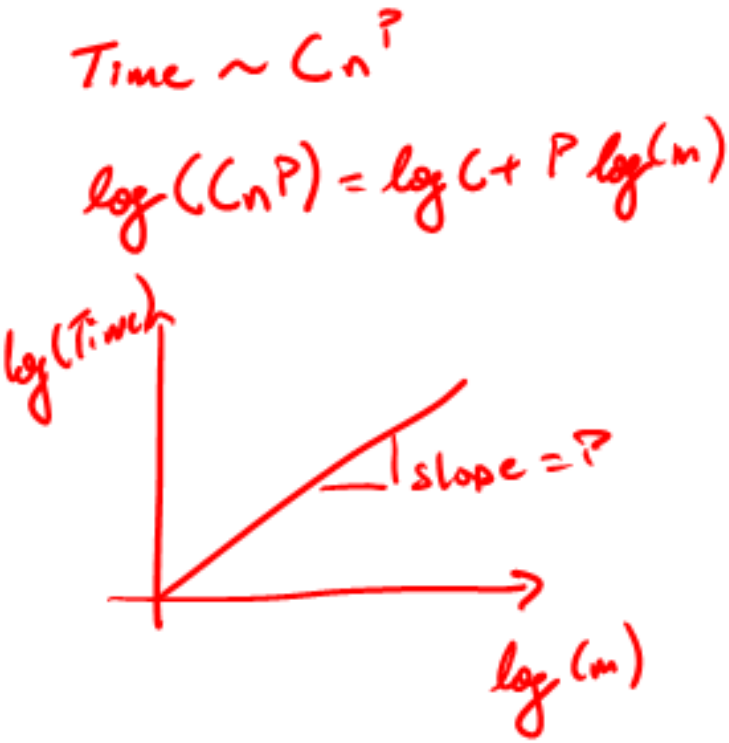
Lecture 2:
Single processor architecture and memory



David Bindel

30 Aug 2011

Teaser

What will this plot look like?

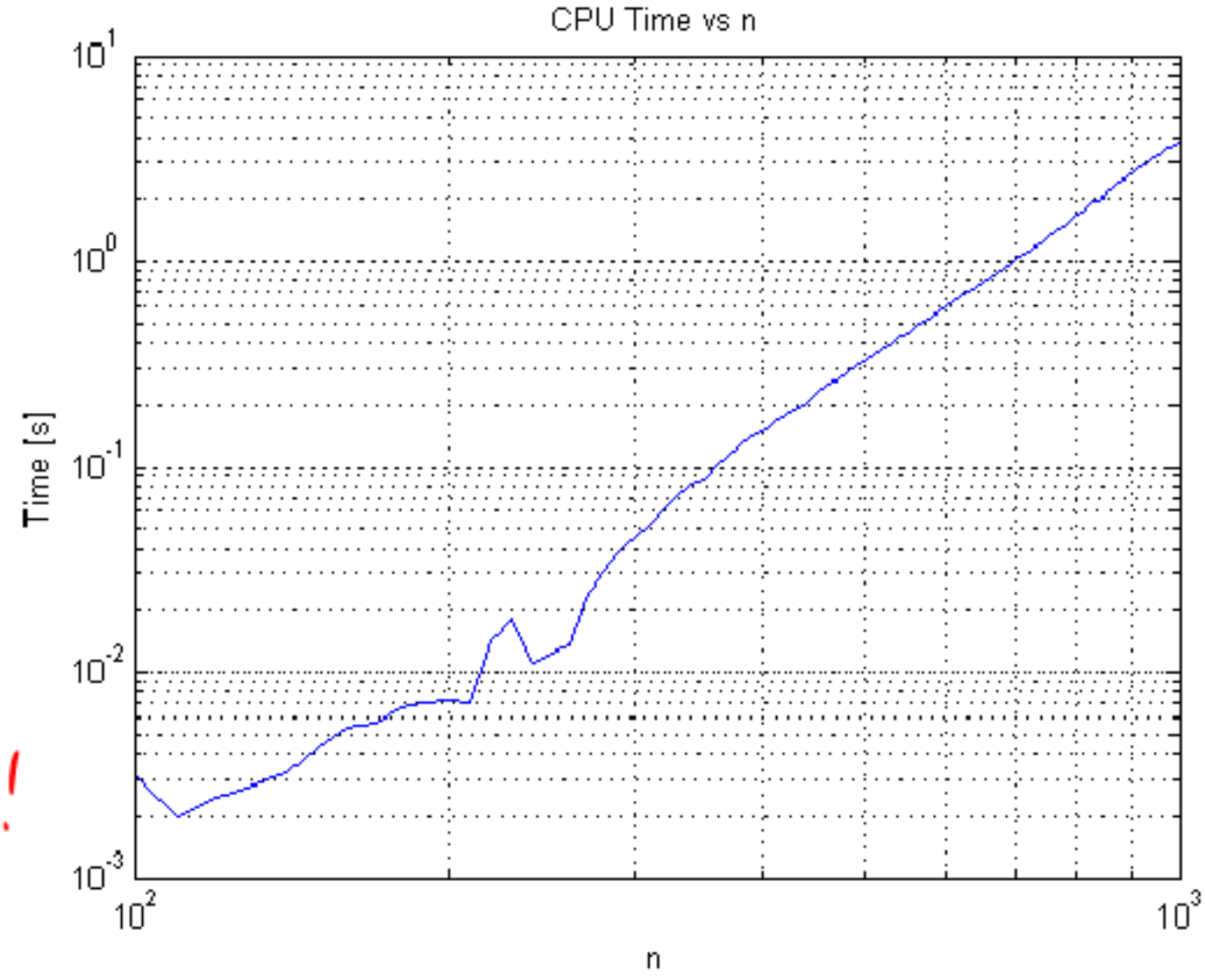
```
% builds the identity matrix I_n a bunch of times
for n = 100:10:1000
  tic; % start timer
  A = []; % empty array
  for i = 1:n
    A(i,i) = 1; % keep adding a 1
  end
  times(n) = toc; % stop timer & store in vector "times"
end
ns = 100:10:1000;
loglog(ns, times(ns)); % plot
```

adding this seems like O(n)

O(n) x n times ⇒ $O(n^2)$

BUT, must copy prev I_{n-1} ⇒ $O(n^3)$

recopying

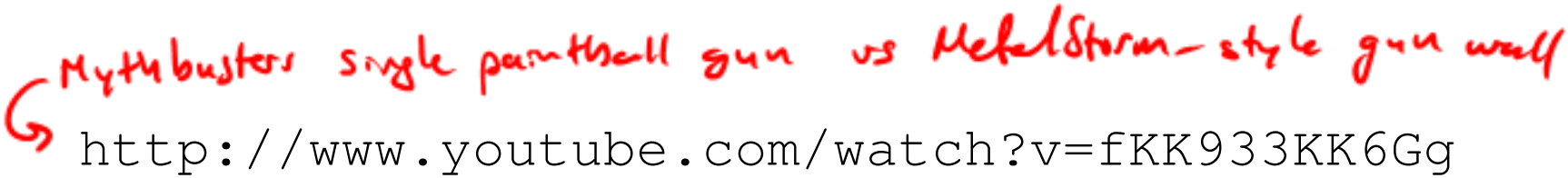


Note the slope is $p \approx 3$!
$\Rightarrow O(n^3)$

Logistics

- Raised enrollment cap from 50 to 80 on Friday.
- Some new background pointers on references page.
- Will set up cluster accounts in next week or so.

Just for fun

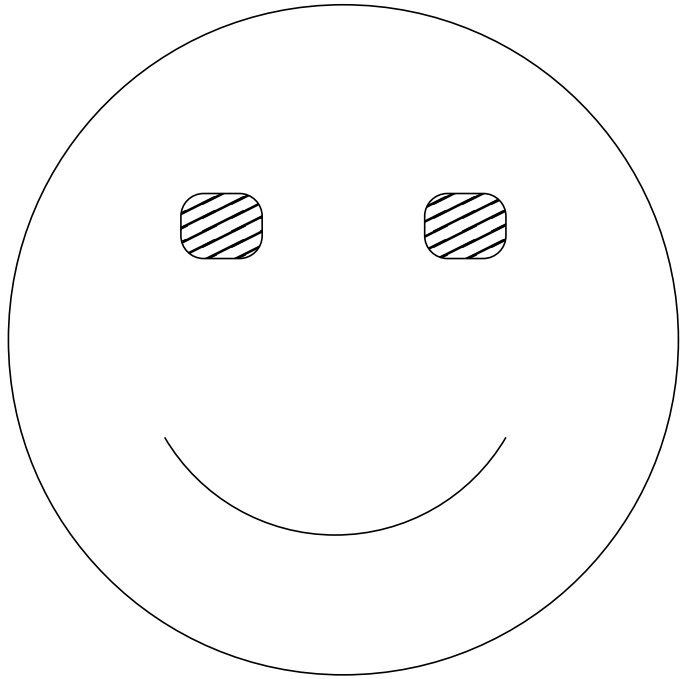


http://www.youtube.com/watch?v=fKK933KK6Gg

Is this a fair portrayal of your CPU?

(See Rich Vuduc's talk, "Should I port my code to a GPU?")

The idealized machine



- Address space of named words
- Basic operations are register read/write, logic, arithmetic
- Everything runs in the program order
- High-level language translates into “obvious” machine code
- All operations take about the same amount of time

The real world

- Memory operations are *not* all the same!
  - Registers and caches lead to variable access speeds
  - Different memory layouts dramatically affect performance
- Instructions are non-obvious!
  - Pipelining allows instructions to overlap
  - Functional units run in parallel (and out of order)
  - Instructions take different amounts of time
  - Different costs for different orders and instruction mixes

Our goal: enough understanding to help the compiler out.

A sketch of reality

Today, a play in two acts:[1]

1. Act 1: One core is not so serial → idealized core is serial.
2. Act 2: Memory matters

[1] If you don't get the reference to *This American Life*, go find the podcast!

Act 1

One core is not so serial.

Parallel processing at the laundromat

- Three stages to laundry: wash, dry, fold.
- Three loads: darks, lights, underwear
- How long will this take?

Parallel processing at the laundromat

- Serial version:

1	2	3	4	5	6	7	8	9
wash	dry	fold						
			wash	dry	fold			
						wash	dry	fold

- Pipeline version:

1	2	3	4	5	
wash	dry	fold			Dinner?
	wash	dry	fold		Cat videos?
		wash	dry	fold	Gym and tanning?

Pipelining

loads/time

time/load

- Pipelining improves *bandwidth*, but not *latency*
- Potential speedup = number of stages → assuming great ILP & no stalls/flushes
  - But what if there's a branch?

Example: My laptop

Core Duo! 2 Cores

2.5 GHz MacBook Pro with Intel Core 2 Duo T9300 processor.

- 14 stage pipeline (P4 was 31; longer isn't always better)
- Wide dynamic execution: up to four full instructions at once (4 issue)
- Operations internally broken down into "micro-ops"
  - Cache micro-ops – like a hardware JIT?!

In principle, two cores can handle 20 Giga-op/s peak?

As we all know x86 ASM is old, bloated, and crufty.

They help address this issue by doing this:

x86 ASM → JIT → cache micro ops

↑ this is what's pipelined.

SIMD

- *S*ingle *I*nstruction *M*ultiple *D*ata
- Old idea had a resurgence in mid-late 90s (for graphics)
- Now short vectors are ubiquitous...

My laptop

- SSE (Streaming SIMD Extensions)
- Operates on 128 bits of data at once
  1. Two 64-bit floating point or integer ops
  2. Four 32-bit floating point or integer ops
  3. Eight 16-bit integer ops
  4. Sixteen 8-bit ops
- Floating point handled slightly differently from “main” FPU
- Requires care with data alignment

Also have vector processing on GPU

→ note that this required some thinking. The semantics of the floaty point are diffrent then the "main" GPU.

Punchline

Modern CPU's are complicated.

- Special features: SIMD instructions, maybe FMAs, ...
- Compiler understands how to utilize these *in principle*
  - Rearranges instructions to get a good mix
  - Tries to make use of FMAs, SIMD instructions, etc
- In practice, needs some help:
  - Set optimization flags, pragmas, etc
  - Rearrange code to make things obvious and predictable
  - Use special intrinsics or library routines
  - Choose data layouts, algorithms that suit the machine
- Goal: You handle high-level, compiler handles low-level.

You must write efficient algorithms and make sure memory accesses are well-structured to get good cache behaviour, etc.

must allow compiler to make use of architectural features and ISA extensions if you actually want them to be used.

Act 2

Memory matters.

My machine

- Clock cycle: 0.4 ns
- DRAM access: 60 ns (about)
- *Getting data* $>$ 100$\times$ slower than *computing*!
- So what can we do?

Standard Memory Wall

Cache basics

Programs usually have *locality*

- *Spatial locality*: things close to each other tend to be accessed consecutively
- *Temporal locality*: use a “working set” of data repeatedly

Cache hierarchy built to use locality.

Cache basics

- Memory *latency* = how long to get a requested item
- Memory *bandwidth* = how fast memory can provide data
- Bandwidth improving faster than latency

Caches help:

- Hide memory costs by reusing data
  - Exploit temporal locality
- Use bandwidth to fetch a *cache line* all at once
  - Exploit spatial locality
- Use bandwidth to support multiple outstanding reads
- Overlap computation and communication with memory
  - Prefetching

This is mostly automatic and implicit.

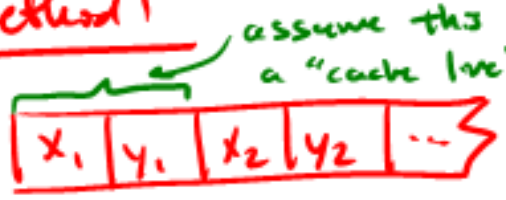
Teaser

We have $N = 10^6$ two-dimensional coordinates, and want their centroid. Which of these is faster and why?

1. Store an array of (x_i, y_i) coordinates. Loop i and simultaneously sum the x_i and the y_i.
2. Store an array of (x_i, y_i) coordinates. Loop i and sum the x_i, then sum the y_i in a separate loop.
3. Store the x_i in one array, the y_i in a second array. Sum the x_i, then sum the y_i.

Let's see!

method 1



⇒ computation is "free" in comparison to memory access ⇒ memory bound

method 2

this causes 2x the cache misses since you have to pull in the cache lines twice. You're pulling in a pair of data values and only operating on one of them, unlike method 1 in which you operate on both values ⇒ Time(method 2) ≈ 2 Time(method 1)

method 3

similar to 2.

Notes if you're following along at home

- Try the experiment yourself (~~`lec01mean.c`~~ lec 02mean.c is posted online) — I'm not giving away the punchline!
- If you use high optimization (`-O3`, compiler may pre-compute as input is static) the compiler may optimize away your timing loops! This is a common hazard in timing. You could get around this by puting `main` and the test stubs in different modules; but for the moment, just compile with `-O2`.

Cache basics

Stride pattern matters! Apparently bad strides are 2^k in size.

- Store cache *line*s of several bytes
- Cache *hit* when copy of needed data in cache
- Cache *miss* otherwise. Three basic types:
  - *Compulsory* miss: never used this data before
  - *Capacity* miss: filled the cache with other things since this was last used – working set too big
  - *Conflict* miss: insufficient associativity for access pattern
- *Associativity*
  - Direct-mapped: each address can only go in one cache location (e.g. store address xxxx1101 only at cache location 1101) ⇒ lots of collisions and evictions
  - *n*-way: each address can go into one of *n* possible cache locations (store up to 16 words with addresses xxxx1101 at cache location 1101). ← specific implementation

  aka Set-associative cache

  Higher associativity is more expensive. (in hardware)

Caches on my laptop (I think)

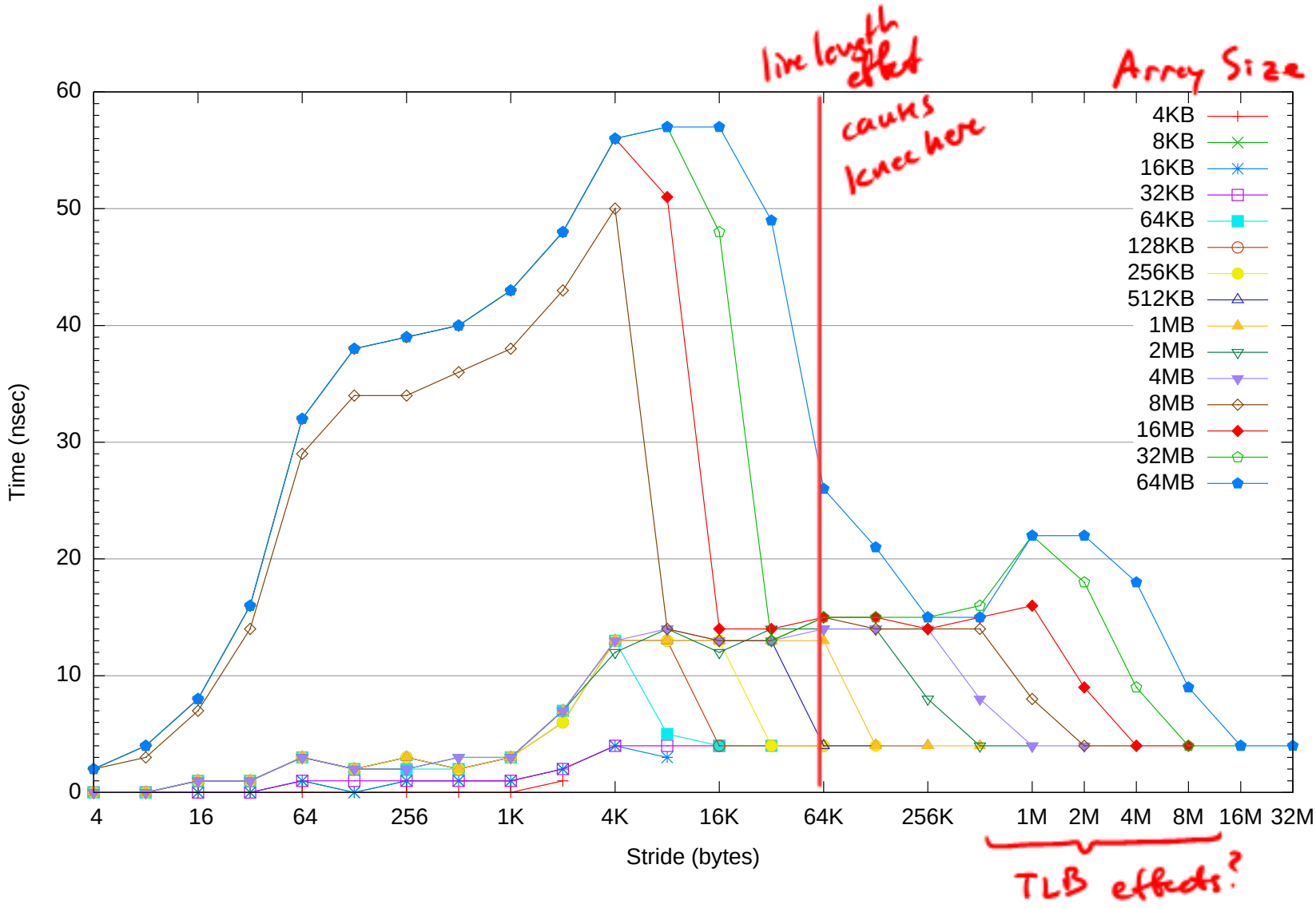
- 32K L1 data and ~~memory~~ instruction caches (per core)
  - 8-way set associative
  - 64-byte cache line
- 6 MB L2 cache (shared by both cores)
  - 16-way set associative
  - 64-byte cache line

one core could misbehave and negatively impact the other core's cache performance
⇒ bad cache contention

A memory benchmark (membench)

```
for array A of length L from 4 KB to 8MB by 2x
  for stride s from 4 bytes to L/2 by 2x
  time the following loop
    for i = 0 to L by s
      load A[i] from memory
```

membench on my laptop



Visible features

- Line length at 64 bytes (prefetching?)
- L1 latency around 4 ns, 8 way associative
- L2 latency around 14 ns
- L2 cache size between 4 MB and 8 MB (actually 6 MB)
- 4K pages, 256 entries in TLB

The moral

Even for simple programs, performance is a complicated function of architecture!

- Need to understand at least a little to write fast programs
- Would like simple models to help understand efficiency
- Would like common tricks to help design fast codes
  - Example: *blocking* (also called *tiling*)

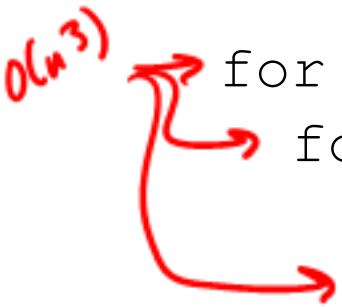
=> Again, we have a memory wall. We are memory bound most of the time

Matrix multiply

Consider naive square matrix multiplication:

```
#define A(i,j) AA[j*n+i]
#define B(i,j) BB[j*n+i]
#define C(i,j) CC[j*n+i]

for (i = 0; i < n; ++i) {
  for (j = 0; j < n; ++j) {
    C(i,j) = 0;
    for (k = 0; k < n; ++k)
      C(i,j) += A(i,k)*B(k,j);
  }
}
```



How fast can this run?

Note on storage

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Two standard matrix layouts:

- Column-major (Fortran): A(i,j) at A+j*n+i $[1\ 4\ 2\ 5\ 3\ 6]$
- Row-major (C): A(i,j) at A+i*n+j $[1\ 2\ 3\ 4\ 5\ 6]$

I default to column major.

Also note: C doesn't really support matrix storage.

1000-by-1000 matrix multiply on my laptop

- Theoretical peak: 10 Gflop/s using both cores
- Naive code: 330 MFlops (3.3% peak)
- Vendor library: 7 Gflop/s (70% peak)

Tuned code is 20$\times$ faster than naive!

Can we understand naive performance in terms of membench?

1000-by-1000 matrix multiply on my laptop

- Matrix sizes: about 8 MB.
- Repeatedly scans B in memory order (column major)
- 2 flops/element read from B
- 3 ns/flop = 6 ns/element read from B
- Check membench — gives right order of magnitude!

Simple model

Consider two types of memory (fast and slow) over which we have complete control.

- m = words read from slow memory
- t_m = slow memory op time
- f = number of flops
- t_f = time per flop
- $q = f/m$ = average flops / slow memory access

Time: ← aka "Computational Intensity"

$$ft_f + mt_m = ft_f \left(1 + \frac{t_m/t_f}{q}\right)$$

← effective slow down due to memory

↑ computation cost

Larger q means better time.

How big can q be?

1. Dot product: n data, $2n$ flops
2. Matrix-vector multiply: n^2 data, $2n^2$ flops
3. Matrix-matrix multiply: $2n^2$ data, $2n^3$ flops

These are examples of level 1, 2, and 3 routines in *Basic Linear Algebra Subroutines* (BLAS). We like building things on level 3 BLAS routines.

q for naive matrix multiply

$q \approx 2$ (on board)

Better locality through blocking

Basic idea: rearrange for smaller working set.

```
for (I = 0; I < n; I += bs) {
  for (J = 0; J < n; J += bs) {
    block_clear(&(C(I,J)), bs, n);
    for (K = 0; K < n; K += bs)
      block_mul(&(C(I,J)), &(A(I,K)), &(B(K,J)),
                bs, n);
  }
}
```

Q: What do we do with “fringe” blocks?

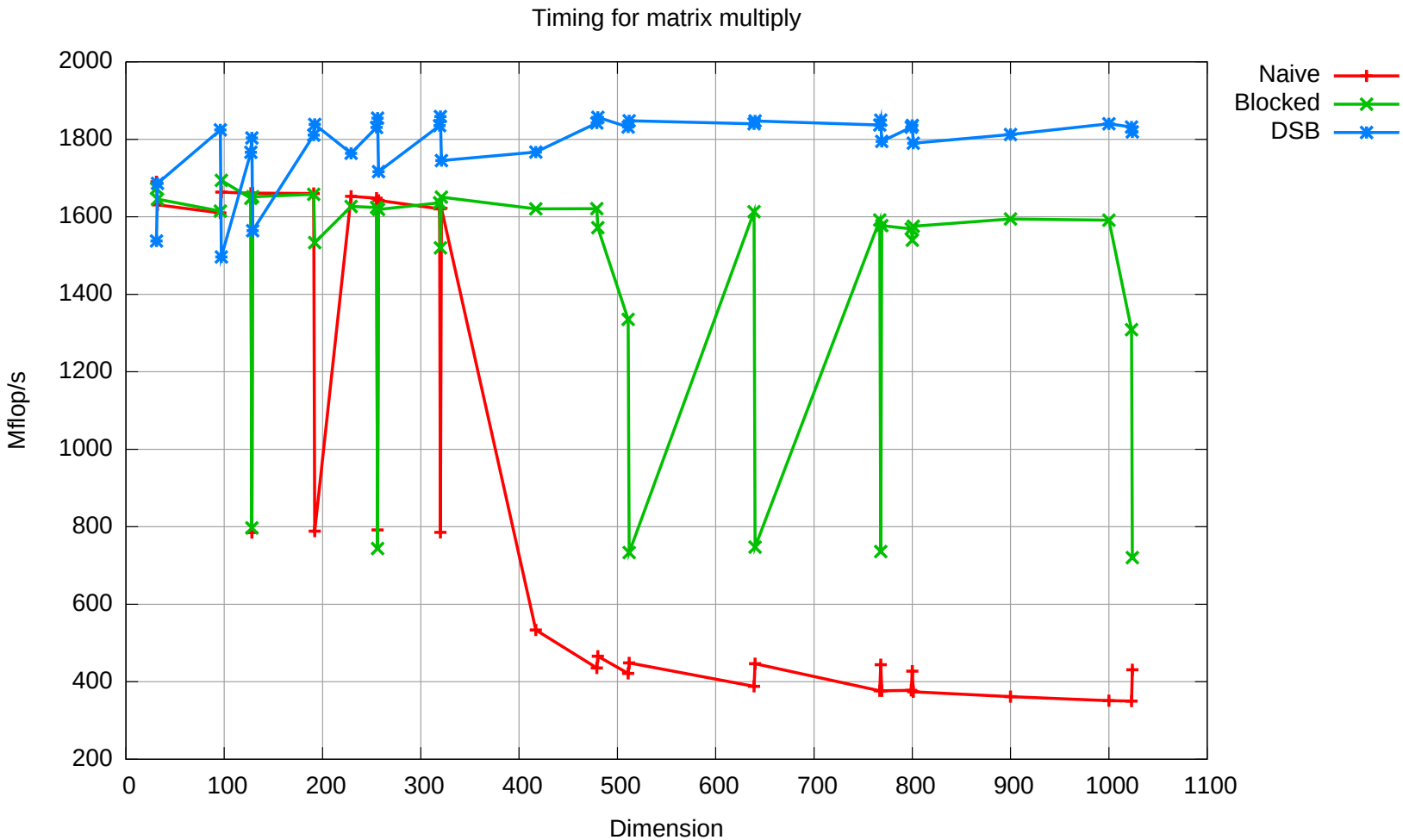
q for naive matrix multiply

$q \approx b$ (on board). If M_f words of fast memory, $b \approx \sqrt{M_f/3}$.

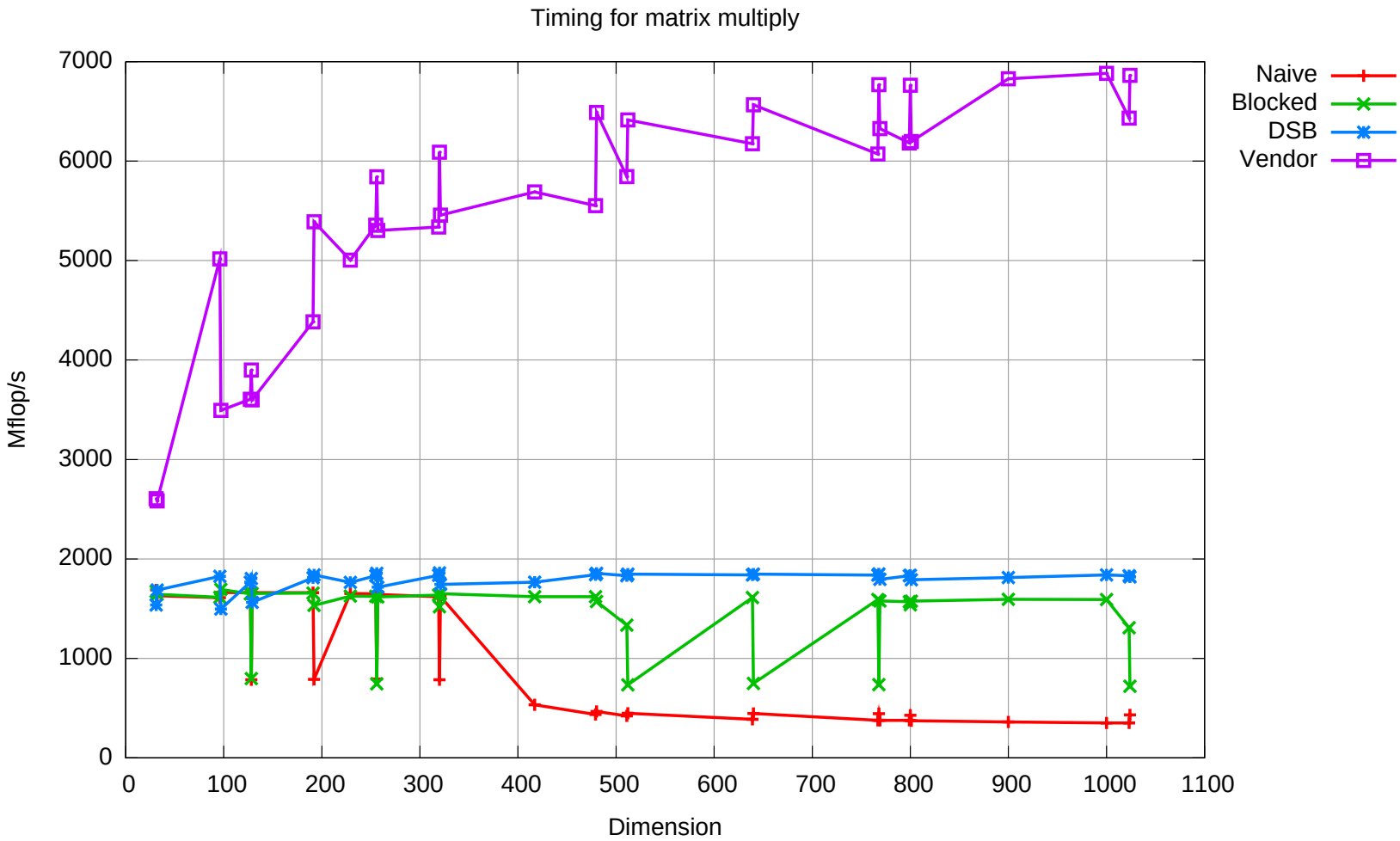
Th: (Hong/Kung 1984, Irony/Tishkin/Toledo 2004): Any reorganization of this algorithm that uses only associativity and commutativity of addition is limited to $q = O(\sqrt{M_f})$

Note: Strassen uses distributivity...

Better locality through blocking



Truth in advertising



Coming attractions

HW 1: You will optimize matrix multiply yourself!

Some predictions:

- You will make no progress without addressing memory.
- It will take you longer than you think.
- Your code will be rather complicated.
- Few will get anywhere close to the vendor.
- Some of you will be sold anew on using libraries!

Not all assignments will be this low-level.

A little perspective

> *"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."*
>
> *– C.A.R. Hoare (quoted by Donald Knuth)*

- Best case: good algorithm, efficient design, *obvious code*
- Speed vs readability, debuggability, maintainability?
- A sense of balance:
  - Only optimize when needed
  - Measure before optimizing
  - Low-hanging fruit: data layouts, libraries, compiler flags
  - Concentrate on the bottleneck
  - Concentrate on inner loops
  - Get correctness (and a test framework) first