



CS 240

Foundations of Computer Systems



Memory Hierarchy and Cache

- Memory hierarchy
- Cache basics
- Locality
- Cache organization
- Cache-aware programming

Big picture: memory layout impacts program performance in sometimes surprising ways!

<https://cs.wellesley.edu/~cs240/>

1

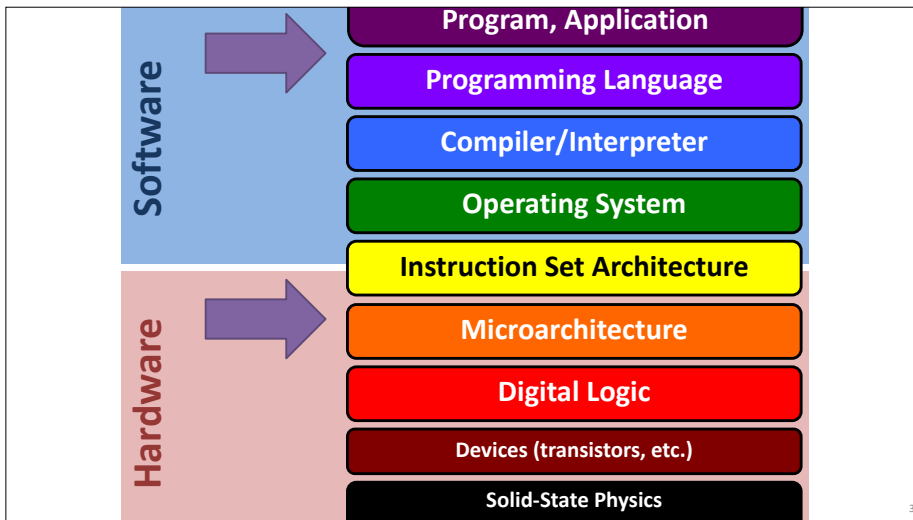
When someone says "clear your browser's cache", what does that do?

0

Nobody has responded yet.

Hang tight! Responses are coming in.

Start the presentation to see live content. For screen share software, share the entire screen. Get help at polllev.com/app



Examples: Nvidia documentation

"CUDA, a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU."

- Nvidia CUDA C++ Programming Guide

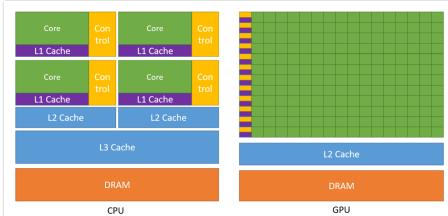


Figure 1: The GPU Devotes More Transistors to Data Processing

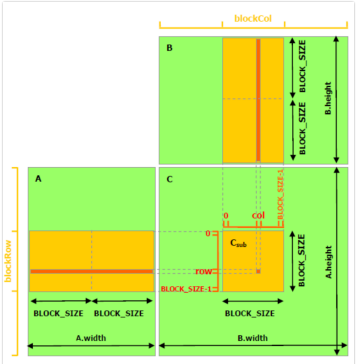


Figure 9: Matrix Multiplication with Shared Memory

What are these caches? Why the complex matrix multiply?

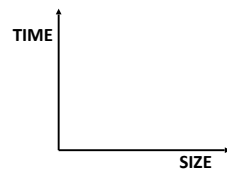
<https://docs.nvidia.com/cuda/cuda-c-programming-guide>

4

Expectation: How does execution time grow with SIZE?

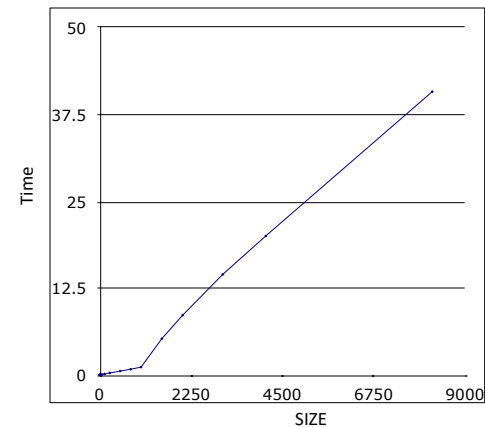
```
int array[SIZE];
fillArrayRandomly(array);
int s = 0;

for (int i = 0; i < 200000; i++) {
    for (int j = 0; j < SIZE; j++) {
        s += array[j];
    }
}
```



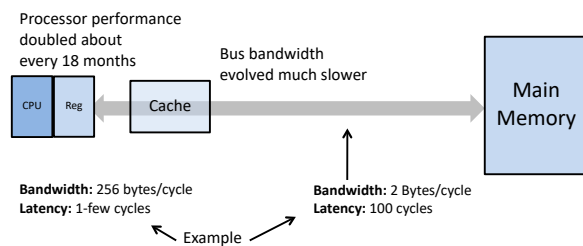
5

Reality



6

Processor-memory bottleneck



Solution: caches

7

Cache

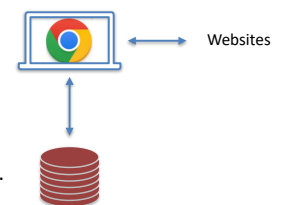
English:

- n.** a hidden storage space for provisions, weapons, or treasures
- v.** to store away in hiding for future use

Computer Science:

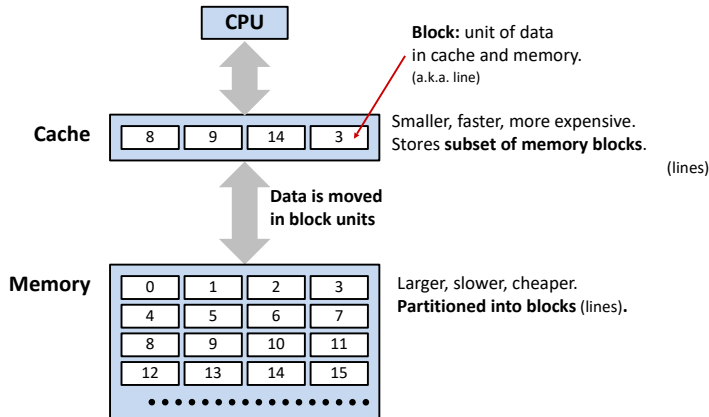
- n.** a computer memory with short access time used to store frequently or recently used instructions or data
- v.** to store [data/instructions] temporarily for later quick retrieval

Also used more broadly in CS: software caches, file caches, etc.



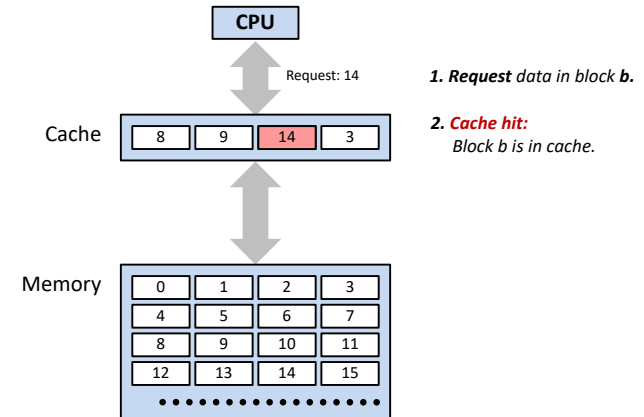
8

General cache mechanics



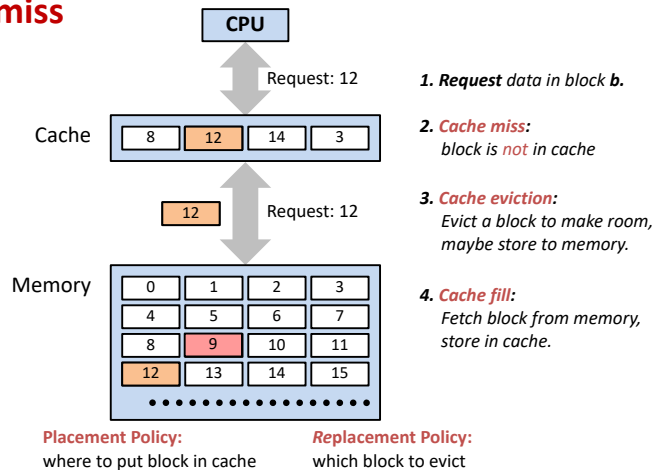
9

Cache hit



10

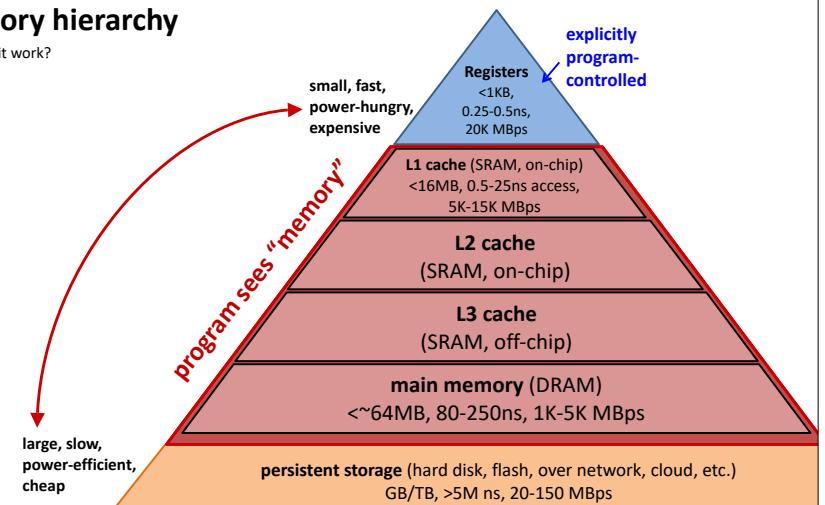
Cache miss



11

Memory hierarchy

Why does it work?



Locality: why caches work

Programs tend to use data and instructions at addresses near or equal to those they have used recently.

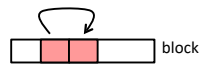
Temporal locality:

Recently referenced items are *likely* to be referenced again in the near future.



Spatial locality:

Items with nearby addresses are *likely* to be referenced close together in time.



How do caches exploit temporal and spatial locality?

13

In terms accessing the input array, does this function have temporal or spatial locality?

0

```
int sum(int *arr) {  
    int s = 0;  
    while (*arr != -1) {  
        s += *arr;  
        arr++;  
    }  
}
```

Temporal

Spatial

None of the above

Start the presentation to see live content. For screen share software, share the entire screen. Get help at polllev.com/app

Locality #1: Basic iteration over array

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

What is stored in memory?

15

Locality #2: iteration over 2D array

row-major M x N 2D array in C

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

ex

16

Locality #3: iteration over 2D array

ex

row-major M x N 2D array in C

Swapped
loop order

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

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3] ...
a[2][0]	a[2][1]	a[2][2]	a[2][3]
...			

17

Locality #4

What is "wrong" with this code?

How can it be fixed?

```
int sum_array_3d(int a[M][N][N]) {
    int sum = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            for (int k = 0; k < M; k++) {
                sum += a[k][i][j];
            }
        }
    }
    return sum;
}
```

18

Cost of cache misses

Miss cost could be $100 \times$ hit cost.

99% hits could be **twice** as good as 97%. How?

Assume cache hit time of 1 cycle, miss penalty of 100 cycles

Mean access time:

97% hits: $(0.97 * 1 \text{ cycle}) + (0.03 * 100 \text{ cycles}) = 3.97 \text{ cycles}$

99% hits: $(0.99 * 1 \text{ cycle}) + (0.01 * 100 \text{ cycles}) = 1.99 \text{ cycles}$

hit/miss rates

19

Cache performance metrics

Miss Rate

Fraction of memory accesses to data not in cache (misses / accesses)

Typically: 3% - 10% for L1; maybe < 1% for L2, depending on size, etc.

Hit Time

Time to find and deliver a block in the cache to the processor.

Typically: 1 - 2 clock cycles for L1; 5 - 20 clock cycles for L2

Miss Penalty

Additional time required on cache miss = main memory access time

Typically 50 - 200 cycles for L2 (trend: increasing!)

20

Cache organization

Block

Fixed-size unit of data in memory/cache

Placement Policy

Where in the cache should a given block be stored?

Replacement Policy

What if there is no room in the cache for requested data?

- least recently used, most recently used

Write Policy

When should writes update lower levels of memory hierarchy?

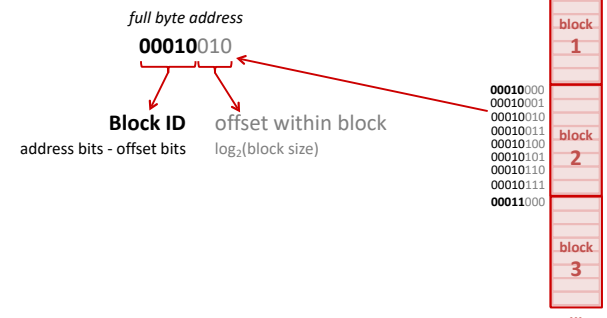
- write back, write through, write allocate, no write allocate

21

Blocks

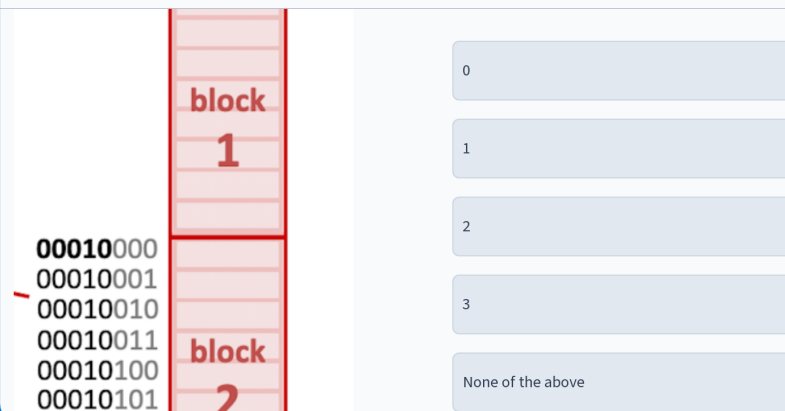
Divide address space into fixed-size aligned blocks.
power of 2

Example: block size = 8



22

With a block size of 8 and assuming the cache starts empty, how many cache *misses* would the following sequence of accesses have? 00010010 00010011 00000001



Start the presentation to see live content. For screen share software, share the entire screen. Get help at polllev.com/app

Cache size puzzle

Cache starts *empty*.

Access (address, hit/miss) stream:

(0xA, miss), (0xB, hit), (0xC, miss)

What could the block size be?

ex

24

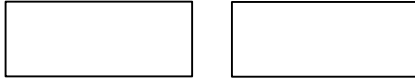
Cache size puzzle

ex

Cache starts *empty*.

Access (address, hit/miss) stream:

(0xA, miss), (0xB, hit), (0xC, miss)



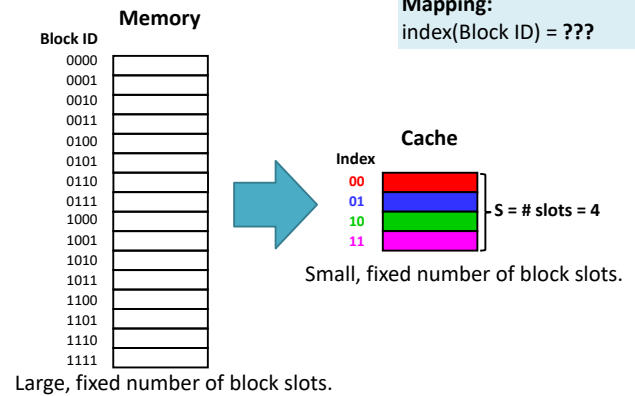
What could the block size be?

1. First, convert the hex to integers
2. Remember that blocks must be aligned to the block size
3. Hint: there are two possible block sizes!

25

Placement policy

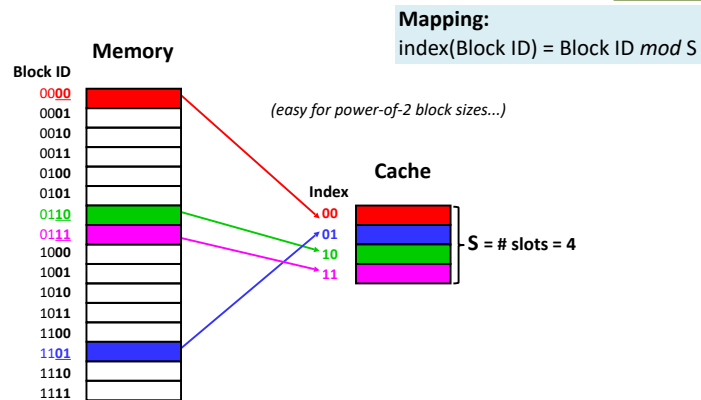
optional



26

Placement: *direct-mapped*

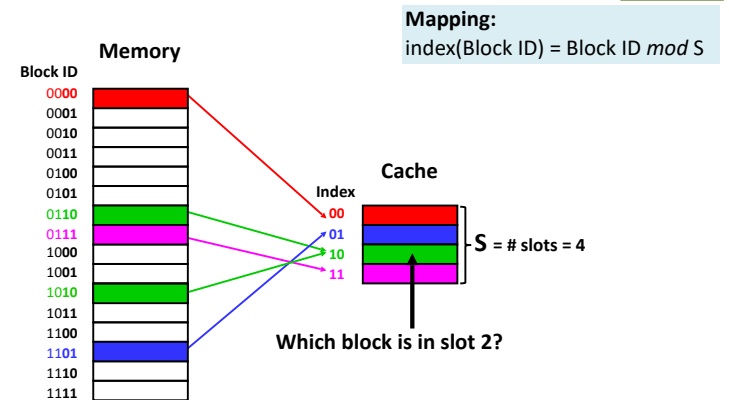
optional



27

Placement: mapping ambiguity?

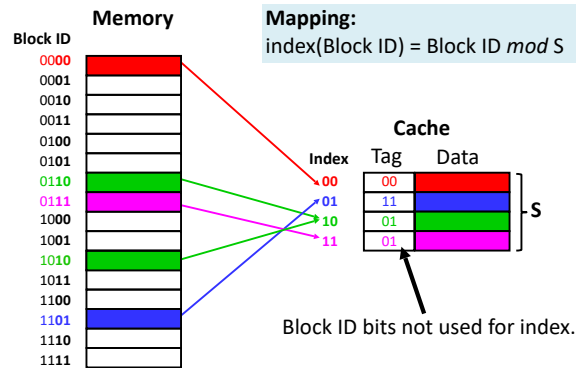
optional



28

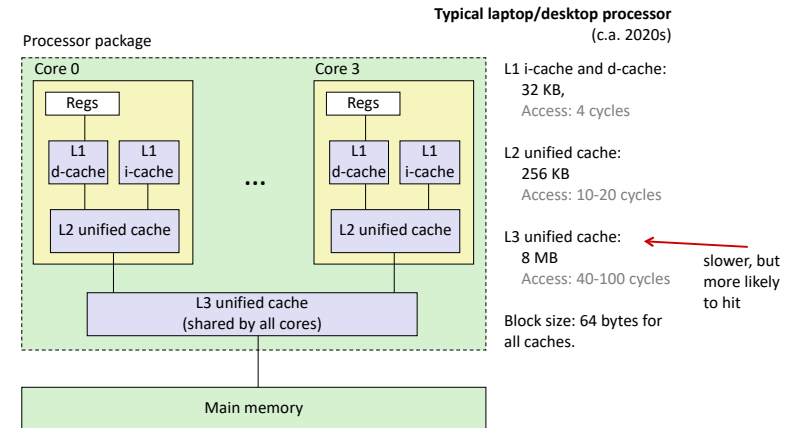
Placement: tags resolve ambiguity

optional



29

Example memory hierarchy



30

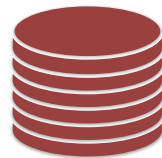
Software caches

Examples

File system buffer caches, web browser caches, database caches, network CDN caches, etc.

Some design differences

Often use complex replacement policies
 Not necessarily constrained to single "block" transfers



31

Cache-friendly code

Locality, locality, locality.

Programmer can optimize for cache performance

Data structure layout

Data access patterns

Nested loops

Blocking / tiling

All systems favor "cache-friendly code"

Performance is hardware-specific

Generic rules capture most advantages

Keep working set small (temporal locality)

Use small strides (spatial locality)

Focus on *inner loop* code



32

Example: Matrix Multiplication

optional

```
c = (double *) calloc(sizeof(double), n*n);
/* Multiply n x n matrices a and b */
void mmm(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];
}
```

$$(AB)_{ij} = \sum_{k=1}^m A_{ik} B_{kj}.$$

memory access pattern?

33

Cache Miss Analysis

optional

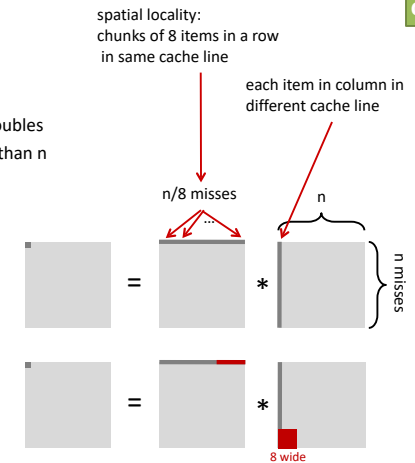
Assume:

- Matrix elements are doubles
- Cache block = 64 bytes = 8 doubles
- Cache size C is much smaller than n

First iteration:

$n/8 + n = 9n/8$ misses
(omitting matrix c)

Afterwards **in cache**:
(schematic)



34

Cache Miss Analysis

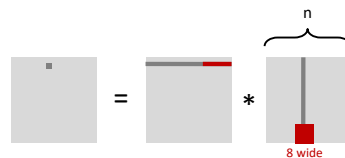
optional

Assume:

- Matrix elements are doubles
- Cache block = 64 bytes = 8 doubles
- Cache size C is much smaller than n

Other iterations:

Again:
 $n/8 + n = 9n/8$ misses
(omitting matrix c)



Total misses:

$$9n/8 * n^2 = (9/8) * n^3$$

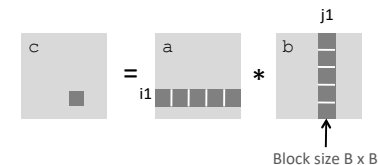
once per element

35

Blocked Matrix Multiplication

optional

```
c = (double *) calloc(sizeof(double), n*n);
/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int 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*n + j1] += a[i1*n + k1]*b[k1*n + j1];
}
```



36

Cache Miss Analysis

optional

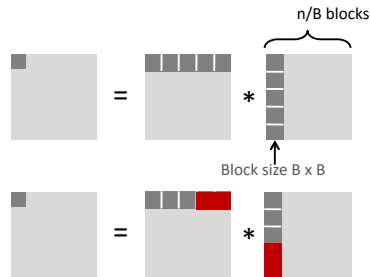
Assume:

Cache block = 64 bytes = 8 doubles

Cache size $C \ll n$ (much smaller than n)

Three blocks fit into cache: $3B^2 < C$

First (block) iteration:
 $B^2/8$ misses for each block
 $2n/B * B^2/8 = nB/4$
 (omitting matrix c)



37

Cache Miss Analysis

optional

Assume:

Cache block = 64 bytes = 8 doubles

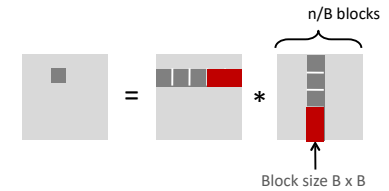
Cache size $C \ll n$ (much smaller than n)

Three blocks fit into cache: $3B^2 < C$

Other (block) iterations:

Same as first iteration

$2n/B * B^2/8 = nB/4$



Total misses:

$nB/4 * (n/B)^2 = n^3/(4B)$

38

Summary: blocking/tiling

No blocking: $(9/8) * n^3 \approx 1.1n^3$

Blocking: $1/(4B) * n^3 \approx 0.1-0.3 n^3$

If $B = 8$ difference is $4 * 8 * 9 / 8 = 36x$

If $B = 16$ difference is $4 * 16 * 9 / 8 = 72x$

Reason for dramatic difference:

Matrix multiplication has inherent temporal locality:

Input data: $3n^2$, computation $2n^3$

Every array element used $O(n)$ times!

But program has to be written properly

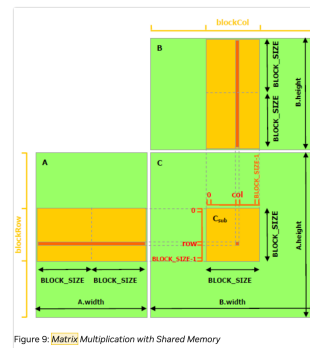


Figure 9: Matrix Multiplication with Shared Memory

39

Exercise: order these 3 functions by locality

ex

```
typedef struct {
    int vel[3];
    int acc[3];
} point;
```

```
#define N 100
point p[N];
```

```
void clear1(point *p, int n) {
    int i, j;
    for (i=0; i<n; i++){
        for (j=0; j<3; j++){
            p[i].vel[j] = 0;
            p[i].acc[j] = 0;
        }
    }
}

void clear2(point *p, int n) {
    int i, j;
    for (i=0; i<n; i++){
        for (j=0; j<3; j++){
            p[i].vel[j] = 0;
            for (j=0; j<3; j++){
                p[i].acc[j] = 0;
            }
        }
    }
}

void clear3(point *p, int n) {
    int i, j;
    for (j=0; j<3; j++){
        for (i=0; i<n; i++){
            p[i].vel[j] = 0;
            for (i=0; i<n; i++){
                p[i].acc[j] = 0;
            }
        }
    }
}
```

40

Modern machine learning/scientific computing frameworks leverage this!
 Including blocking/tiling on GPUs. Often, with specific matrix multiply units.