Announcements

- The class capacity has been increased to 100.
- Based on preliminary results from the survey, Abhishek’s office hours will be **in person on Mondays 2:30-3:30p in Klaus (KACB) 3121**. To get in, the passcode is 421, then press enter. The other office hours will be online in the same zoom room as the classroom at their original times.
- Survey due tomorrow by ~5pm
- HW1 is out - due Monday, Jan 22 by ~5pm
- Pre-proposal is due Thursday, Jan 18 by ~5pm

Superlinear speedup in practice

Amdahl's law tells us that the maximum theoretical speedup we can get on P processors is P.

In reality, there cases when we might achieve **more than P speedup**.

One possible reason is that **more cache memory** is available when running on multiple processors.

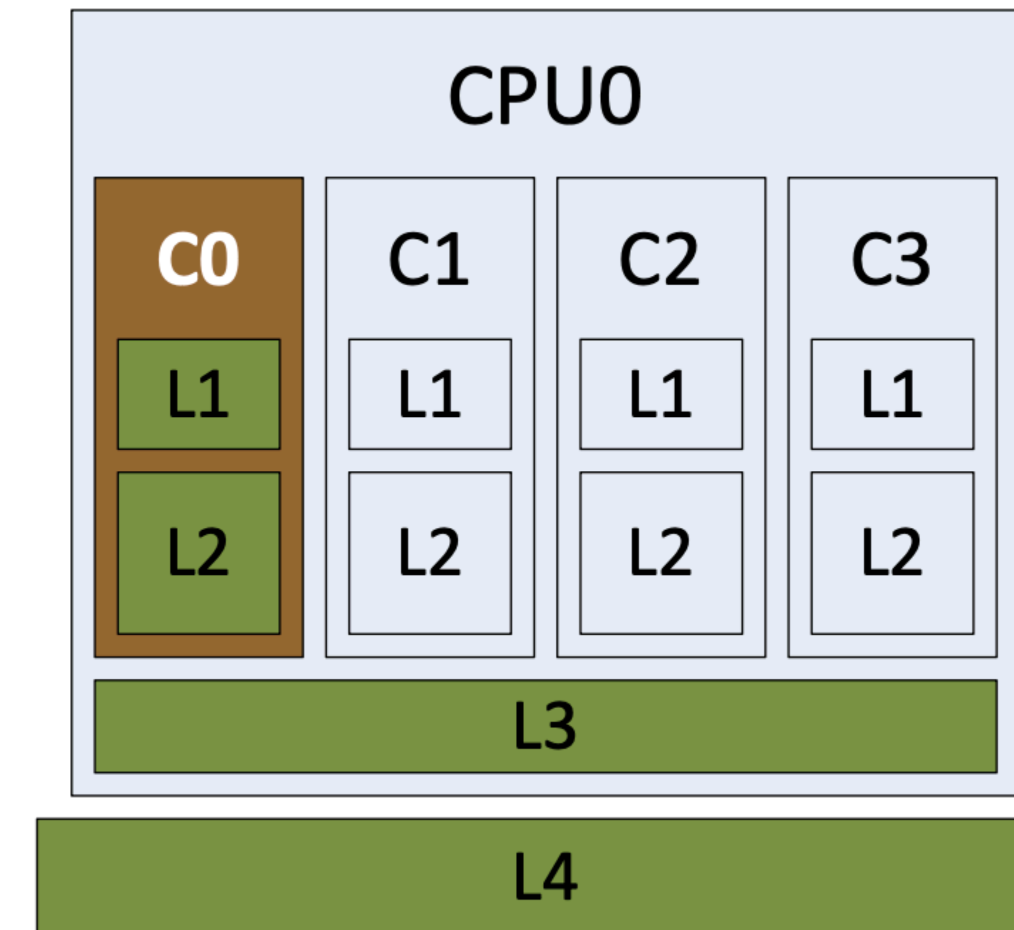


Fig. 3. Utilized memory for sequential execution

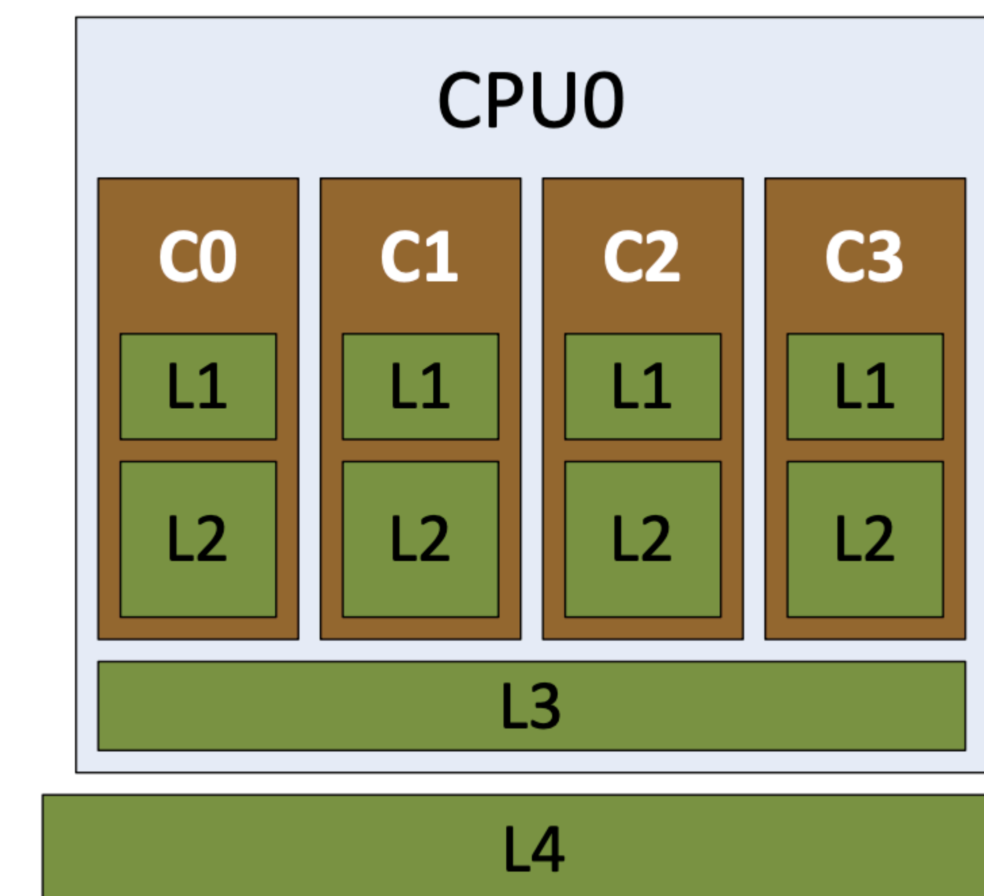
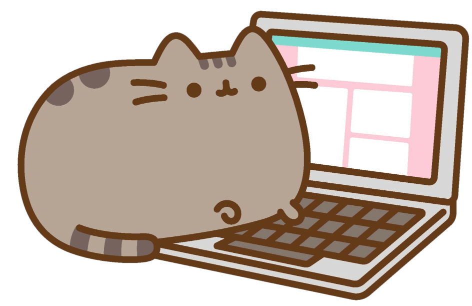


Fig. 4. Utilized multi tiered memory for loosely coupled processors for parallel execution

From "Superlinear Speedup in HPC Systems: why and when?" - Ristov et al. 2016

CSE 6230:
HPC Tools and Applications

+



Lecture 2: Memory Hierarchies and Matrix Multiplication

Helen Xu
hxu615@gatech.edu

Georgia Tech College of Computing
School of Computational Science and Engineering

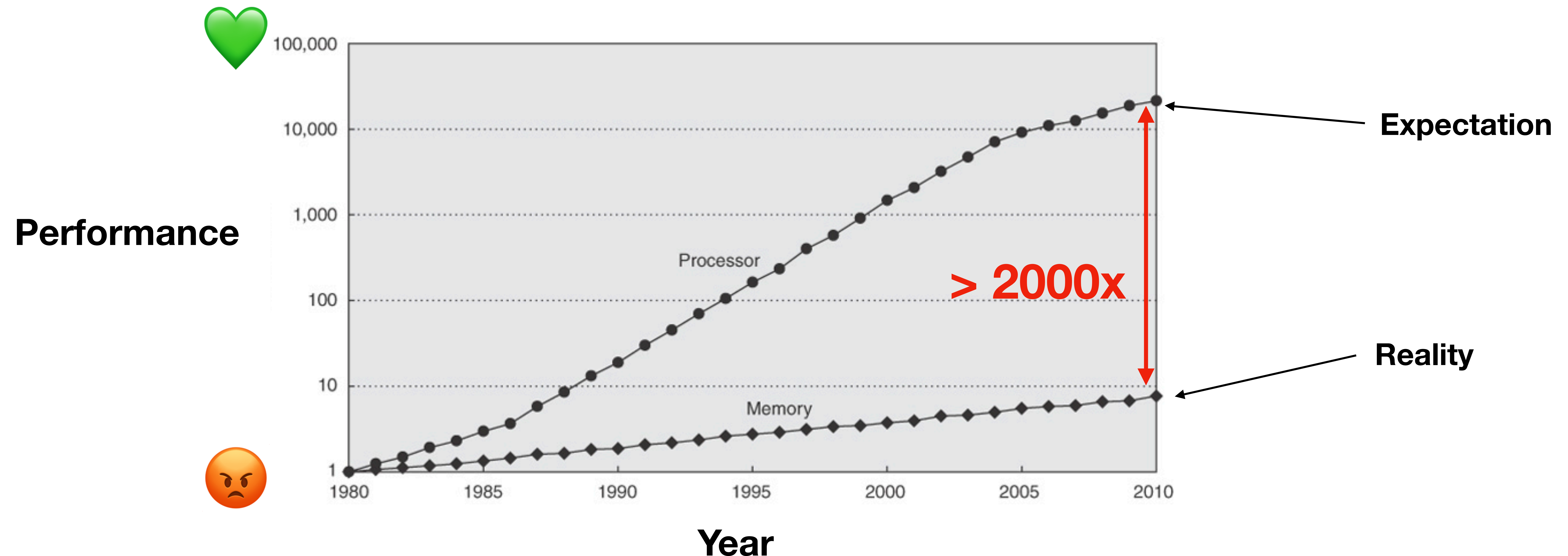
(Some slides from MIT's OCW 6.172, UC Berkeley CS267)

Let’s start with single processors - why?

Memory limitations affect overall performance

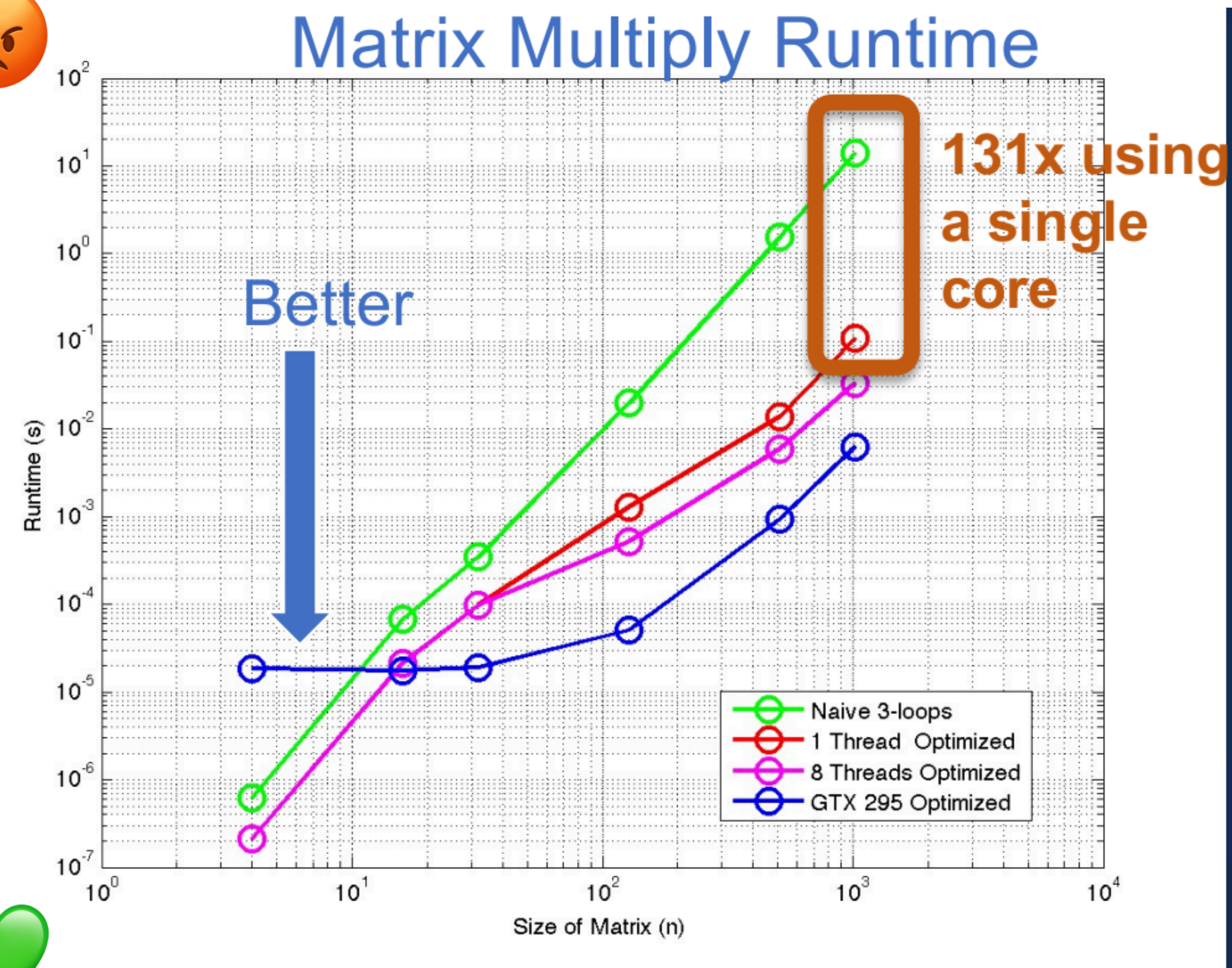
Most applications run at **<10%** of “peak” performance (max possible rate of flops).

Much of the performance is lost on a single processor due to **data movement**.



From UC Berkeley CS267

Why should we care about serial performance in a course on high-performance computing?



Starting with a fast serial implementation is an **important first step** towards a fast parallel implementation.

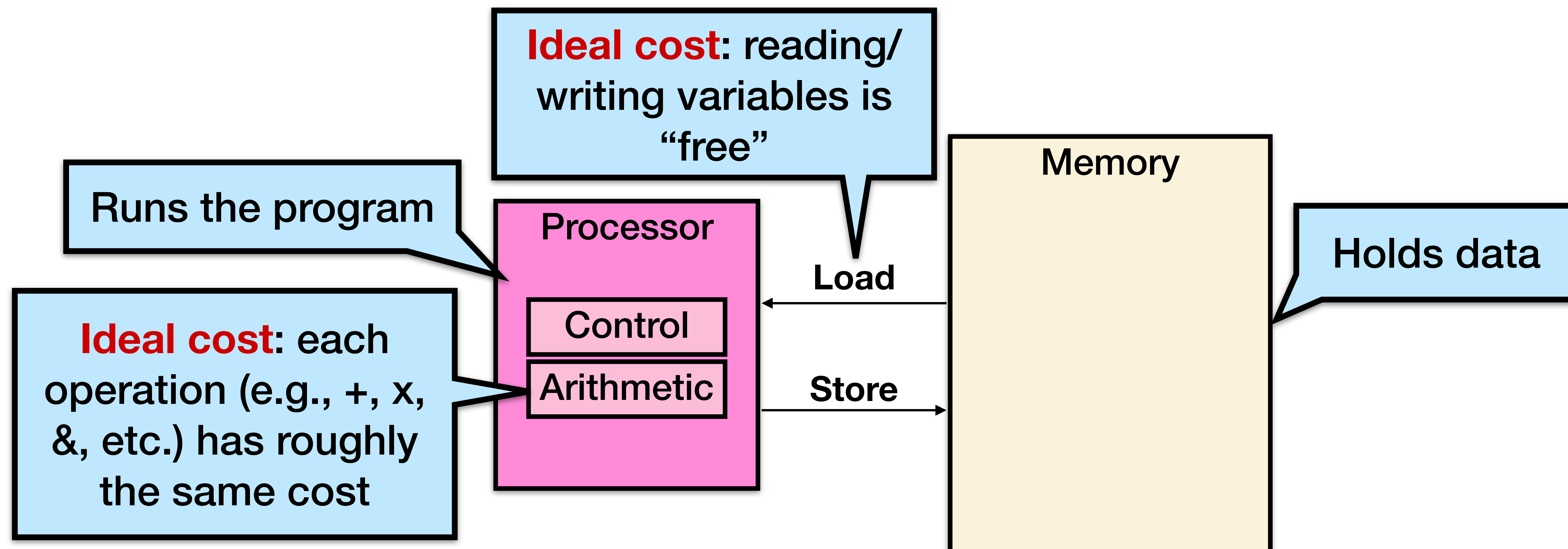
Another way to say it: **If you parallelize slow serial code, you will get slow parallel code.**

Matrix multiply is extreme, but ~10x improvement from **sequential optimizations** is more common.

Idealized uniprocessor cost model

Processor can:

- Name **variables** (e.g., integers, floats, doubles, arrays, pointers, structs, etc.)
- Perform **operations** (e.g., arithmetic, logical, etc.) on those variables
- **Control** the flow of execution as specified by the program (branches (if statements), loops, function calls, etc.)



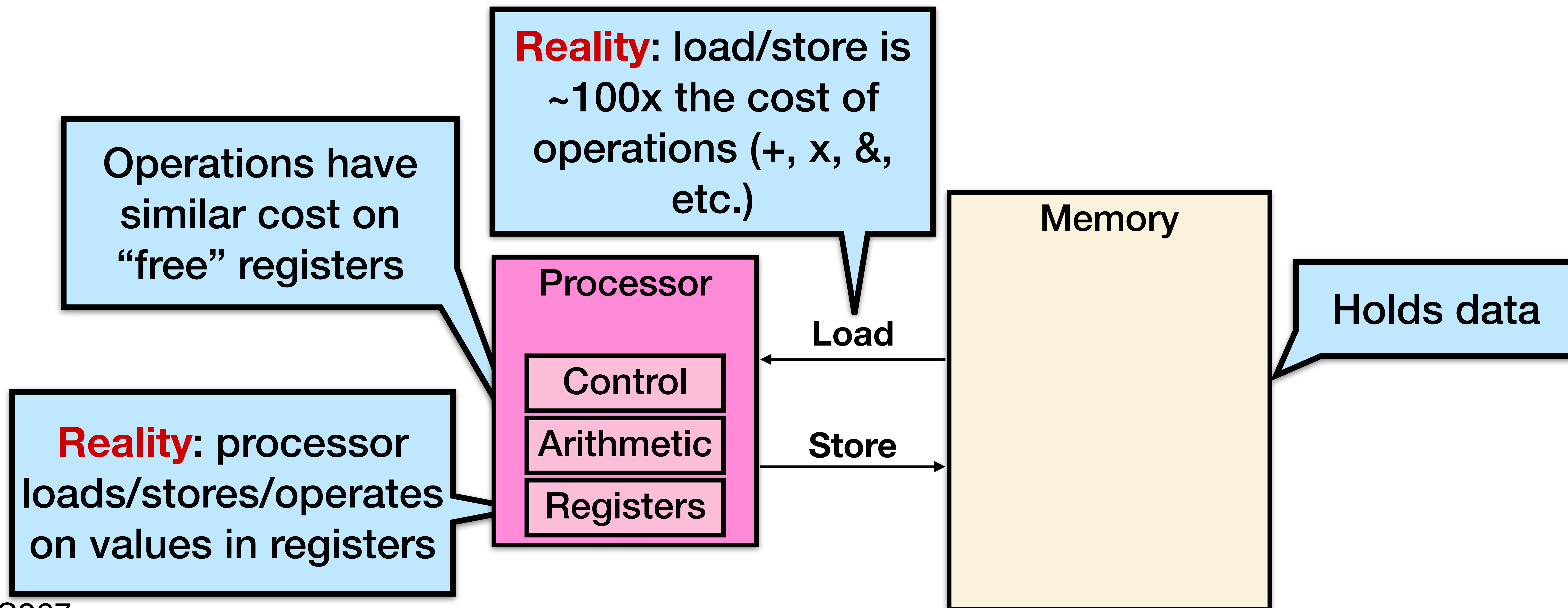
More realistic uniprocessor cost model

In reality, processors have **registers**, or quickly accessible locations.

- **Variables** have to be loaded into registers to operate on them.
- **Load/Store** variables between memory and registers.

Control flow (not free) is determined by the program.

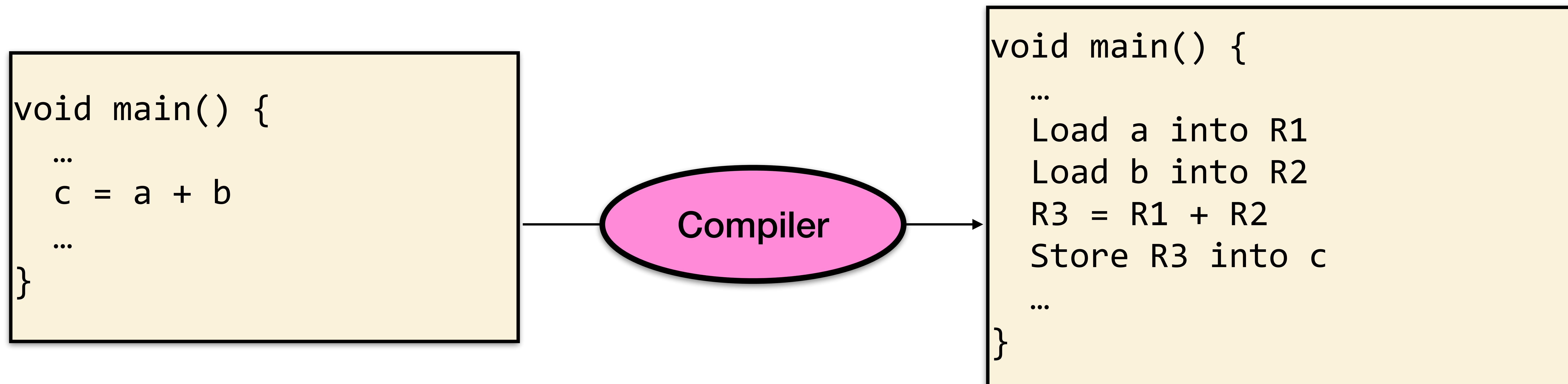
- The compiler **translates** higher-level code into lower-level instructions.



Compilers and assembly code

Compilers for languages like C/C++ and Fortran:

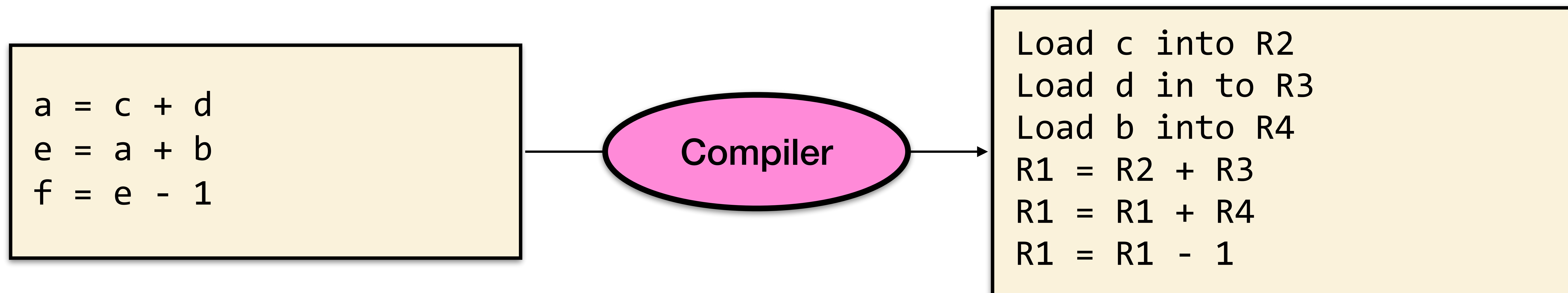
- **Check** that the program is legal,
- **Translate** into assembly code, and
- **Optimize** the generated code.



For more details, see the lecture "C to Assembly" from MIT 6.172.

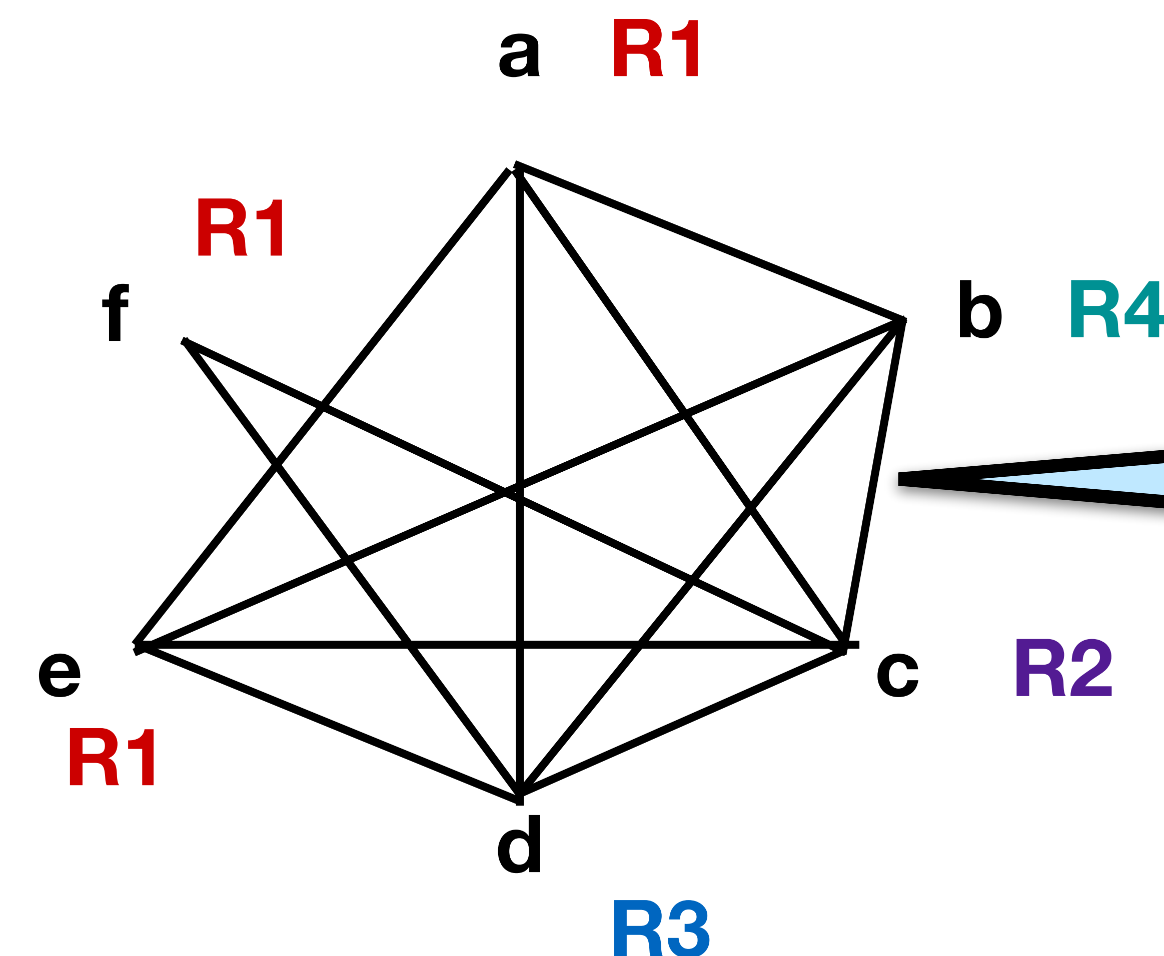
Compilers manage memory and registers

The compiler performs **register allocation**, which decides when to load/store variables vs when to reuse variables.



Register allocation appeared in the first Fortran compiler in the 1950s, Chaitin et al. introduced the idea of doing it via **graph coloring** in the 1980s.

Nodes represent live ranges (variables, temporaries, etc.)



Edges connect live ranges that are simultaneously live

Compilers optimize code

Besides register allocation, **the compiler performs optimizations**:
- Loop unrolling (because control isn't free)
- Fuses loops (merge two loops together)
- Interchanges loops (reorders them)
- Dead code elimination (if branch is never taken)
- Reordering instructions (to improve register reuse)
- Strength reduction (e.g., shift left rather than multiply by 2)
- Loop vectorization (uses SIMD registers)

Why is this the programmer's problem?
- Sometimes the compiler does not do as much as you want it to…

For more details, see the lecture "What Compilers Can and Cannot Do" from MIT 6.172.

Proebsting's Law

Controversial?

Proebsting's Law: Compilers double performance **every 18 years**

(Compared to two years for Moore's law, or 3 years for memory bandwdith)

Benchmark	Optimized (Peak)	Unoptimized (Base)	Ratio
tomcatv	28.3	6.1	4.6
swim	36.9	6.3	5.9
su2cor	10.9	3.1	3.5
hydro2d	9.8	2.6	3.8
mgrid	18.6	1.1	16.9
applu	10.4	0.6	17.3
turb3d	24.4	2.7	9.0
apsi	25.3	3.7	6.8
fpppp	51.3	9.7	5.3

Table 2: SPECfp95 Results. The average ratio between optimized and unoptimized performance is 8.1 with a standard deviation of 5.4.

Assuming compiler research has been going since 1955 (this paper is from 2001), so ~8x improvement over 45 years = **doubling every 15 years or so**

From "On Proebsting's Law" - Kevin Scott (2001)

Slide inspired by Rich Vuduc

Memory Hierarchies

Even more realistic uniprocessor model

Memory accesses (load/store) have two costs:

- **Latency** - the cost to load or store one word (α)
- **Bandwidth** - the average rate (bytes / sec) to load/store a large chunk of data (β)

Bandwidth

≈ data throughput (bytes/second)

Low Bandwidth

High Bandwidth

Latency

≈ delay due data travel time (secs)

Low Latency

High Latency

Image: Katie Hempenius

Memory bandwidth gap

Memory Bandwidth is Falling Behind: (GFLOP/s) / (GWord/s)

Balance Ratio (FLOPS/memory access)

10,000
1,000
100
10
1

+14.2%/year
2x / 5.2 years

RISC systems (IBM, MIPS, Alpha)
x86-64 systems (AMD & Intel)

1990 1995 2000 2005 2010 2015 2020

Date of Introduction

THE UNIVERSITY OF TEXAS AT AUSTIN
TEXAS ADVANCED COMPUTING CENTER

From UC Berkeley CS267

John McCalpin, SC16 invited talk

Memory latency gap is worse

Memory Latency is much worse: (GFLOP/s) / (Memory Latency)

Peak FLOPS per Idle Memory Latency
Peak FLOPS / Word of Sustained Memory BW
+24.5%/year
2x / 3.2 years
+14.2%/year
2x / 5.2 years
RISC systems (IBM, MIPS, Alpha)
x86-64 systems (AMD & Intel)
Balance Ratio (FLOPS/memory access)
10,000
1,000
100
10
1
1990
1995
2000
2005
2010
2015
2020
Date of Introduction

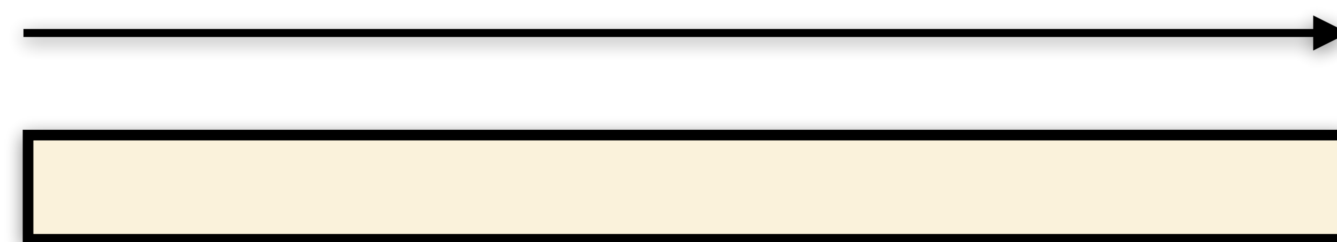
THE UNIVERSITY OF TEXAS AT AUSTIN
TEXAS ADVANCED COMPUTING CENTER

Two main types of locality: Spatial and Temporal

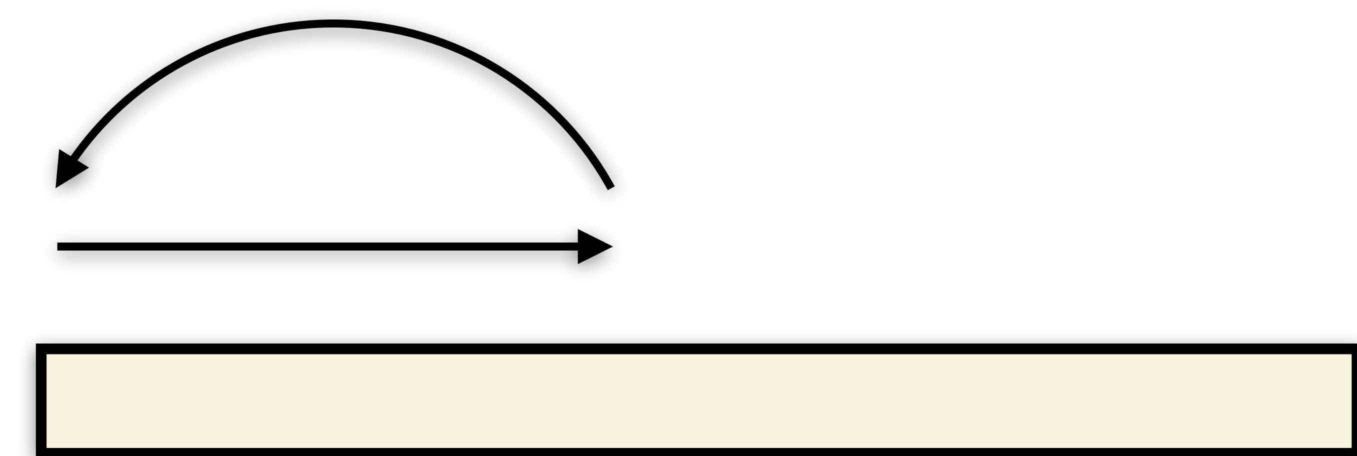
Most programs have a high degree of **locality**.

Spatial locality: how many accesses an algorithm makes to **nearby** data over a short period of time [Denning72, Denning05].

Temporal locality: how many repeated accesses an algorithm makes to **the same** data over a short period of time [Denning72, Denning05].



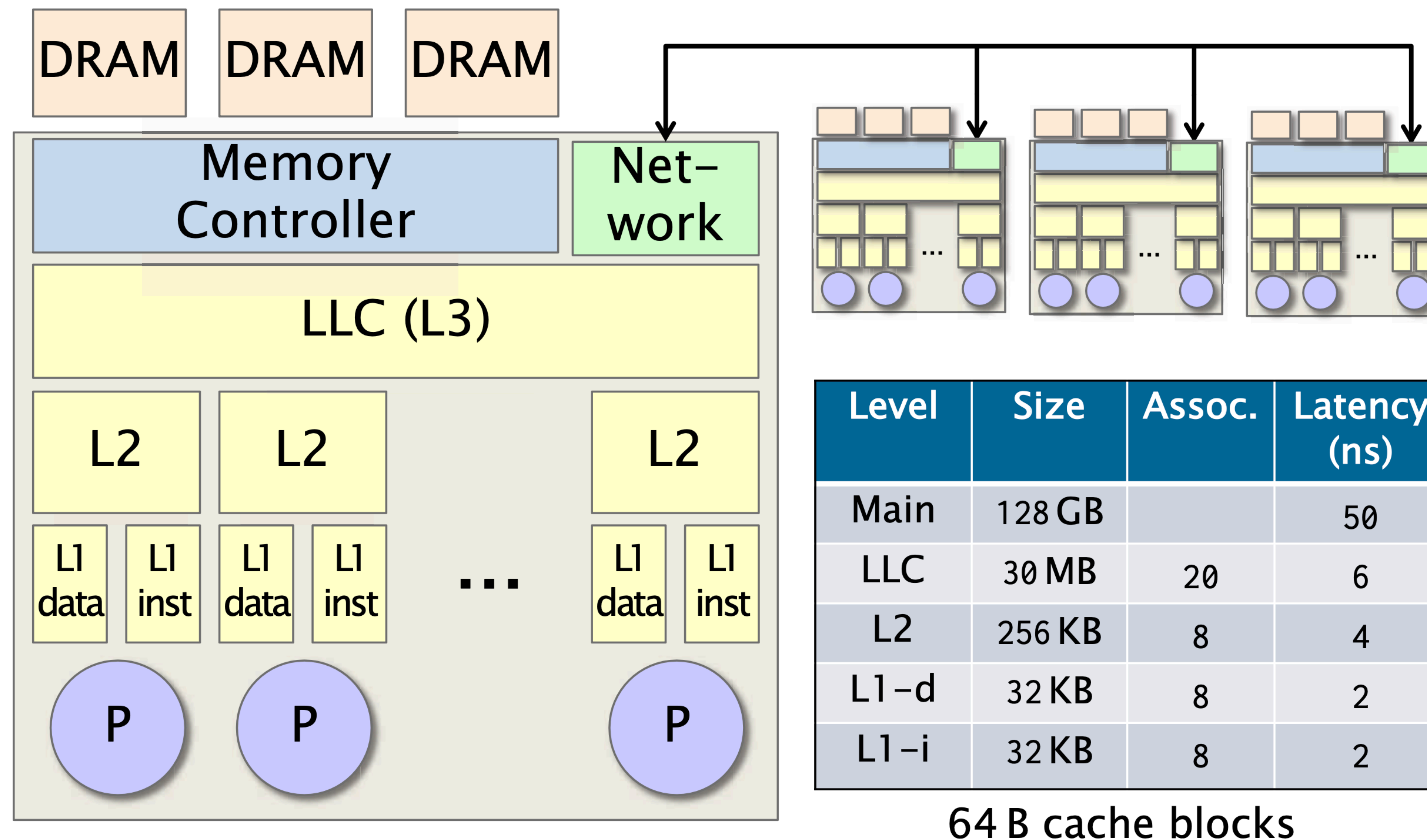
Makes use of **multiple elements transferred together**



Makes use of **efficient hierarchical accesses**

Memory hierarchy

The **memory hierarchy** takes advantage of locality to speed up the average case to handle memory latency.



Level	Size	Assoc.	Latency (ns)
Main	128 GB		50
LLC	30 MB	20	6
L2	256 KB	8	4
L1-d	32 KB	8	2
L1-i	32 KB	8	2

64 B cache blocks

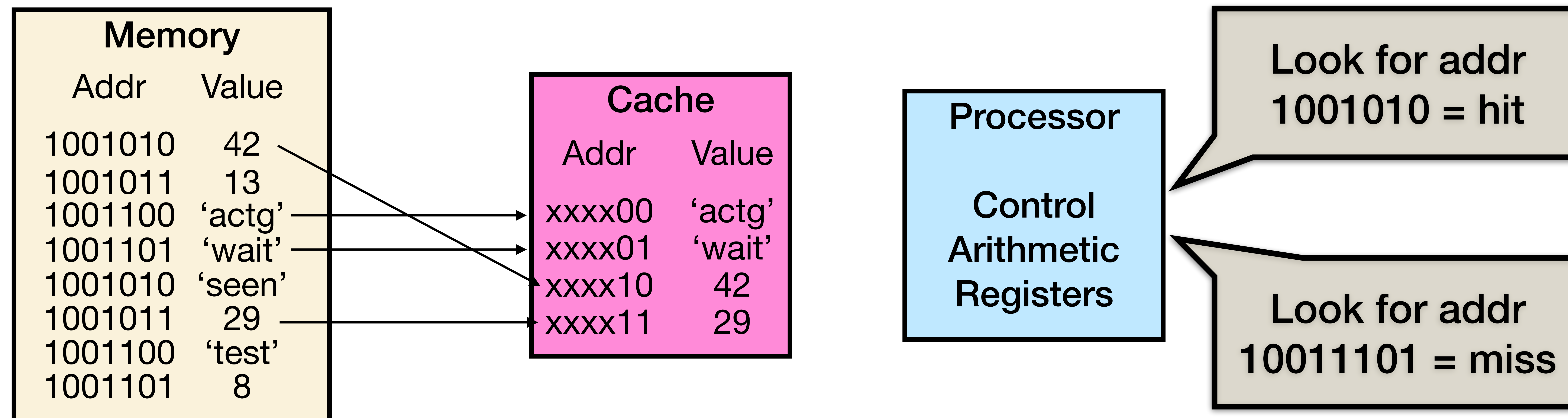
Figure from MIT 6.172

Cache basics

Cache is fast (expensive) memory which keeps a copy of the data; it is hidden from software.

Cache-line length: number of bytes loaded together in one entry (often 64 bytes).

Simple example: data at address xxxxx10 is stored at cache location 10.

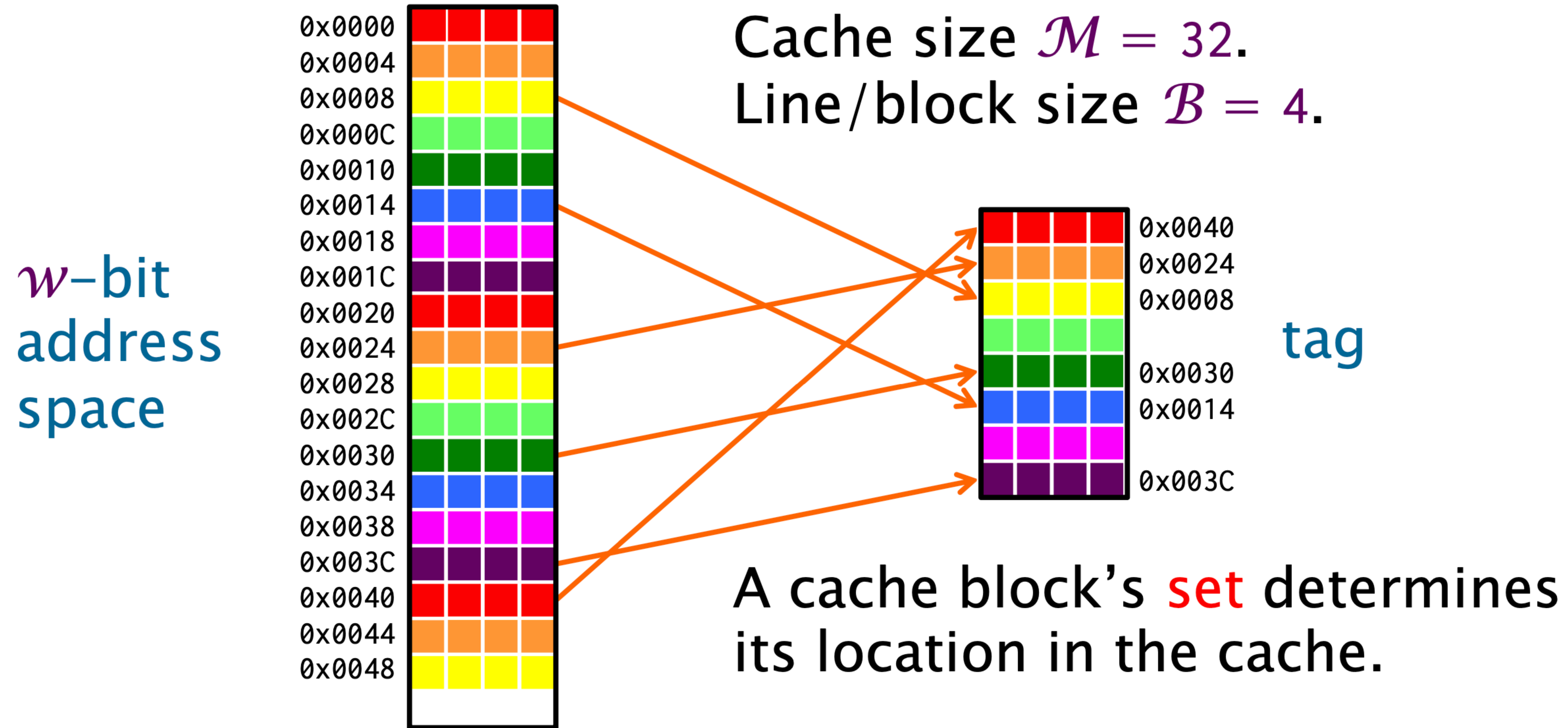


Cache **hit**: access to a memory address in cache - cheap

Cache **miss**: non-cached memory access - expensive

Need to look in next, slower level of memory.

Direct-mapped cache



Cache size $\mathcal{M} = 32$.
Line/block size $\mathcal{B} = 4$.

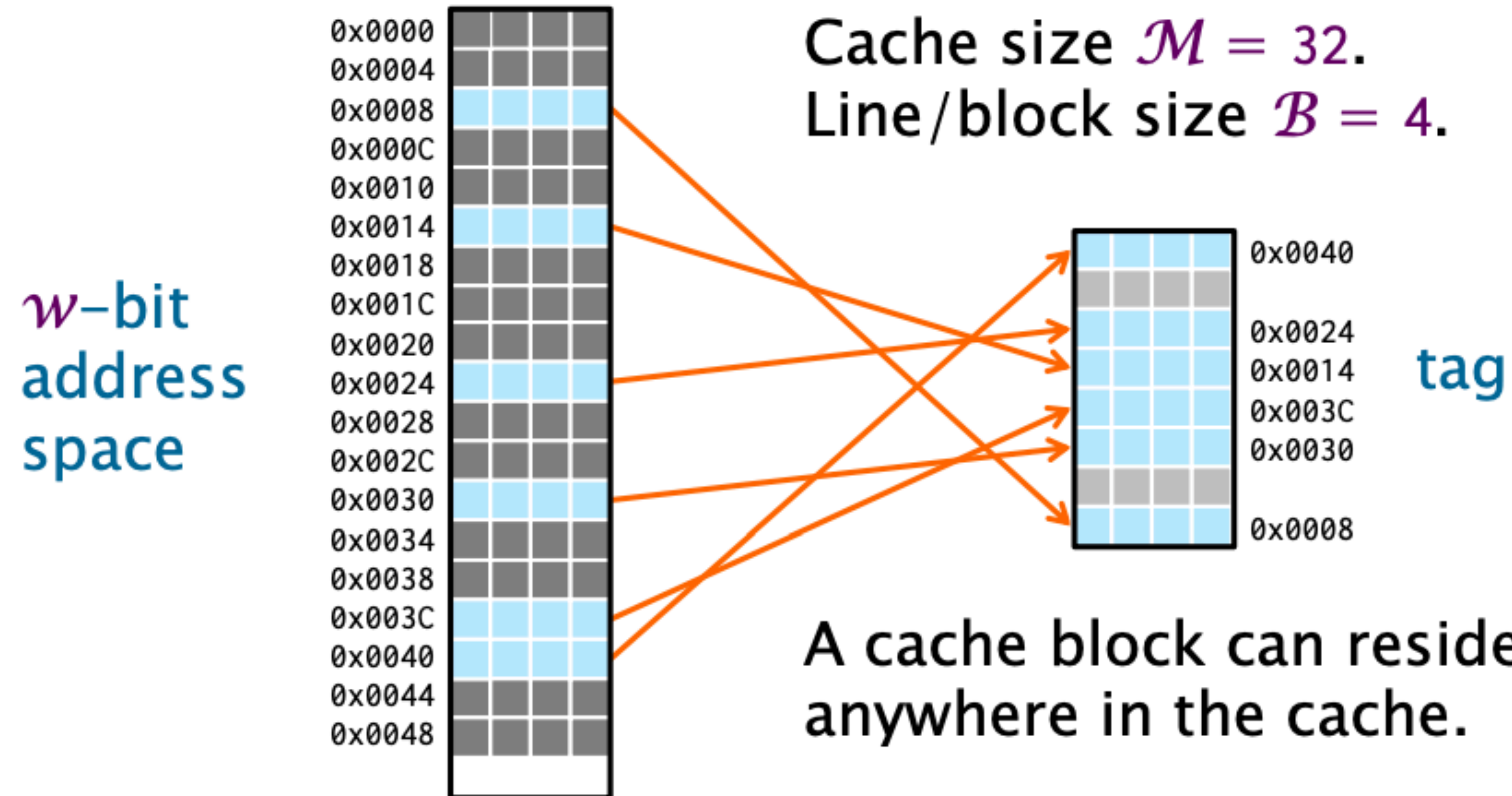
A cache block's set determines its location in the cache.

address

	tag	set	offset
bits	w – lg $\mathcal{M}$	lg($\mathcal{M}/\mathcal{B}$)	lg $\mathcal{B}$

To find a block in the cache, only a single location in the cache need be searched.

Fully-associative cache

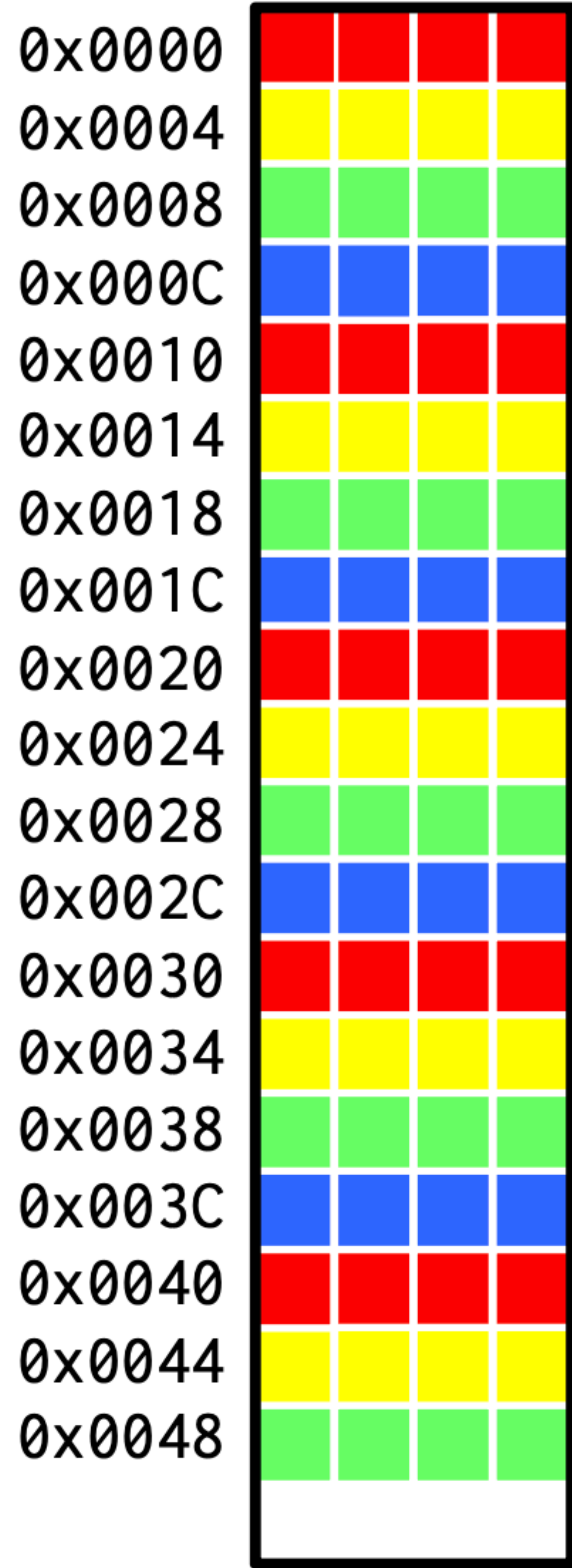


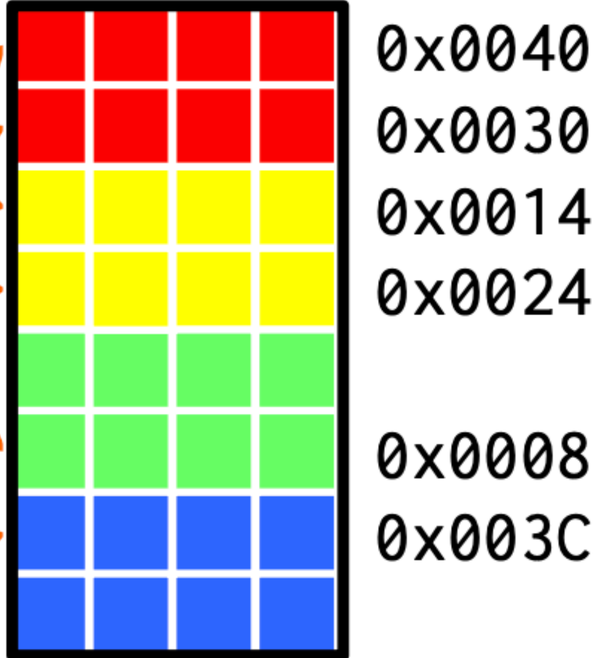
To find a block in the cache, the entire cache must be searched for the tag. When the cache becomes full, a block must be **evicted** to make room for a new block. The **replacement policy** determines which block to evict.

Set-associative cache

Cache size $\mathcal{M} = 32$.
Line/block size $\mathcal{B} = 4$.
k=2–way associativity.

w–bit address space





tag

A cache block's set determines k possible cache locations.

address

	tag	set	offset
bits	$w - \lg(\mathcal{M}/k)$	$\lg(\mathcal{M}/k\mathcal{B})$	$\lg \mathcal{B}$

To find a block in the cache, only the k locations of its set must be searched.

Type of cache misses - Three C’s

Cold miss

The first time a cache block is accessed.

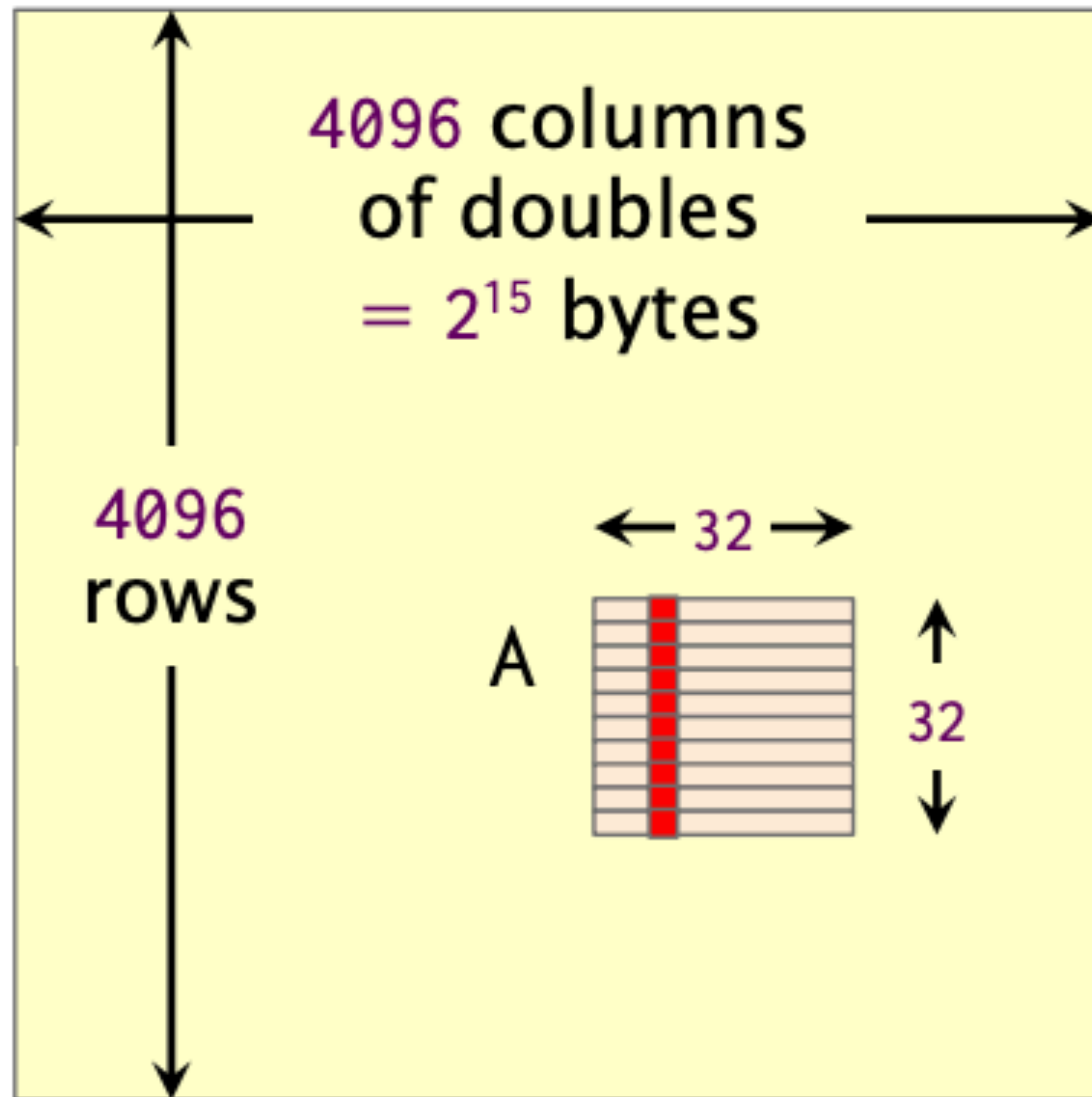
Capacity miss

The previous cached copy would have been evicted even with a fully-associative cache.

Conflict miss

Too many blocks from the same set in cache. The block would not have been evicted with a fully-associative cache.

Conflict misses for submatrices



Conflict misses can be problematic for caches with limited associativity.

address

	tag	set	offset
bits	$w - \lg(\mathcal{M}/k)$	$\lg(\mathcal{M}/k\mathcal{B})$	$\lg \mathcal{B}$
	51	7	6

Assume:

- Word width $w = 64$.
- Cache size $\mathcal{M} = 32K$.
- Line (block) size $\mathcal{B} = 64$.
- $k=4$-way associativity.

Analysis

Look at a column of submatrix A. The addresses of the elements are $x, x+2^{15}, x+2\cdot2^{15}, \ldots, x+31\cdot2^{15}$.
They all fall into the same set!

Solutions

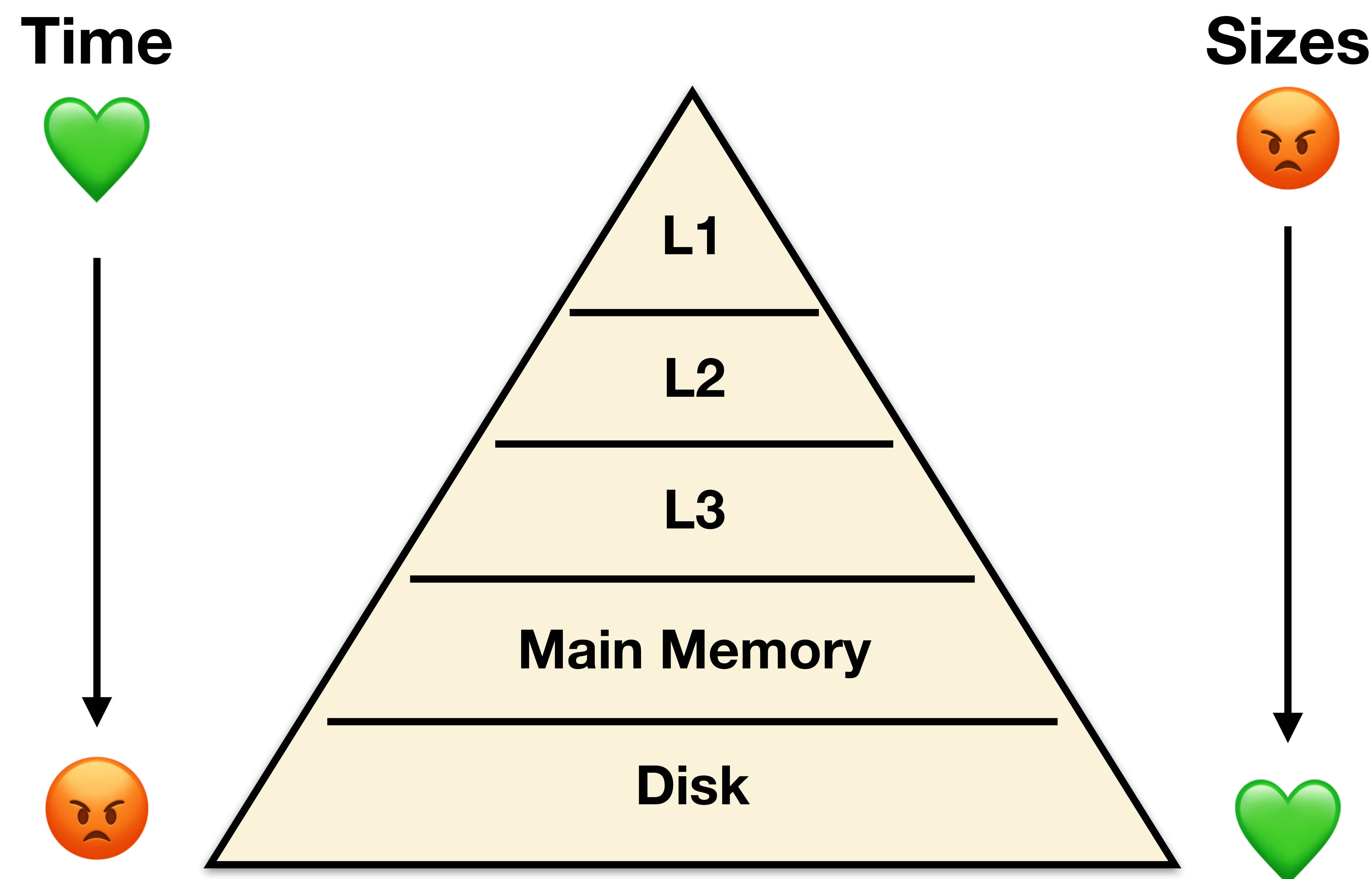
Copy A into a temporary 32×32 matrix, or pad rows.

Why have multiple levels of cache?

On-chip caches are **faster but smaller** compared to off-chip caches.

A large cache has delays:

- **Hardware** to check longer addresses in cache takes more time.
- **Associativity**, which gives a more general set of data in cache, also takes more time.



Approaches to handling memory latency

Reuse values in fast memory (bandwidth filtering)
Needs temporal locality in program

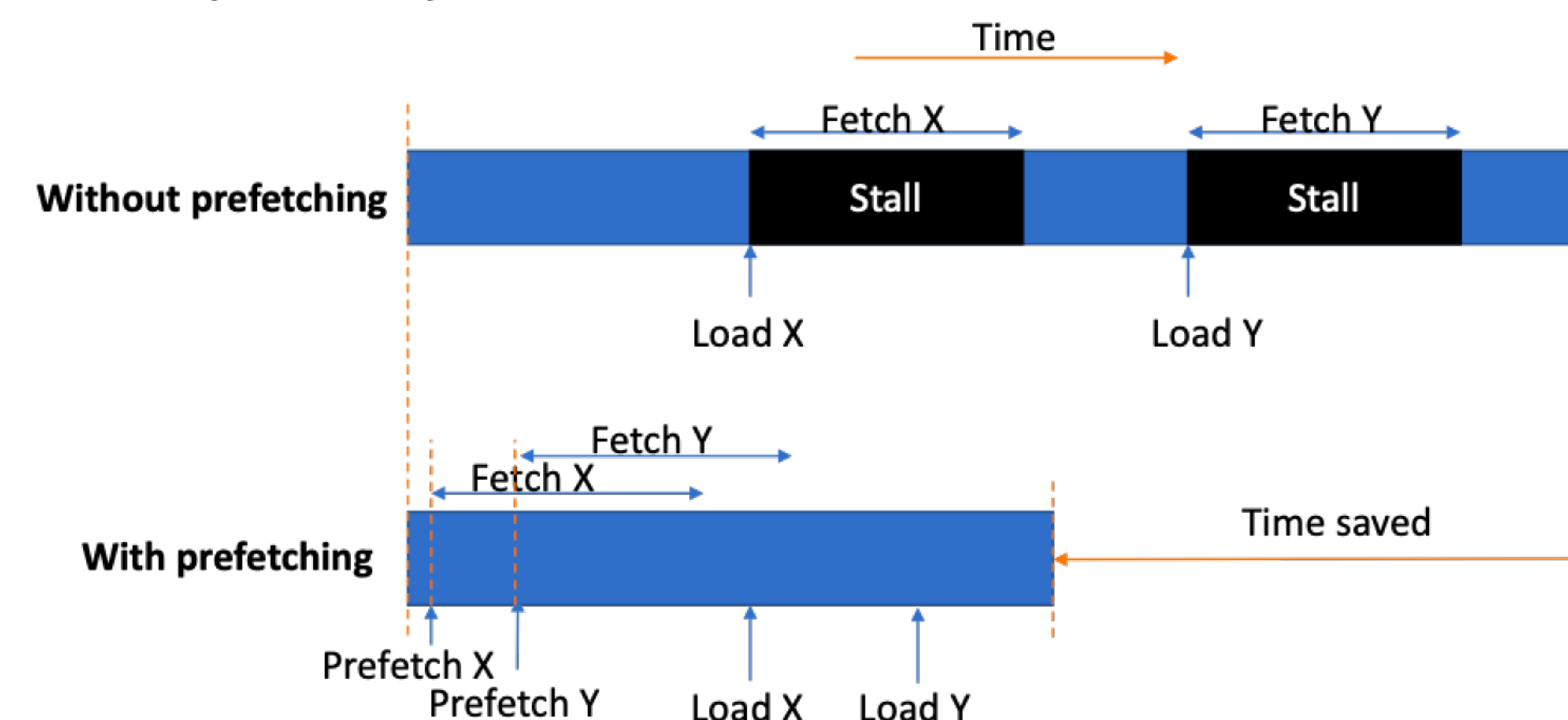
Move larger chunks (achieve higher bandwidth)
Needs spatial locality in program

Concurrency

Issue multiple reads/writes in a single instruction (higher bandwidth)
Vector operations require access to a set of locations (typically neighboring)

Issue multiple reads/writes in parallel (hide latency)
Prefetching issues read hint
Delayed writes (write buffering) stages writes for later operation

Requires that nothing dependent is happening (parallelism)



How much concurrency do you need?
(To run at bandwidth speeds rather than latency)

Little's Law from queueing theory says:

concurrency = latency * bandwidth

For example, let:
  Latency = 10 sec
  Bandwidth = 2 bytes / sec

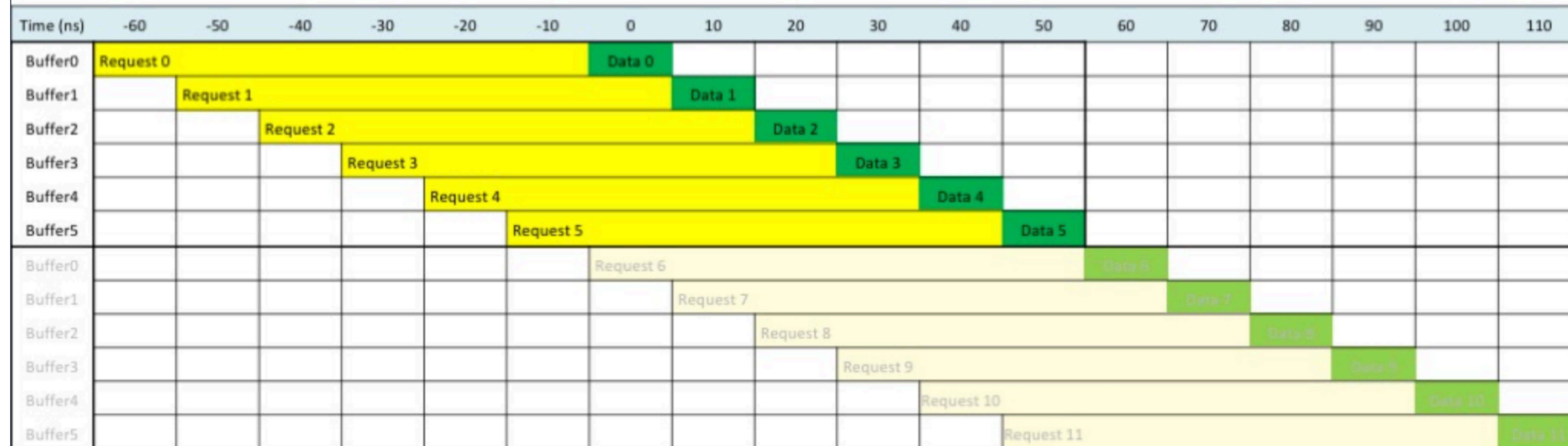
-> requires finding 20 bytes in flight to hit bandwidth speeds, or finding 20 independent things to issue.

Little's law explains how **concurrency helps to hide latency**.

Real-world example

Little's Law: illustration for 2005-era Opteron processor

60 ns latency, 6.4 GB/s (=10ns per 64B cache line)



- 60 ns * 6.4 GB/s = 384 Bytes = 6 cache lines
- To keep the pipeline full, there must always be 6 cache lines "in flight"
- Each request must be launched at least 60 ns before the data is needed

THE UNIVERSITY OF TEXAS AT AUSTIN

TEXAS ADVANCED COMPUTING CENTER

Ideal-Cache Model

Ideal-Cache Model

Parameters

- Two-level hierarchy
- Cache size of *M* bytes
- Cache-line length of *B* bytes
- Fully associative
- Optimal, omniscient replacement.

memory
cache
P
$\mathcal{M}/\mathcal{B}$ cache lines
$\mathcal{B}$

Performance Measures

- **Work** W (ordinary running time)
- **Cache misses** Q (number of cache lines that need to be transferred between cache and memory)

How reasonable are ideal caches?

"LRU" Lemma [ST85]. Suppose that an algorithm incurs Q cache misses on an ideal cache of size M. Then on a fully associative cache of size $2M$ that uses the **least-recently used (LRU)** replacement policy, it incurs at most $2Q$ cache misses.

Implication

For asymptotic analyses, one can assume optimal or LRU replacement, as convenient.

Software engineering

- Design a **theoretically good** algorithm.
- **Engineer** for detailed performance.
  - Real caches are not fully associative.
  - Loads and stores have different costs with respect to bandwidth and latency.

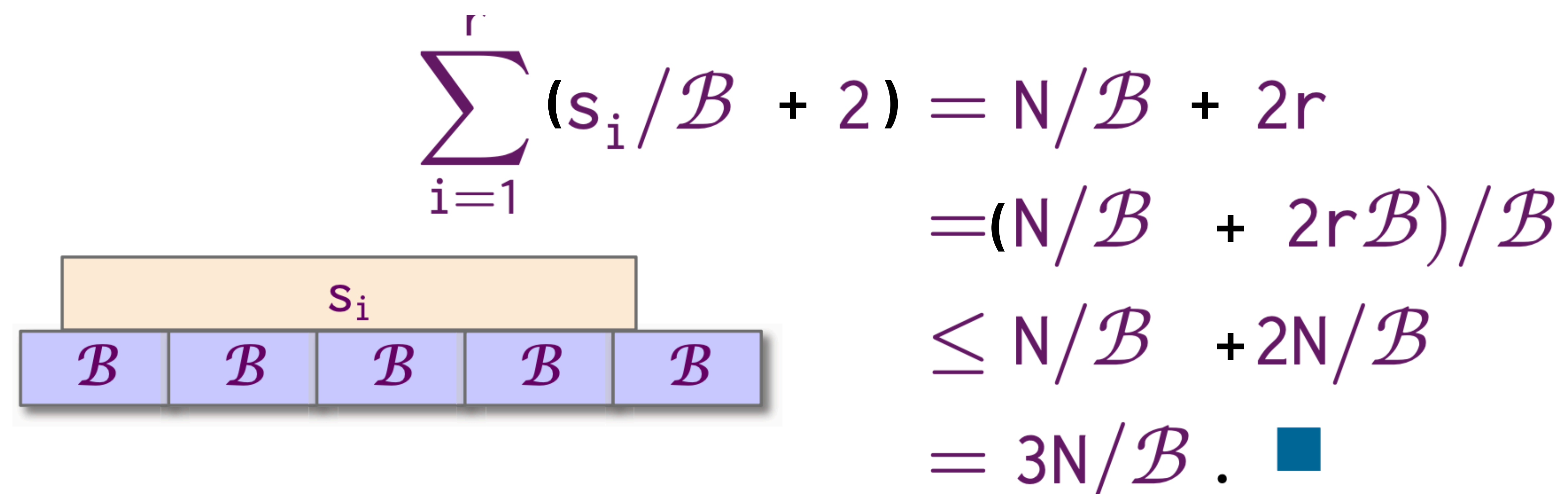
Cache-miss lemma

Lemma. Suppose that a program reads a set of r data segments, where the ith segment consists of s_i bytes, and suppose that

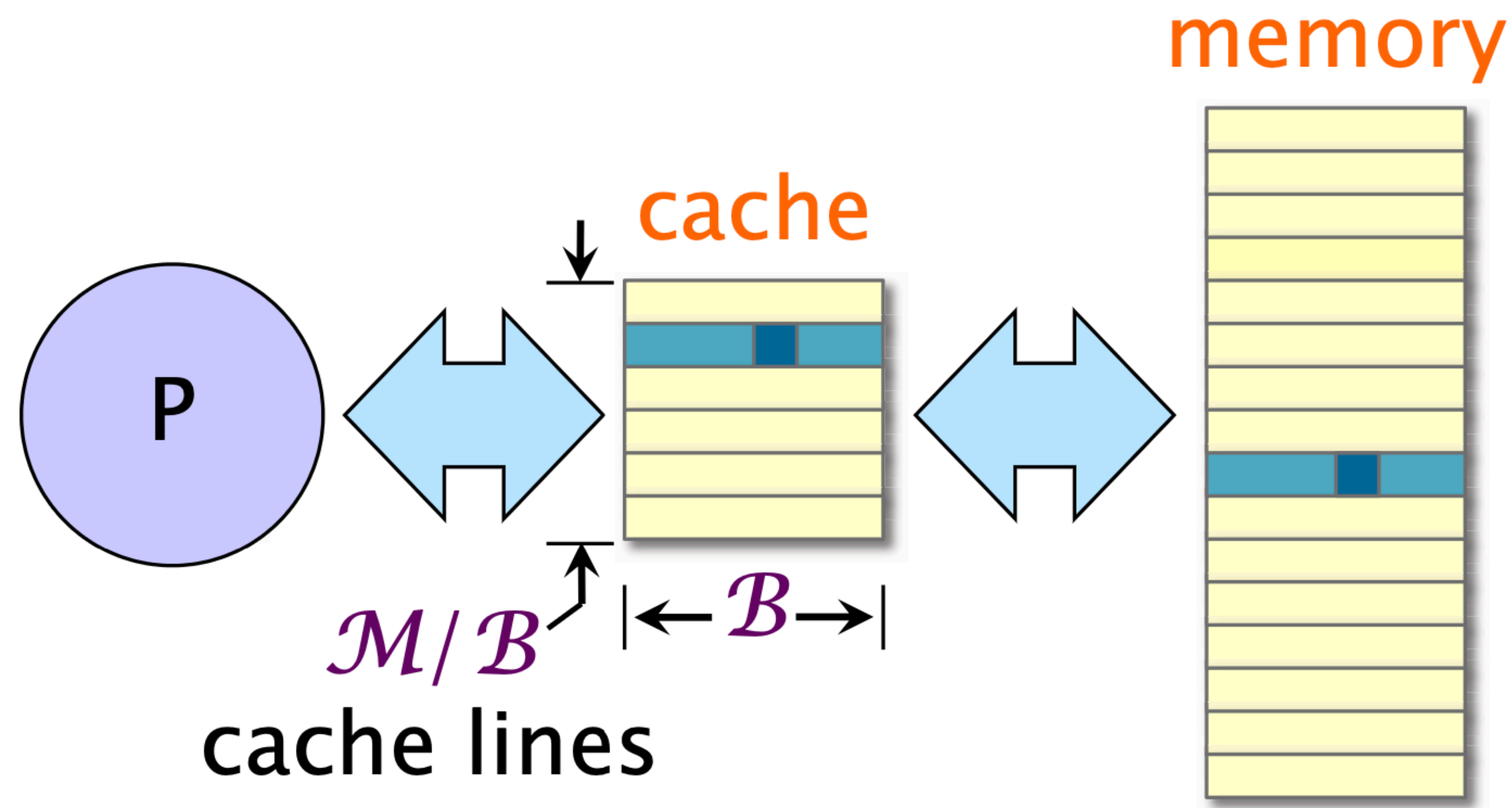
$$\sum_{i=1}^{r} s_i = N < \mathcal{M}/3 \text{ and } N/r \geq \mathcal{B} .$$

Then all of the segments fit into cache, and the number of misses to read them all is at most $3N/B$.

Proof. Suppose that a program reads a set of r data segments, where the ith segment consists of A single segment s_i incurs at most s_i/B misses, and so



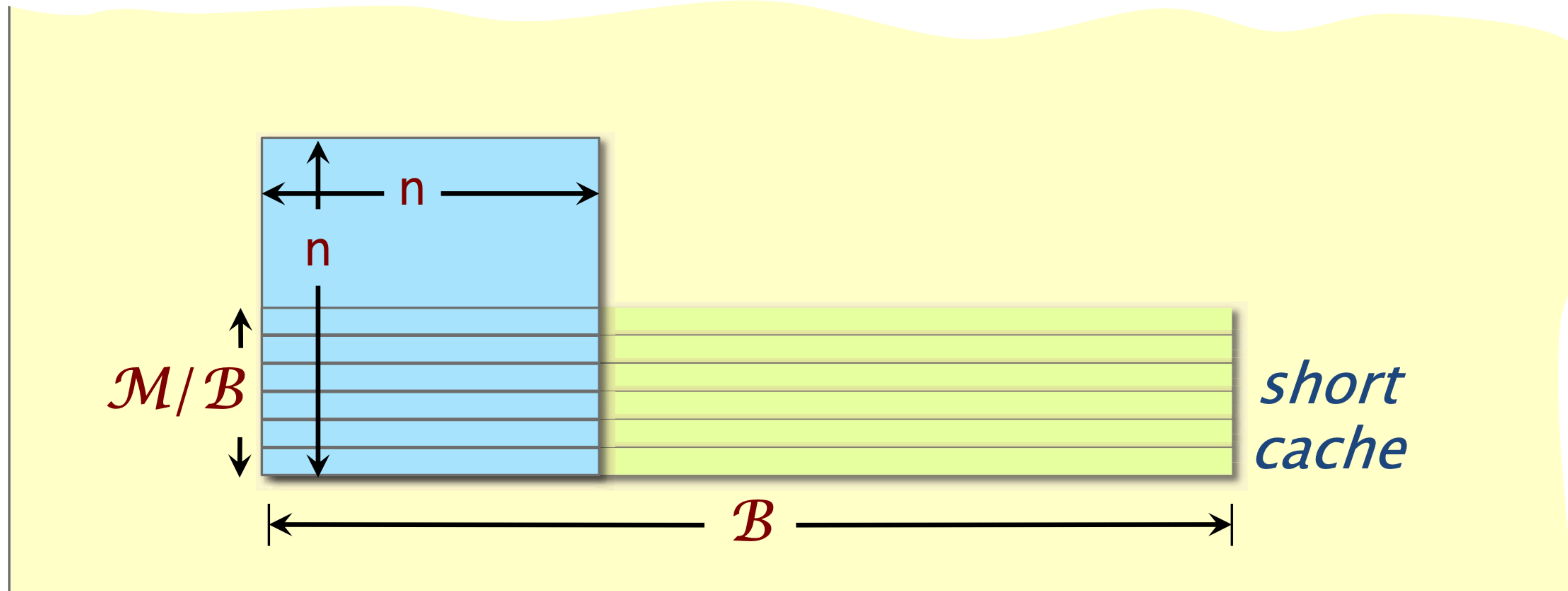
Tall caches



Tall-cache assumption
$\mathcal{B}^2 < c\mathcal{M}$ for some sufficiently small constant $c \leq 1$.

Example: Intel Xeon E5-2666 v3
- Cache-line length = 64 bytes.
- L1-cache size = 32 Kbytes.

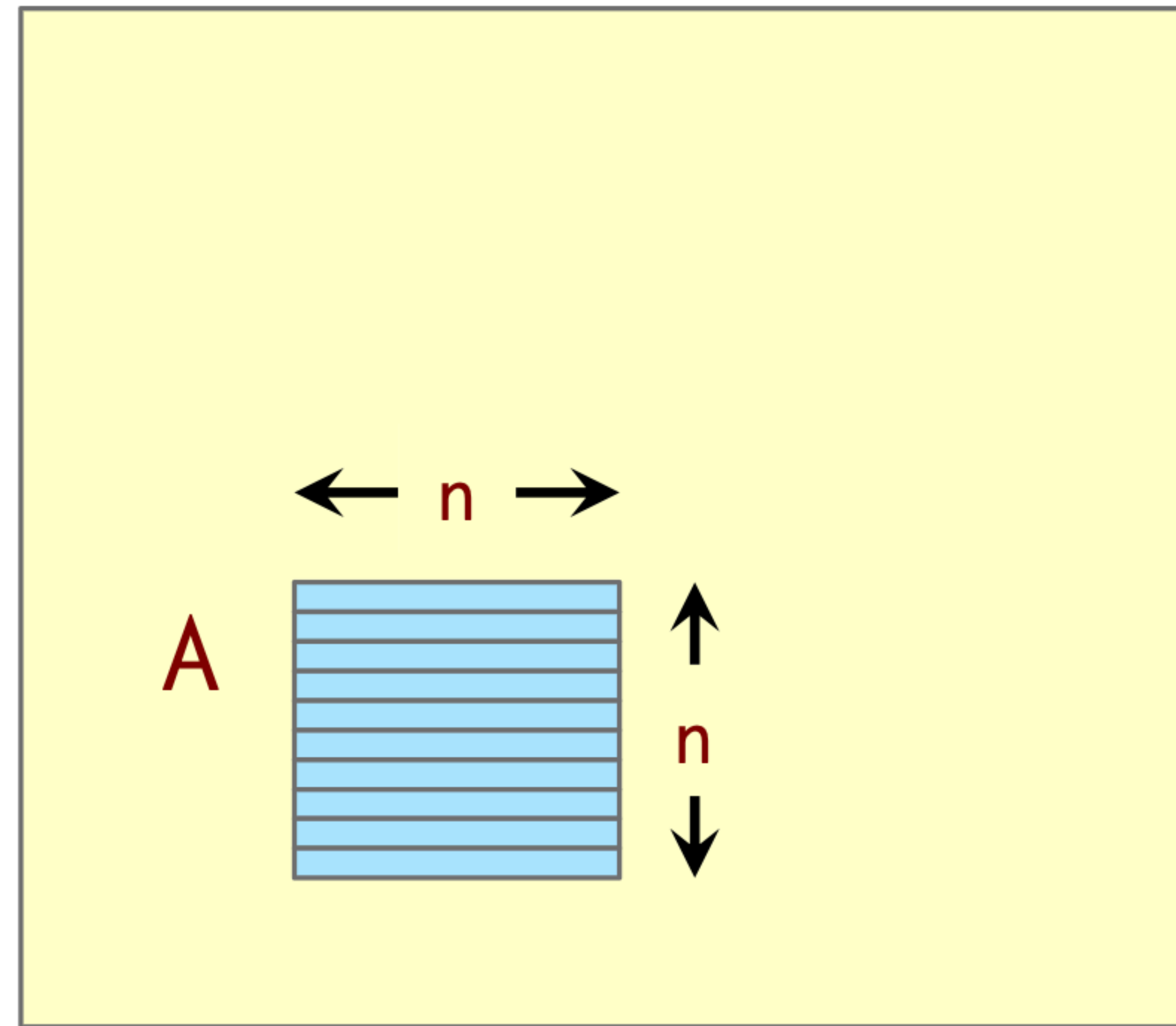
What's wrong with short caches?



Tall-cache assumption
$\mathcal{B}^2 < c\mathcal{M}$ for some sufficiently small constant $c \leq 1$.

An $n \times n$ submatrix stored in row-major order may not fit in a short cache even if $n^2 < c\mathcal{M}$!

Submatrix caching lemma



Lemma. Suppose that an $n \times n$ submatrix A is read into a tall cache satisfying $\mathcal{B}^2 < c\mathcal{M}$, where $c \leq 1$ is constant, and suppose that $c\mathcal{M} \leq n^2 < \mathcal{M}/3$. Then A fits into cache, and the number of misses to read all A's elements is at most $3n^2/\mathcal{B}$.

Proof. We have $N = n^2$, $n = r = s_i$, $\mathcal{B} \leq n = N/r$, and $N < \mathcal{M}/3$. Thus, the Cache-Miss Lemma applies. ■

Cache Analysis of Matrix Multiplication

Why matrix multiplication?

Matrix multiplication is an **important kernel** in many problems:

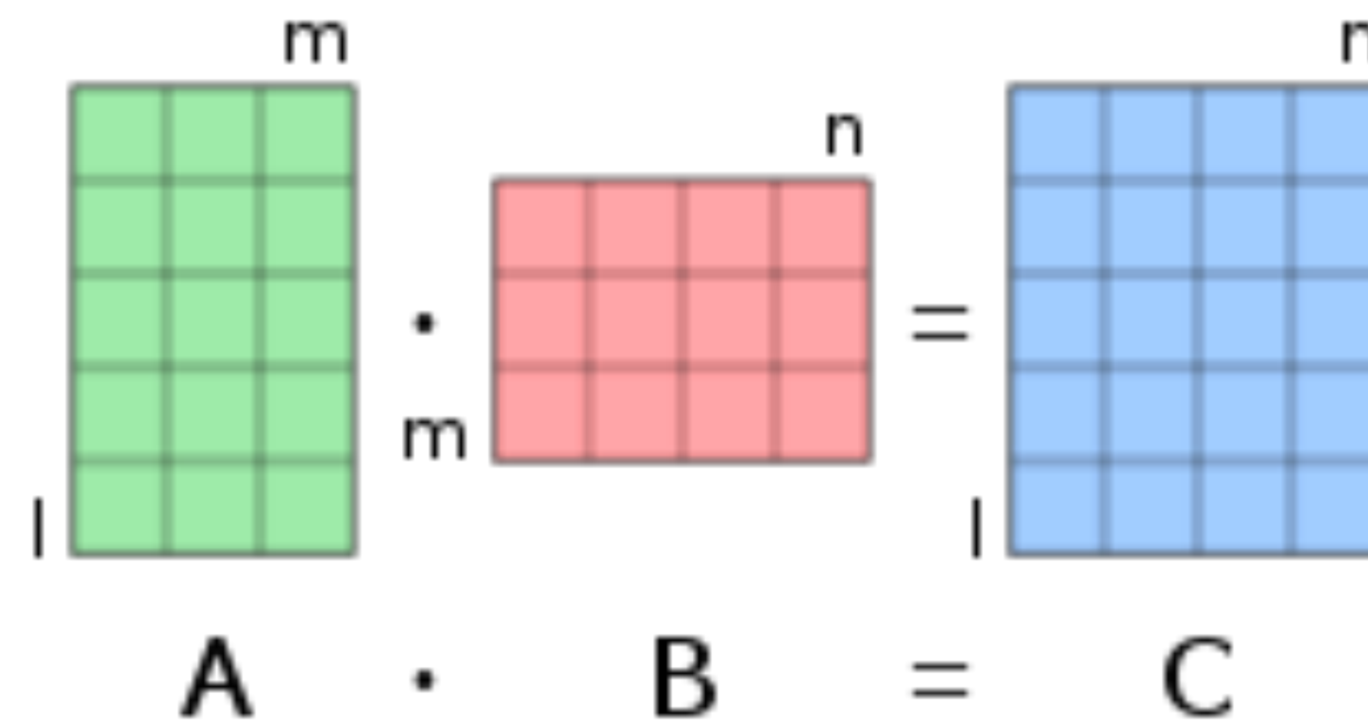
– Dense linear algebra is a motif in every list,

– Closely related to other algorithms, e.g., transitive closure on a graph,

– And dominates training time in deep learning (CNNs)

Good model problem (well-studied, illustrates ideas).

Easy to find good libraries that are hard to beat! (e.g., Intel MKL, etc.)



https://en.wikipedia.org/wiki/Matrix_multiplication

Multiply square matrices

```
void Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i=0; i < n; i++)
    for (int64_t j=0; j < n; j++)
      for (int64_t k=0; k < n; k++)
        C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```

Analysis of work:

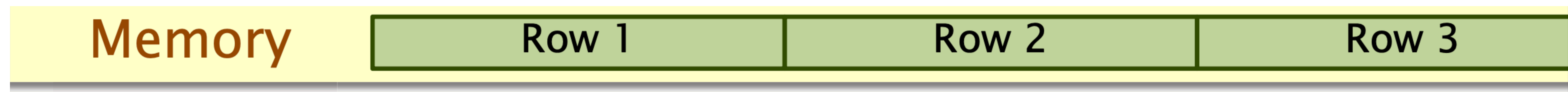
$W(n) = \Theta(n^3)$

Memory layout of matrices

In this matrix-multiplication code, matrices are laid out in memory in **row-major order**.

Matrix

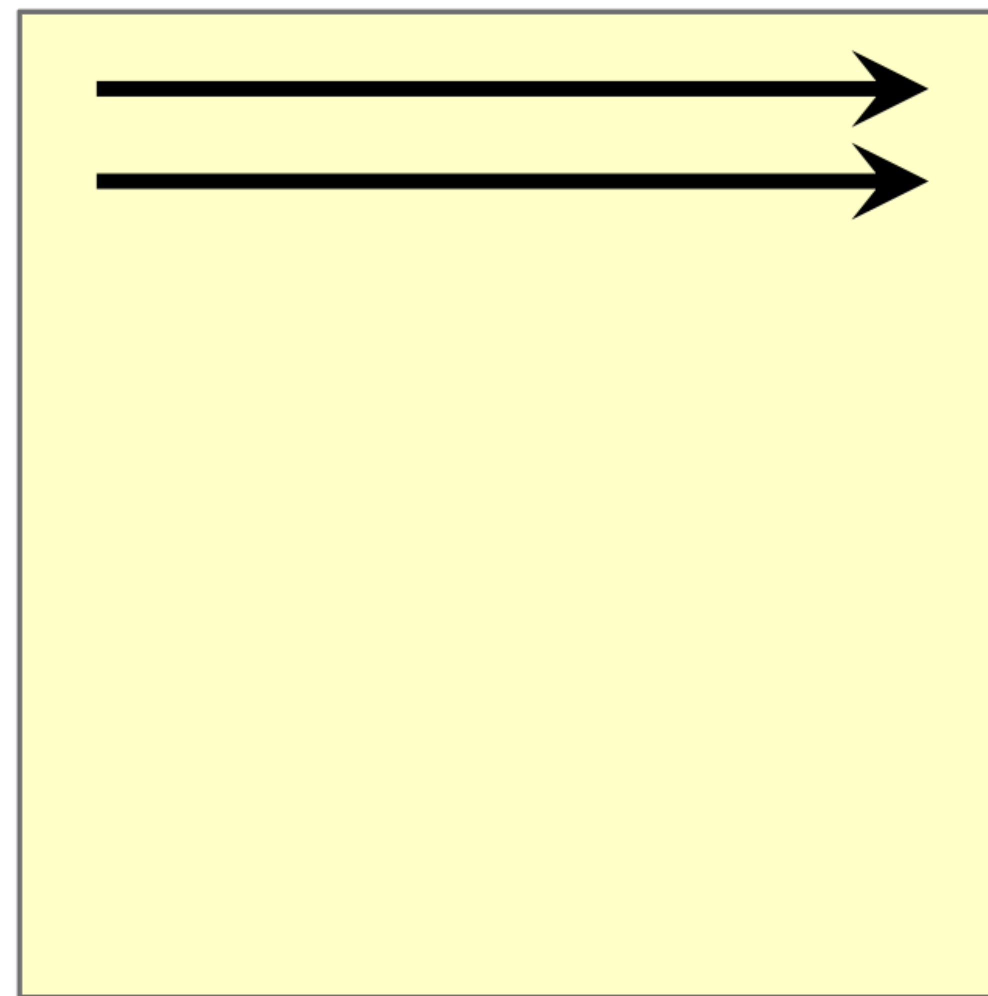
Row 1
Row 2
Row 3
Row 4
Row 5
Row 6
Row 7
Row 8

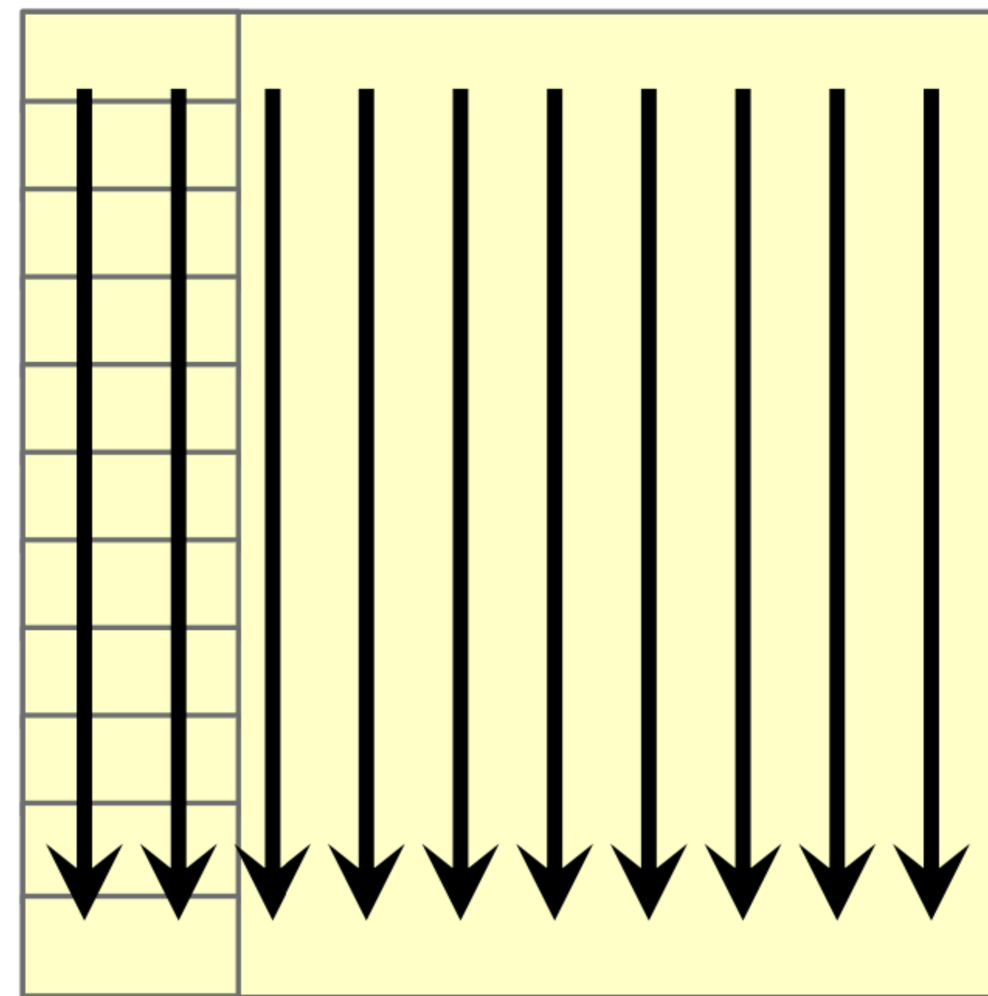


Analysis of cache misses

```
void Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i=0; i < n; i++)
    for (int64_t j=0; j < n; j++)
      for (int64_t k=0; k < n; k++)
        C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```

Assume row major and tall cache





Case 1

$n > cM/B$. Analyze matrix B.

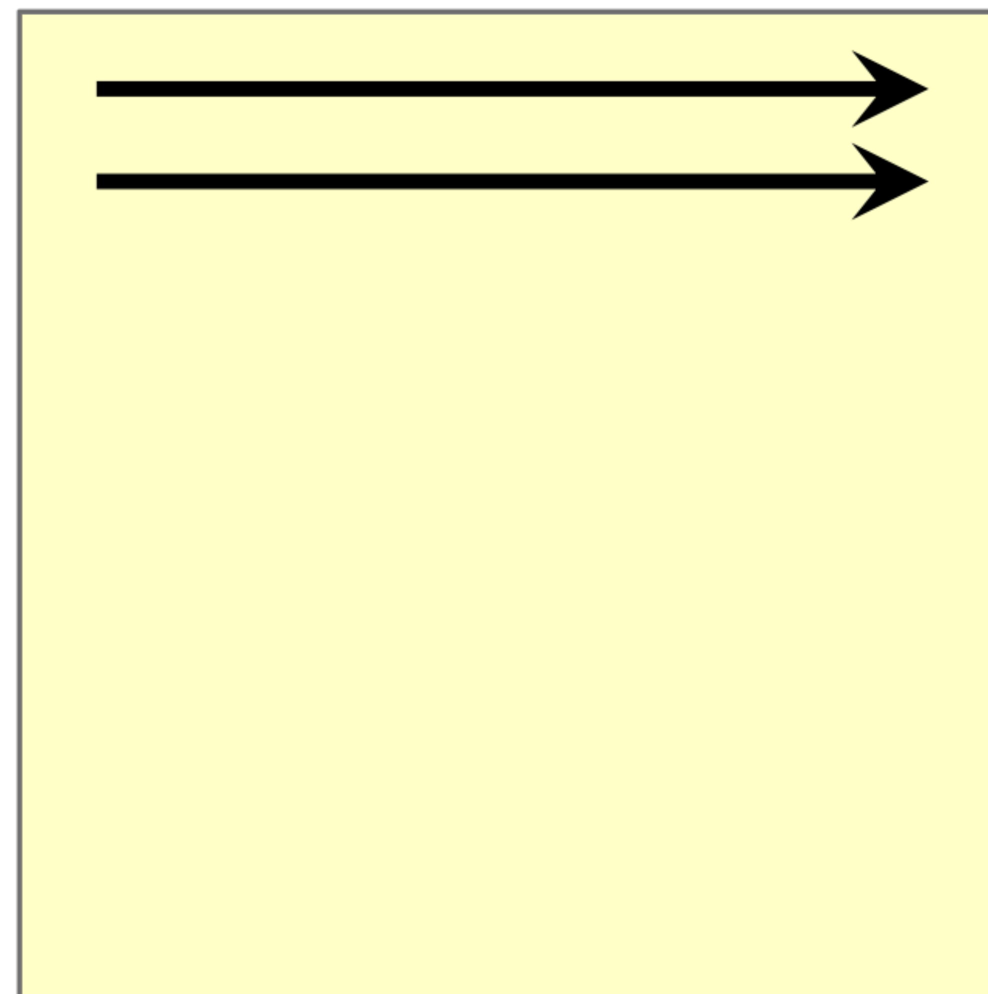
Assume LRU.

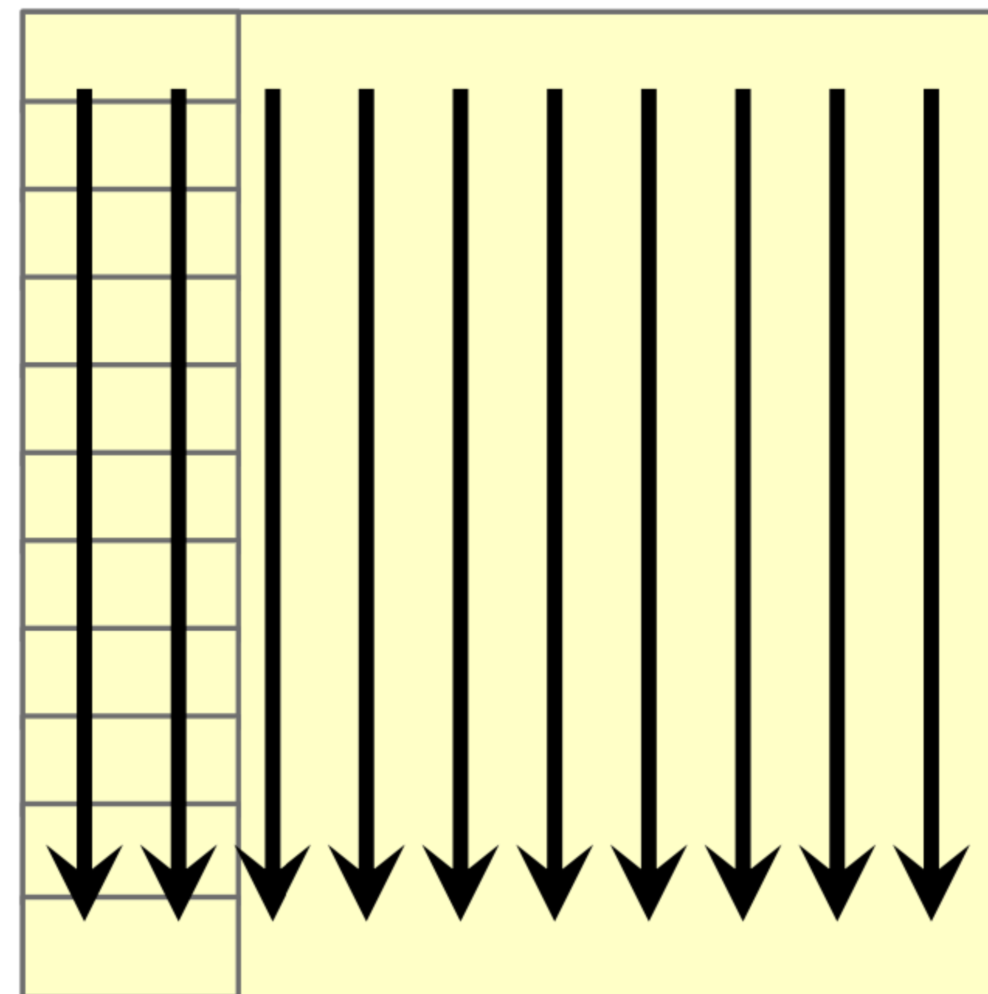
$Q(n) = \Theta(n^3)$, since matrix B misses on every access.

Analysis of cache misses

```
void Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i=0; i < n; i++)
    for (int64_t j=0; j < n; j++)
      for (int64_t k=0; k < n; k++)
        C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```

Assume row major and tall cache





Case 2

$c'M^{1/2} < n < cM/B$. Analyze matrix B.

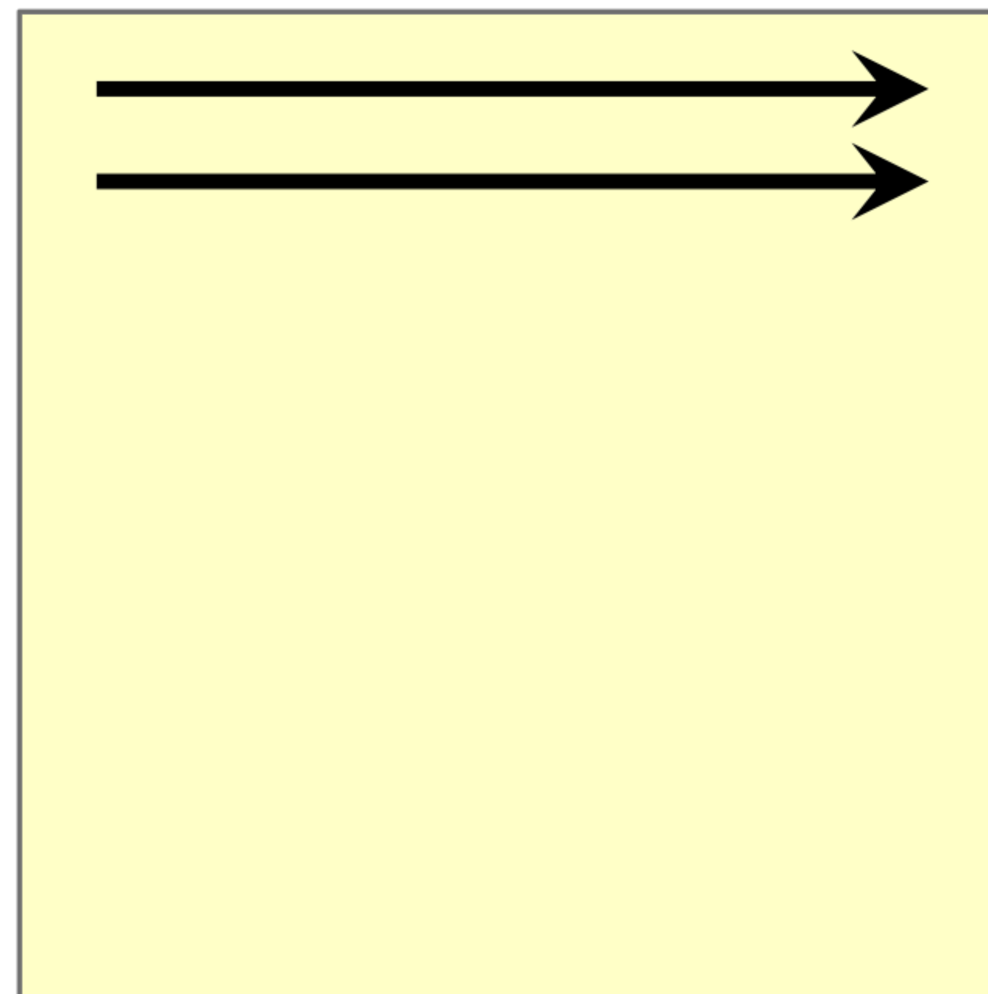
Assume LRU.

$Q(n) = n \cdot \Theta(n^2/B) = \Theta(n^3/B)$, since matrix B can exploit spatial locality.

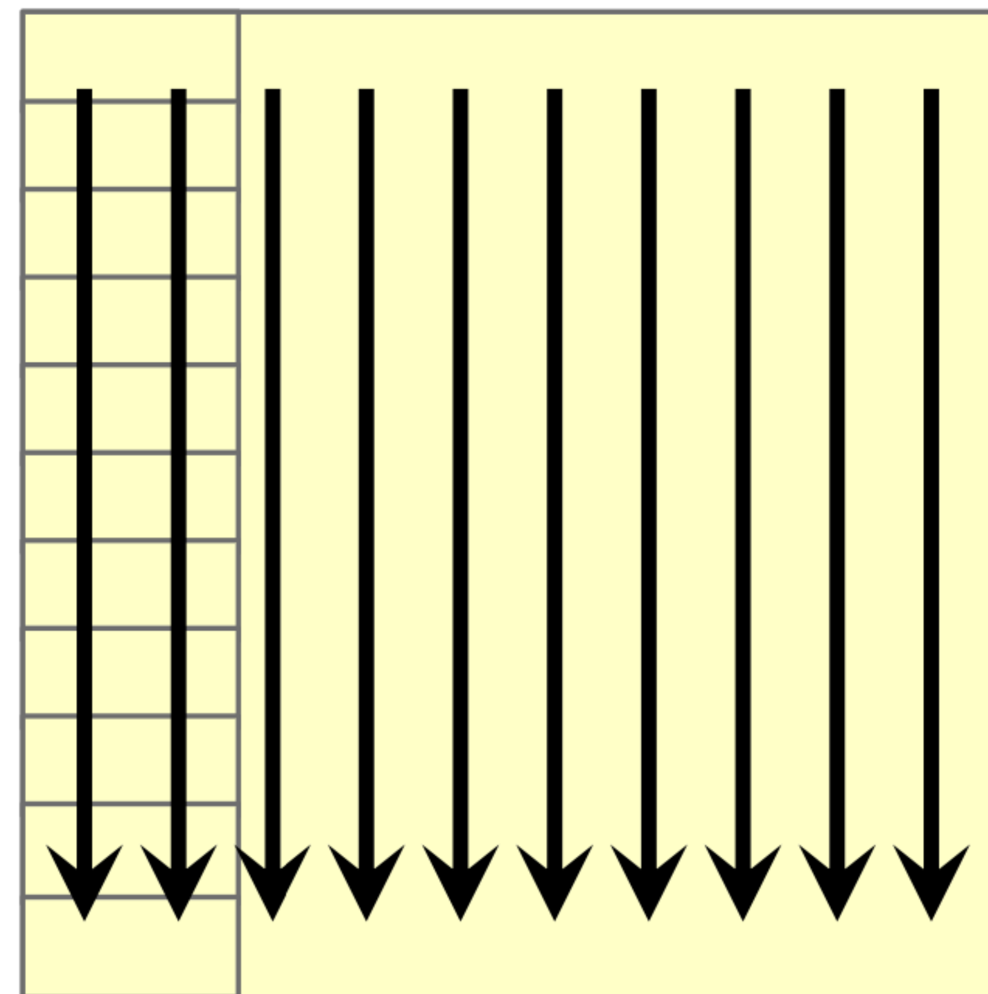
Analysis of cache misses

```
void Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i=0; i < n; i++)
    for (int64_t j=0; j < n; j++)
      for (int64_t k=0; k < n; k++)
        C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```

Assume row major and tall cache



A



B

Case 3

$n < c'M^{1/2}$. Analyze matrix B.

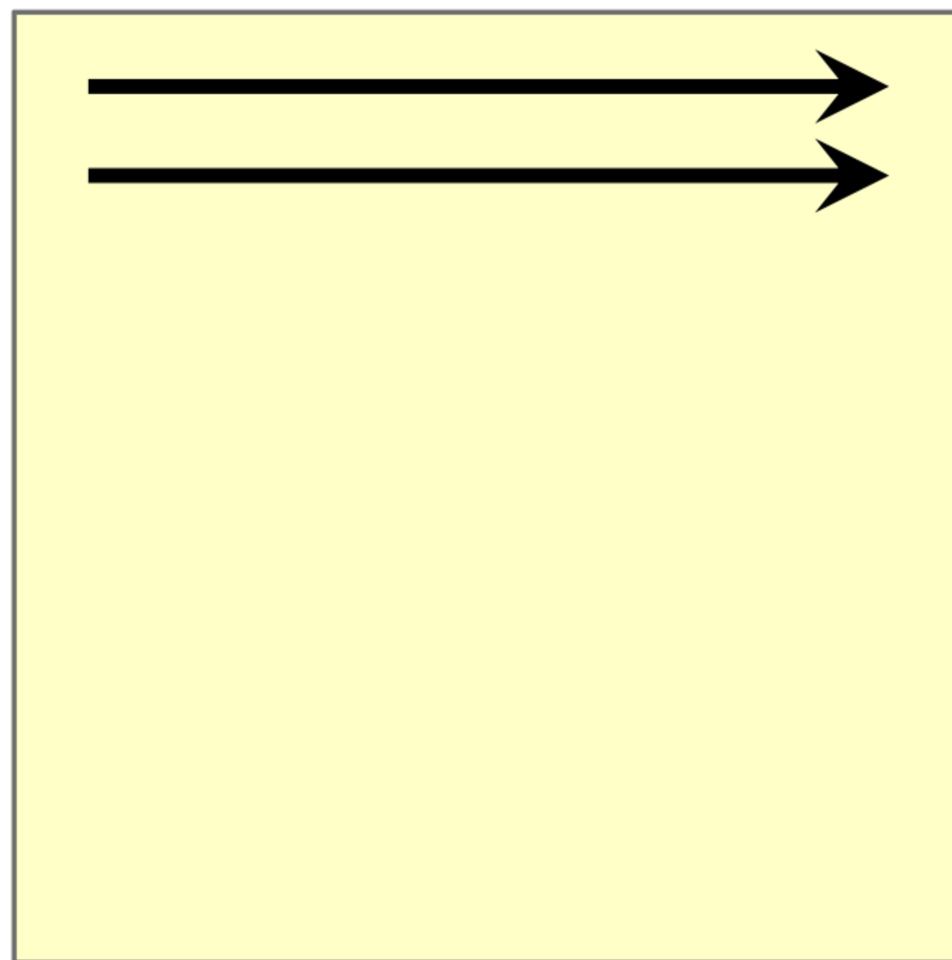
Assume LRU.

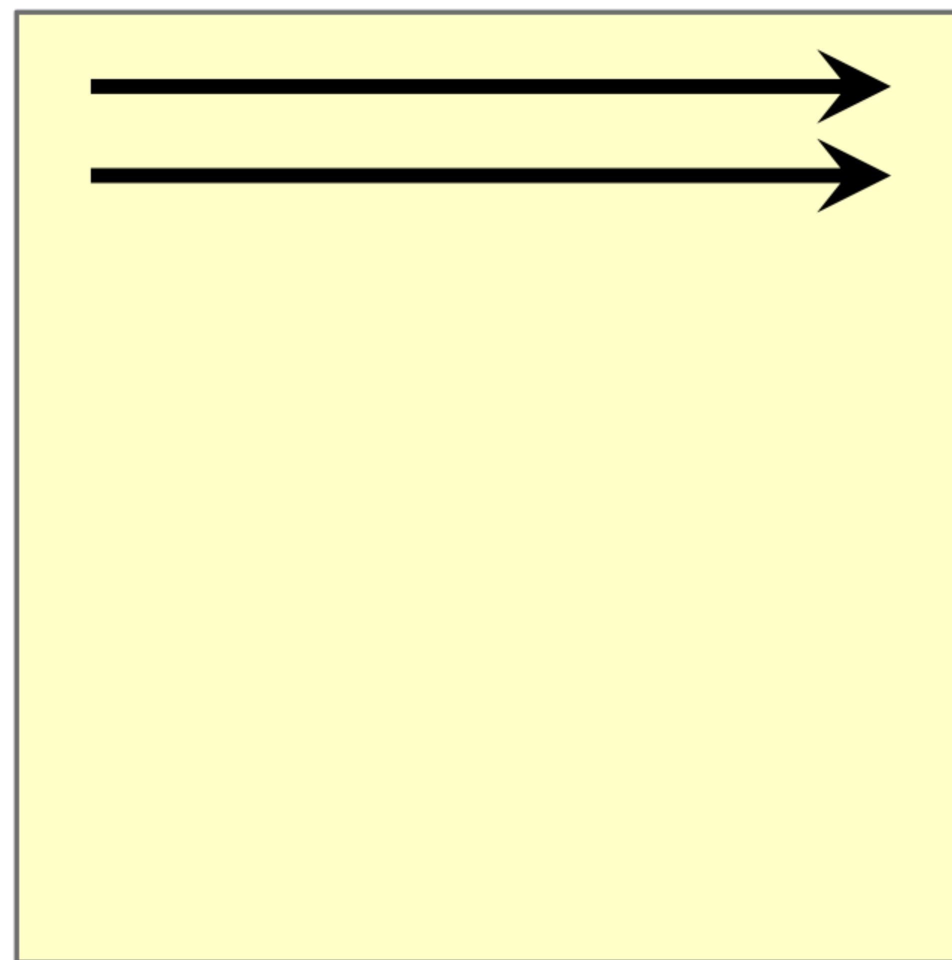
$Q(n) = \Theta(n^2/B)$, since everything fits in cache!

Swapping inner loop order

```
void Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i=0; i < n; I++)
    for (int64_t k=0; k < n; k++)
      for (int64_t j=0; j < n; j++)
        C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```

Assume row major and tall cache





C

B

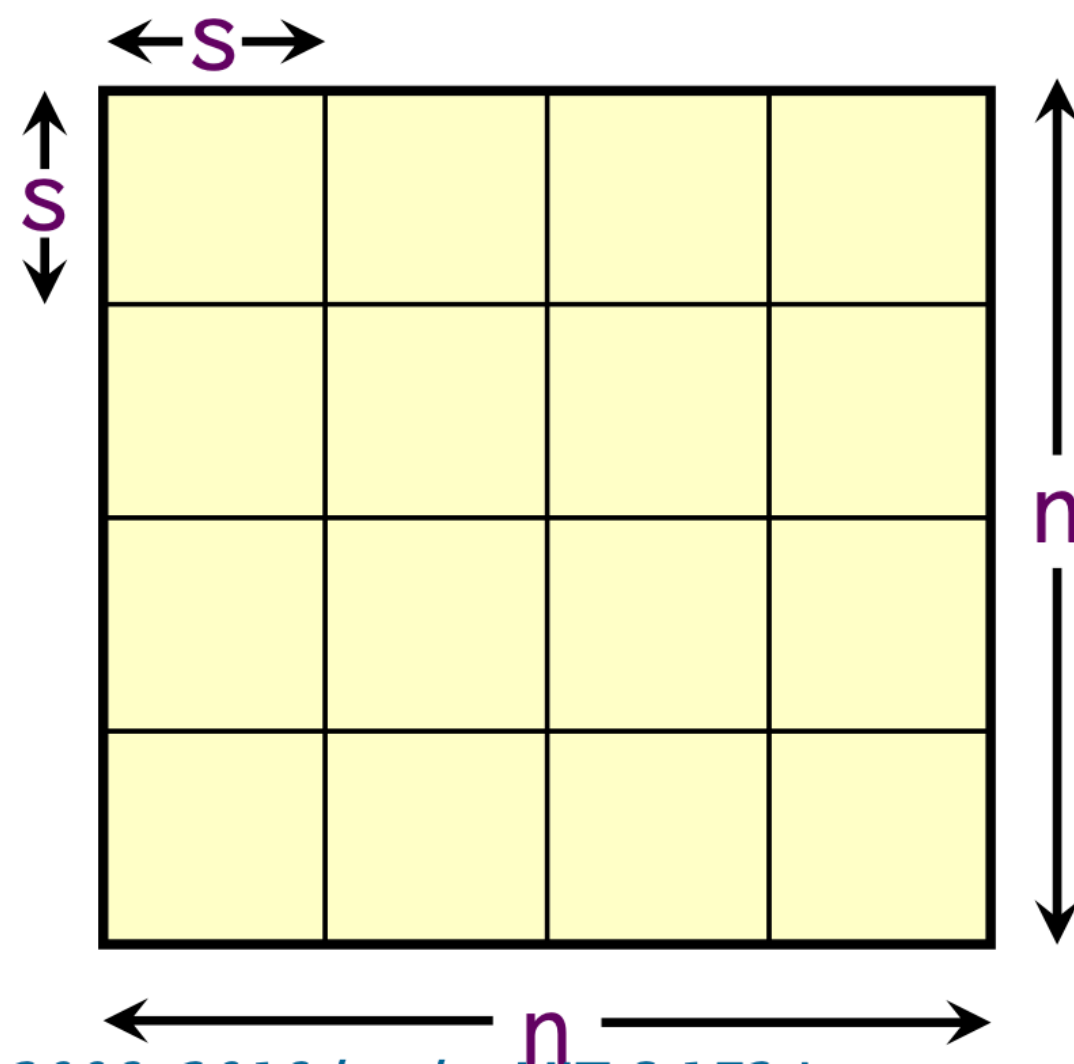
Assume matrix B. Assume LRU.

$Q(n) = n \cdot \Theta(n^2/B) = \Theta(n^3/B)$, since matrix B can exploit spatial locality.

Tiling (aka Blocking)

Tiled (Blocked) matrix multiply

```
void Tiled_Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i1=0; i1<n; i1+=s)
    for (int64_t j1=0; j1<n; j1+=s)
      for (int64_t k1=0; k1<n; k1+=s)
        for (int64_t i=i1; i<i1+s && i<n; i++)
          for (int64_t j=j1; j<j1+s && j<n; j++)
            for (int64_t k=k1; k<k1+s && k<n; k++)
              C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```



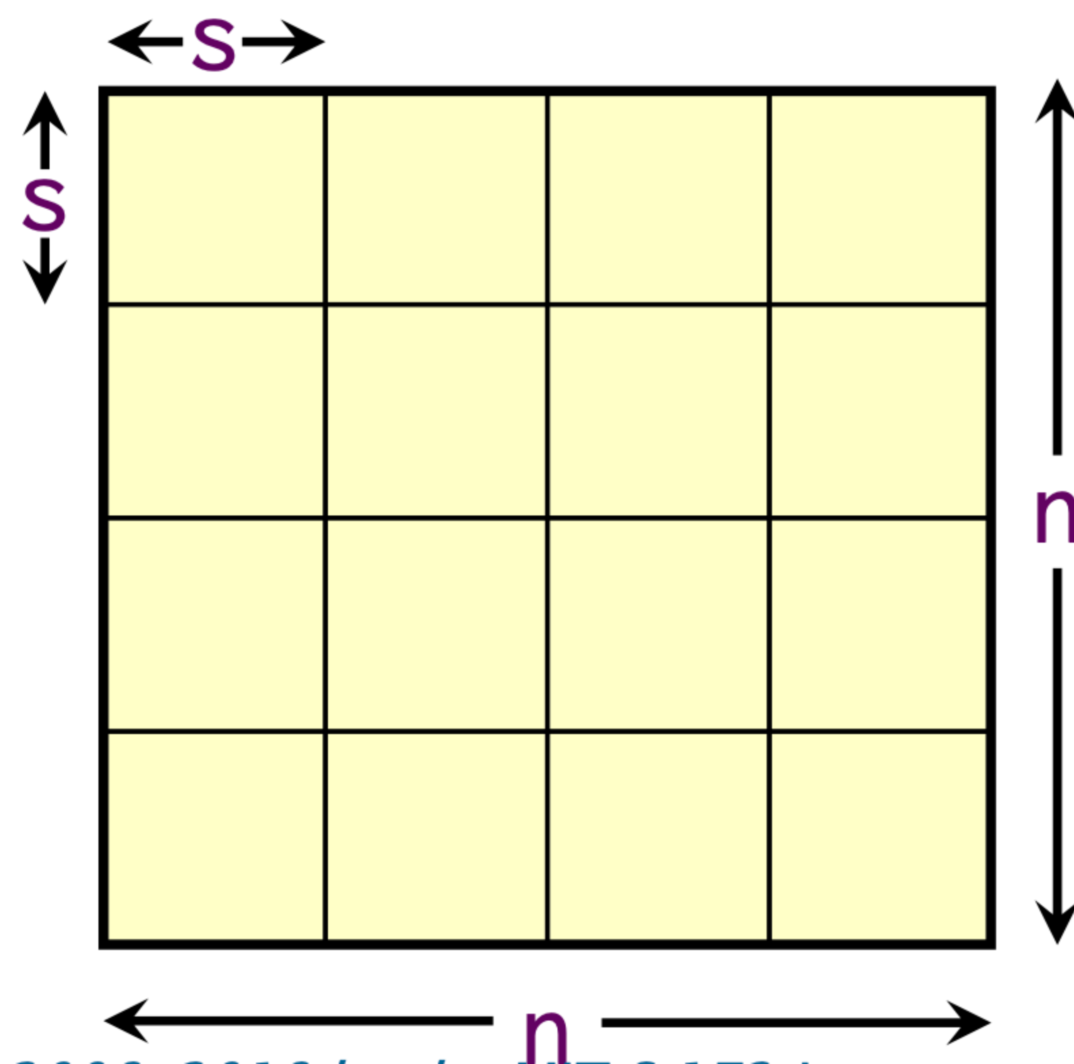
Analysis of work:

$$W(n) = \Theta((n/s)^3(s)^3)$$
$$= \Theta(n^3)$$

Tile size (or block size)

Tiled (Blocked) matrix multiply

```
void Tiled_Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i1=0; i1<n/s; i1+=s)
    for (int64_t j1=0; j1<n; j1+=s)
      for (int64_t k1=0; k1<n; k1+=s)
        for (int64_t i=i1; i<i1+s && i<n; i++)
          for (int64_t j=j1; j<j1+s && j<n; j++)
            for (int64_t k=k1; k<k1+s && k<n; k++)
              C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```



Analysis of cache misses

Tune s so that the submatrices just fit into cache:

$s = \Theta(M^{1/2})$.

Submatrix Caching Lemma implies $\Theta(s^2/B)$ misses per submatrix.

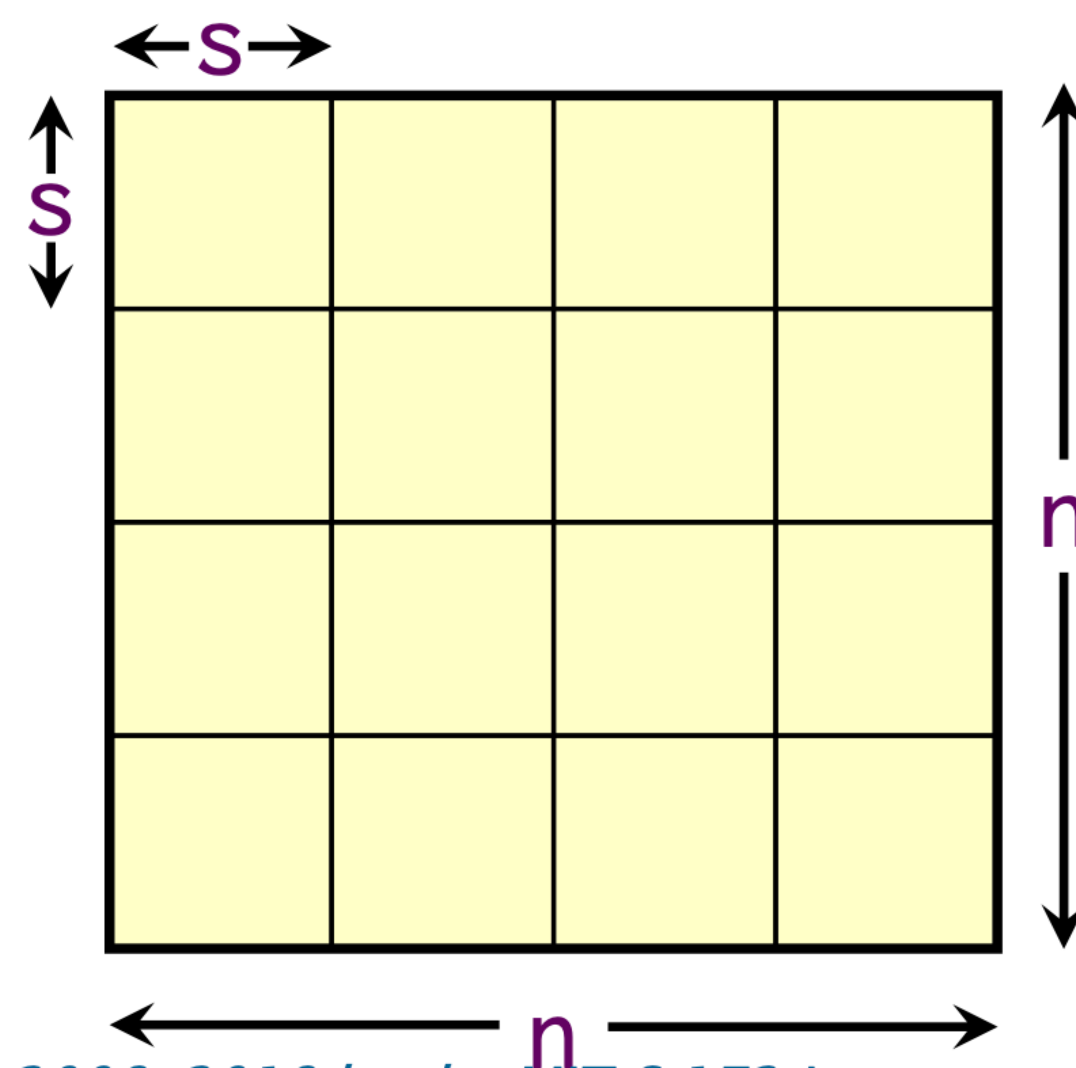
$Q(n) = \Theta((n/s)^3(s^2/B)) = \Theta(n^3/(BM^{1/2}))$.

Optimal
[HK81]

Tiled (Blocked) matrix multiply

```
void Tiled_Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i1=0; i1<n/s; i1+=s)
    for (int64_t j1=0; j1<n; j1+=s)
      for (int64_t k1=0; k1<n; k1+=s)
        for (int64_t i=i1; ... i++)
          for (int64_t j=j ... n; j++)
            for (int64_t k ... k<n; k++)
              C[i*n+j] += ... n+j];
}
```

How?



Analysis of cache misses

Tune s so that the submatrices just fit into cache:

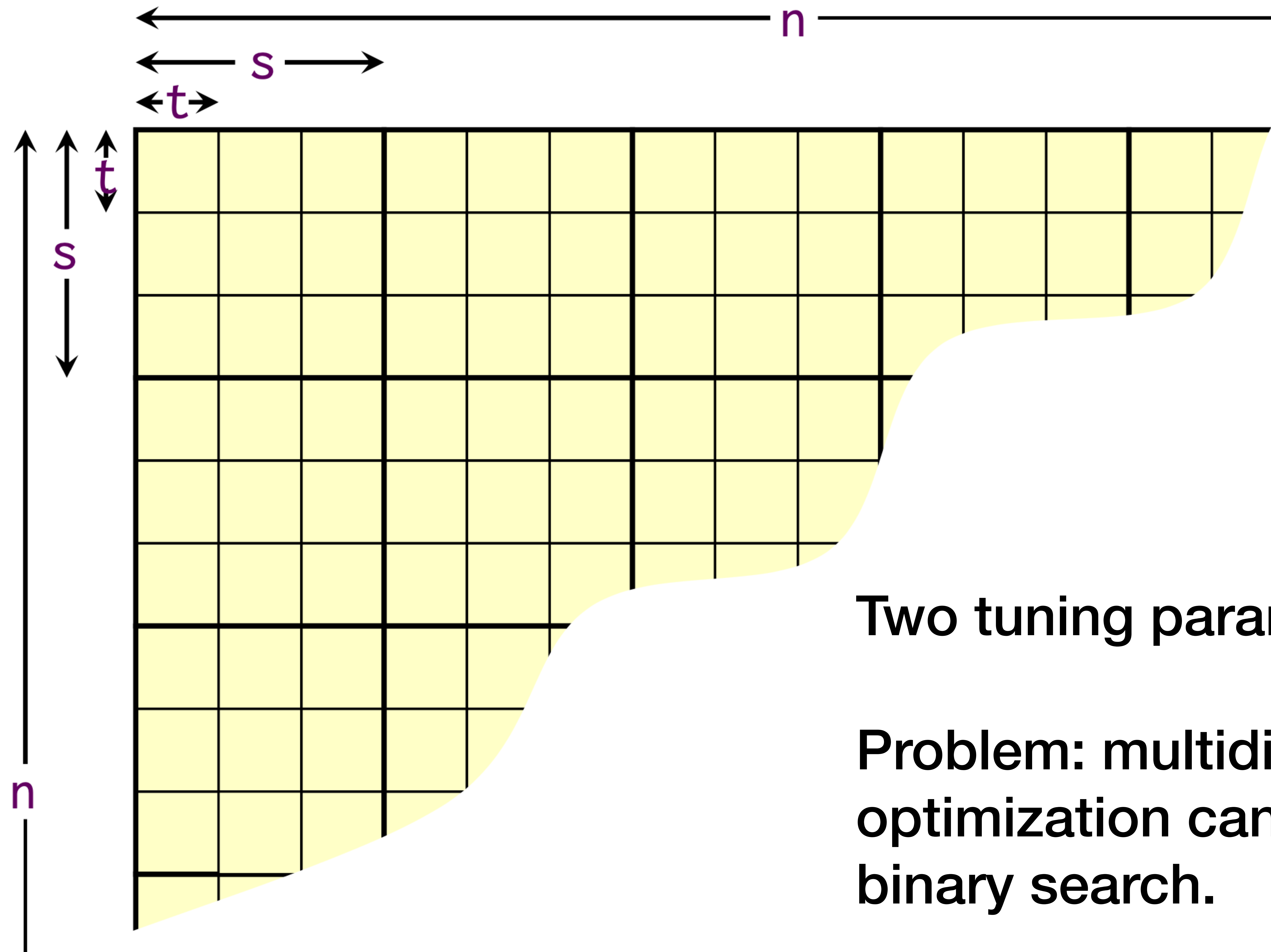
$s = \Theta(M^{1/2})$.

Submatrix Caching Lemma implies $\Theta(s^2/B)$ misses per submatrix.

$Q(n) = \Theta((n/s)^3(s^2/B)) = \Theta(n^3/(BM^{1/2}))$.

Optimal [HK81]

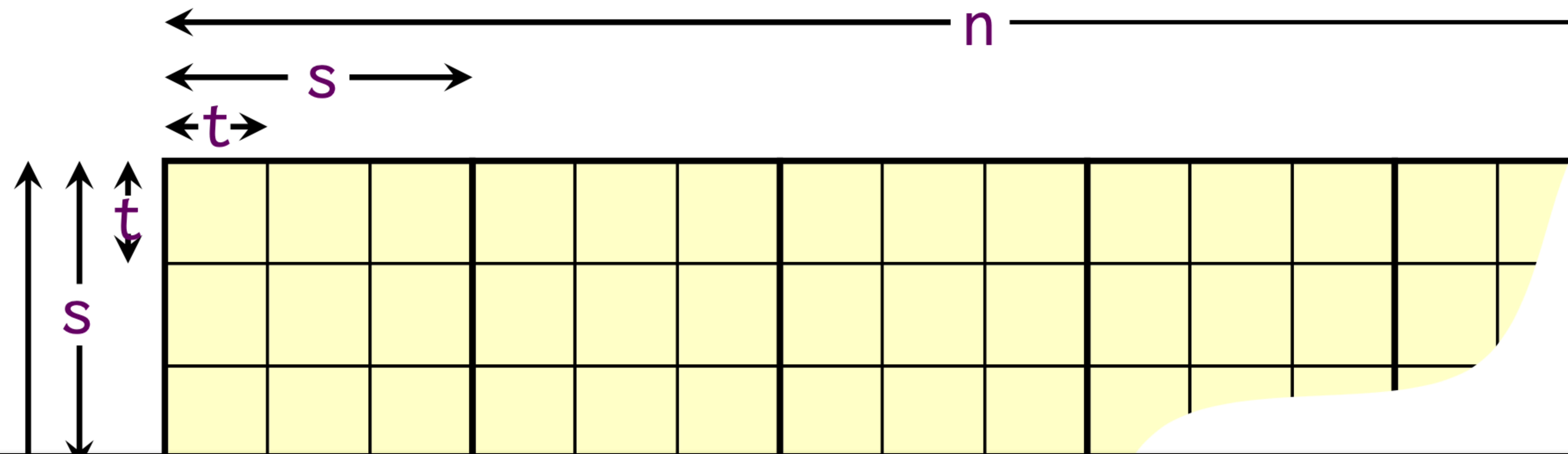
Two-level cache



Two tuning parameters: s and t.

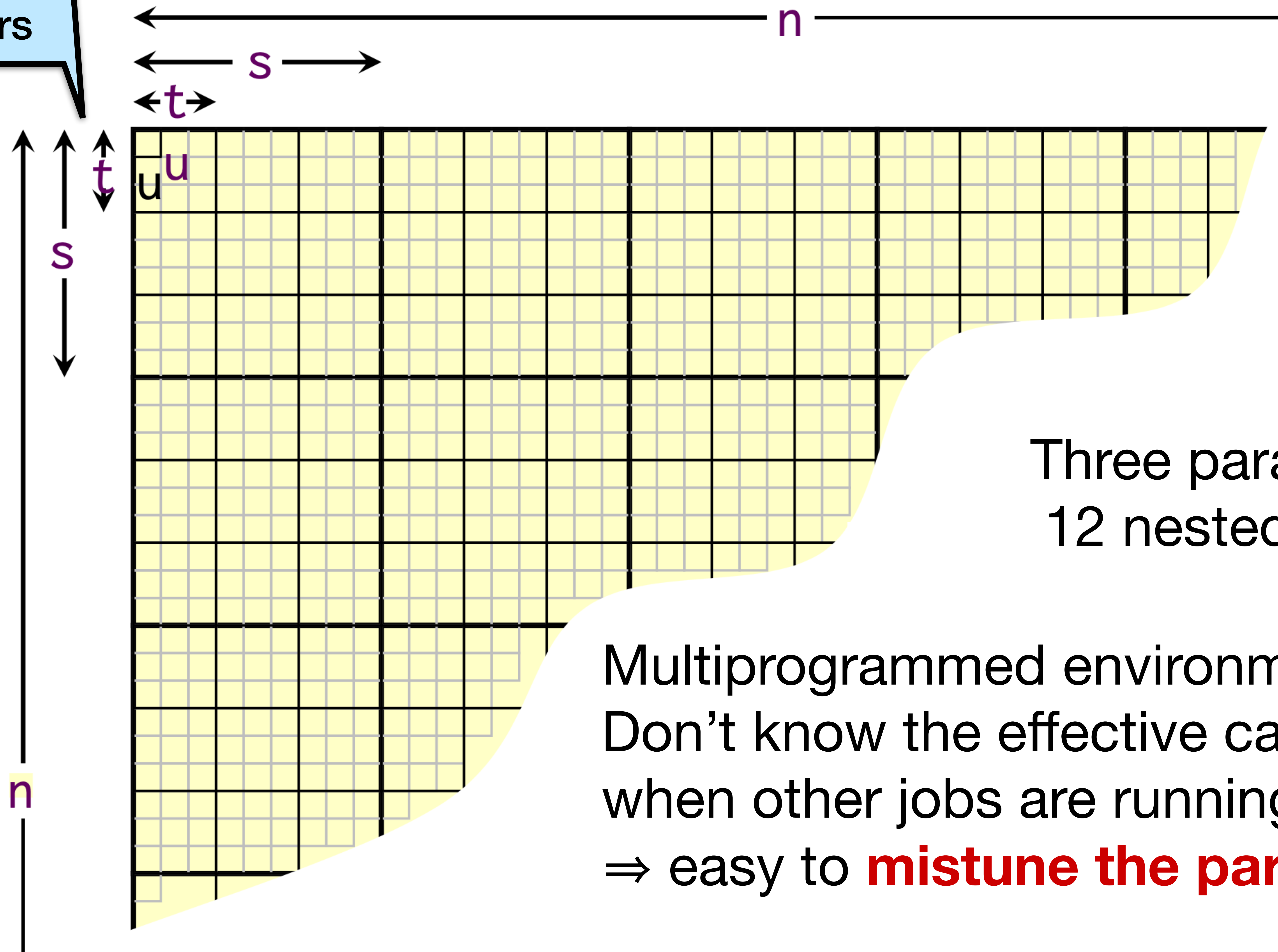
Problem: multidimensional tuning optimization cannot be done with binary search.

Two-level cache




```
void Tiled_Mult2(double *C, double *A, double *B, int64_t n) {
  for (int64_t i2=0; i2<n; i2+=s)
    for (int64_t j2=0; j2<n; j2+=s)
      for (int64_t k2=0; k2<n; k2+=s)
        for (int64_t i1=i2; i1<i2+s && i1<n; i1+=t)
          for (int64_t j1=j2; j1<j2+s && j1<n; j1+=t)
            for (int64_t k1=k2; k1<k2+s && k1<n; k1+=t)
              for (int64_t i=i1; i<i1+s && i<i2+t && i<n; i++)
                for (int64_t j=j1; j<j1+s && j<j2+t && j<n; j++)
                  for (int64_t k=k1; k1<k1+s && k<k2+t && k<n; k++)
                     C[i*n+j] += A[i*n+k] * B[k*n+j];
}
```

Three-level cache



Three parameters ->
12 nested for loops

Multiprogrammed environment:
Don’t know the effective cache size when other jobs are running
⇒ easy to **mistune the parameters**!