

Algorithms for GIS

csci3225

Laura Toma

Bowdoin College

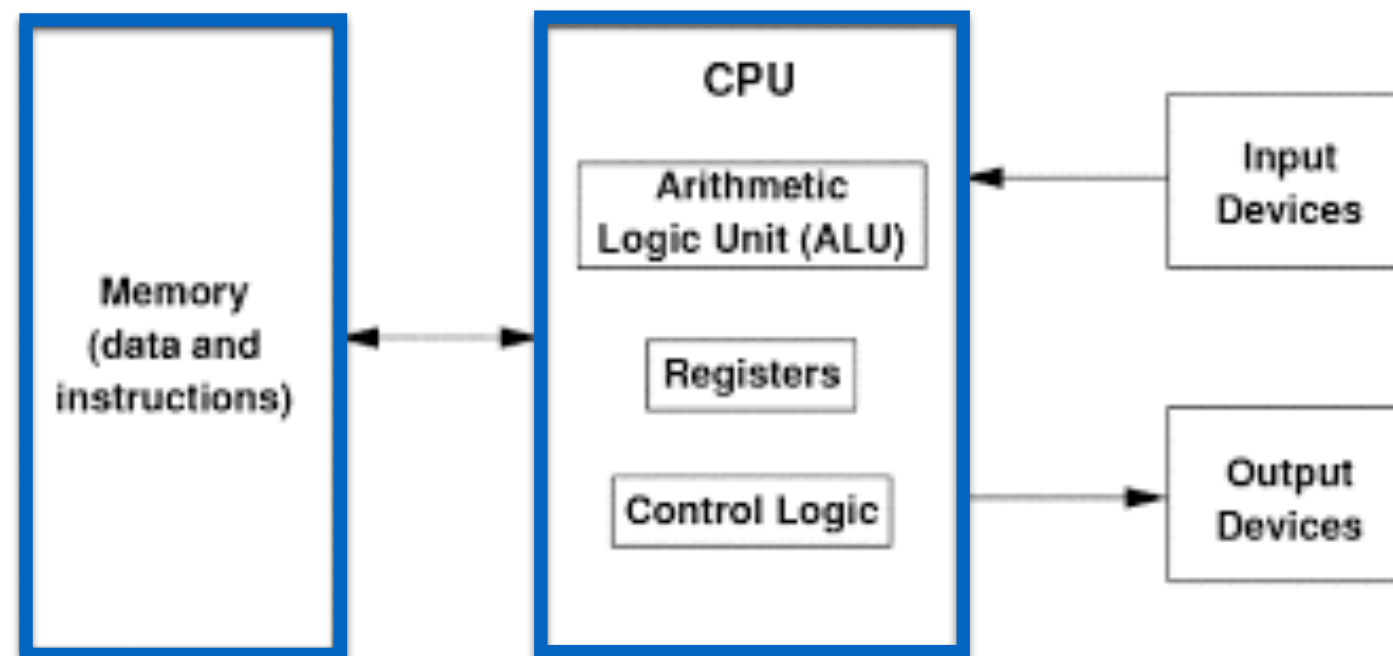
Memory-efficient algorithms

(Standard) algorithm analysis

- Running time = number of instructions in the **RAM model of computation**

RAM model = abstract model of a machine

- all arithmetic/logic/control instructions
- every instruction takes 1 step
- each memory access takes 1 step
- memory is infinite (data always fits in memory)

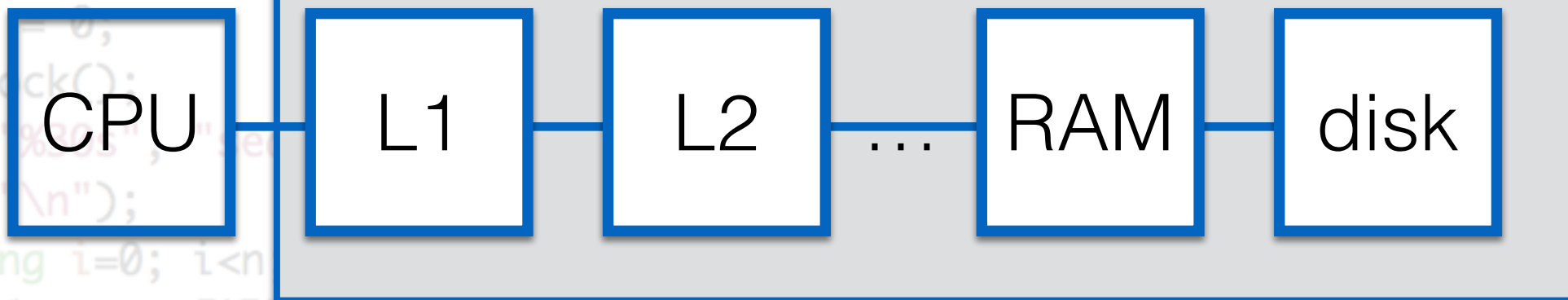


Algorithms run on real machines

```
//allocate and initialize a
t1 = clock();
printf("%30s", "allocate and
int* a = (int*)malloc(n * si
assert(a);
for (long i=0; i<n; i++) {
    a[i]=i;
}
t2 = clock();
printf("time elapsed %.3f seconds\n", (double)(t2-t1)/CLOCKS_PER_SEC);
```

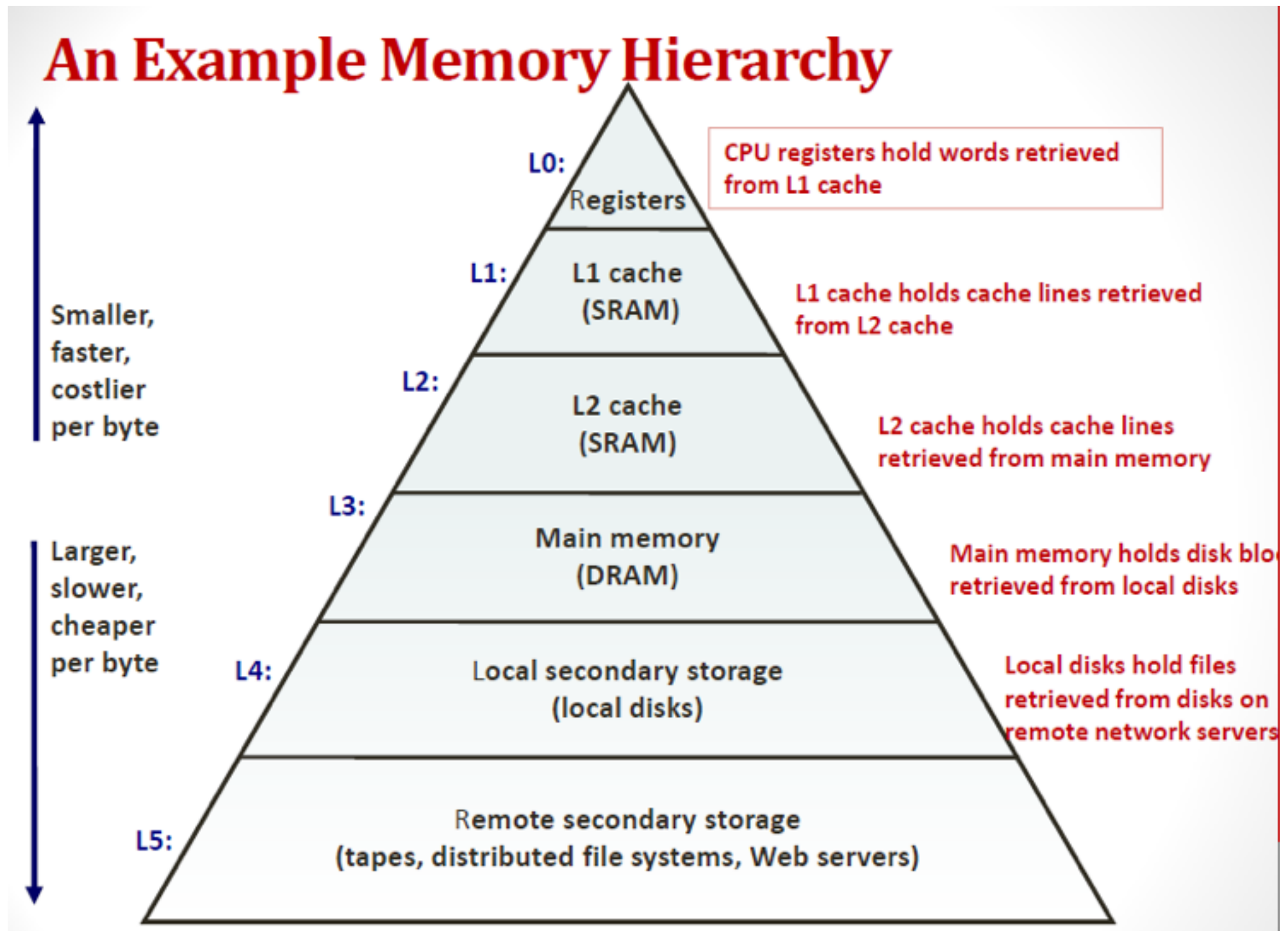


memory hierarchy

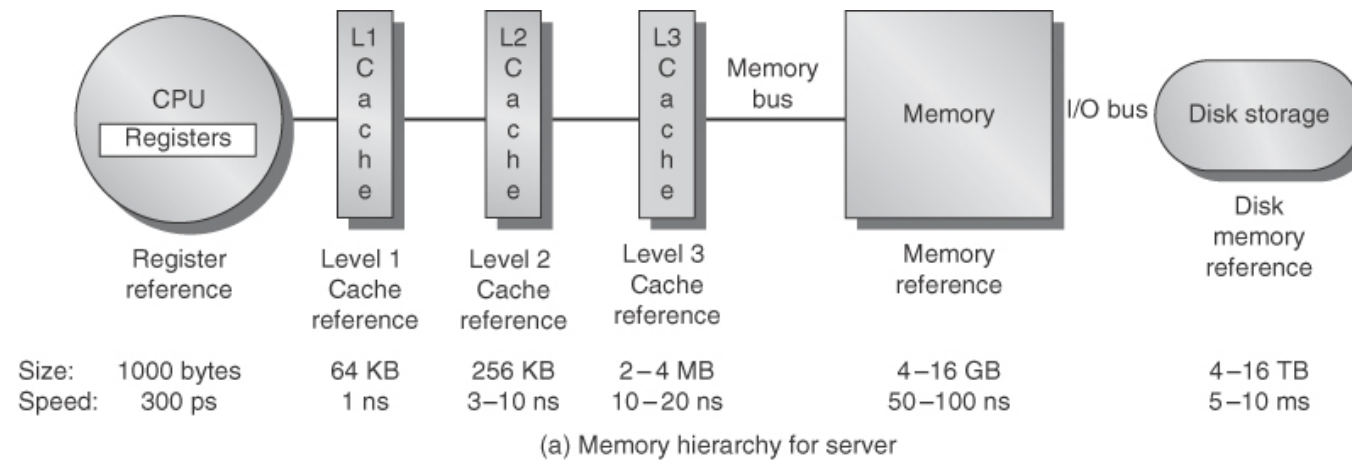


—————larger, slower—————→

The memory hierarchy



Example



RAM is 50 times slower than L1

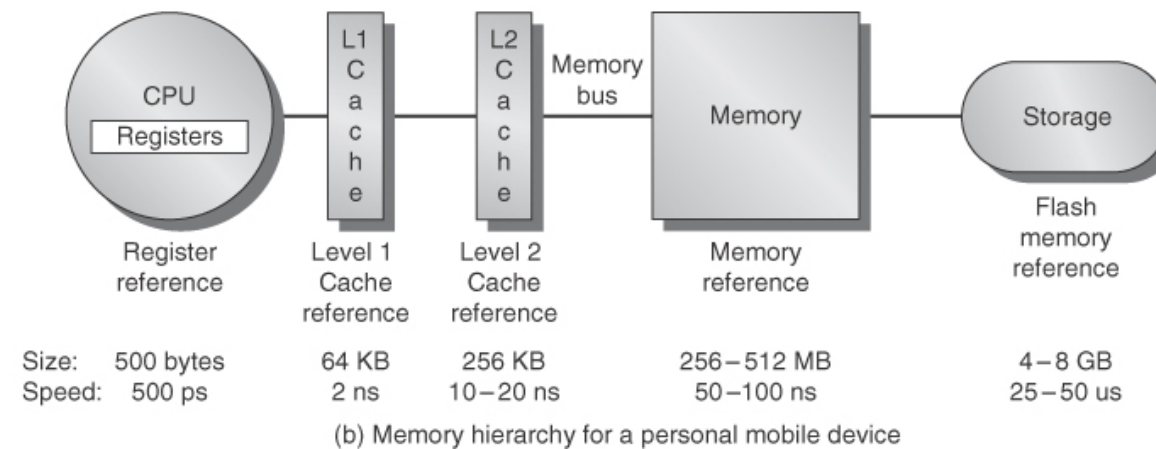
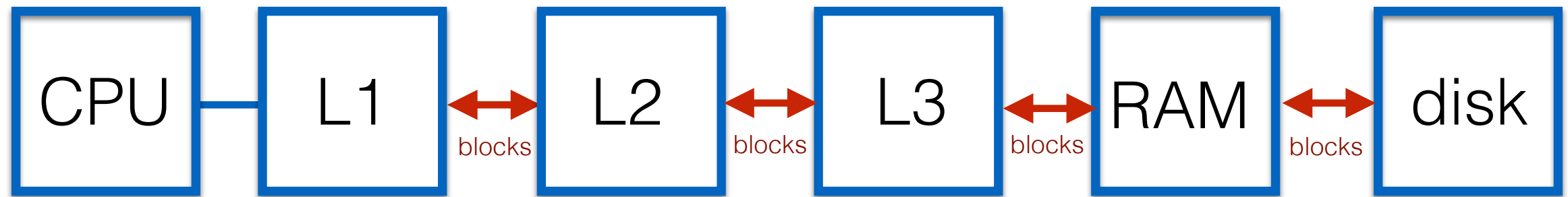


Figure 2.1 The levels in a typical memory hierarchy in a server computer shown on top (a) and in a personal mobile device (PMD) on the bottom (b). As we move farther away from the processor, the memory in the level below becomes slower and larger. Note that the time units change by a factor of 10^9 —from picoseconds to milliseconds—and that the size units change by a factor of 10^{12} —from bytes to terabytes. The PMD has a slower clock rate and smaller caches and main memory. A key difference is that servers and desktops use disk storage as the lowest level in the hierarchy while PMDs use Flash, which is built from EEPROM technology.

The memory hierarchy



- At all levels, data is organized and moved in **blocks/pages**
- Each level acts as a “cache” for the next level (stores most recently used blocks)
- When CPU needs to do a memory access, it first checks if the block that contains that data is (already) in the cache
 - **cache hit**: the block that contains the data item is in cache; read it from cache
 - **cache miss**: the block that contains the data item is not in cache; read the block in cache (and, if necessary, evict a block to make space)
 - **cache replacement policy**: OPT, LRU, FIFO, ..

Why does it matter anyways? Aren't computers getting faster?

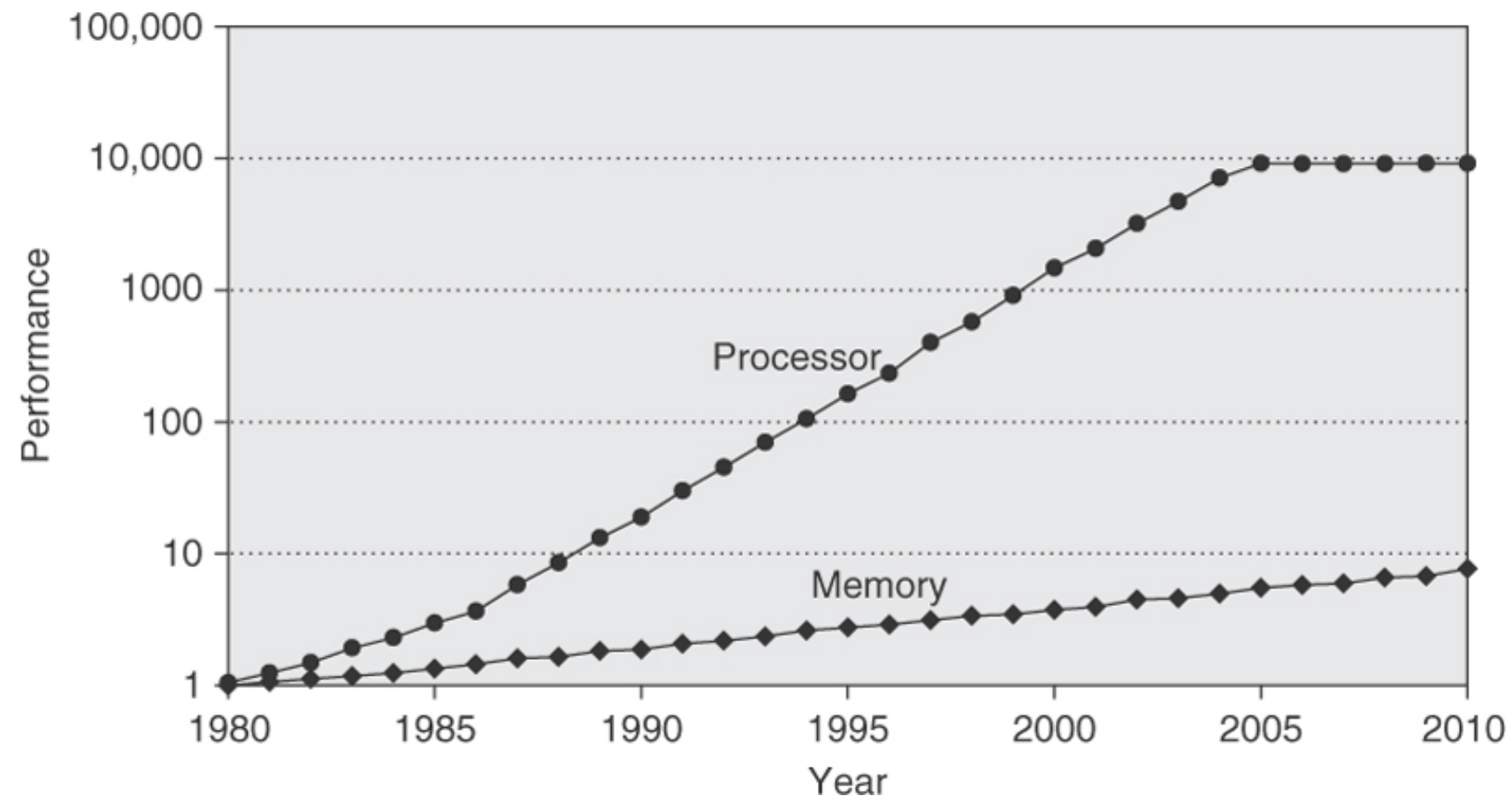
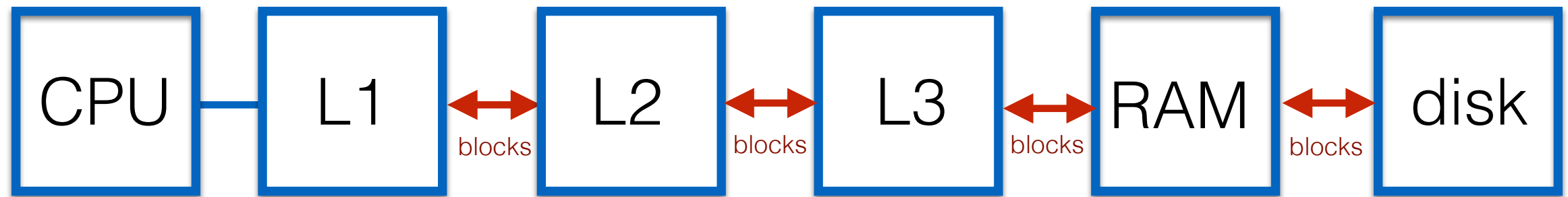


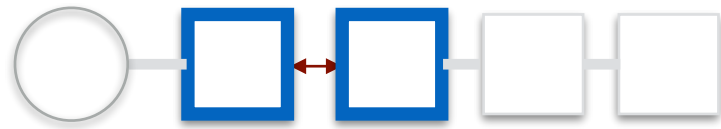
Figure 2.2 Starting with 1980 performance as a baseline, the gap in performance, measured as the difference in the time between processor memory requests (for a single processor or core) and the latency of a DRAM access, is plotted over time. Note that the vertical axis must be on a logarithmic scale to record the size of the processor–DRAM performance gap. The memory baseline is 64 KB DRAM in 1980, with a 1.07 per year performance improvement in latency (see Figure 2.13 on page 99). The processor line assumes a 1.25 improvement per year until 1986, a 1.52 improvement until 2000, a 1.20 improvement between 2000 and 2005, and no change in processor performance (on a per-core basis) between 2005 and 2010; see Figure 1.1 in Chapter 1.

Refined algorithm analysis

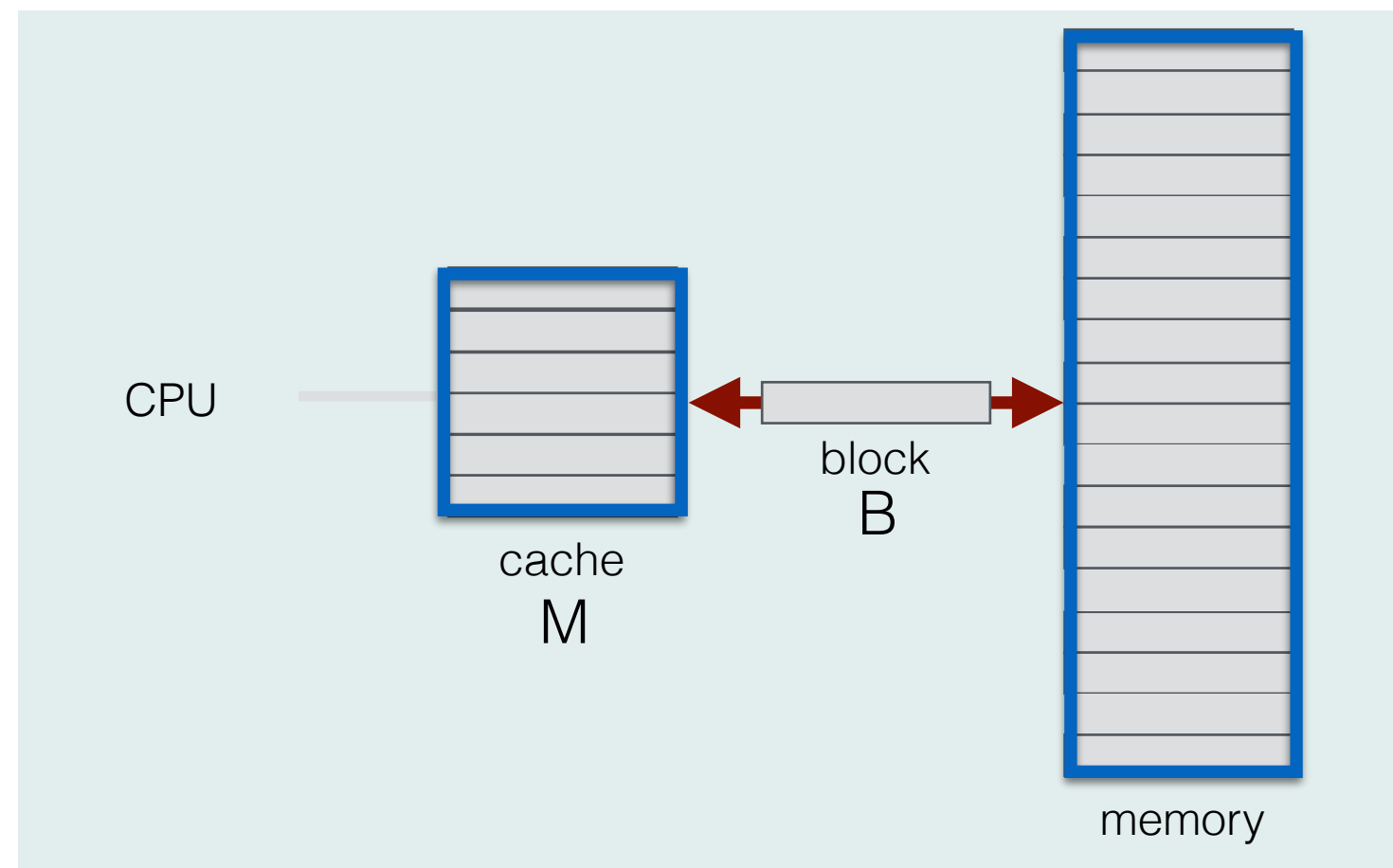


- CPU analysis
 - number of instructions
- Memory access analysis
 - number of blocks transferred between levels of the memory hierarchy

Memory access analysis: **The ideal-cache model**



- Consider only two levels of the memory hierarchy
- Cache size **M** bytes
- Cache block (line) size **B** bytes
- Assume LRU or FIFO block replacement
- Fully associative cache (a block can go anywhere)
- Count only one direction (what goes in must come out)



Performance measure: **number of cache misses**

$$Q(n) = Q(n, M, B)$$

Analysis in the ideal-cache model

Example: Scanning

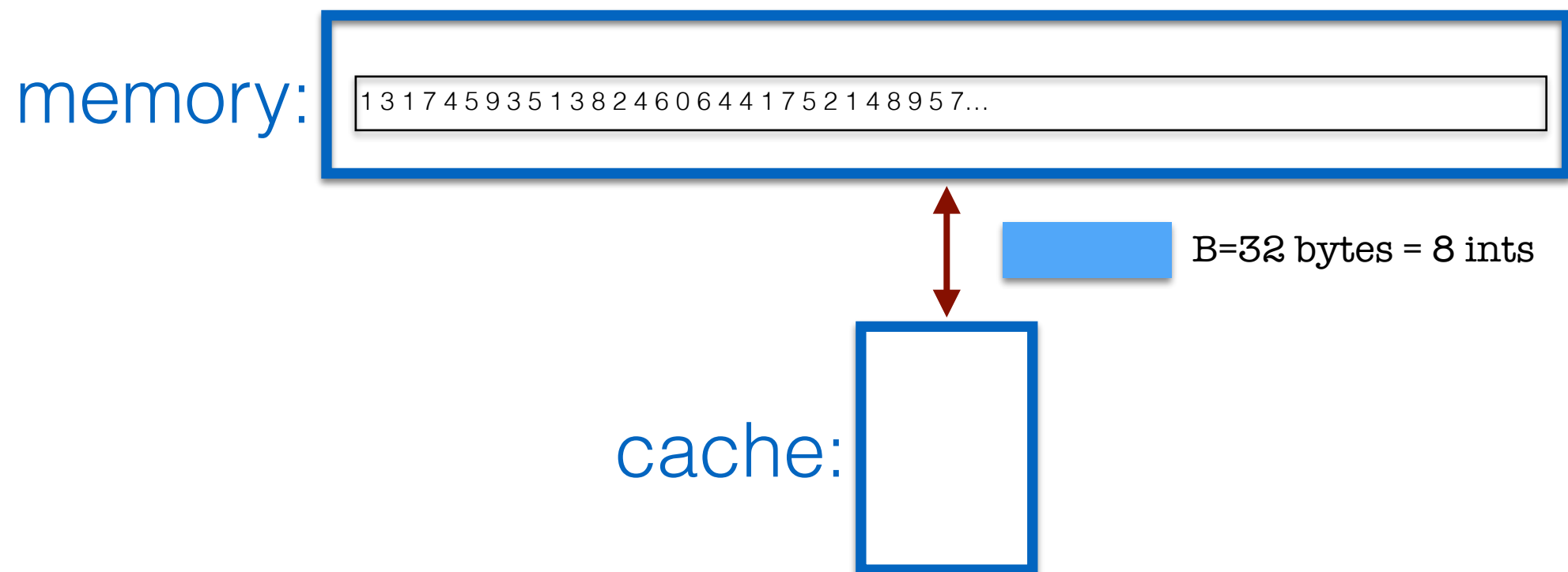
```
//array a was initialized with n elements  
sum = 0  
for (i=0; i<n; i++)  
    sum += a[i]
```

Example: Scanning

```
//array a was initialized with n elements  
sum = 0  
for (i=0; i<n; i++)  
    sum += a[i]
```

Notation:

- n: array size
- M: cache size
- B: block size

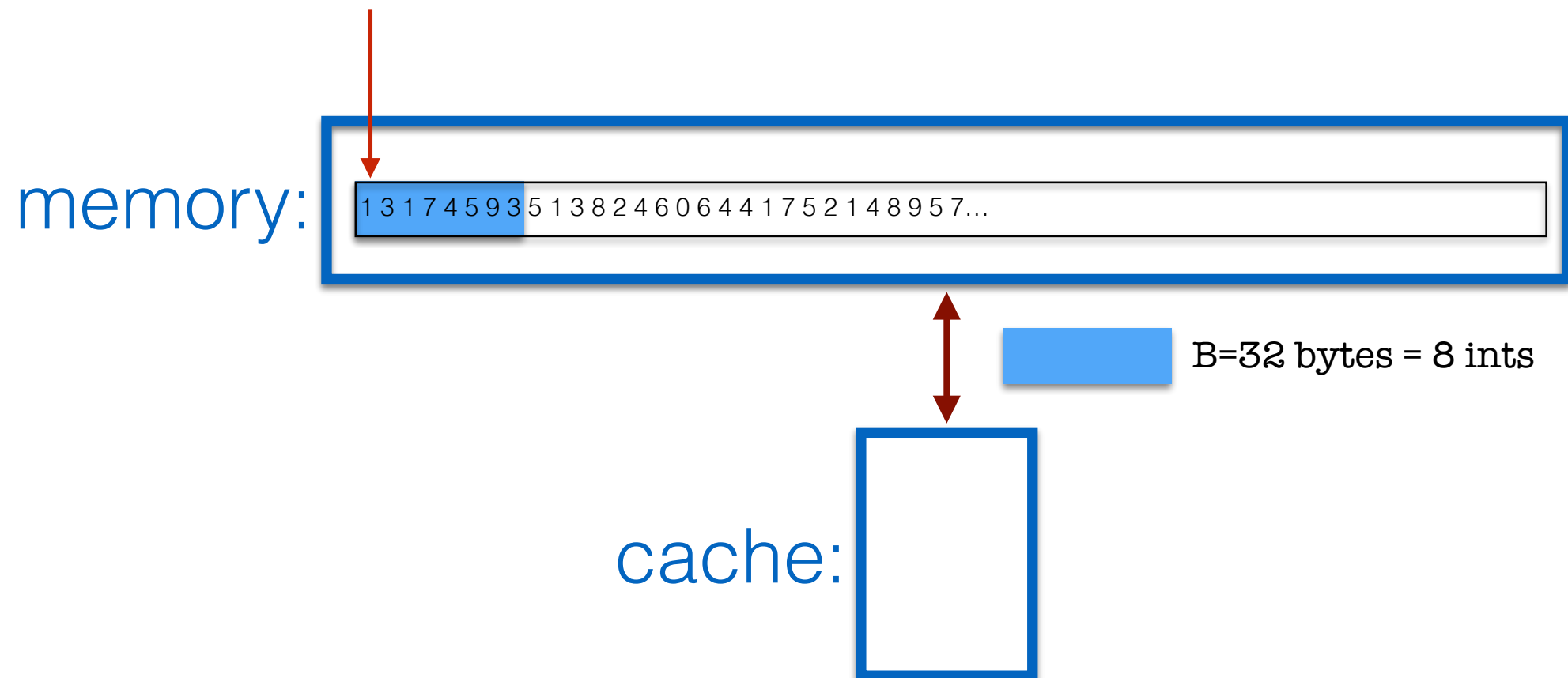


Example: Scanning

```
//array a was initialized with n elements  
sum = 0  
for (i=0; i<n; i++)  
    sum += a[i]
```

Notation:

- n: array size
- M: cache size
- B: block size

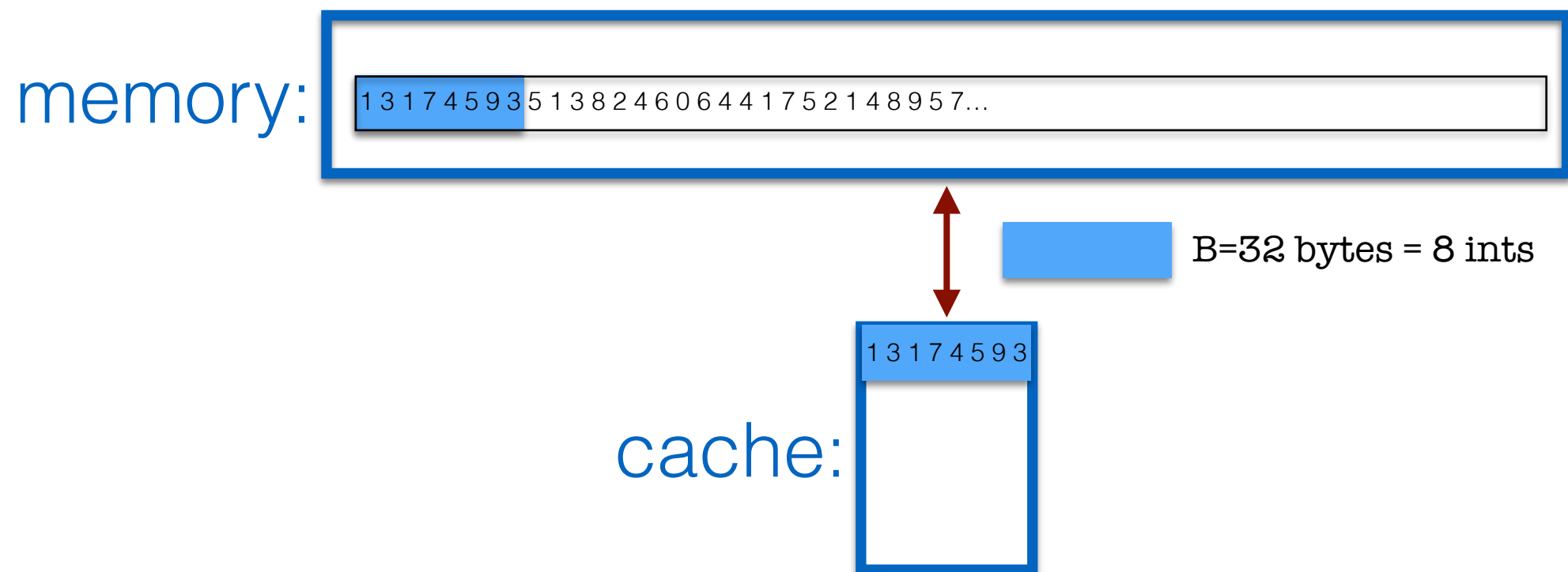


Example: Scanning

```
//array a was initialized with n elements  
sum =0  
for (i=0; i<n; i++)  
    sum += a[i]
```

Notation:

- n: array size
- M: cache size
- B: block size

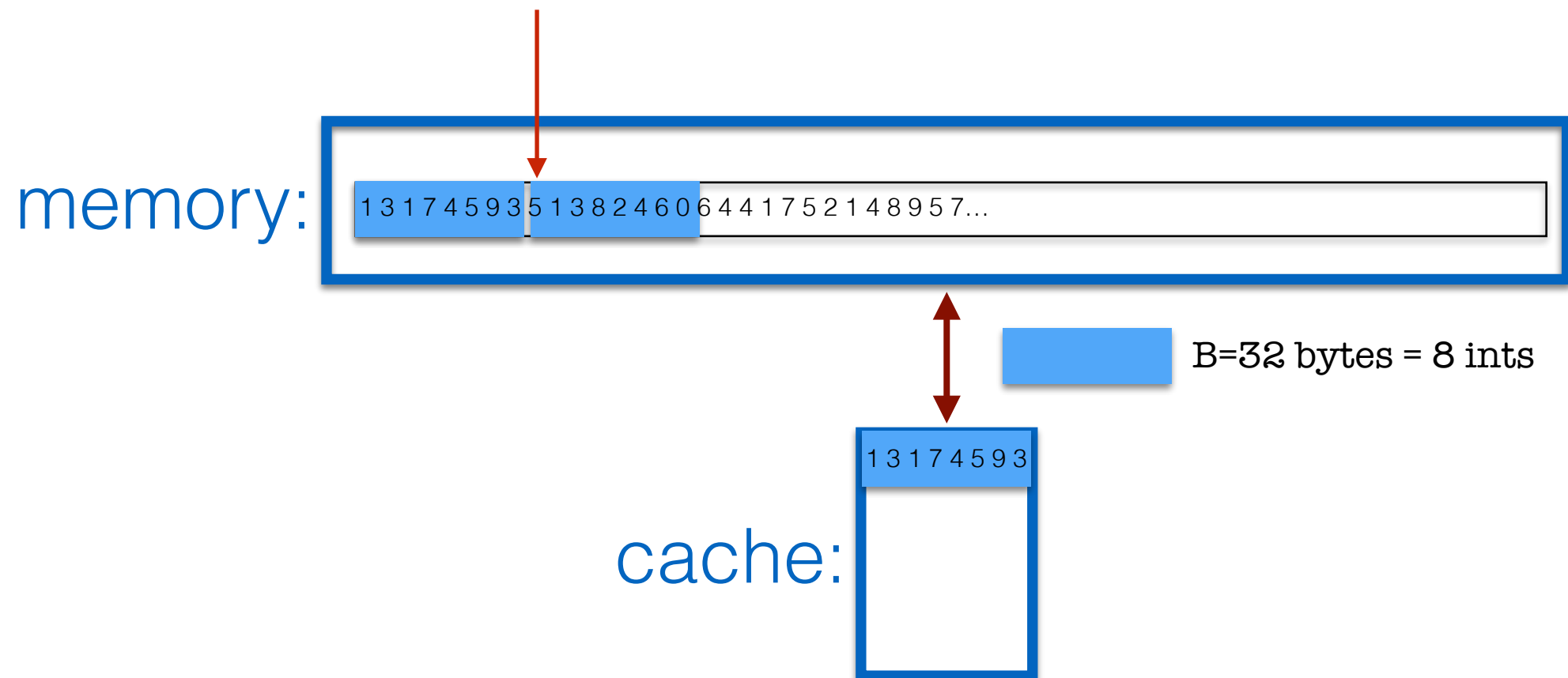


Example: Scanning

```
//array a was initialized with n elements  
sum = 0  
for (i=0; i<n; i++)  
    sum += a[i]
```

Notation:

- n: array size
- M: cache size
- B: block size

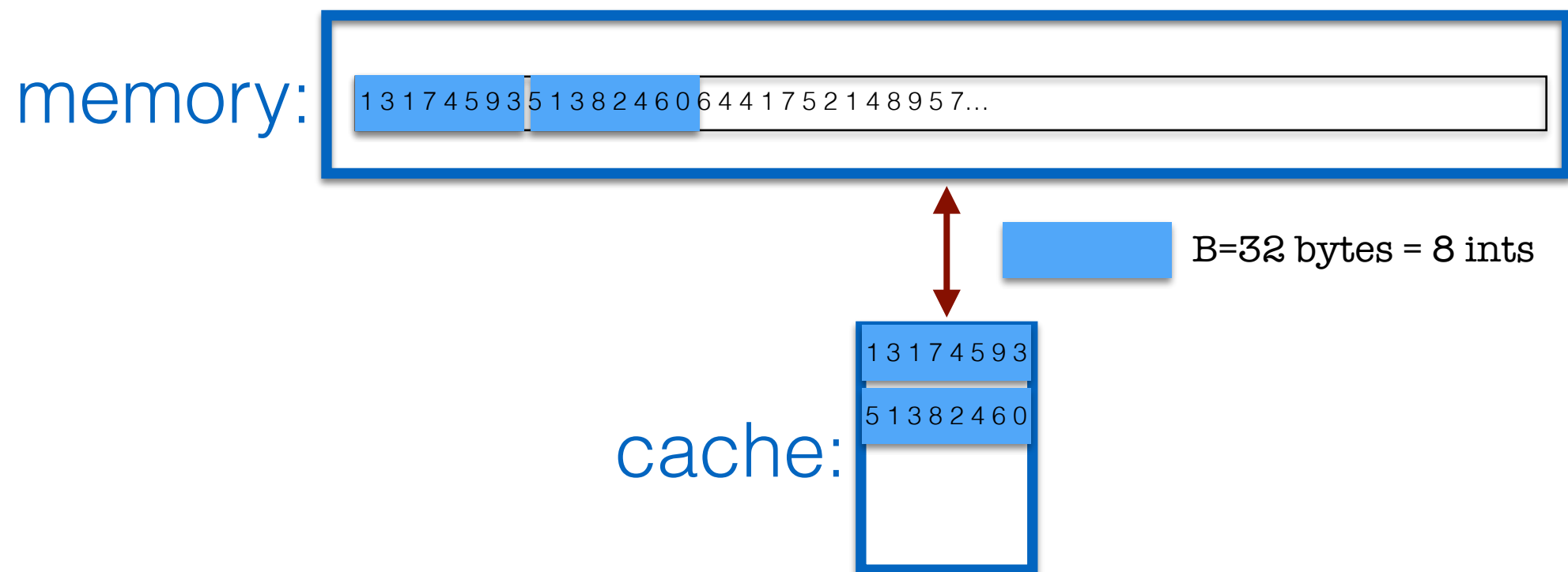


Example: Scanning

```
//array a was initialized with n elements  
sum = 0  
for (i=0; i<n; i++)  
    sum += a[i]
```

Notation:

- n: array size
- M: cache size
- B: block size

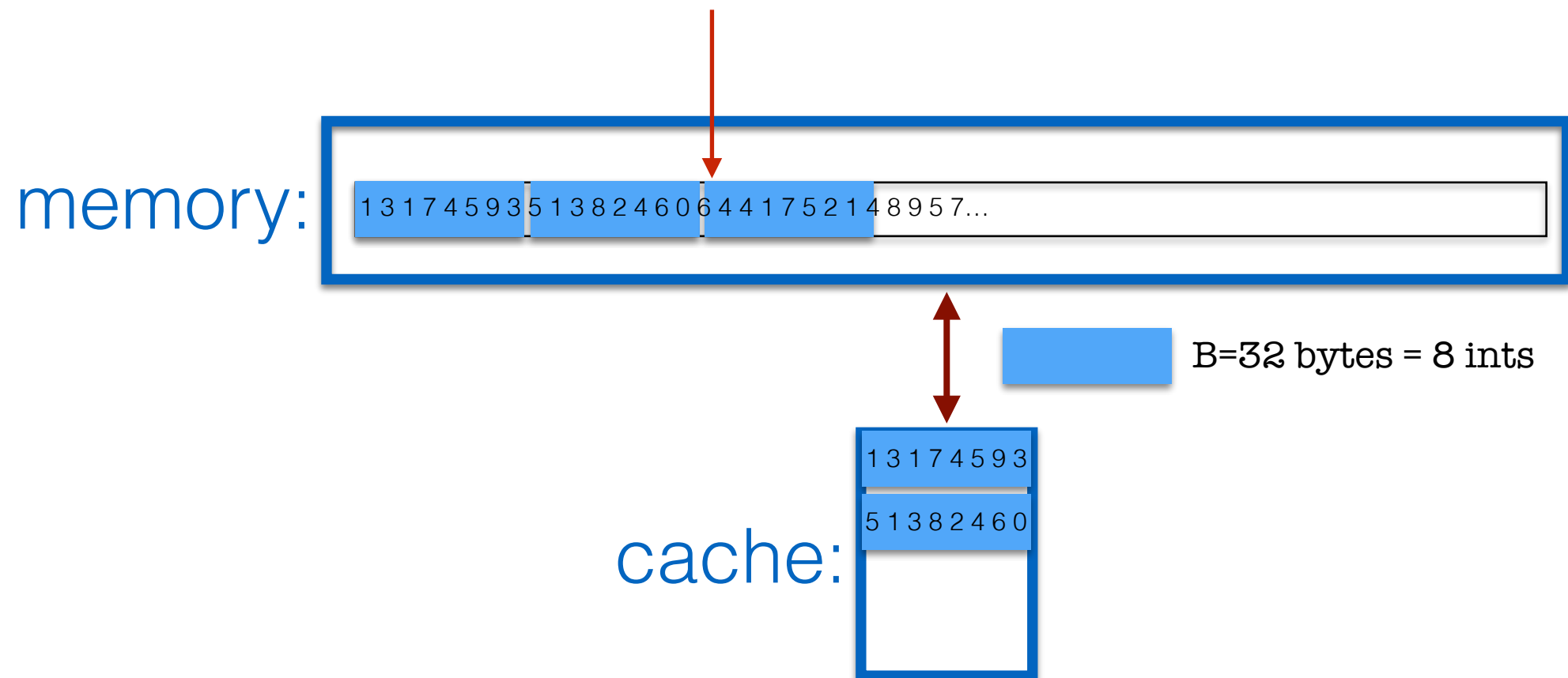


Example: Scanning

```
//array a was initialized with n elements  
sum = 0  
for (i=0; i<n; i++)  
    sum += a[i]
```

Notation:

- n: array size
- M: cache size
- B: block size

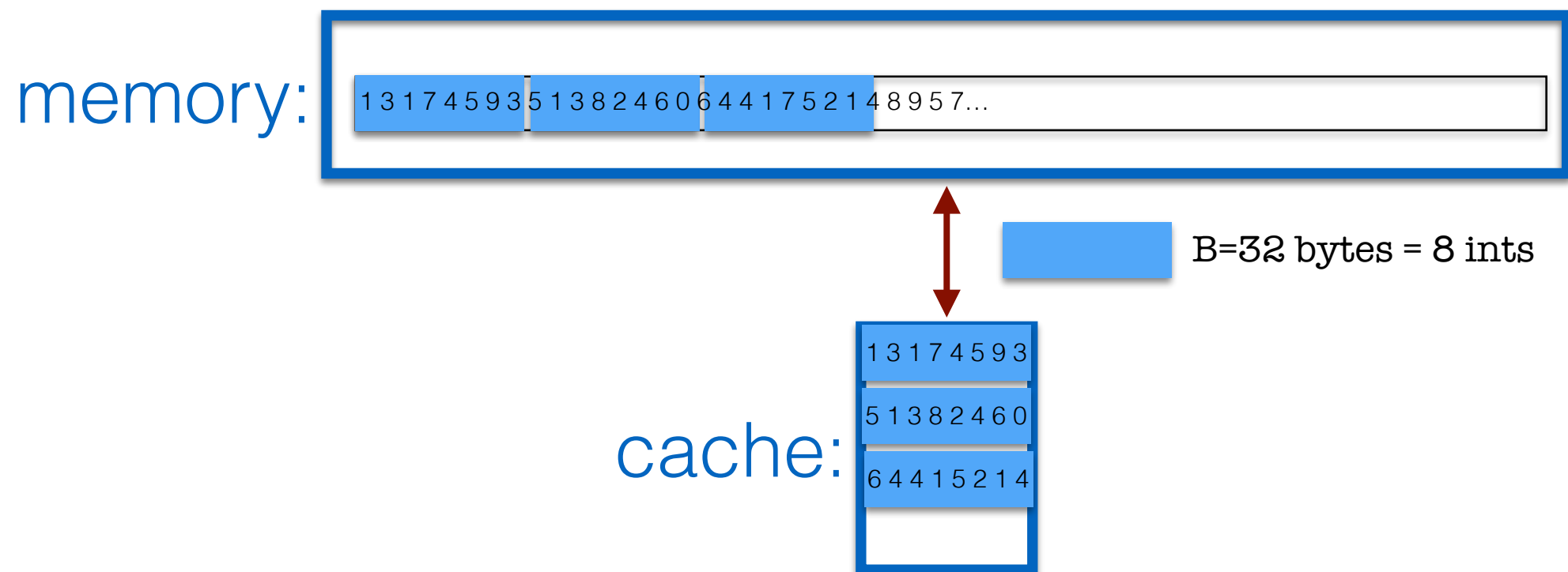


Example: Scanning

```
//array a was initialized with n elements  
sum = 0  
for (i=0; i<n; i++)  
    sum += a[i]
```

Notation:

- n: array size
- M: cache size
- B: block size

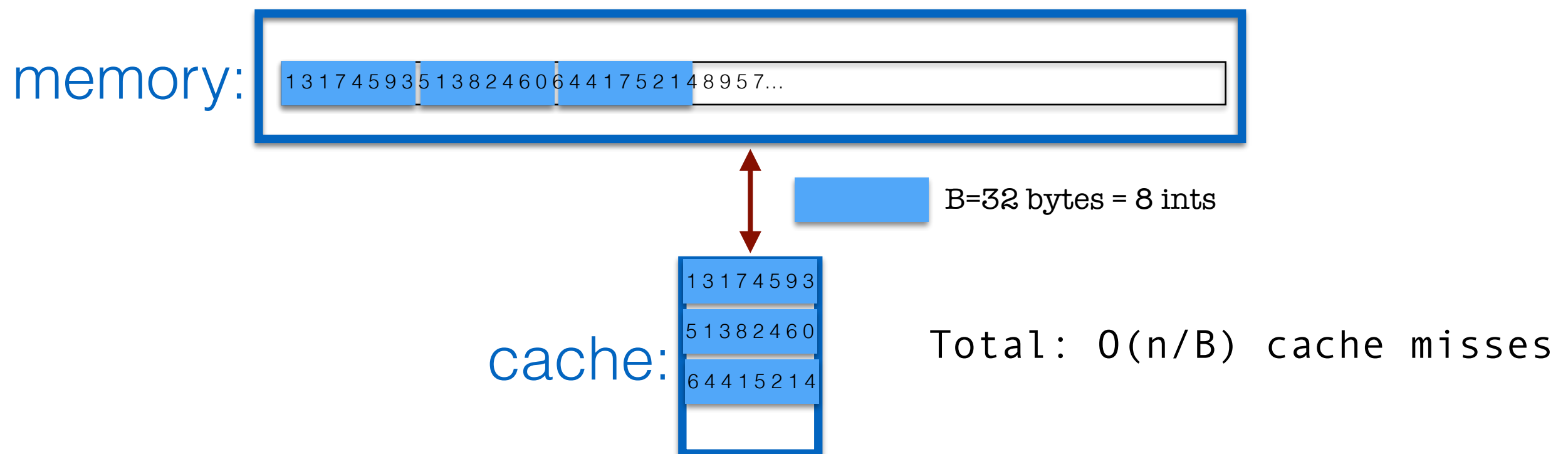


Example: Scanning

```
//array a was initialized with n elements  
sum = 0  
for (i=0; i<n; i++)  
    sum += a[i]
```

Notation:

- n : array size
- M : cache size
- B : block size



Example: Random access

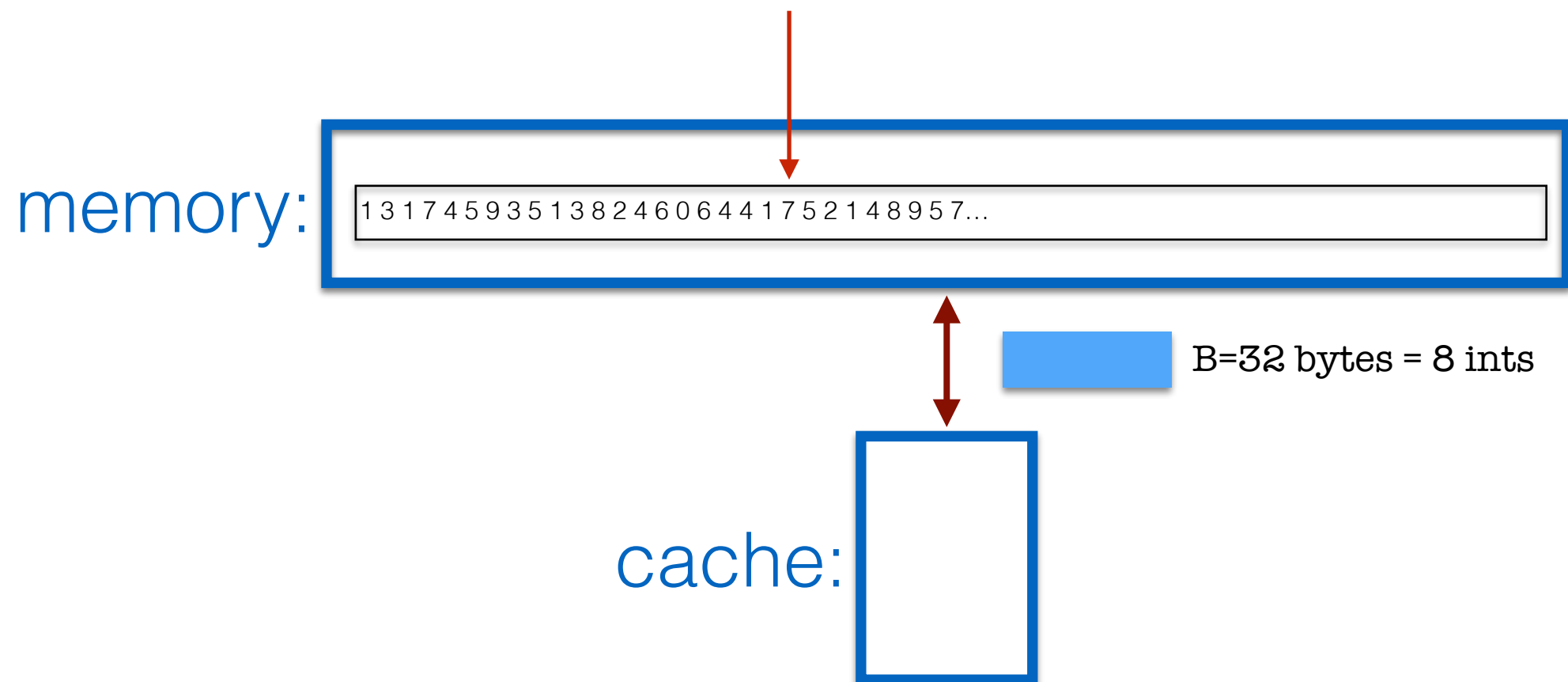
```
//array a was initialized with n elements  
sum =0  
for (i=0; i<n; i++)  
    k = random() %n ;  
    sum += a[k]
```

Example: Random access

```
//array a was initialized with n elements  
sum = 0  
for (i=0; i<n; i++)  
    k = random() %n ;  
    sum += a[k]
```

Notation:

- n: array size
- M: cache size
- B: block size

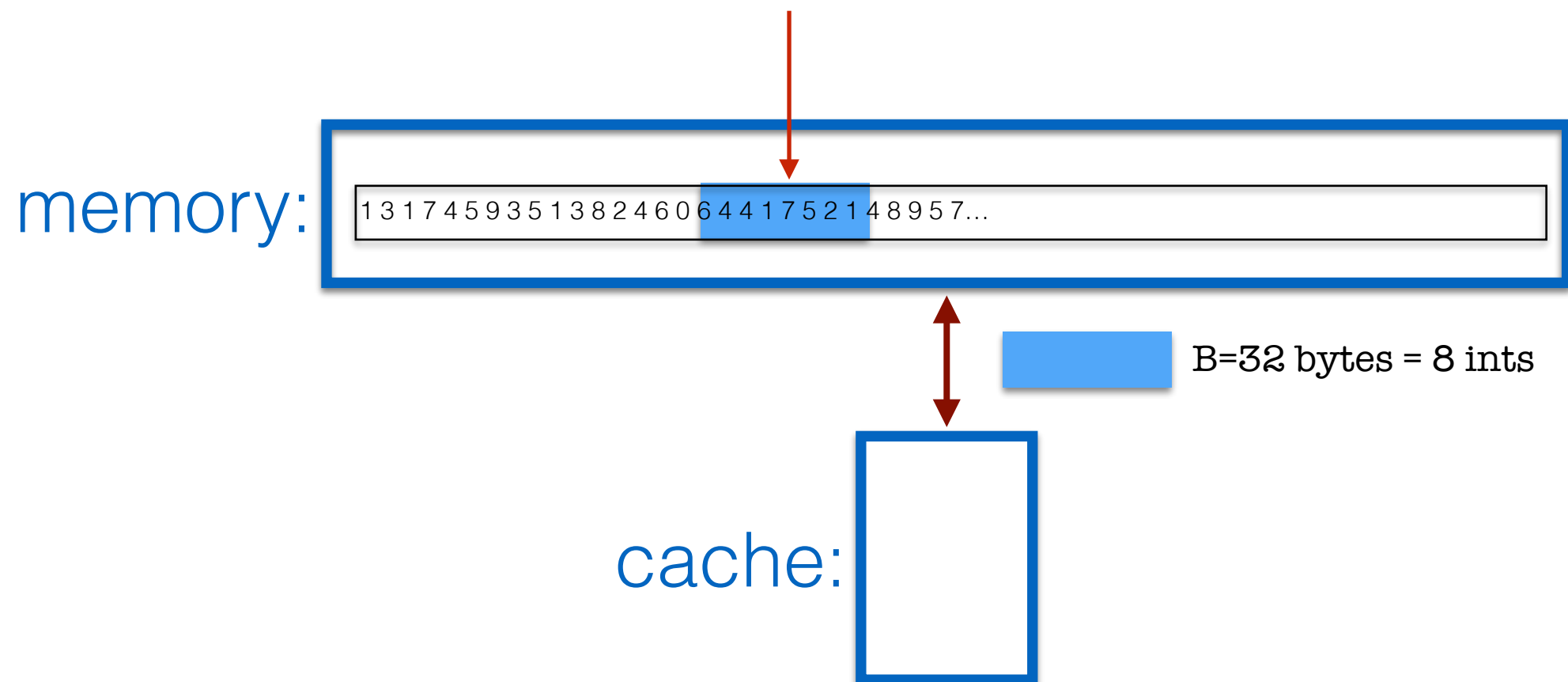


Example: Random access

```
//array a was initialized with n elements  
sum = 0  
for (i=0; i<n; i++)  
    k = random() %n ;  
    sum += a[k]
```

Notation:

- n: array size
- M: cache size
- B: block size

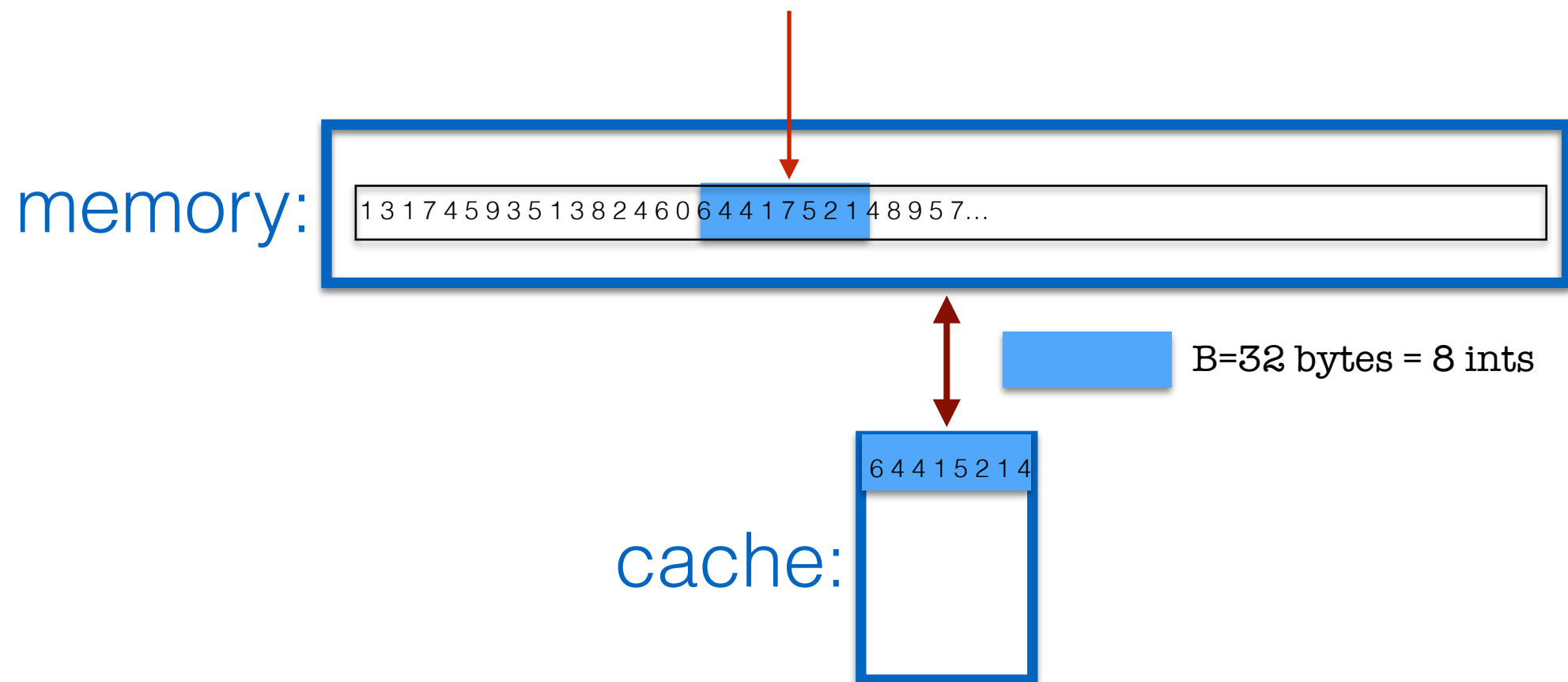


Example: Random access

```
//array a was initialized with n elements  
sum = 0  
for (i=0; i<n; i++)  
    k = random() %n ;  
    sum += a[k]
```

Notation:

- n: array size
- M: cache size
- B: block size

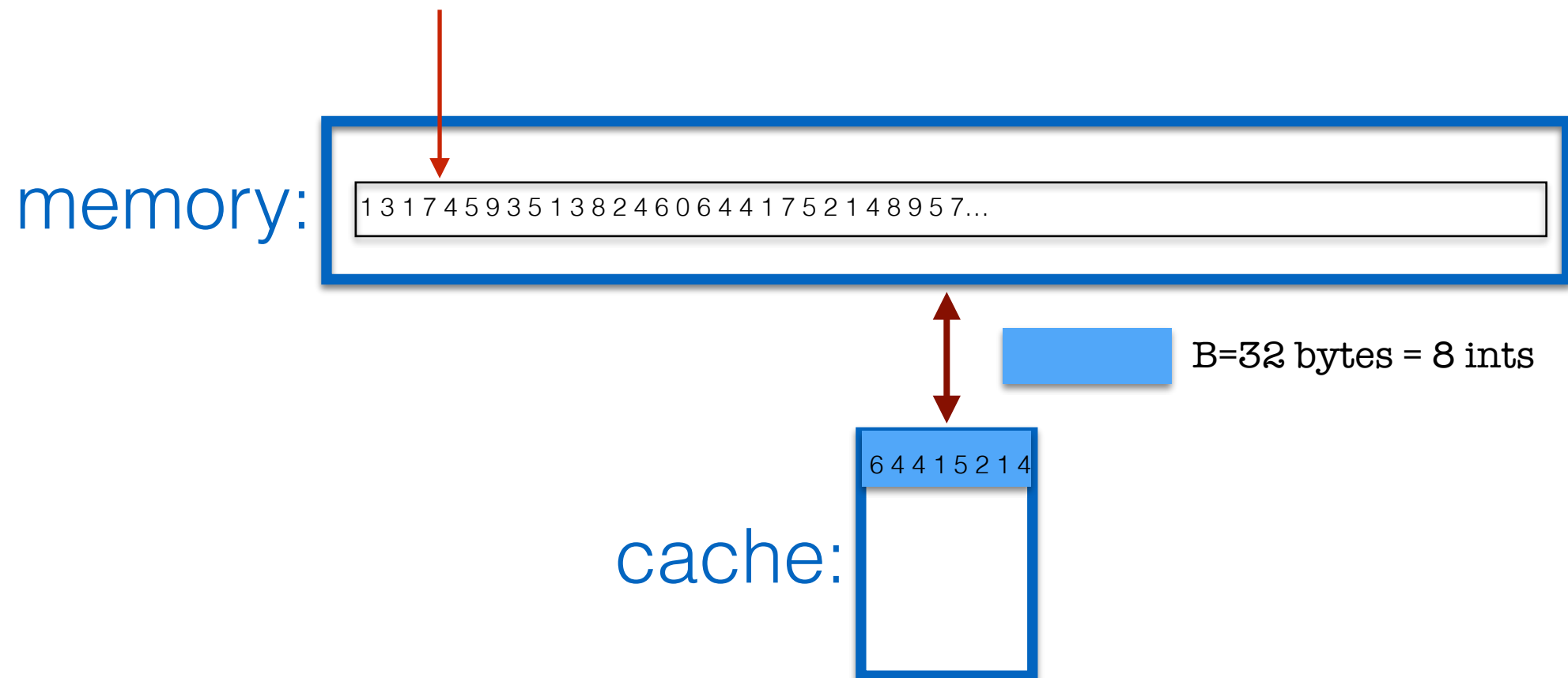


Example: Random access

```
//array a was initialized with n elements  
sum = 0  
for (i=0; i<n; i++)  
    k = random() %n ;  
    sum += a[k]
```

Notation:

- n: array size
- M: cache size
- B: block size

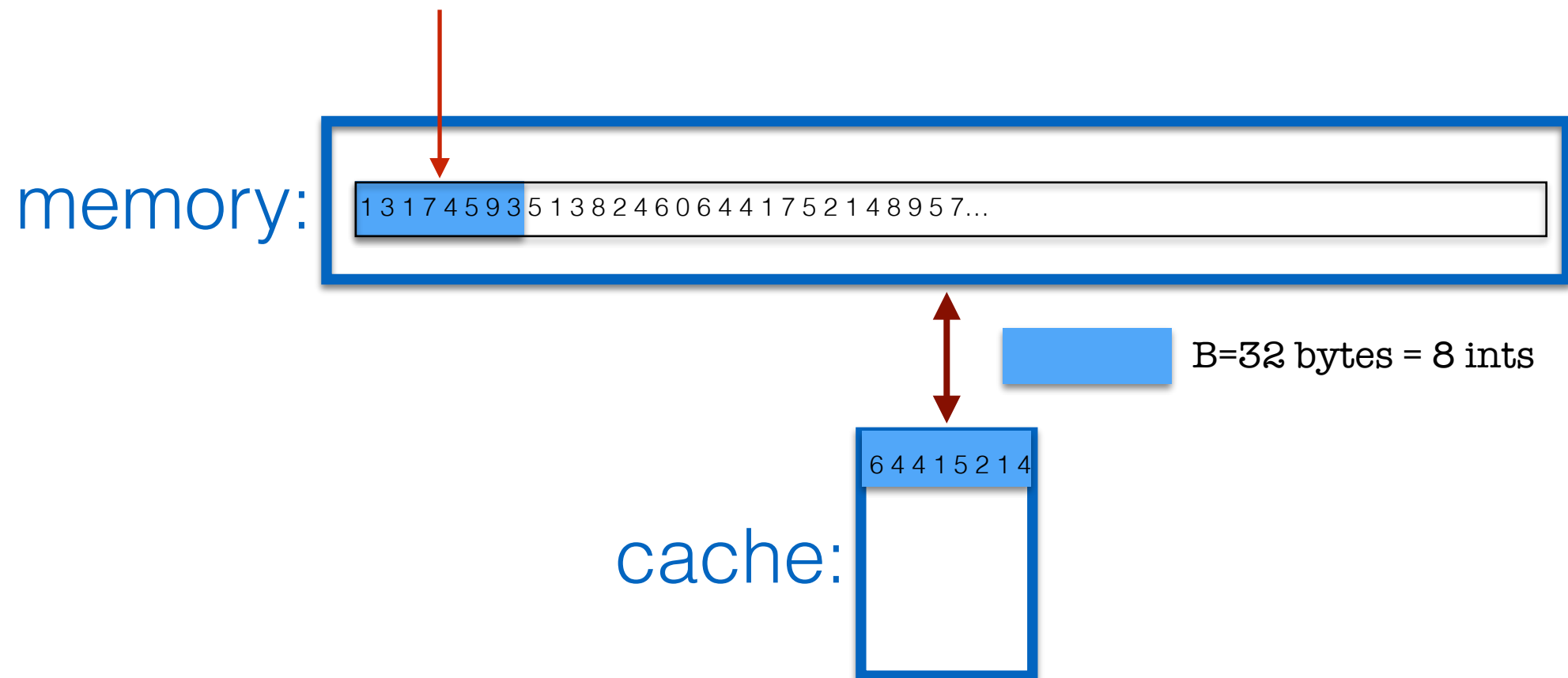


Example: Random access

```
//array a was initialized with n elements  
sum = 0  
for (i=0; i<n; i++)  
    k = random() %n ;  
    sum += a[k]
```

Notation:

- n: array size
- M: cache size
- B: block size

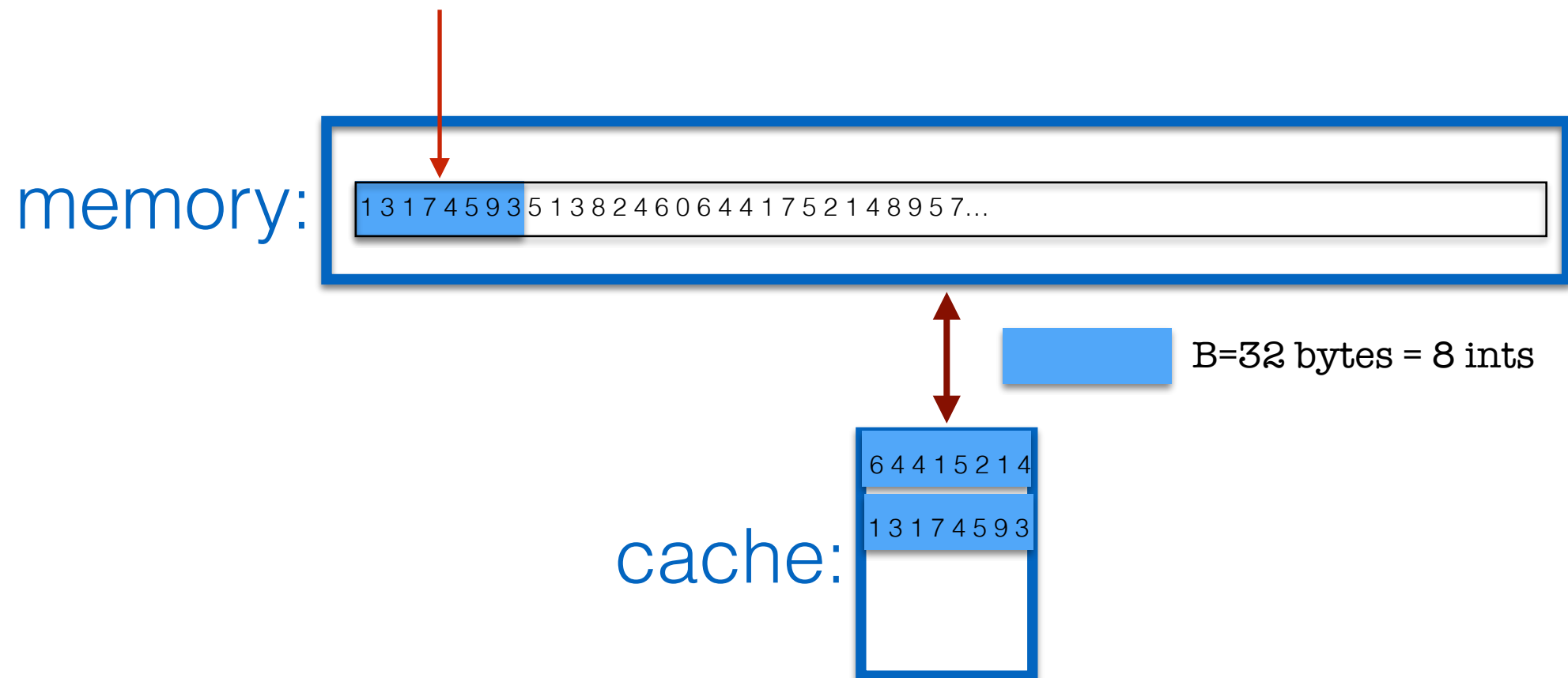


Example: Random access

```
//array a was initialized with n elements  
sum = 0  
for (i=0; i<n; i++)  
    k = random() %n ;  
    sum += a[k]
```

Notation:

- n: array size
- M: cache size
- B: block size

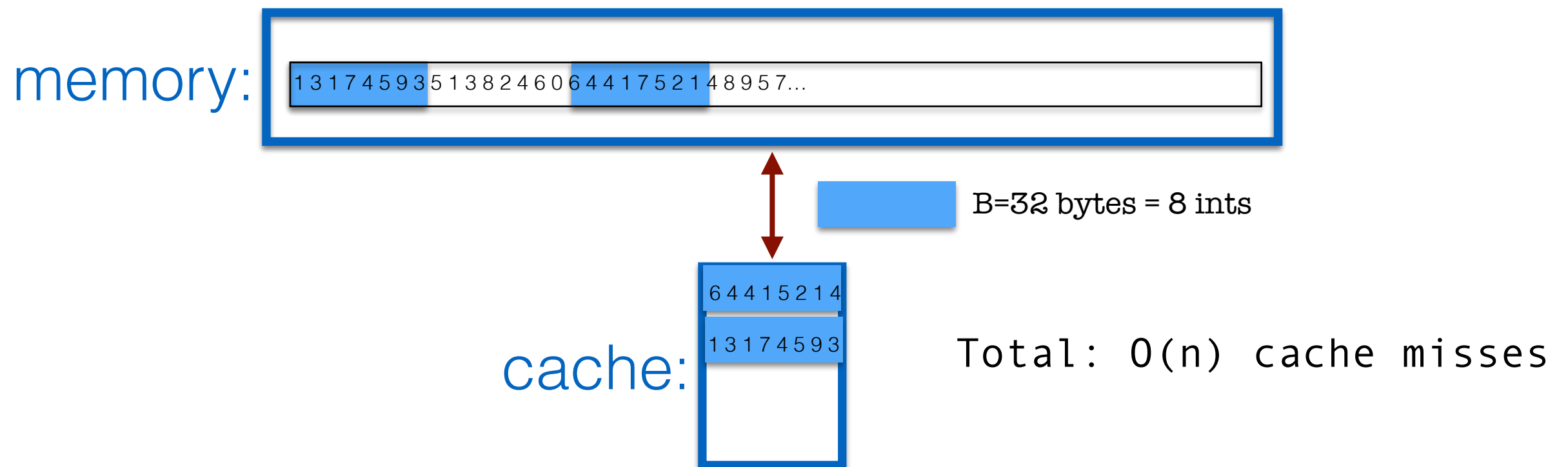


Example: Random access

```
//array a was initialized with n elements  
sum = 0  
for (i=0; i<n; i++)  
    k = random() %n ;  
    sum += a[k]
```

Notation:

- n: array size
- M: cache size
- B: block size

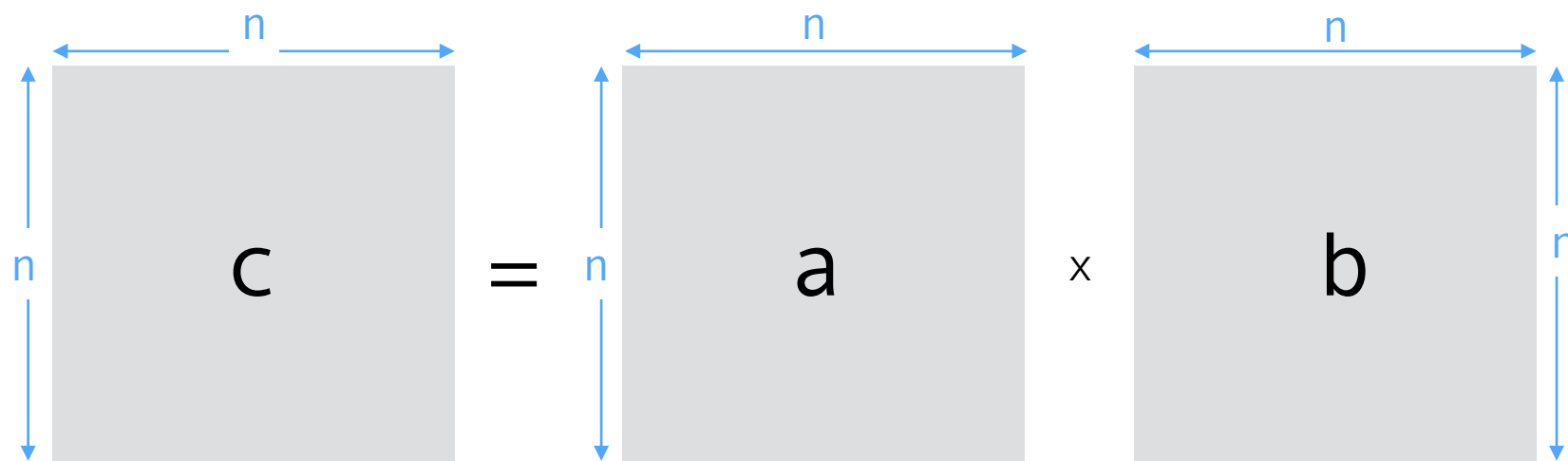


Memory-efficient algorithms

- **Summary**
 - Sequential access: $O(n/B)$ cache misses
 - Random accesses: $O(n)$ cache misses
- **Spatial locality**
 - If a memory location is accessed, it's likely that a nearby memory location will be accessed in the near future
- **Temporal locality**
 - If a memory location is accessed, it's likely that the same location will be accessed again in the near future
- Memory-efficient algorithms need to improve (spatial and/or temporal) locality

Matrix multiplication

Given two n -by- n matrices, compute their product



Three algorithms:

- standard
 - blocked/tiling
 - divide-and-conquer
-
- Illustrate the analysis and design of memory-efficient algorithms

Matrix multiplication: standard

```
//we'll use linearized arrays for efficiency

c = (double*) calloc(sizeof(double), n*n)

void mmult(double* a, double* b, double*c, int n) {
    int i, j, k;
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            for (k=0; k<n; k++)
                //c[i][j] += a[i][k]*b[k][j]
                c[i*n+j] += a[i*n+k]*b[k*n+j]
}
```

Matrix multiplication: standard

```
//we'll use linearized arrays for efficiency

c = (double*) calloc(sizeof(double), n*n)

void mmult(double* a, double* b, double*c, int n) {
    int i, j, k;
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            for (k=0; k<n; k++)
                //c[i][j] += a[i][k]*b[k][j]
                c[i*n+j] += a[i*n+k]*b[k*n+j]
}
```

CPU: $O(n^3)$

cache-misses:

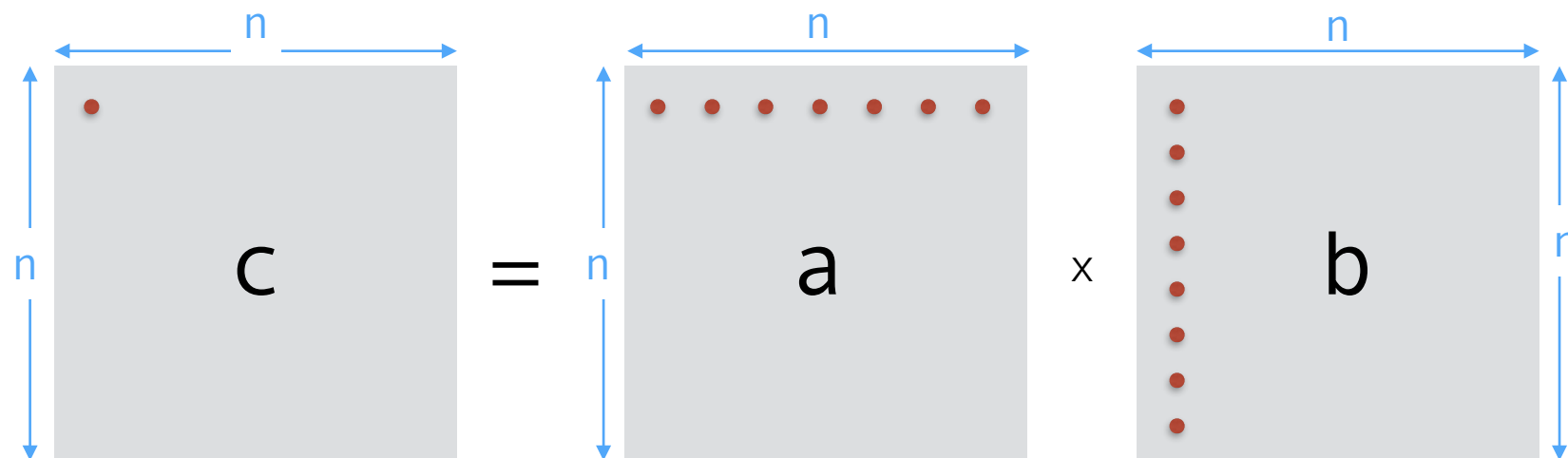
Cache-miss analysis

```
c = (double*) calloc(sizeof(double), n*n)
void mmult(double* a, double* b, double* c, int n) {
    int i, j, k;
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            for (k=0; k<n; k++)
                c[i*n+j] += a[i*n+k]*b[k*n+j]
}
```

Notation:

- n : array size
- M : cache size
- B : block size

Assume $B \ll n$



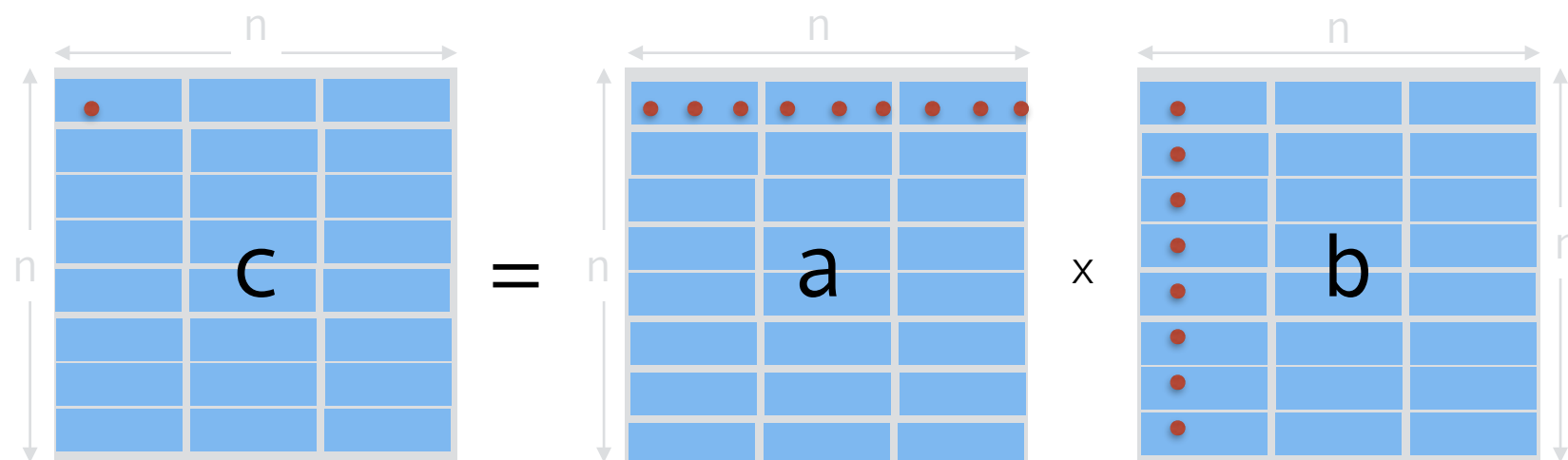
Cache-miss analysis

```
c = (double*) calloc(sizeof(double), n*n)
void mmult(double* a, double* b, double* c, int n) {
    int i, j, k;
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            for (k=0; k<n; k++)
                c[i*n+j] += a[i*n+k]*b[k*n+j]
}
```

Notation:

- n : array size
- M : cache size
- B : block size

Assume $B \ll n$



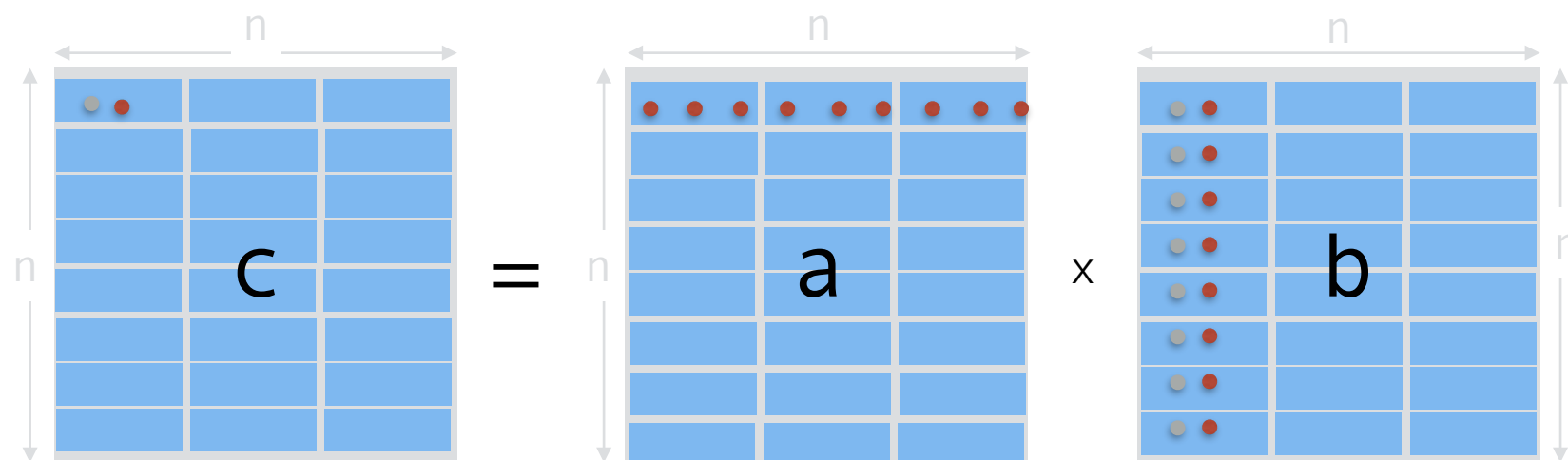
Cache-miss analysis

```
c = (double*) calloc(sizeof(double), n*n)
void mmult(double* a, double* b, double* c, int n) {
    int i, j, k;
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            for (k=0; k<n; k++)
                c[i*n+j] += a[i*n+k]*b[k*n+j]
}
```

Notation:

- n : array size
- M : cache size
- B : block size

Assume $B \ll n$



each element in c accesses a row of a (n/B blocks) and a column of b (B blocks)

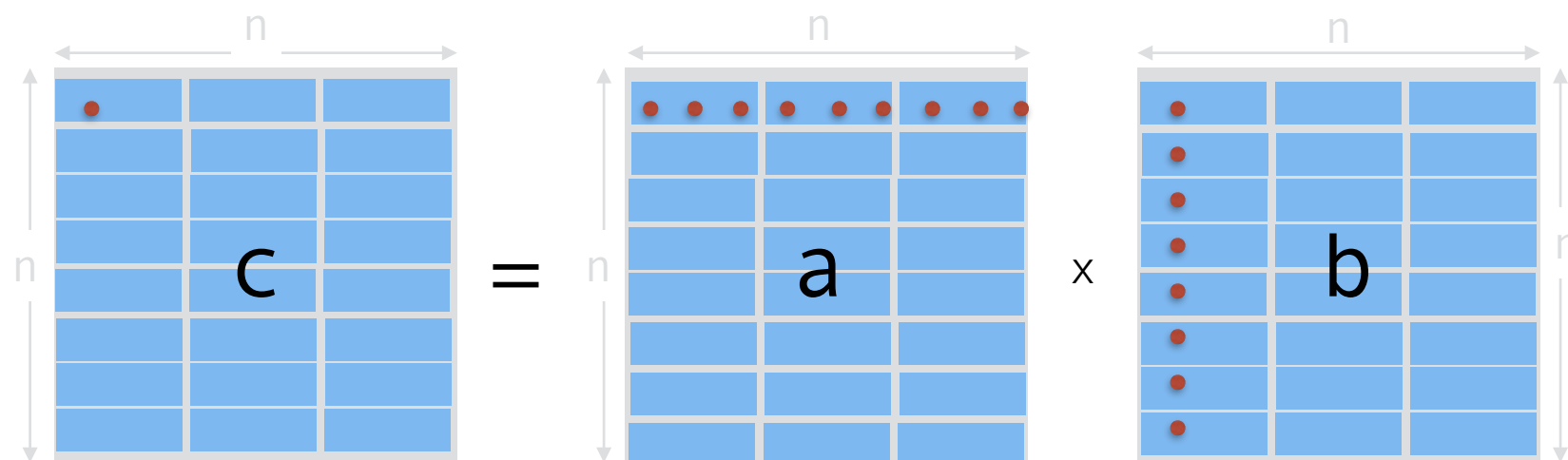
Cache-miss analysis

```
c = (double*) calloc(sizeof(double), n*n)
void mmult(double* a, double* b, double*c, int n) {
    int i, j, k;
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            for (k=0; k<n; k++)
                c[i*n+j] += a[i*n+k]*b[k*n+j]
}
```

Notation:

- n : array size
- M : cache size
- B : block size

Assume $B \ll n$

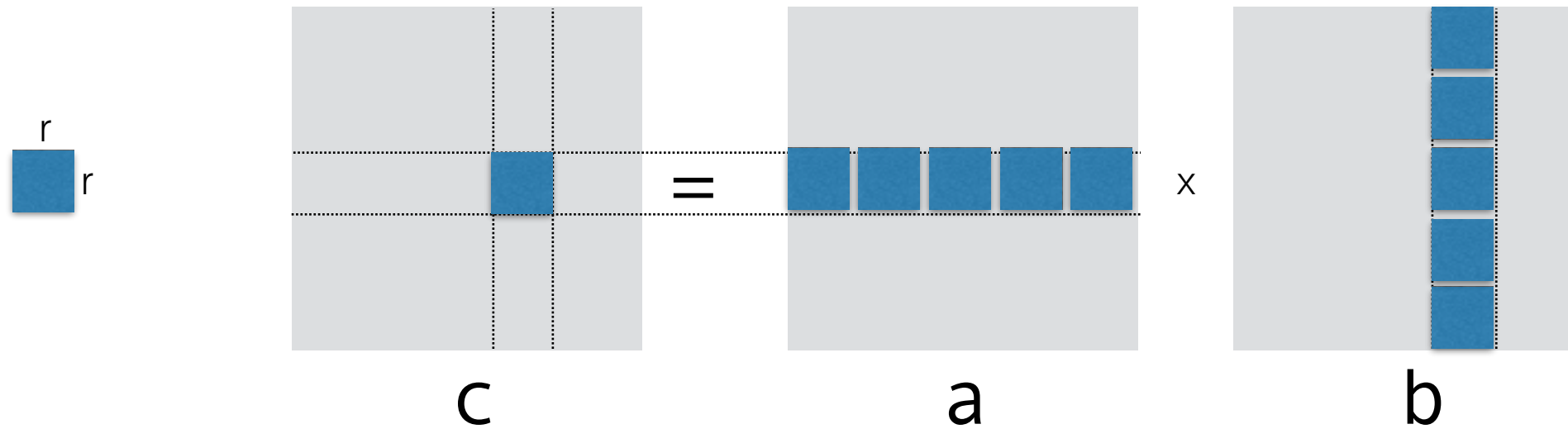


cache-misses: $O((n + n/B)n^2) = O(n^3)$

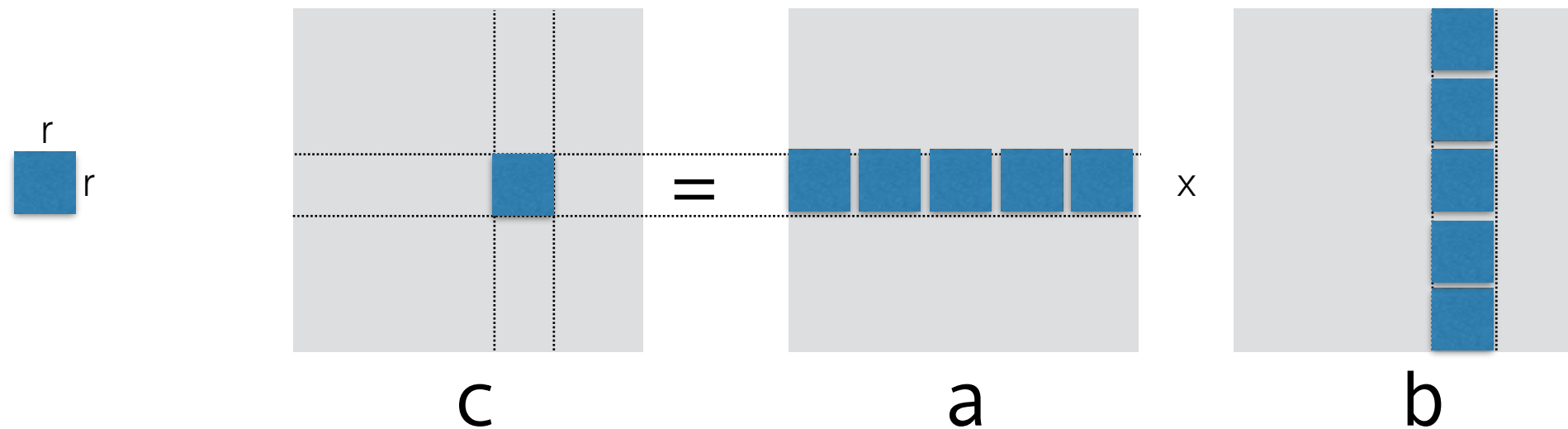
if $n < M/B$: b columns share same blocks $\Rightarrow O(n^3/B)$

Matrix multiplication:blocked

Do it block by block, instead of element by element.



Matrix multiplication:blocked



Partition the matrices in blocks of size r -by- r

Denote C_{ij} : the r -by- r block with upper-left corner (i,j)

Then $C_{ij} = A_{i0}B_{0j} + \dots$

Matrix multiplication:blocked

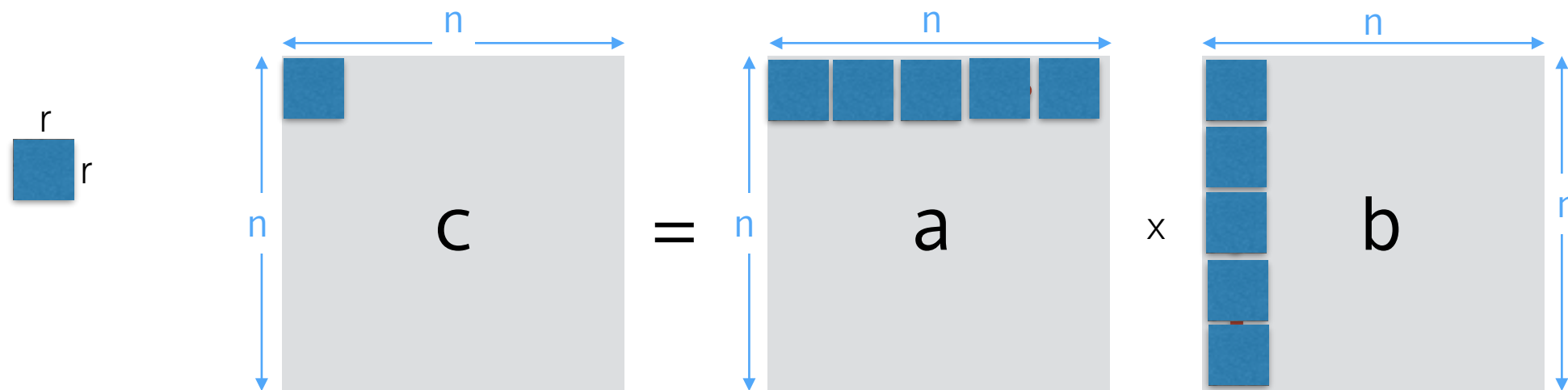
```
c = (double*) calloc(sizeof(double), n*n)

void mmult(double* a, double* b, double*c, int n) {
    int i, j, k;
    for (i=0; i<n; i+=r)
        for (j=0; j<n; j+=r)
            //compute block of c with upper left corner at(i,j)
            for (k=0; k<n; k+=r)
                //Cij += Aik*Bkj //<----- matrix multiplication
            }
}
```

Matrix multiplication:blocked

[illegible]

Cache-miss analysis



Notation:

- n : array size
- M : cache size
- B : block size

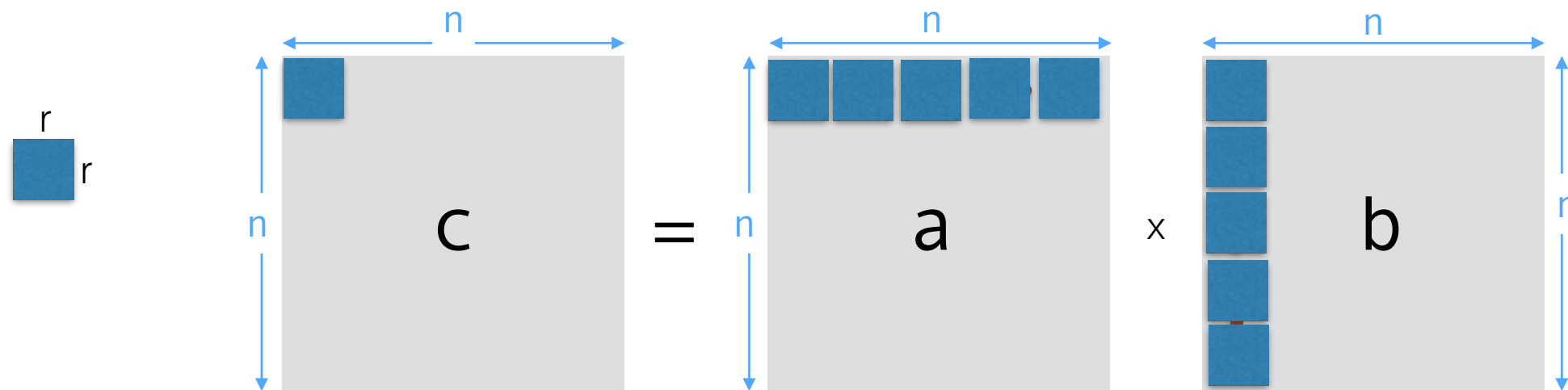
For each block of c :

read n/r blocks in a and n/r blocks in b

In total there are $(n/r)^2$ blocks in c

How to choose r ?

Cache-miss analysis



Notation:

- n : array size
- M : cache size
- B : block size

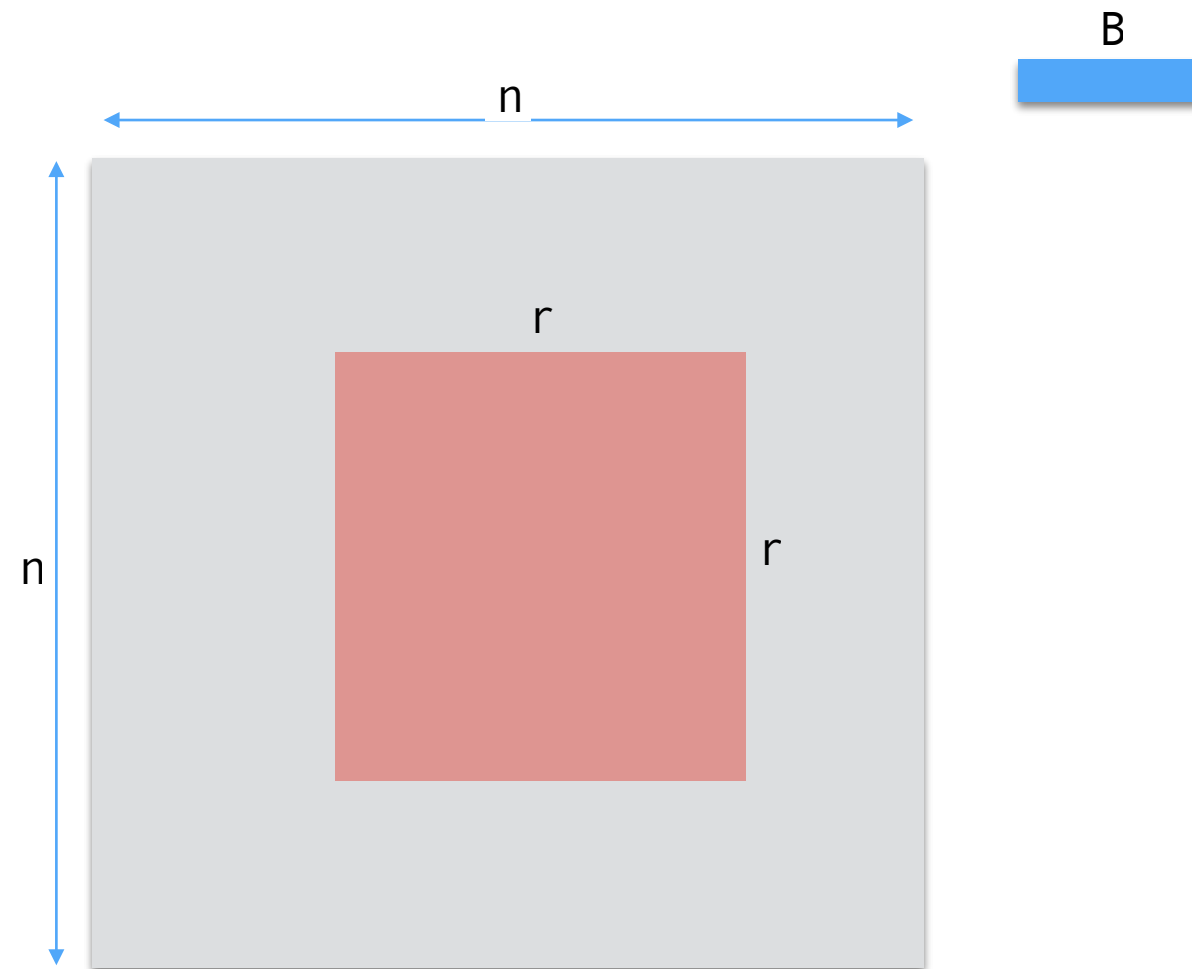
For each block of c :

read n/r blocks in a and n/r blocks in b

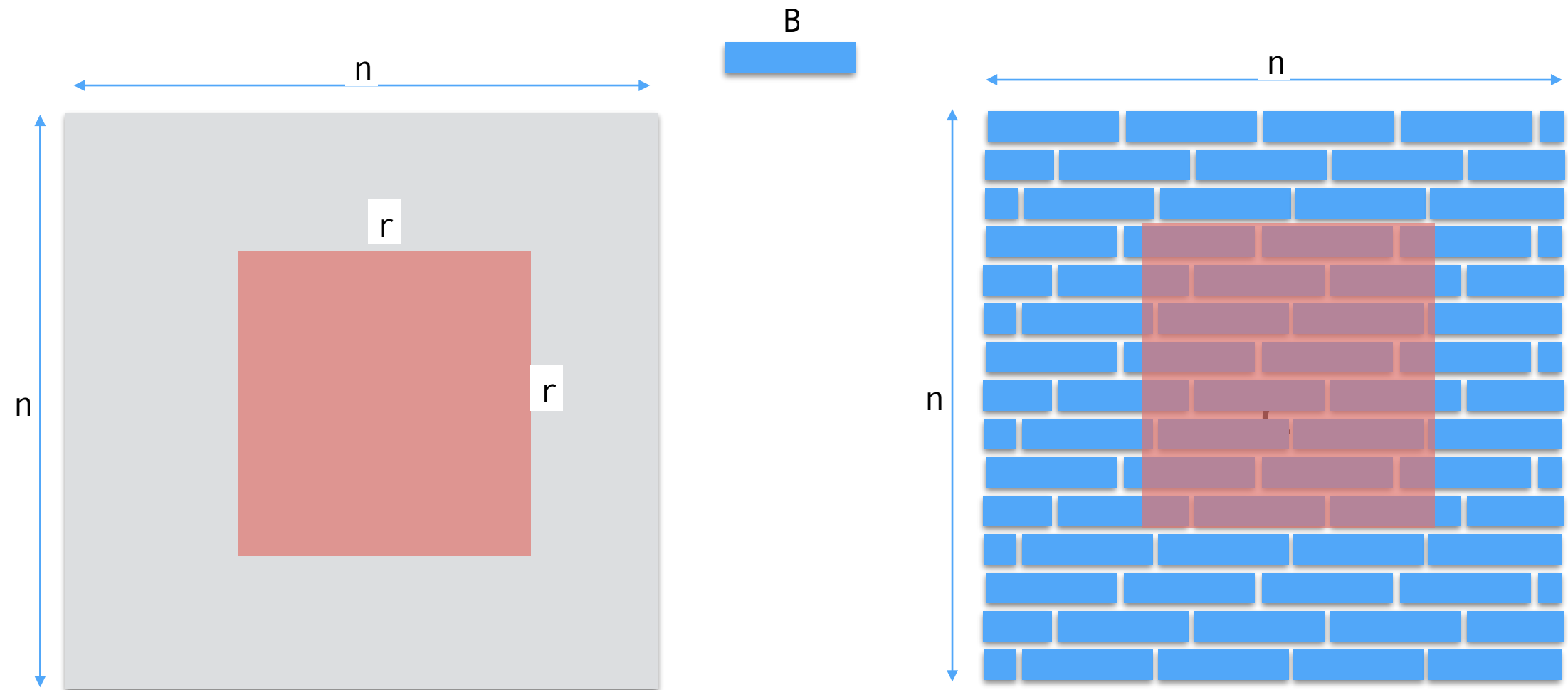
In total there are $(n/r)^2$ blocks in c

We'll choose the size of a block such that three blocks fit in cache: $3r^2 = M$, or $r = \Theta(\sqrt{M})$

How many cache misses to read a block of size r -by- r , in a matrix laid out in row-major order?

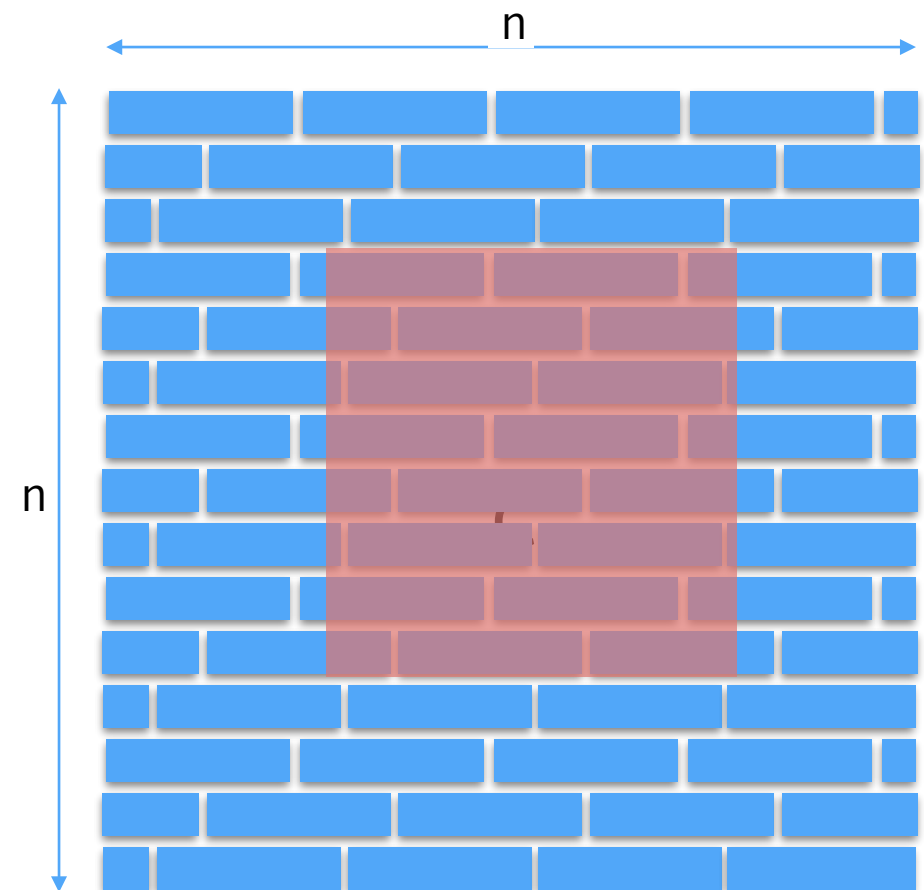
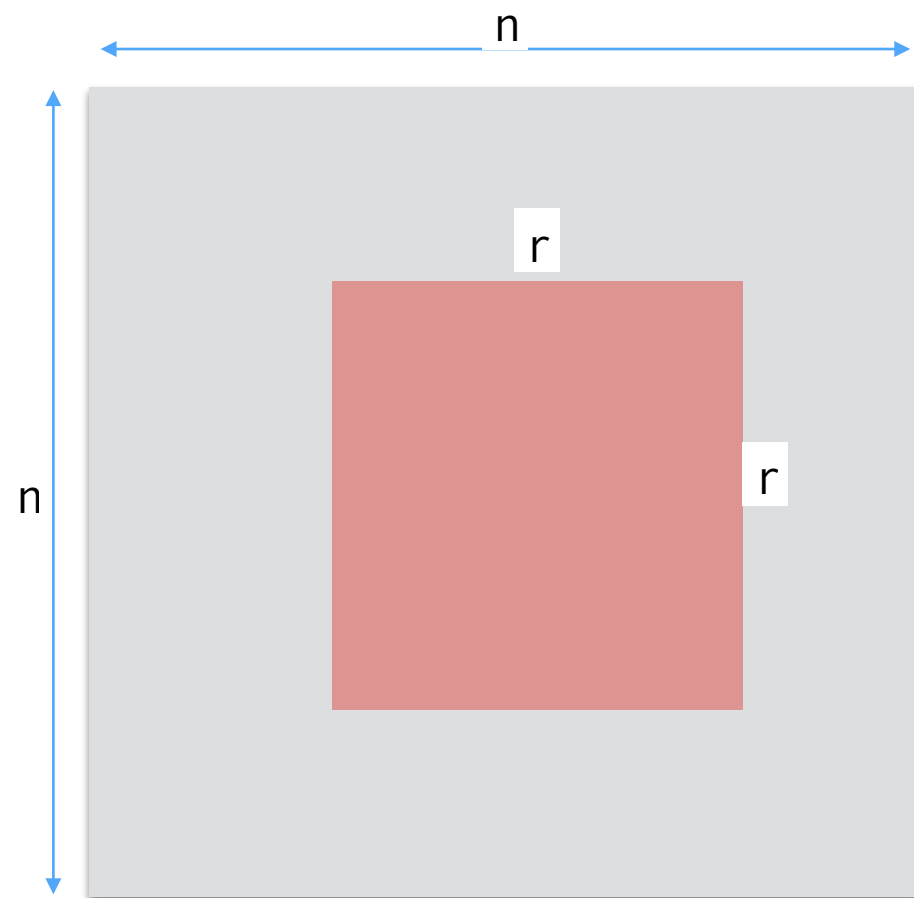


How many cache misses to read a block of size r -by- r , in a matrix laid out in row-major order?



How many blocks span a sub-matrix of size r -by- r , in a matrix laid out in row-major order?

How many cache misses to read a block of size r -by- r , in a matrix laid out in row-major order?



cache-misses: $O(r^2/B + r)$

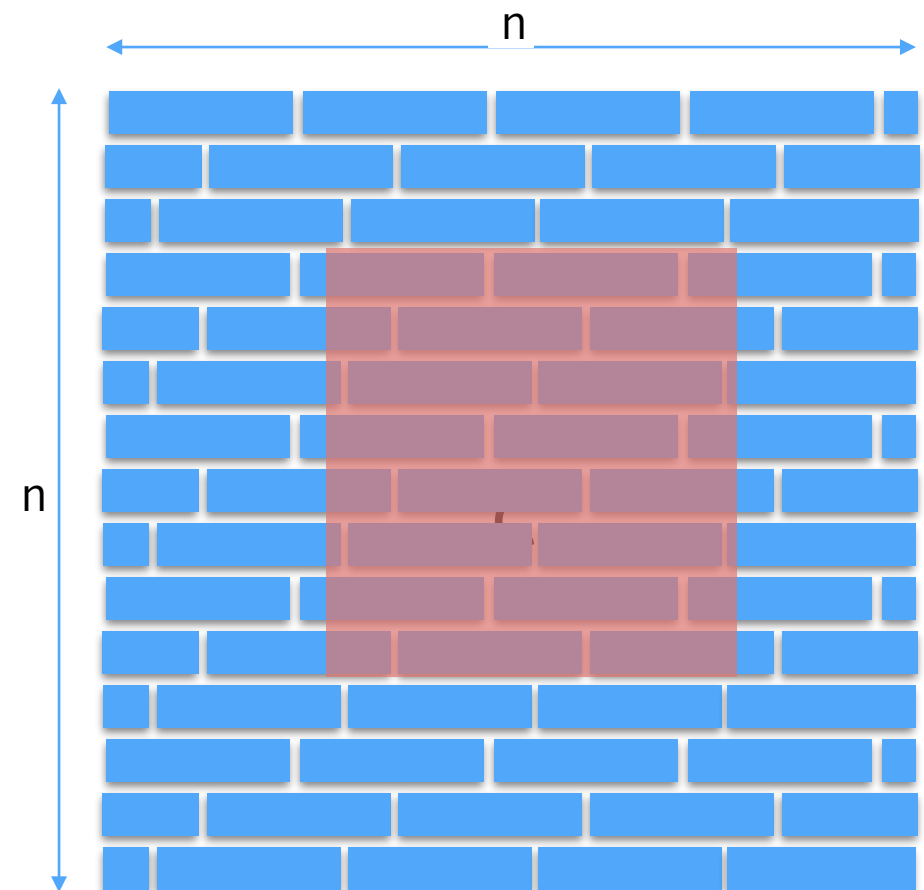
← this term is annoying but it's there

\Rightarrow Reading a block of size M takes $O(M/B + \sqrt{M})$

Note that $O(M/B + \sqrt{M})$ is $O(M/B)$ when $M > B^2$

tall cache assumption

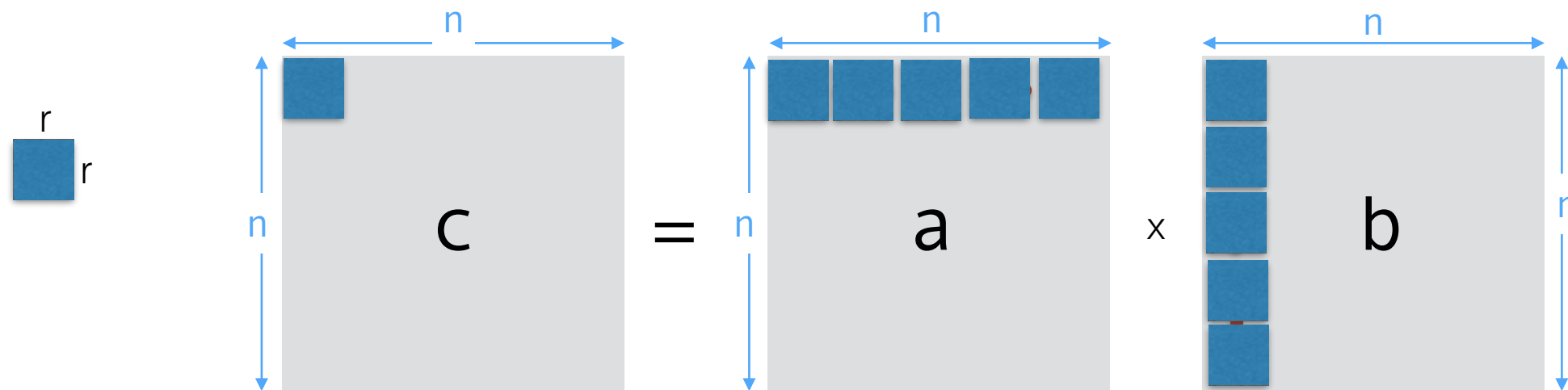
A cache is called **tall** if $M > B^2$.



Reading a block of size M in a matrix laid out in row-major order takes $O(M/B)$ cache misses with the tall cache assumption.

If the cache is not tall, it takes $O(\sqrt{M})$ cache misses.

Cache-miss analysis



Notation:

- n : array size
- M : cache size
- B : block size

Assume the size of a matrix block is chosen such that three blocks fit in cache: $3r^2 < M$, or $r = \sqrt{M}$

For each block of c :

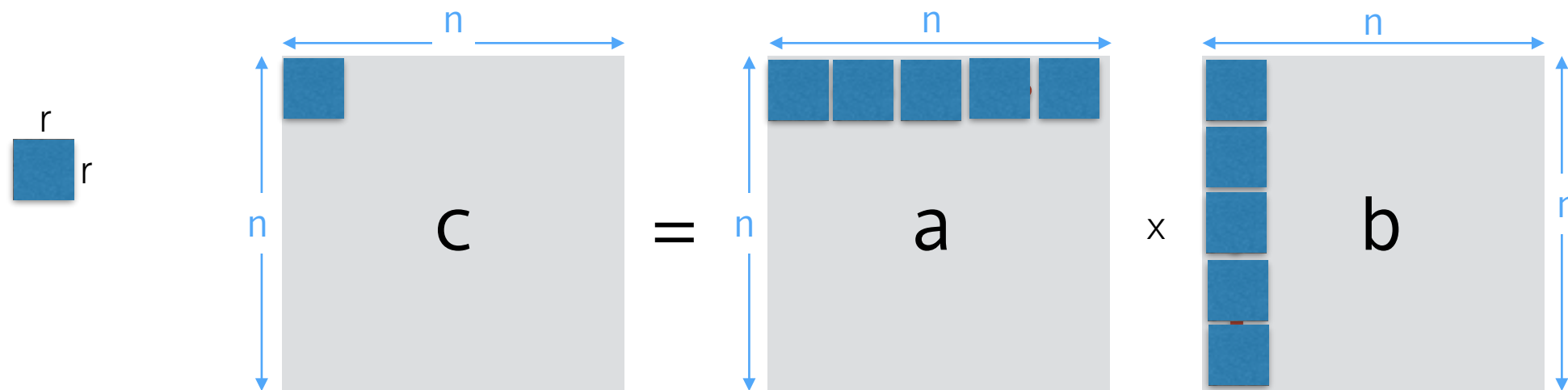
read n/r blocks in a and n/r blocks in b

In total there are $(n/r)^2$ blocks in c

A block has size $O(M)$ and reading it takes $O(M/B)$ misses with the tall cache assumption.

TOTAL: $(n/r)^2 \cdot 2n/r \cdot r^2/B$

Cache-miss analysis



Notation:

- n : array size
- M : cache size
- B : block size

TOTAL: $O(n^3 / (B \sqrt{M}))$

The point is, it's much better (although messy to analyze)

•

Summary

Notation:

- n : array size
- M : cache size
- B : block size

standard: $O(n^3)$

blocked: $O(n^3 / (B \sqrt{M}))$

Example:

cache block $B = 32$ bytes = 4 doubles

cache size $M = 768$ elements $\Rightarrow b = 16$ elements

standard: $O(n^3)$

blocked: $O(n^3/64)$ < — — — — — — — — 64 times fewer cache misses



standard: $O(n^3)$

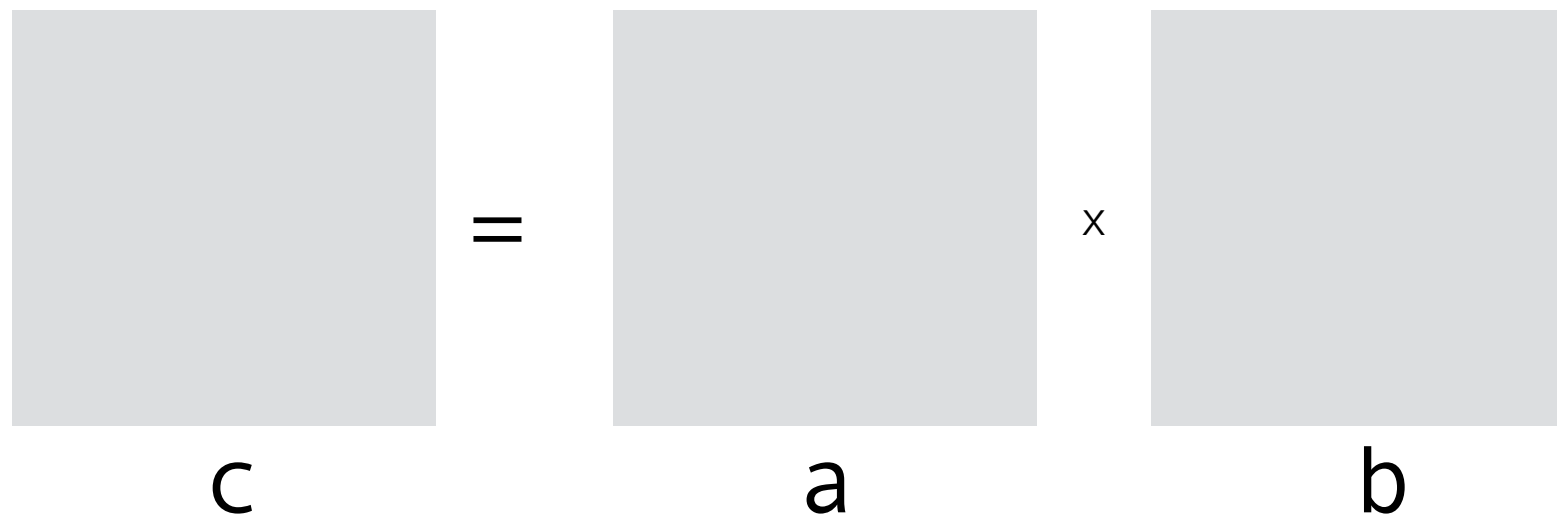
blocked: $O(n^3 / (B \sqrt{M}))$

Can we do better?

Matrix multiplication: divide & conquer

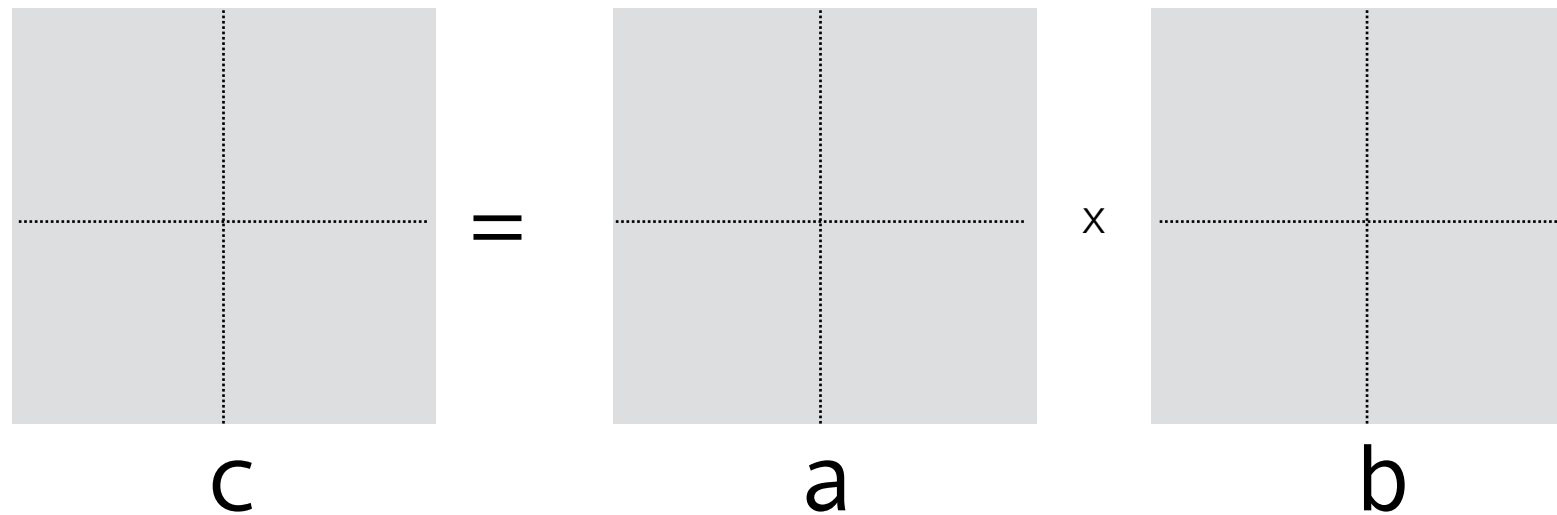
Matrix multiplication: divide & conquer

Assume n is a power of 2.



Matrix multiplication: divide & conquer

Assume n is a power of 2.



Matrix multiplication: divide & conquer

Assume n is a power of 2.

The diagram illustrates the divide-and-conquer matrix multiplication process. It shows a 2x2 matrix C equal to the product of two 2x2 matrices A and B . Each matrix is divided into four quadrants labeled 11, 12, 21, and 22.

$$\begin{bmatrix} 11 & 12 \\ 21 & 22 \end{bmatrix} = \begin{bmatrix} 11 & 12 \\ 21 & 22 \end{bmatrix} \times \begin{bmatrix} 11 & 12 \\ 21 & 22 \end{bmatrix}$$

$C \qquad \qquad \qquad A \qquad \qquad \qquad B$

Matrix multiplication: divide & conquer

```
c = (double*) calloc(sizeof(double), n*n)

void mmult(double* a, double* b, double*c, int n) {
    //base case
    if (n==1)
        c = a*b
        return
    else
        compute mmult(a11, b11, p1, n/2)
        compute mmult(a12, b21, p2, n/2)
        compute a11b12 ...
        compute a12b22
        compute a21b11
        compute a22b21
        compute a21b12
        compute a22b22
        add p1+p2 and put it in c11
        add p3+p4 and put it in c12
        ...
}
```

a11, a12, ... b11, b12, ... can be created
by copying from a and b

implement w/o copying?

can these sums be avoided?

8 recursive calls to multiply matrices of size $n/2$ -by- $n/2$

Matrix multiplication: divide & conquer

```
c = (double*) calloc(sizeof(double), n*n)

void mmult(double* a, double* b, double*c, int n) {
    //base case
    if (n==1)
        c = a*b
        return
    else
        compute mmult(a11, b11, p1, n/2)
        compute mmult(a12, b21, p2, n/2)
        compute a11b12 ...
        compute a12b22
        compute a21b11
        compute a22b21
        compute a21b12
        compute a22b22
        add p1+p2 and put it in c11
        add p3+p4 and put it in c12
        ...
}
```

CPU analysis

8 recursive calls to multiply matrices of size $n/2$ -by- $n/2$

Matrix multiplication: divide & conquer

```
c = (double*) calloc(sizeof(double), n*n)

void mmult(double* a, double* b, double*c, int n) {
    //base case
    if (n==1)
        c = a*b
        return
    else
        compute mmult(a11, b11, p1, n/2)
        compute mmult(a12, b21, p2, n/2)
        compute a11b12 ...
        compute a12b22
        compute a21b11
        compute a22b21
        compute a21b12
        compute a22b22
        add p1+p2 and put it in c11
        add p3+p4 and put it in c12
        ...
}
```

CPU analysis

- $T(n) = 2$ if $n=1$
- $T(n) = 8T(n/2) + n^2$ otherwise

solves to $O(n^3)$

8 recursive calls to multiply matrices of size $n/2$ -by- $n/2$

Matrix multiplication: divide & conquer

Notation:

- n : array size
- M : cache size
- B : block size

```
c = (double*) calloc(sizeof(double), n*n)

void mmult(double* a, double* b, double*c, int n) {
    //base case
    if (n==1)
        c = a*b
        return
    else
        compute mmult(a11, b11, p1, n/2)
        compute mmult(a12, b21, p2, n/2)
        compute a11b12 ...
        compute a12b22...
        compute a21b11...
        compute a22b21...
        compute a21b12...
        compute a22b22...
        add p1+p2 and put it in c11
        add p3+p4 and put it in c12
        ...
}
```

Cache miss analysis

- $C(n) = 8C(n/2) + O(n^2/B)$
- $C(n) = ??$ basecase

8 recursive calls to multiply matrices of size $n/2$ -by- $n/2$

Matrix multiplication: divide & conquer

Notation:

- n : array size
- M : cache size
- B : block size

Cache miss analysis:

- $C(n) = 8C(n/2) + O(n^2/B)$
- $C(n) = ??$ for basecase

What is a good base-case to analyze cache misses?

The recursive algorithm proceeds until matrix size = 1.

Let n' be the size at which a, b, c all fit in cache, ie $3 n'^2 = M$

From that point on, the algorithm keeps dividing and recursing. No matter in what order the elements of the matrices are brought in cache, once all blocks have been loaded, they stay in cache, since they fit in cache.

Reading a matrix of size $O(M)$ takes $O(M/B)$ with the tall cache assumption .

\Rightarrow Basecase: $C(\sqrt{M}) = O(M/B)$

This solves to $O(n^3/(B \sqrt{M}))$.

Matrix layout

- $a_{11}, a_{12}, \dots, b_{11}, b_{12}, \dots$ are **not** contiguous in a . Reading a matrix of size r -by- r takes $O(r^2/B + r)$ misses, which is $O(r^2/B)$ with the tall cache assumption. However the $O(r)$ term will add to the constant factors.
- Can this be avoided?

The diagram illustrates the matrix multiplication $C = A \times B$. Each matrix is represented as a 2x2 grid of elements, with the top row containing 11 and 12, and the bottom row containing 21 and 22. The matrices are labeled C , A , and B below them. The multiplication is shown as $C = A \times B$.

Matrix layout

- $a_{11}, a_{12}, \dots, b_{11}, b_{12}, \dots$ are **not** contiguous in a . Reading a matrix of size r -by- r takes $O(r^2/B + r)$ misses, which is $O(r^2/B)$ with the tall cache assumption. However the $O(r)$ term will add to the constant factors.
- Can this be avoided?

The diagram illustrates the matrix multiplication $C = A \times B$ using 2x2 block matrices. Each matrix is represented as a 2x2 grid of blocks, with the top-left block labeled 11, top-right 12, bottom-left 21, and bottom-right 22. The matrices are labeled c , a , and b below them. The equation is shown as $c = a \times b$.

Yes, with a different matrix layout!
More on this next time

Matrix multiplication: summary

Cache miss analysis:

standard:	$O(n^3)$
blocked:	$O(n^3 / (B \sqrt{M}))$
divide-and-conquer:	$O(n^3 / (B \sqrt{M}))$

Notation:

- n : array size
- M : cache size
- B : block size

Matrix multiplication: summary

blocked

```
void mmult(double*a, double*b, double*c, int n) {  
  
    int i, j, k;  
    for (i=0; i<n; i+=r)  
        for (j=0; j<n; j+=r)  
            //compute block of c with upper  
            //left corner at(i,j)  
            for (k=0; k<n; k+=r)  
                //Cij += Aik*Bkj  
}
```

$O(n^3 / (B \sqrt{M}))$

divide-and-conquer

```
void mmult(double* a, double* b, double*c, int n) {  
    //base case  
    if (n==1)  
        c = a*b  
        return  
    else  
        compute mmult(a11, b11, p1, n/2)  
        compute mmult(a12, b21, p2, n/2)  
        compute mmult(a11, b12, p3, n/2)  
        compute a12b22..  
        compute a21b11..  
        compute a22b21..  
        compute a21b12..  
        compute a22b22..  
        add p1+p2 and put it in c11  
        add p3+p4 and put it in c12  
        ...  
}
```

$O(n^3 / (B \sqrt{M}))$

Matrix multiplication: summary

blocked

```
void mmult(double*a, double*b, double*c, int n) {  
  
    int i, j, k;  
    for (i=0; i<n; i+=r)  
        for (j=0; j<n; j+=r)  
            //compute block of c with upper  
            //left corner at(i,j)  
            for (k=0; k<n; k+=r)  
                //Cij += Aik*Bkj  
}
```

$O(n^3 / (B \sqrt{M}))$

cache-aware

divide-and-conquer

```
void mmult(double* a, double* b, double*c, int n) {  
    //base case  
    if (n==1)  
        c = a*b  
        return  
    else  
        compute mmult(a11, b11, p1, n/2)  
        compute mmult(a12, b21, p2, n/2)  
        compute mmult(a11, b12, p3, n/2)  
        compute a12b22..  
        compute a21b11..  
        compute a22b21..  
        compute a21b12..  
        compute a22b22..  
        add p1+p2 and put it in c11  
        add p3+p4 and put it in c12  
        ...  
}
```

$O(n^3 / (B \sqrt{M}))$

cache-oblivious

Aware or oblivious?

Notation:

- n : array size
- M : cache size
- B : block size

Cache-aware algorithms

- The algorithm needs to know M and B

Cache-oblivious algorithms

- We use M and B to analyze it, but the algorithm does not need knowledge of M and B

Aware or oblivious?

Notation:

- n : array size
- M : cache size
- B : block size

Cache-aware algorithms

- The algorithm needs to know M and B
- Parameters need to be tuned

Cache-oblivious algorithms

- We use M and B to analyze it, but the algorithm does not need knowledge of M and B
- They are efficient at all levels of the memory hierarchy, simultaneously
- Elegant !

Project1

- Implement the three algorithms for matrix multiplication
 - 1. standard
 - 2. blocked
 - 3. divide-and-conquer
- For each algorithm, plot running time function of problem size n
- For the blocked algorithm, pick a large n and plot running time function of matrix block size r ; choose the r that optimizes running time
- Compare the algorithms and discuss your findings

Will be in GitHub soon!

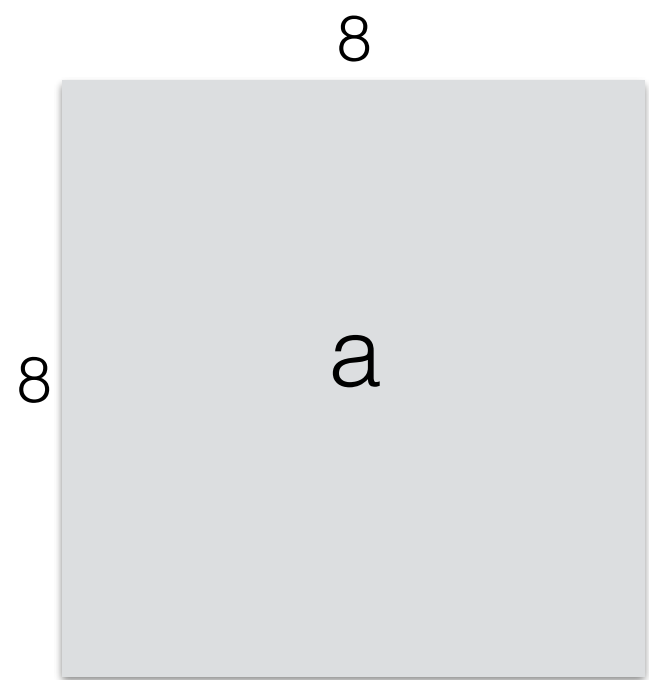
Resources

- Cache-oblivious algorithms were introduced by

Matteo Frigo, Charles E. Leiserson, Harald Prokop, Sridhar Ramachandran, "Cache-Oblivious Algorithms", FOCS'99 (pdf)

- Here are some links that may be helpful
 - Charles Leiserson, MIT, lecture on cache-efficient algorithms (part of course Performance engineering of software systems). <https://www.youtube.com/watch?v=T9LkSKK075M>
 - Erik Demaine, MIT, lecture on cache-oblivious median finding and matrix multiplication. It's long (1.5hr) but has all details in the analysis. <https://www.youtube.com/watch?v=CSqbjfCCLrU>
 - cache-oblivious matrix multiply, part of Udacity course High-performance computing, 3min video, great for getting the essential, but misses the subtleties of the analysis, like the necessity of the tall-cache assumption. - (<https://www.youtube.com/watch?v=vxkZkcbwU40>)

Matrix layout

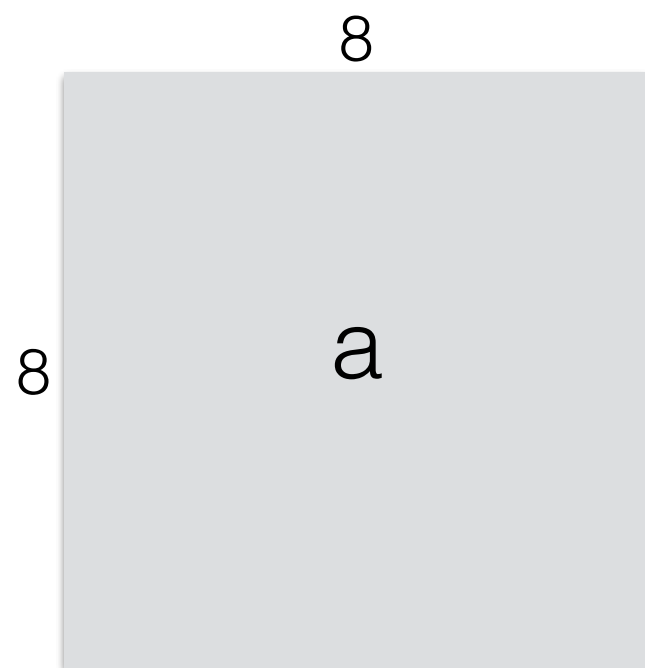


row-major order

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Matrix layout



row-major order

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Show where the elements of a_{11} reside in this array

Matrix multiplication with blocked layout

```
c = (double*) calloc(sizeof(double), n*n)

void mmult(double* a, double* b, double*c, int n) {
    //base case
    if (n==1)
        c = a*b
        return
    else
        compute mmult(a11, b11, p1, n/2)
        compute mmult(a12, b21, p2, n/2)
        compute a11b12 ...
        compute a12b22
        compute a21b11
        compute a22b21
        compute a21b12
        compute a22b22
        add p1+p2 and put it in c11
        add p3+p4 and put it in c12
        ...
}
```

a11, a12, ... b11, b12, ... have to be created
by copying from a and b

c11, c12, have to be copied into c

implement w/o copying??

8 recursive calls to multiply matrices of size $n/2$ -by- $n/2$