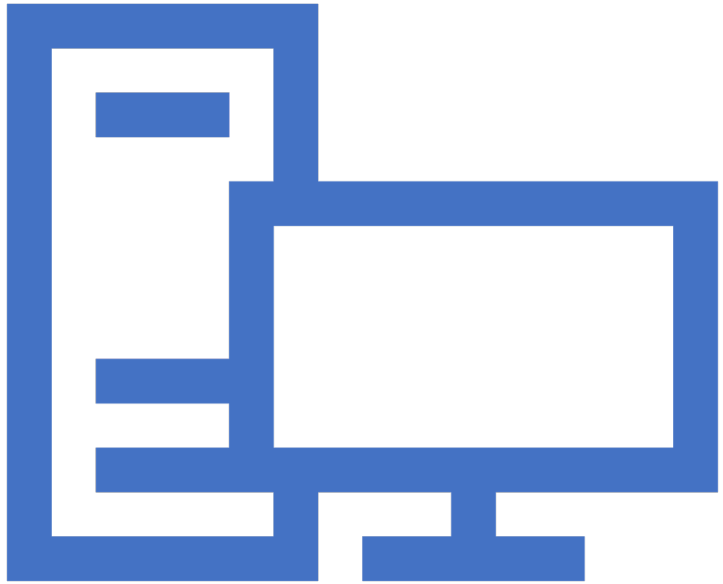
Applied Algorithms

Lecture 5: Cache efficiency



Admin

- Assignment 2 out
- Assignment 1 back soon
- Short(ish) code review Thursday

- Mini-midterm next week

Assignment 1

- Went pretty well
- Lots of great ideas and discussion!
- Everyone had very fast code
  - I thought I'd be near the middle ☹

Reminders about submissions

- Don’t need to beat my time (of course)
- Ties get good points
  - You are motivated to collaborate so long as you wind up in the ballpark of each other
- Times get much lower (~10% or so) during tests
  - One motivation to submit early

Changes

Feedback / late days

- The script should now tell you something about what went wrong
  - Wrong answer/seg fault/bad output/worse time/etc.
  - In errors.txt
- Assignments get 20% off per day late
  - Syllabus updated later today

Testing code

- Guess and check coding
  - Not reflective of the task
  - Debugging is a skill!
  - Loop invariants, etc: can you prove that your code always works?
- But: sometimes it happens
  - Time/results tradeoff
  - Black box issues: works on every input except the secret test

Testing moving forward

- I gave many more tests on Assignment 2
- Can “buy” five more tests for 5 points
  - Minor penalty
  - But, encourages you to exhaust other options

- This will all be updated on the assignment overview handout soon

In-class descriptions

- My goal was to give a “high-level” description in class, and a more detailed description on the assignment itself
- Interpreting a description into code is a part of the class
- Midterms don’t have a leaderboard, etc.

- But I think I’m going to back off this a bit and try to address more details
- I will edit slides over the next couple days

Edit distance

A few important points

- First: strings in C
- “Null-terminated”: just an array that ends in a special character
- ‘\0’
- This character has value 0 if (say) cast to an int

Storing solutions

- I mentioned last time that there are a few good ways
- Probably the easiest: build them bottom-up using arrays
  - Create a simple solution for the base case
  - After both recursive calls, find out the size of both solutions
  - Create a new array large enough to handle both together
  - Copy them over
  - Delete the solutions from the recursive calls
  - Pass the solution to the next level up

Subproblems

- Make sure your subproblems don't overlap!
- Think of it as partitioning both strings

- Hirshberg's works because:

  If ED(X,Y) = c, then if we divide X in half to get X[1,…,n/2] and X[n/2+1,…n], there must be a partition of Y into Y[1,…a] and Y[a+1,…n] such that

  ED(X[1,…,n/2],Y[1,…a]) + ED(X[n/2+1,…,n],Y[a+1,…,n]) = c

  (Proof on board)

Subproblems: example

- Edit distance between cde and aab
- String length/2 = 1

		a	a	b
	0	1	2	3
c	1	1	2	3

		b	a	a
e	1	1	2	3
d	2	2	2	3

Subproblems: example

- Edit distance between cde and aab
- String length/2 = 1

- Most efficient to match c to a (cost 1) and de to ab (cost 2)

- Recurse on Hirshberg's(c,a) and Hirshberg's(de, ab)

		a	a	b
	0	1	2	3
c	1	1	2	3

		b	a	a
e	1	1	2	3
d	2	2	2	3

Two towers

An optimization

- Almost all of the fastest solutions used an algorithmi
- Original idea: generate a table, binary search over it f
- $O\left(n2^{n/2}\right)$ time

0	0000
1.73	0010
3.87	0100
5.6	0110
8.88	1000
10.61	1010
12.75	1100
14.48	1110

31.76	0001
33.49	0011
35.63	0101
37.36	0111
40.64	1001
42.37	1011
44.51	1101
46.24	1111

Idea

- Generate two tables, sort bo
- Then do the binary search
- This improved performance

0	0000
1.63	0010
4.76	0100
5.9	0110
9.74	1000
12.15	1010
15.97	1100
18.72	1110

21.78	0001
24.83	0011
36.02	0101
37.01	0111
40.56	1001
40.84	1011
43.92	1101
49.26	1111

0	0000
1.73	0010
3.87	0100
5.6	0110
8.88	1000
10.61	1010
12.75	1100
14.48	1110

31.76	0001
33.49	0011
35.63	0101
37.36	0111
40.64	1001
42.37	1011
44.51	1101
46.24	1111

Why?

- Are the asymptotics better?
  - Nope
- Constants?
  - No, we're doing the same work
  - (We'll discuss Thursday: can avoid the searches entirely. But even implementations that kept binary searches were sped up)

The answer

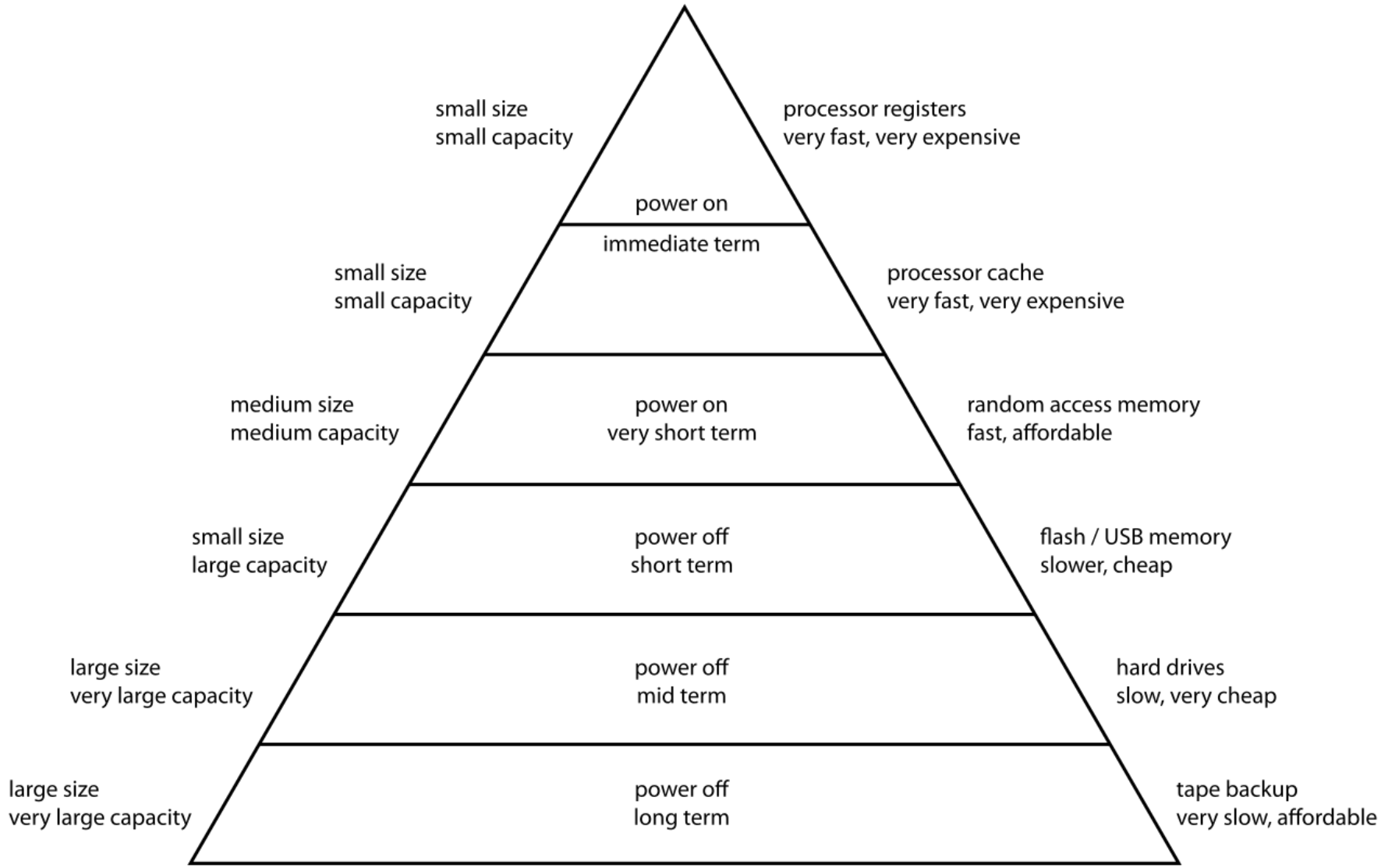
- Cache efficiency!

Cache

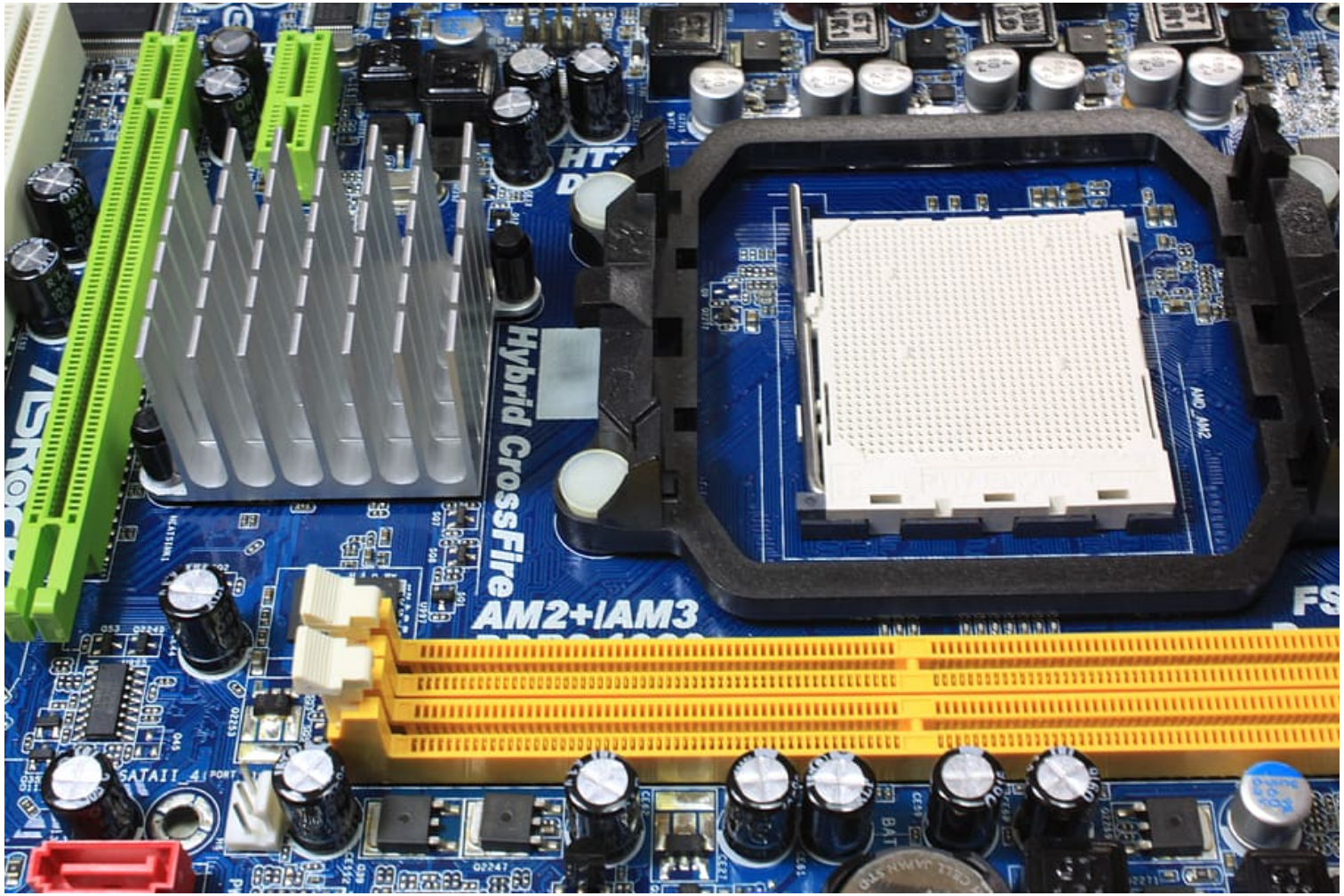
- Your CPU needs to fetch data before it can process it
- Your computer would never run if it ran off the hard drive
- Caching to the rescue!

- Fast memory close to the processor, to minimize time spent reading data

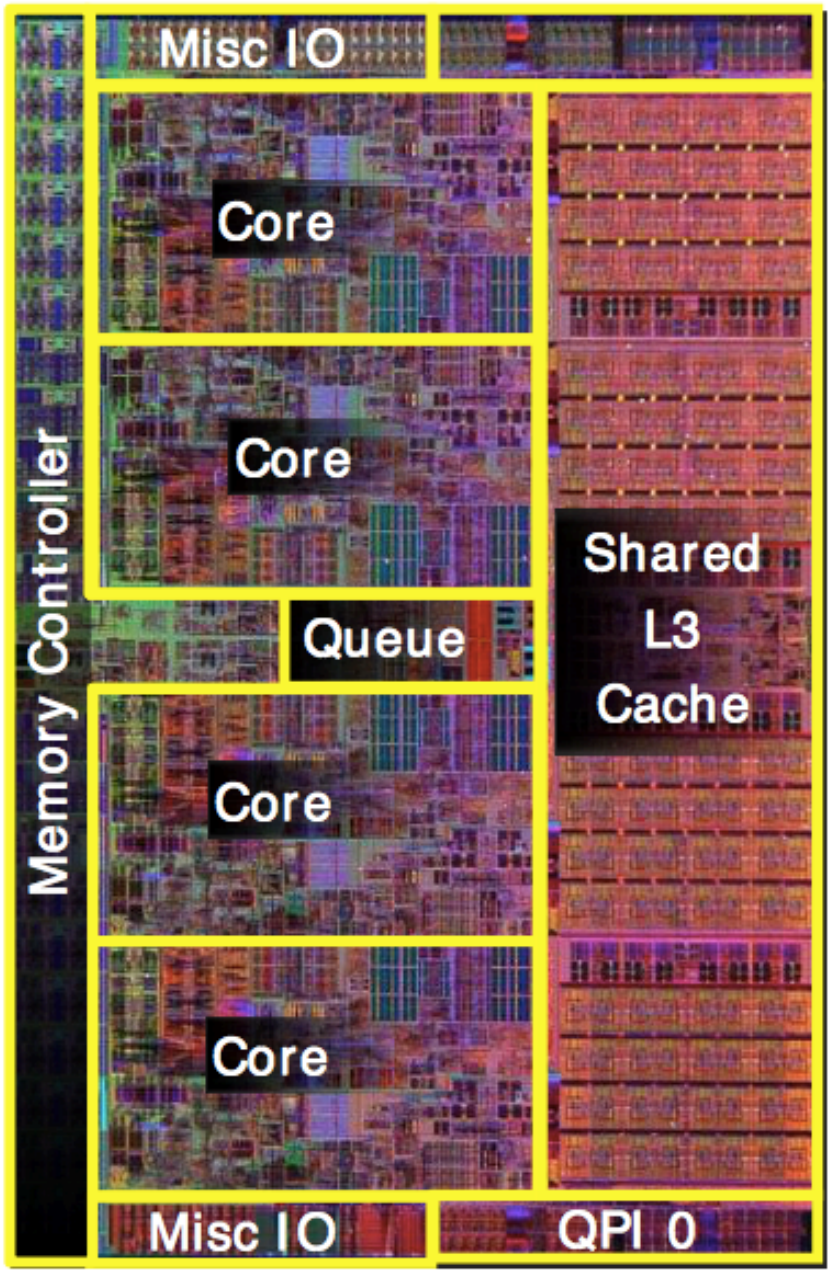
Computer Memory Hierarchy



RAM: pretty big and slow

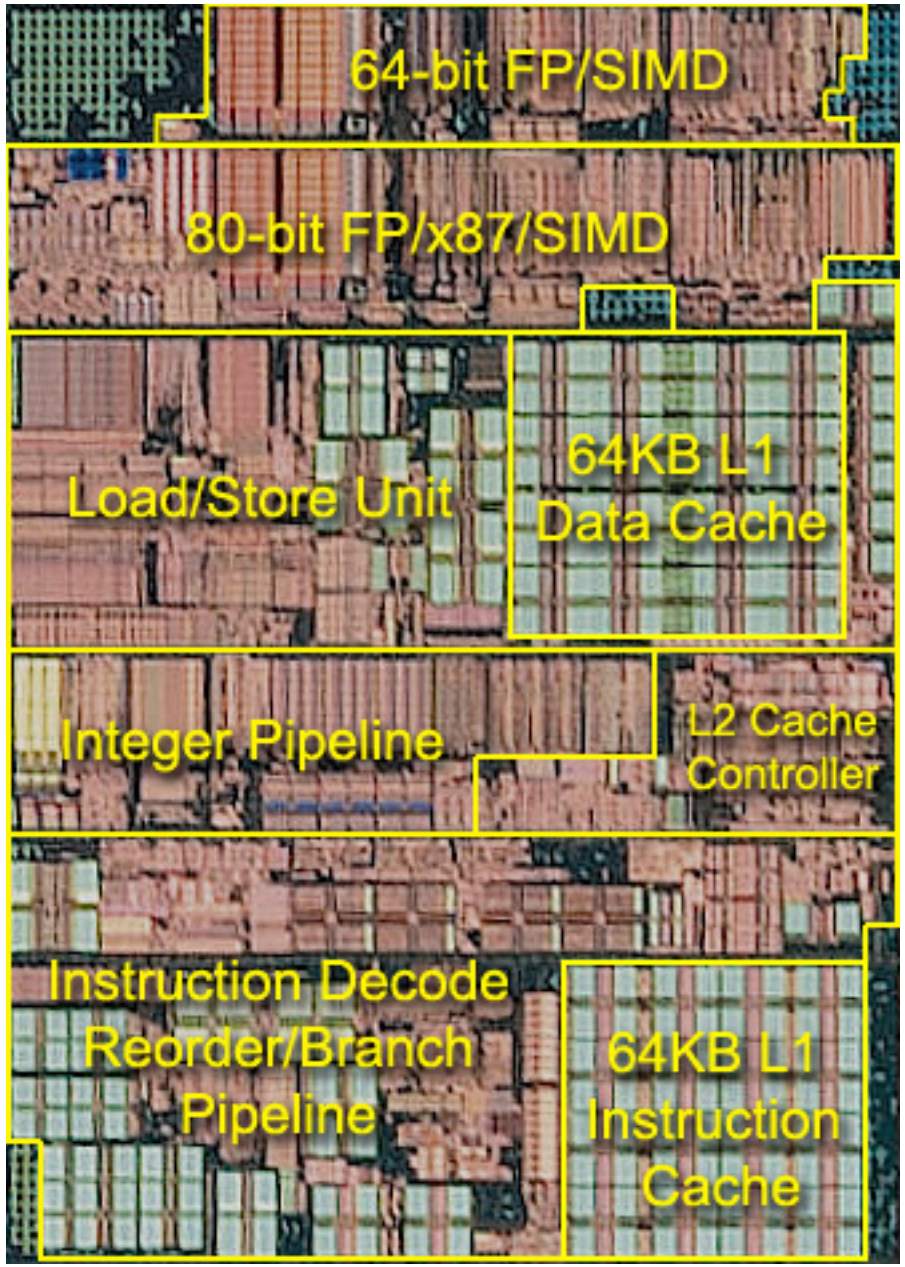


L3/L2 Cache



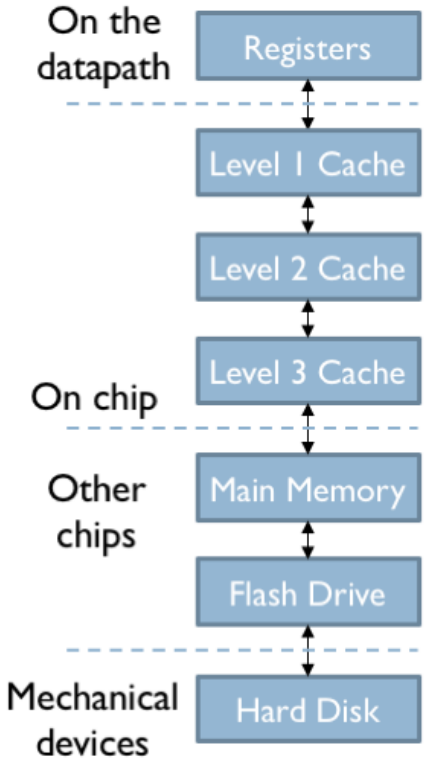
QPI: Intel® QuickPath Interconnect (Intel® QPI)

L1 Cache


64-bit FP/SIMD
80-bit FP/x87/SIMD
Load/Store Unit
64KB L1
Data Cache
Integer Pipeline
L2 Cache
Controller
Instruction Decode
Reorder/Branch
Pipeline
64KB L1
Instruction
Cache

A Typical Memory Hierarchy

- Everything is a cache for something else…



Access time	Capacity	Managed By
1 cycle	1 KB	Software/Compiler
2-4 cycles	32 KB	Hardware
10 cycles	256 KB	Hardware
40 cycles	10 MB	Hardware
200 cycles	10 GB	Software/OS
10-100us	100 GB	Software/OS
10ms	1 TB	Software/OS

What goes in the fast memory?

- Can’t fit much
- Who controls it?

How do these work?

- Your computer decides what is stored where
  - It's very good at it
- Also maintains consistency, etc.

- How to decide what data should be stored in the valuable L1/L2 caches?
  - Computer knows nothing about your program or your data
  - Most recently used data! It's likely to be accessed again

One note:

- This is a massive simplification on modern machines
- TLBs, shared memories, new hardware
- Modern machines have many levels of cache
- Not always even clear that “level” is meaningful

Getting back to two towers

0	0000
1.63	0010
4.76	0100
5.9	0110
9.74	1000
12.15	1010
15.97	1100
18.72	1110

21.78	0001
24.83	0011
36.02	0101
37.01	0111
40.56	1001
40.84	1011
43.92	1101
49.26	1111

0	0000
1.73	0010
3.87	0100
5.6	0110
8.88	1000
10.61	1010
12.75	1100
14.48	1110

31.76	0001
33.49	0011
35.63	0101
37.36	0111
40.64	1001
42.37	1011
44.51	1101
46.24	1111

Getting back to two towers

0	0000
1.63	0010
4.76	0100
5.9	0110
9.74	1000
12.15	1010
15.97	1100
18.72	1110

21.78	0001
24.83	0011
36.02	0101
37.01	0111
40.56	1001
40.84	1011
43.92	1101
49.26	1111

0	0000
1.73	0010
3.87	0100
5.6	0110
8.88	1000
10.61	1010
12.75	1100
14.48	1110

31.76	0001
33.49	0011
35.63	0101
37.36	0111
40.64	1001
42.37	1011
44.51	1101
46.24	1111

Getting back to two towers

0	0000
1.63	0010
4.76	0100
5.9	0110
9.74	1000
12.15	1010
15.97	1100
18.72	1110

21.78	0001
24.83	0011
36.02	0101
37.01	0111
40.56	1001
40.84	1011
43.92	1101
49.26	1111

0	0000
1.73	0010
3.87	0100
5.6	0110
8.88	1000
10.61	1010
12.75	1100
14.48	1110

31.76	0001
33.49	0011
35.63	0101
37.36	0111
40.64	1001
42.37	1011
44.51	1101
46.24	1111

Getting back to two towers

0	0000
1.63	0010
4.76	0100
5.9	0110
9.74	1000
12.15	1010
15.97	1100
18.72	1110

21.78	0001
24.83	0011
36.02	0101
37.01	0111
40.56	1001
40.84	1011
43.92	1101
49.26	1111

0	0000
1.73	0010
3.87	0100
5.6	0110
8.88	1000
10.61	1010
12.75	1100
14.48	1110

31.76	0001
33.49	0011
35.63	0101
37.36	0111
40.64	1001
42.37	1011
44.51	1101
46.24	1111

Getting back to two towers

0	0000
1.63	0010
4.76	0100
5.9	0110
9.74	1000
12.15	1010
15.97	1100
18.72	1110

21.78	0001
24.83	0011
36.02	0101
37.01	0111
40.56	1001
40.84	1011
43.92	1101
49.26	1111

0	0000
1.73	0010
3.87	0100
5.6	0110
8.88	1000
10.61	1010
12.75	1100
14.48	1110

31.76	0001
33.49	0011
35.63	0101
37.36	0111
40.64	1001
42.37	1011
44.51	1101
46.24	1111

Getting back to two towers

0	0000
1.63	0010
4.76	0100
5.9	0110
9.74	1000
12.15	1010
15.97	1100
18.72	1110

21.78	0001
24.83	0011
36.02	0101
37.01	0111
40.56	1001
40.84	1011
43.92	1101
49.26	1111

0	0000
1.73	0010
3.87	0100
5.6	0110
8.88	1000
10.61	1010
12.75	1100
14.48	1110

31.76	0001
33.49	0011
35.63	0101
37.36	0111
40.64	1001
42.37	1011
44.51	1101
46.24	1111

Getting back to two towers

0	0000
1.63	0010
4.76	0100
5.9	0110
9.74	1000
12.15	1010
15.97	1100
18.72	1110

21.78	0001
24.83	0011
36.02	0101
37.01	0111
40.56	1001
40.84	1011
43.92	1101
49.26	1111

0	0000
1.73	0010
3.87	0100
5.6	0110
8.88	1000
10.61	1010
12.75	1100
14.48	1110

31.76	0001
33.49	0011
35.63	0101
37.36	0111
40.64	1001
42.37	1011
44.51	1101
46.24	1111

Getting back to two towers

0	0000
1.63	0010
4.76	0100
5.9	0110
9.74	1000
12.15	1010
15.97	1100
18.72	1110

21.78	0001
24.83	0011
36.02	0101
37.01	0111
40.56	1001
40.84	1011
43.92	1101
49.26	1111

0	0000
1.73	0010
3.87	0100
5.6	0110
8.88	1000
10.61	1010
12.75	1100
14.48	1110

31.76	0001
33.49	0011
35.63	0101
37.36	0111
40.64	1001
42.37	1011
44.51	1101
46.24	1111

Getting back to two towers

0	0000
1.63	0010
4.76	0100
5.9	0110
9.74	1000
12.15	1010
15.97	1100
18.72	1110

21.78	0001
24.83	0011
36.02	0101
37.01	0111
40.56	1001
40.84	1011
43.92	1101
49.26	1111

0	0000
1.73	0010
3.87	0100
5.6	0110
8.88	1000
10.61	1010
12.75	1100
14.48	1110

31.76	0001
33.49	0011
35.63	0101
37.36	0111
40.64	1001
42.37	1011
44.51	1101
46.24	1111

Getting back to two towers

0	0000
1.63	0010
4.76	0100
5.9	0110
9.74	1000
12.15	1010
15.97	1100
18.72	1110

21.78	0001
24.83	0011
36.02	0101
37.01	0111
40.56	1001
40.84	1011
43.92	1101
49.26	1111

0	0000
1.73	0010
3.87	0100
5.6	0110
8.88	1000
10.61	1010
12.75	1100
14.48	1110

31.76	0001
33.49	0011
35.63	0101
37.36	0111
40.64	1001
42.37	1011
44.51	1101
46.24	1111

Getting back to two towers

- Your table probably did not fit in cache---it was stored in RAM
- Before: had to access RAM almost every step (~21 steps) during the binary search

- After sorting: only have to access RAM ~once!

- If you sort both tables, the binary search has an almost-negligible cost

How cache works

- Data is moved around in large(ish) “chunks”
  - Idea: nearby data is also, likely, useful
- Called a “cache line”
- Aligned (this is important for fine tuning, but we will mostly ignore)
- L2/L1 cache line: ~64 bytes

What can we do with this?

- Can we model it?

- Can we predict how well an algorithm will perform, or is it purely experimental?

Simplifying assumptions

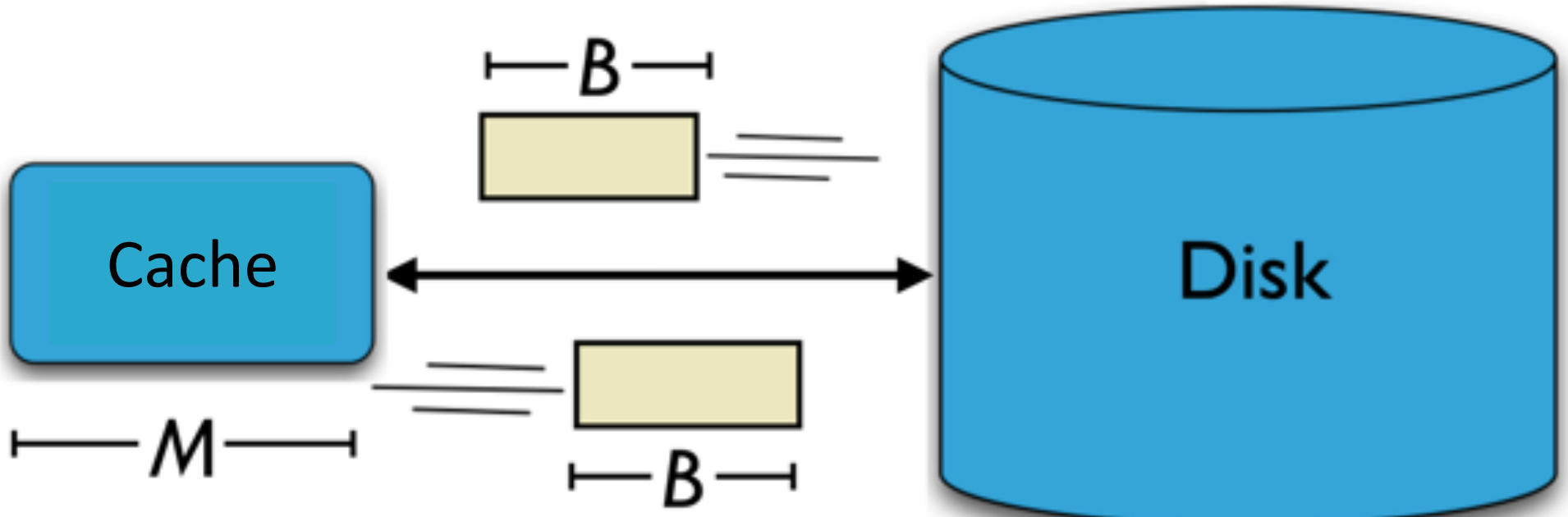
- Just look at *one* level of the memory hierarchy
  - It's very common that one is the bottleneck
- Assume that the computer always keeps the right things in memory
  - Pretty close to practice
- Just focus on moving items around
  - Ignore computation time (can analyze separately)

External Memory Model

Also called “DAM” or “Disk Access Model”

Components

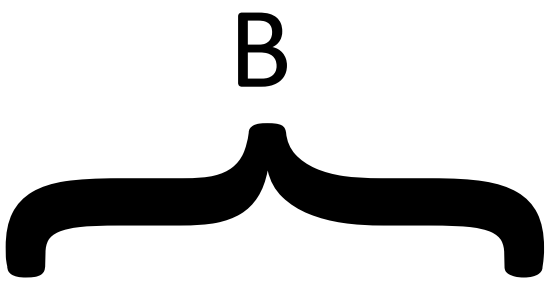
- We have a small cache (of size M)
- Rest of memory is unbounded size. Let's call it the "disk."
  - It's not always a hard disk (oftentimes it's the RAM), but "disk" emphasizes its size and slowness
- Can only compute on data that is in cache
  - But computation is free
- Move data around in blocks of size B
- Each movement is called an "I/O"
  - Some people call it a "disk access" or a "cache miss"
- Analyze using big-O notation, parameterized by M and B


B
Cache
Disk
M
B

Components

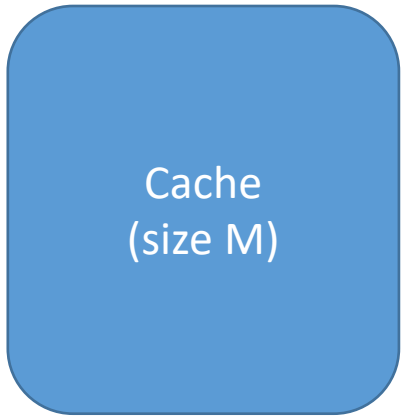
How do we read in an item from disk? (say 78)

Cache
(size M)



1	3	4	3	78	2	99	47	53	61

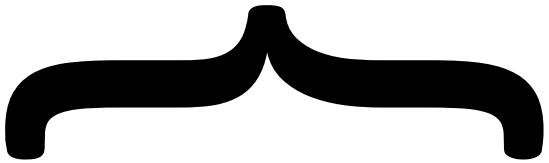
Components



How do we read in an item from disk? (say 78)

Costs 1 to bring it into memory

Now we can compute using 78! And we get 3 and 2 if they're useful.



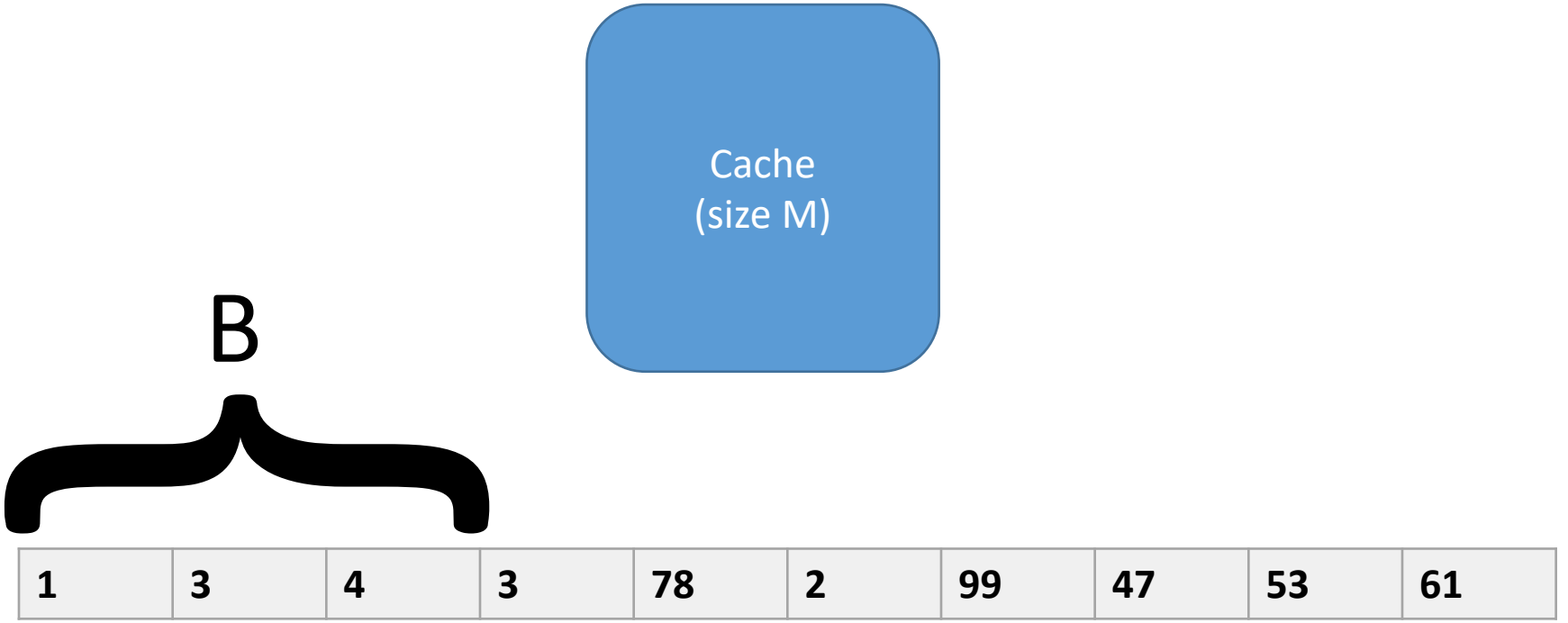
1	3	4	3	78	2	99	47	53	61

Simple example

- How many I/Os does it take to find the minimum element in a list?
- Well, how can we move memory around to do it?

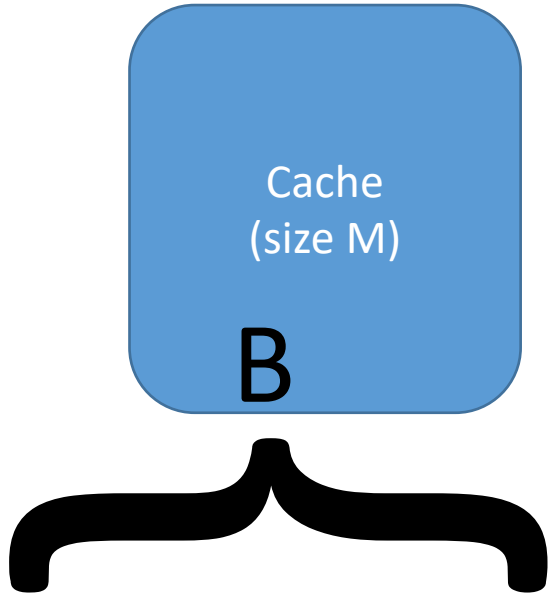
Simple example

- How many I/Os does it take to find the minimum element in a list?
- Well, how can we move memory around to do it?



Simple example

- How many I/Os does it take to find the minimum element in a list?
- Well, how can we move memory around to do it?



1	3	4	3	78	2	99	47	53	61

Finding the minimum

- How many I/Os does it take?

- Why doesn’t M come into the picture? Does any value of M work?

Binary search

- Let’s assume you do just one binary search. How many I/Os does it take?

- How do we find out how long binary search takes when counting operations normally?
  - Each operation divides the size of the input in two
  - We stop, at the latest, when we get to a range of size 1

Binary search

- When do we stop?
  - When we get to a range of size B
- So if h is the height of our recursion, we have $n/2^h = B$
- Solving, $h = \log_2 n/B$
- Not very good! (Although better than 1 I/O for EVERY operation)

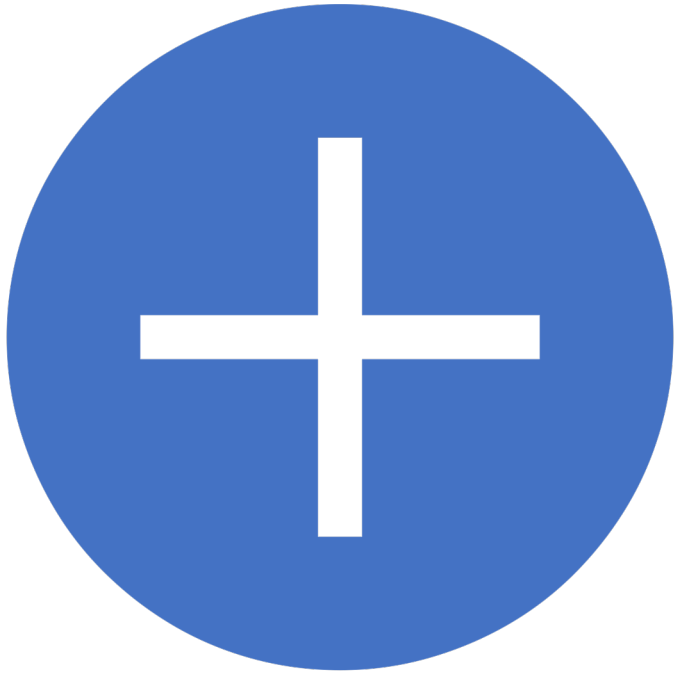
Quicksort

- How many I/Os does it take to partition?
- Basically a linear scan! $O(n/B)$

- At each "level" of quicksort we need to partition all array items
- How many levels?
  - Divide by 2 until size M
  - $O(\log_2 n/M)$

- Overall I/Os: $O((n/B)\log(n/M))$

Matrix multiplication



The problem

- Given two $n \times n$ matrices A, B
- Want to compute their product C:
- $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$

- Example:

1	2
8	-1

$\times$

2	3
-2	7

$=$

-2	17
18	17

How do we do this?

```
for i = 1 to n
        for j = 1 to n
                for k = 1 to n
                        C[i][j] = A[i][k] + B[k][j]
```

How many I/Os does it take?

Every addition requires an I/O for B: $O(n^3)$

Can we improve this?

```
for i = 1 to n
        for k = 1 to n
                for j = 1 to n
                        C[i][j] = A[i][k] + B[k][j]
```

How many I/Os does it take?

Inner loop gets B additions per I/O: $O(n^3)/B$

I am given two functions for finding the product of two matrices:

```
void MultiplyMatrices_1(int **a, int **b, int **c, int n){
     for (int i = 0; i < n; i++)
         for (int j = 0; j < n; j++)
             for (int k = 0; k < n; k++)
                 c[i][j] = c[i][j] + a[i][k]*b[k][j];
 }

void MultiplyMatrices_2(int **a, int **b, int **c, int n){
     for (int i = 0; i < n; i++)
         for (int k = 0; k < n; k++)
             for (int j = 0; j < n; j++)
                 c[i][j] = c[i][j] + a[i][k]*b[k][j];
}
```

I ran and profiled two executables using `gprof`, each with identical code except for this function. The second of these is significantly (about 5 times) faster for matrices of size 2048 x 2048. Any ideas as to why?

c algorithm matrix matrix-multiplication gprof

share improve this question

edited Sep 13 '11 at 2:47
templatetypedef
295k ● 80 ● 725 ● 933

asked Sep 13 '11 at 0:29
kevlar1818
2,639 ● 4 ● 19 ● 39

add a comment

nswers

active oldest votes

I believe that what you're looking at is the effects of locality of reference in the computer's memory

Can we improve further??

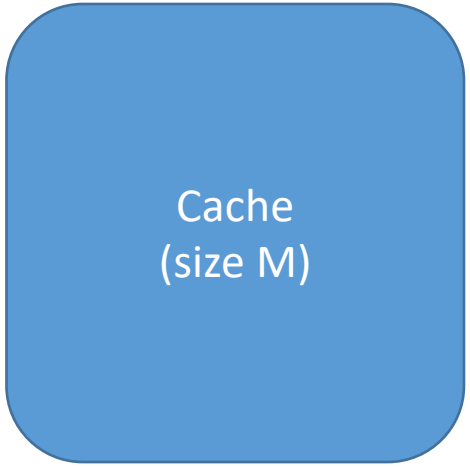
- Our goal is to perform all $O(n^3)$ multiplications with the fewest possible I/Os.

- Restated: our goal is to have each I/O result in the maximum possible number of multiplications.

- What is the most efficient way we can use our cache???

Can we improve further??

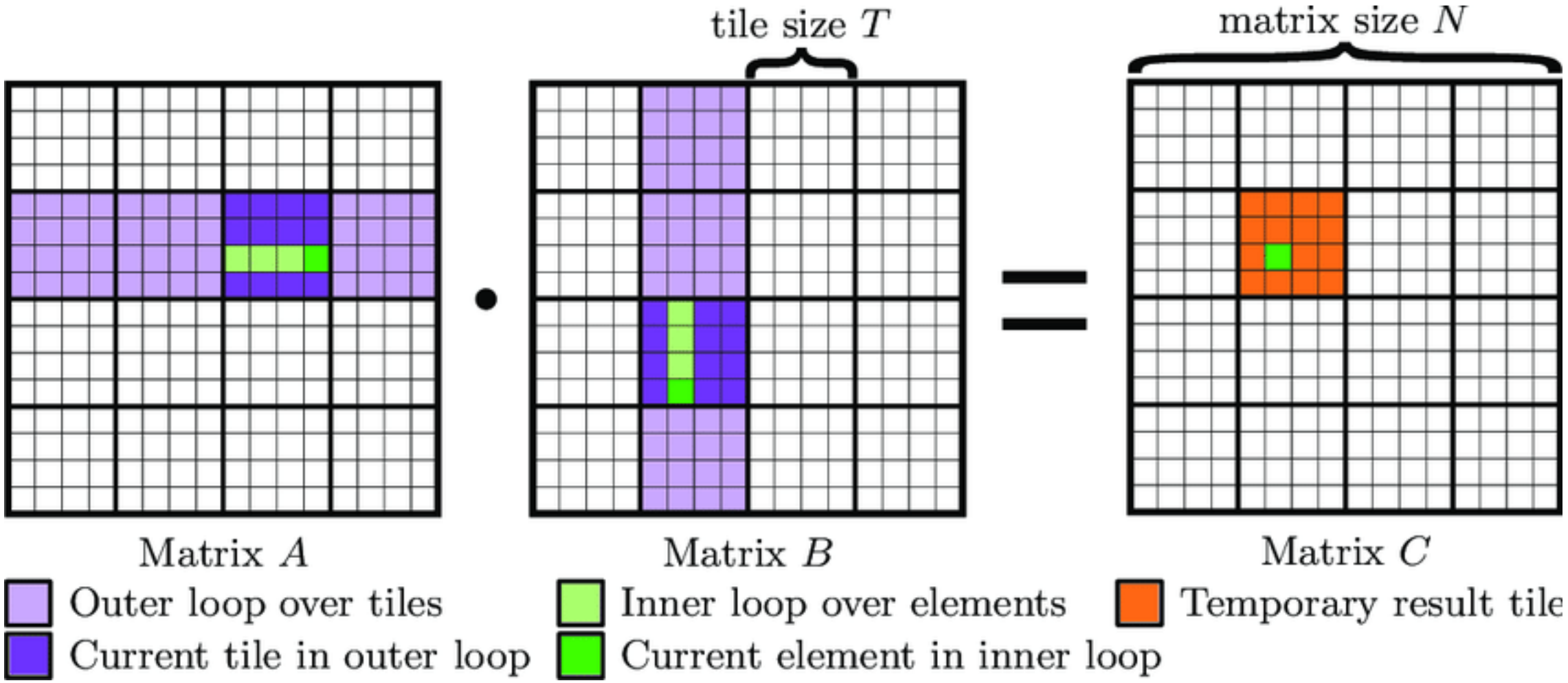
- If we have three matrices, each of total size < M/3, we can fit them all in cache and multiply them
- How many I/Os does this take?
- How many multiplications do we get out of it?



How can we take advantage of this?

- Can we partition matrix multiplication into a series of multiplications of matrices of size at most M/3?

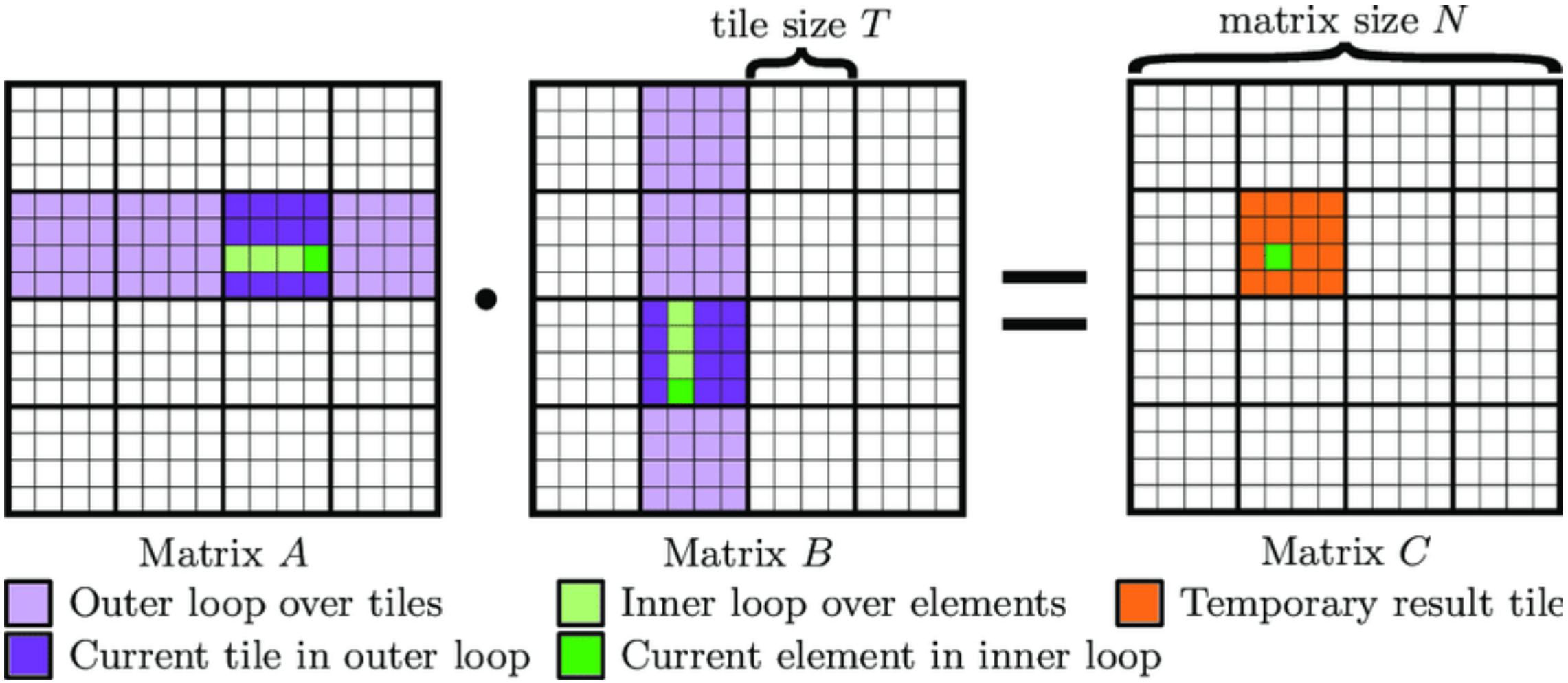
Blocking (tiling)



Blocking (tiling)

- Partition each matrix into ”tiles” (ideally, should fit in memory)
- Outer loop: perform a normal matrix multiplication of two $n/\sqrt{M} \times n/\sqrt{M}$ matrices
- Inner loop: for each tile, multiply the matrices as usual

Blocking (tiling)



Analysis

- How many tiles do we need to multiply?

- How many I/Os does it take to multiply two tiles and store the result?

What about sorting?

- Quicksort: $O((n/B)\log(n/M))$
- Can we do better?
- What does the cache of size M get us?

What about trees?

- What is binary search wasting?
- How can a tree structure resolve this?

Carrying back to practice

- How do we implement this? Do we plug in our best guess for M? What level of the hierarchy do we use?
- Cache improvement: predicted by model
- Constants: experimentation