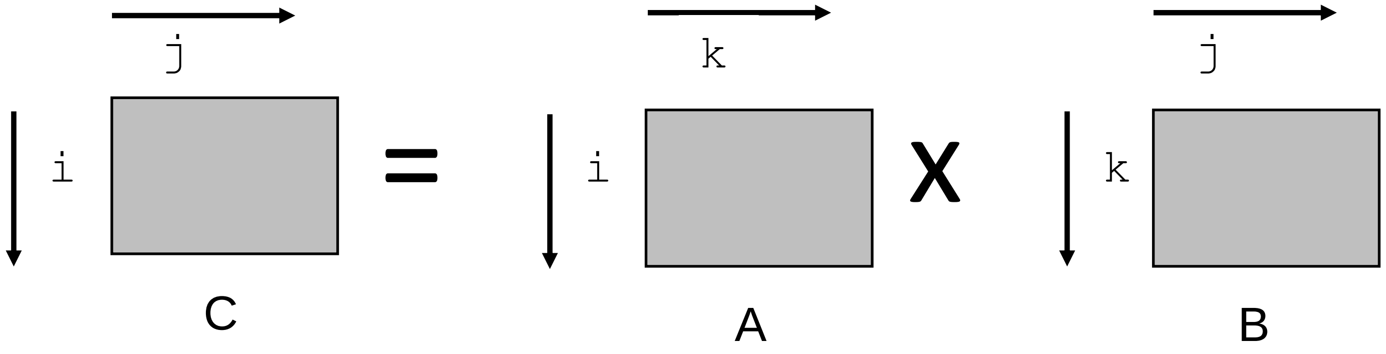
Why Matrix Multiplication?

- AI/ML
- Image Processing
- Scientific Computing
- Graph traversals
- Many more …



Miss Rate Analysis

- Assume:
  - Block size = 32B (big enough for four doubles)
  - Matrix dimension (N) is very large
    - Approximate 1/N as 0.0
  - Cache is not even big enough to hold multiple rows
- Analysis Method:
  - Look at access pattern of inner loop



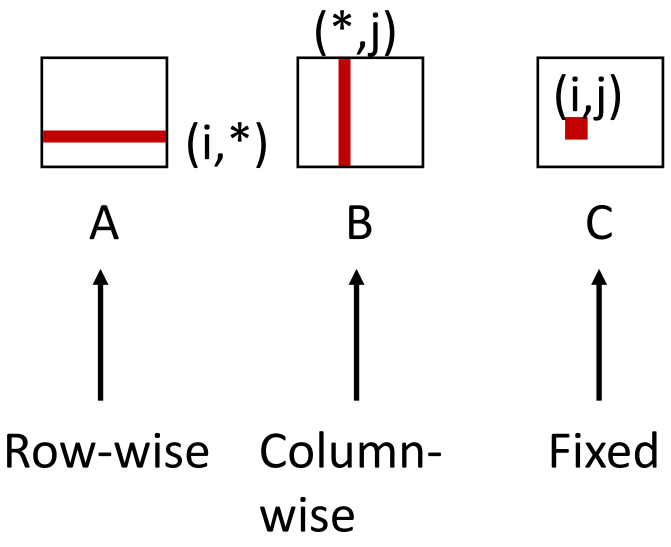
Cache Layout

- C arrays allocated in row-major order
  - each row in contiguous memory locations
- Stepping through columns in one row:
  - ```
    for (i = 0; i < N; i++)
      sum += a[0][i];
    ```
  - accesses successive elements
  - if block size (B) > sizeof(a_{ij}) bytes, exploit spatial locality
    - miss rate = sizeof(a_{ij}) / B
- Stepping through rows in one column:
  - ```
    for (i = 0; i < N; i++)
      sum += a[i][0];
    ```
  - accesses distant elements
  - no spatial locality!
    - miss rate = 1 (i.e. 100%)

Effect of loops (ijk)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Inner loop:



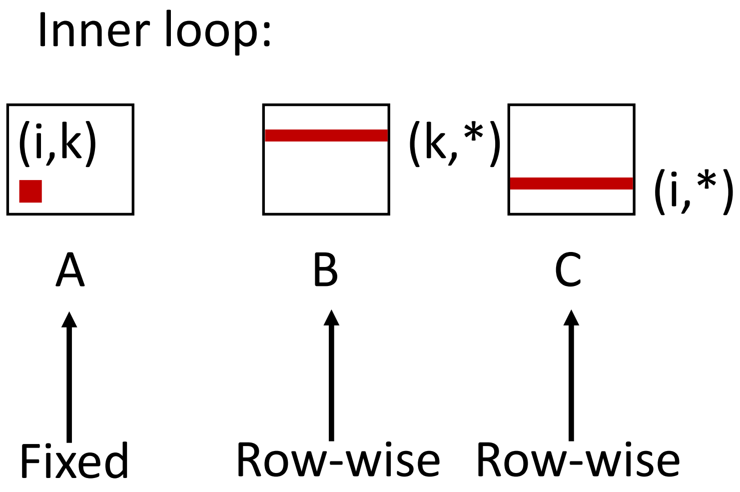
Block size = 32B (four doubles), your laptop will have 64B blocks

Miss rate for inner loop iterations:

A	B	C
0.25	1.0	0.0

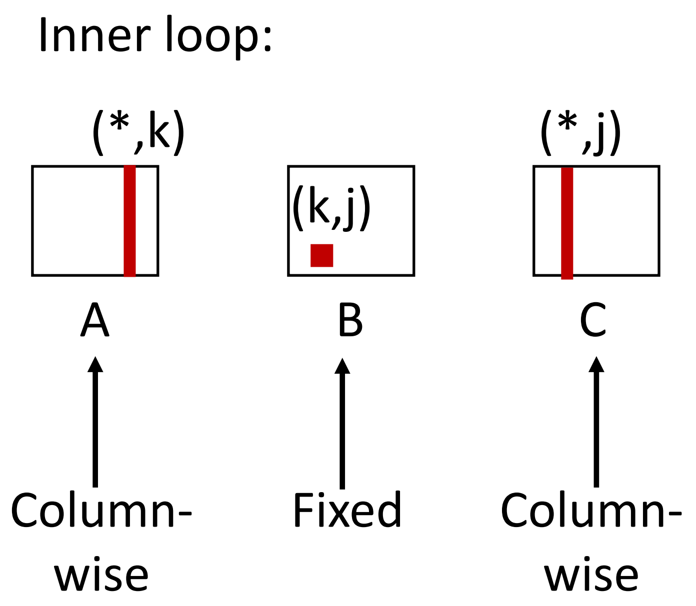
Effect of loops (kij)

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



Effect of loops (jki)

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

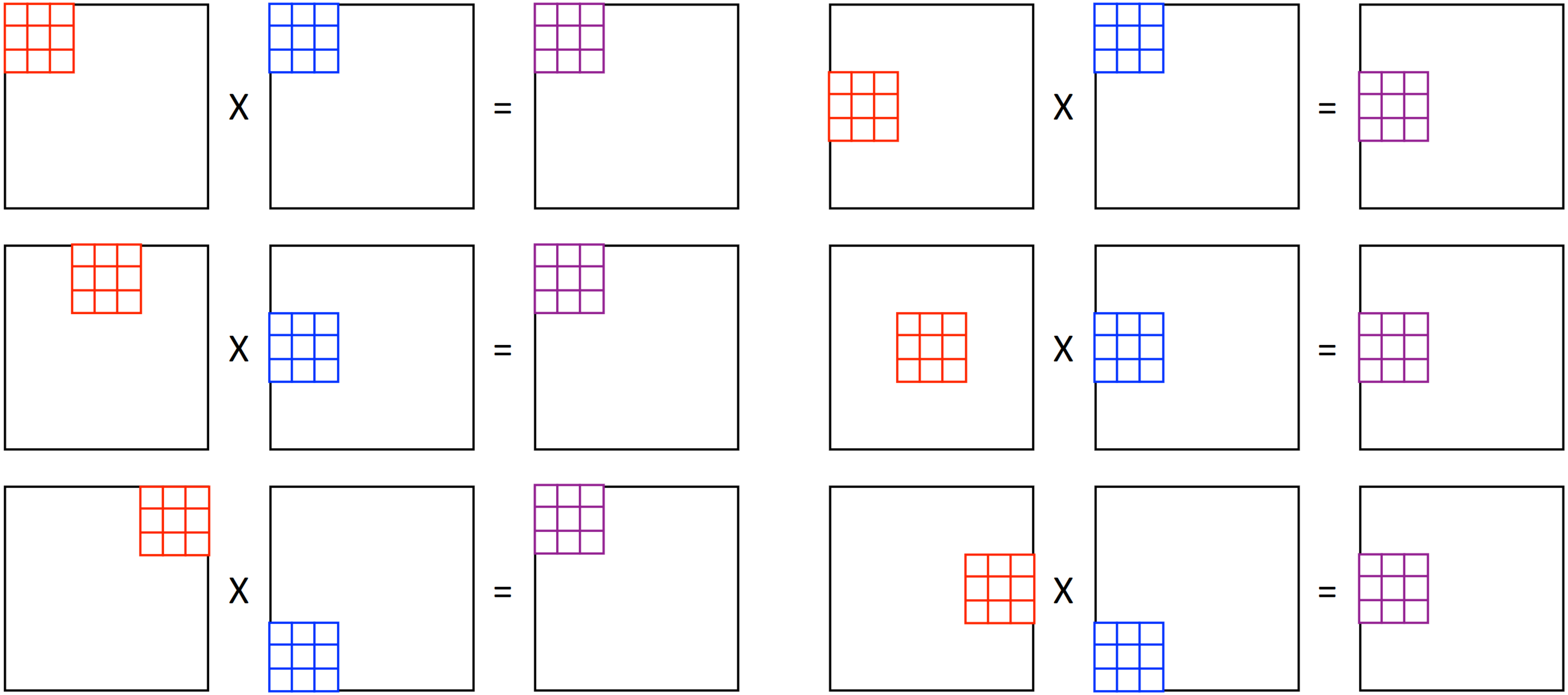


Let's Dig Deep: Where are the Cache misses? Cache grind

```
--------------------------------------------------------------------------------
           Ir I1mr ILmr             Dr          D1mr          DLmr            Dw    D1mw    DLmw

            .    .    .              .             .             .             .       .       .
            .    .    .              .             .             .             .       .       .  #include "matrix.h"
            .    .    .              .             .             .             .       .       .
            .    .    .              .             .             .             .       .       .  #define min(a,b) ((a)<(b)?(a):(b)) //Macro to return small value from a and  b.
            .    .    .              .             .             .             .       .       .
            .    .    .              .             .             .             .       .       .  /*function to perform matric multiplication.*/
            .    .    .              .             .             .             .       .       .  void mat_multiply(void *mat_a, void *mat_b, void *mat_c, int size)
            6    1    1              0             0             0             5       0       0  {
            .    .    .              .             .             .             .       .       .
            .    .    .              .             .             .             .       .       .    //Type conversion generic type to requred data type
            2    1    1              1             0             0             1       0       0    double *a = (double *) mat_a;
            2    0    0              1             0             0             1       0       0    double *b = (double *) mat_b;
            2    0    0              1             0             0             1       0       0    double *c = (double *) mat_c;
            .    .    .              .             .             .             .       .       .    int i,j,k;
            .    .    .              .             .             .             .       .       .
            .    .    .              .             .             .             .       .       .    /* Matrix multiplication starts */
        4,101    1    1          3,074             0             0             1       0       0    for(i=0;i<size;i++)
            .    .    .              .             .             .             .       .       .    {
    4,199,424    0    0      3,147,776             0             0         1,024       0       0        for(j=0;j<size;j++)
            .    .    .              .             .             .             .       .       .        {
   11,534,336    1    1      4,194,304             0             0     1,048,576 131,072 131,072            c[i*size + j]=0;
4,300,210,176    1    1  3,223,322,624             0             0     1,048,576       0       0            for(k=0;k<size;k++)
            .    .    .              .             .             .             .       .       .            {
45,097,156,608   1    1 20,401,094,656 1,076,239,357 1,074,935,805 1,073,741,824       0       0                c[ i*size + j] = c[i*size + j]+a[i*size + k]*b[k*size + j];
            .    .    .              .             .             .             .       .       .            }
            .    .    .              .             .             .             .       .       .        }
            .    .    .              .             .             .             .       .       .    }
            .    .    .              .             .             .             .       .       .
            .    .    .              .             .             .             .       .       .    /* Matrix multiplication Completed. */
            3    0    0              2             0             0             0       0       0  }
```

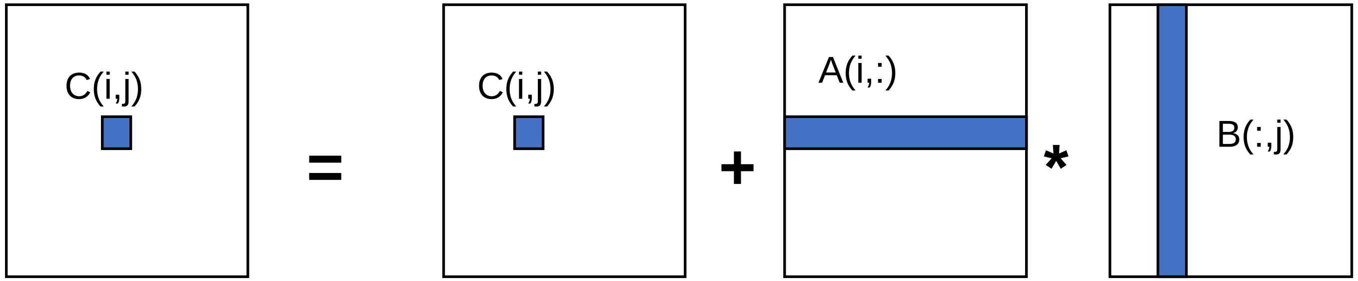
Matrix Multiplication with Tiling/Blocking: 101 ☺



Some More Visibility

Naïve MM

```
{implements C = C + A*B}
for i = 1 to n
  {read row i of A into fast memory}
   for j = 1 to n
      {read C(i,j) into fast memory}
      {read column j of B into fast memory}
      for k = 1 to n
         C(i,j) = C(i,j) + A(i,k) * B(k,j)
      {write C(i,j) back to slow memory}
```



A bit of analysis

- Number of slow memory references on unblocked matrix multiply
- Say a block size is 64 bytes and array elements are of 8 bytes

Here it is

$9/8\ (n^3)$ memory accesses

How: One miss in eight accesses (64B cache block) for the first array (1/8)

Second array: 8/8 misses, total: 9/8 mises per iteration Total number of iterations = you know it ☺

Tiled/Blocked

$$A = \left(\begin{array}{cc|cc} a_{11} & a_{12} & a_{13} & a_{1n} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{array}\right) \implies \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

$$A_{11} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, A_{12} = \begin{pmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{pmatrix}$$

$$A_{21} = \begin{pmatrix} a_{31} & a_{32} \\ a_{41} & a_{42} \end{pmatrix}, A_{22} = \begin{pmatrix} a_{33} & a_{34} \\ a_{43} & a_{44} \end{pmatrix}$$

The Code

```
/* Multiply n x n matrices a and b  */
void mmm(int n, double a[n][n], double b[n][n], double c[n][n]) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i1++)
                    for (j1 = j; j1 < j+B; j1++)
                        for (k1 = k; k1 < k+B; k1++)
                            c[i1][j1] += a[i1][k1]*b[k1][j1];
}
```

Blocked or Tiled MM

- Number of blocks per row or per column= n/B, For two arrays = 2n/B
- Block size = B X B
- Three blocks are in the C, $3B^2$ are in the cache
- For each block: $B^2/8$ misses,
  Two arrays = $2n/B \times B^2/8$
- Total number of misses = $n^3/4B$

Another way

- For each block: : $B^2/8$ misses
- Two blocks (two arrays) = : $B^2/4$ misses
- Total n^3/B^3 iterations
- Total number of misses = $n^3/4B$

Speedup

A: Non-tiled: 9/8 (n^3)

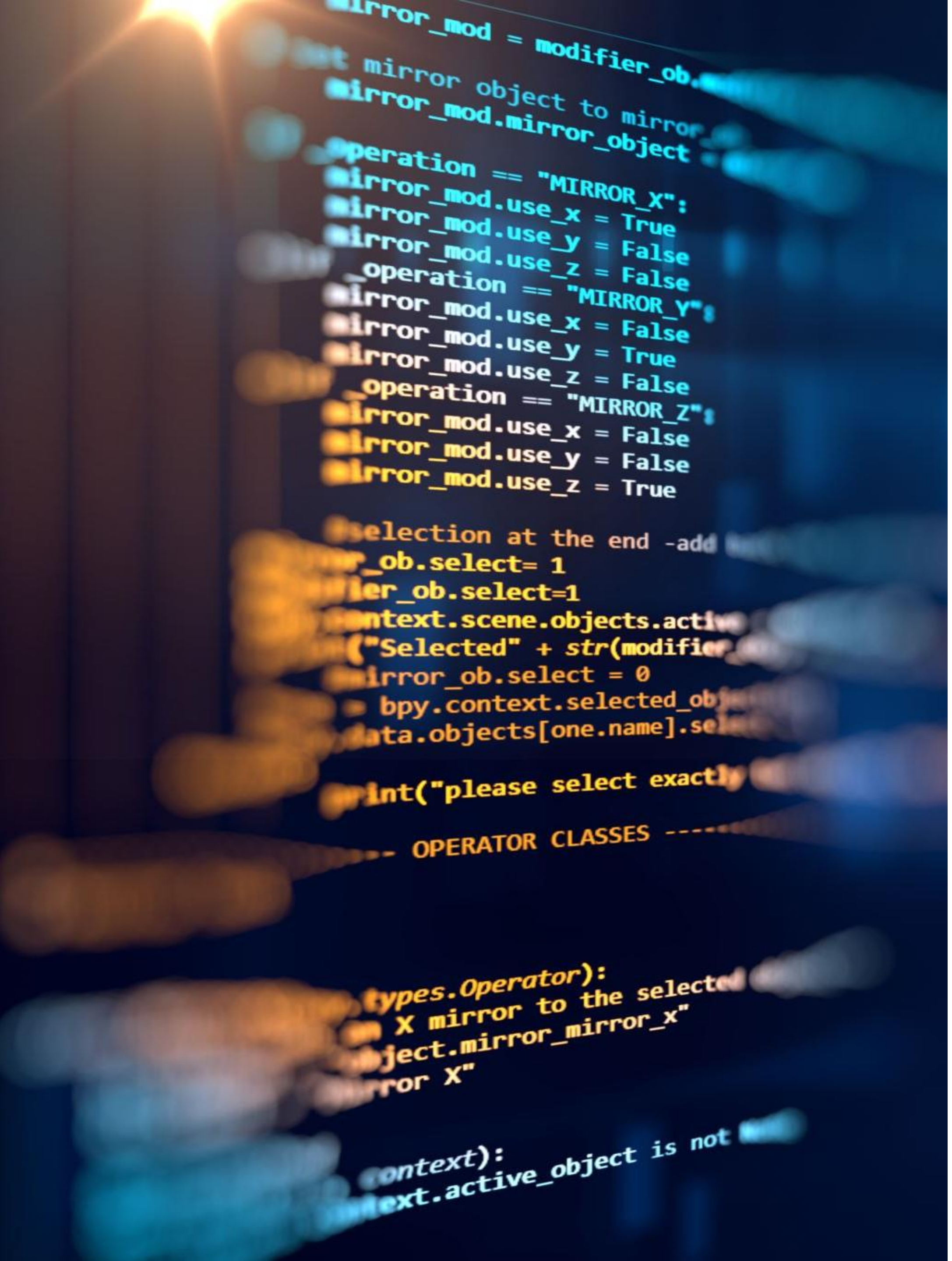
B: Tiled: $n^3/4B$

Speedup =

Speedup

If B = 16, 72X speedup ☺ ☺

Why 16?



Software (compiler/programmer) Prefetching

```
for( i=0; i< 3; i++)
    for (j=0;j<100;j++)
        a[i][j] = b[j][0] * b[j+1][0]
```

Where to add prefetch instructions?

Coffee Credits

Hrishikesh: +1