18.S096 January 2017: Memory and Matrices

Steven G. Johnson, MIT Applied Math

performance experiments (circa 2008):

Hardware: 2.66GHz Intel Core 2 Duo

64-bit mode, double precision, gcc 4.1.2

optimized BLAS dgemm: ATLAS 3.6.0

http://math-atlas.sourceforge.net/

A trivial problem?

$$\underset{m\times p}{C} = \underset{m\times n}{A}\ \underset{n\times p}{B}$$

the "obvious" C code (rows · columns):

```
/* C = A B, where A is m x n, B is n x p,
            and C is m x p, in row-major order */
void matmul(const double *A, const double *B,
            double *C, int m, int n, int p)
{
     int i, j, k;
     for (i = 0; i < m; ++i)
          for (j = 0; j < p; ++j) {
               double sum = 0;
               for (k = 0; k < n; ++k)
                    sum += A[i*n + k] * B[k*p + j];
               C[i*p + j] = sum;
          }
}
```

for $i = 1$ **to** m
  for $j = 1$ **to** p

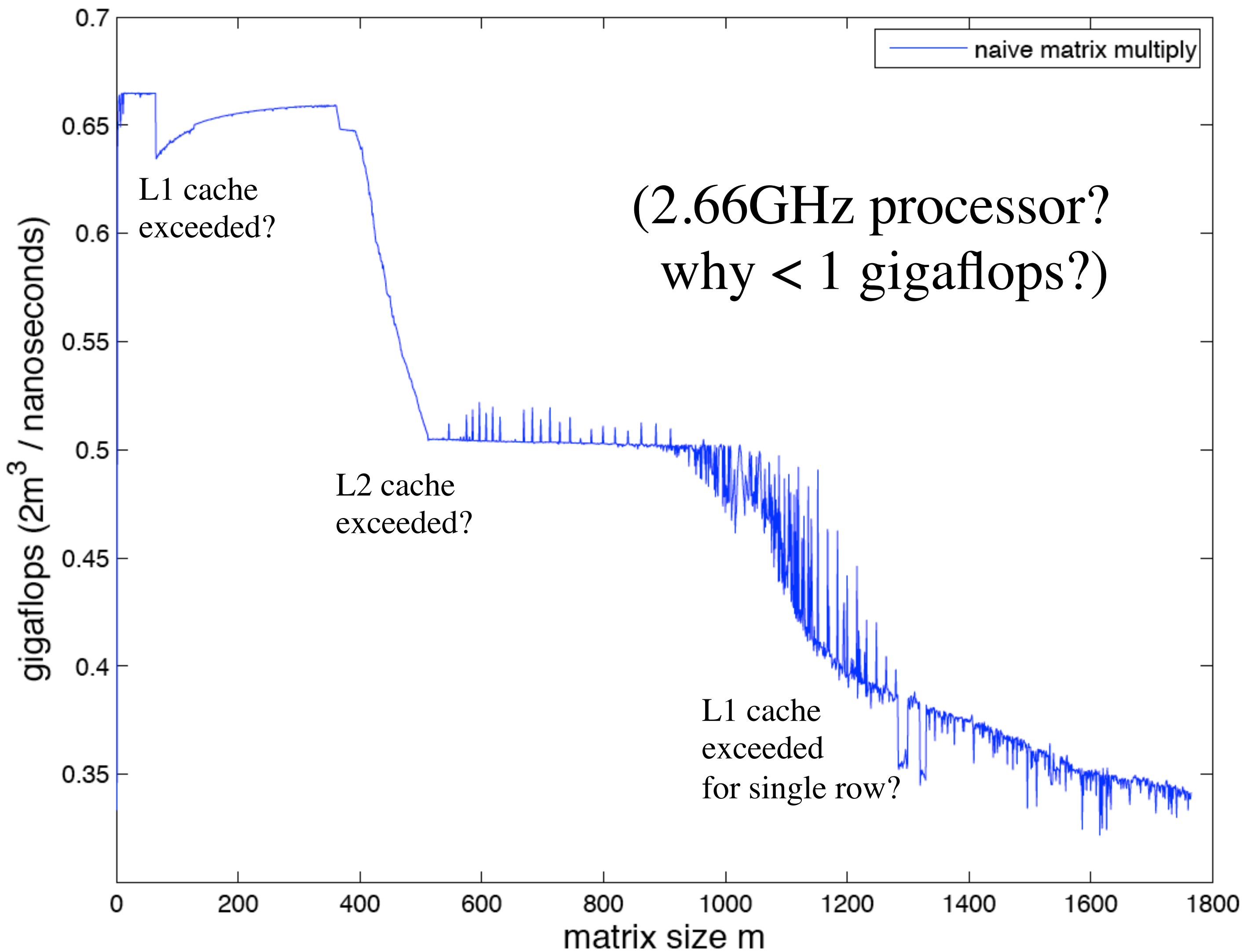
$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

$2mnp$ flops
(adds+mults)
"floating-point operations"

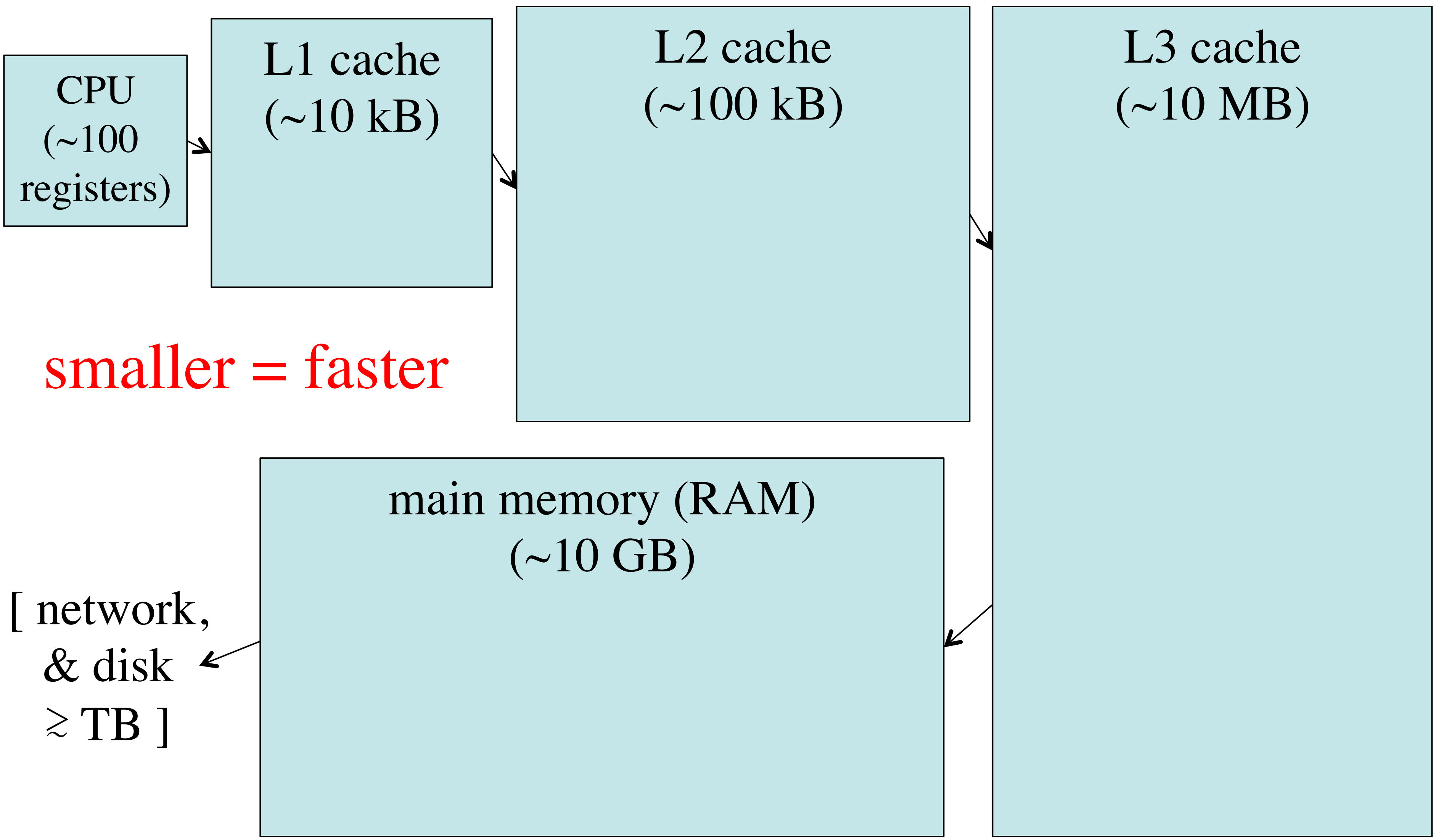
just three loops, how complicated can it get?

flops/time is not constant!

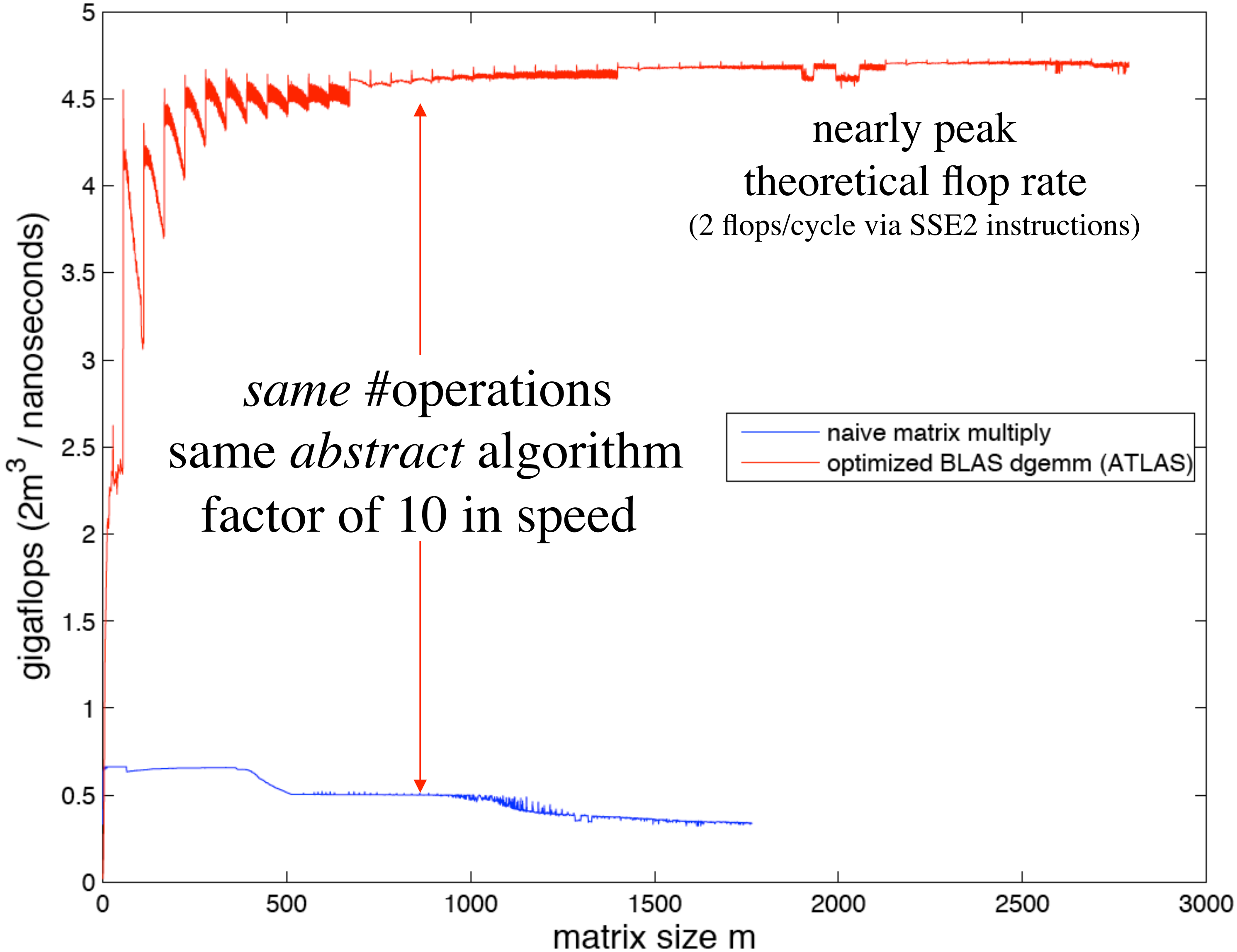
(square matrices, $m=n=p$)



Speed is limited by access to the memory hierarchy *[not to scale!]*



All flops are not created equal



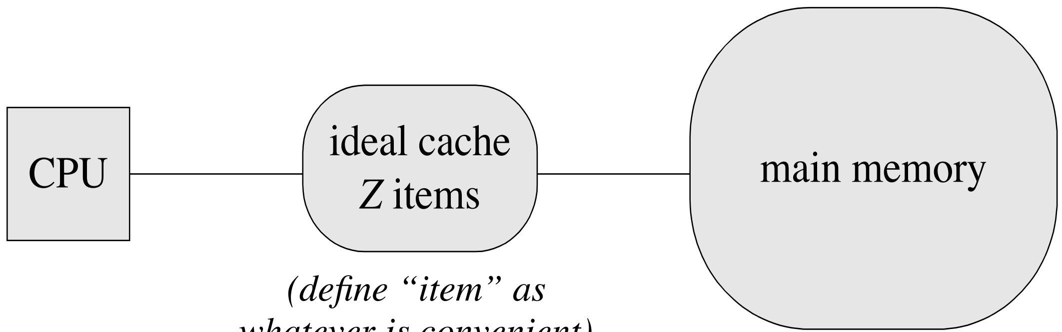
Things to remember

- We often cannot understand performance without understanding memory efficiency (caches).
  - often ~10 times more important than arithmetic count when working with lots of data
- Computers are more complicated than you think.
- Even a trivial algorithm is nontrivial to implement *well*.
  - matrix multiplication: 10 lines of code $\longrightarrow$ 130,000+ (ATLAS)
  - getting the last factor of 2 in speed often requires wizardry (and is usually not worth it)
  - but factors of 10 are often worthwhile and not too hard…

Ideal Cache Model

[Frigo, Leiserson, Prokop, and Ramachandran (1999)]

simplified model of cache to help us understand/design algorithms



(define "item" as whatever is convenient)

when CPU needs an item, either:

- cache hit: already in cache (fast)
- cache miss: load into cache (slow)

goal: analyze # of cache misses

Simplification: cache is "ideal"

— fully associative: any item in memory can replace any item in cache

— optimal replacement: cache miss replaces "best" item

(= item not needed for longest time in the future)

… within a ~constant factor of more realistic caches

Cache complexity: for problem with n items, cache size Z,

want # misses for large n in "big O" or "big Θ" notation

(ignoring constant factors), e.g. $\Theta(n/Z)$

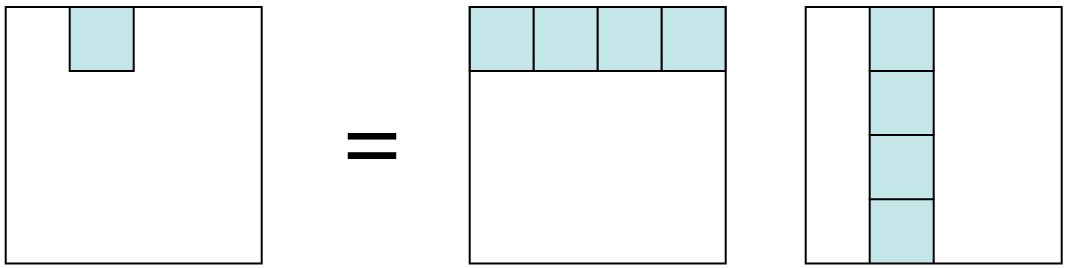
Strategy for efficient cache utilization:

Maximize temporal locality

Once we read an item from memory, we want as much computation as possible before reading the next item…

Re-arrange our algorithm so that computations on the same data occur at close to the same time.

(optimal) Blocked Matrix Multiply



C $\quad$ A $\quad$ B

$n{\times}n$ $\quad$ $n{\times}n$ $\quad$ $n{\times}n$

divide matrices into $b{\times}b$ blocks of $b{=}\sqrt{Z/3}$ numbers

load 3 $b{\times}b$ blocks into cache and *multiply blocks in-cache*

(provably optimal) cache misses: $\Theta(n^3/b^3) \times \Theta(b^2) = \Theta(n^3/\sqrt{Z})$

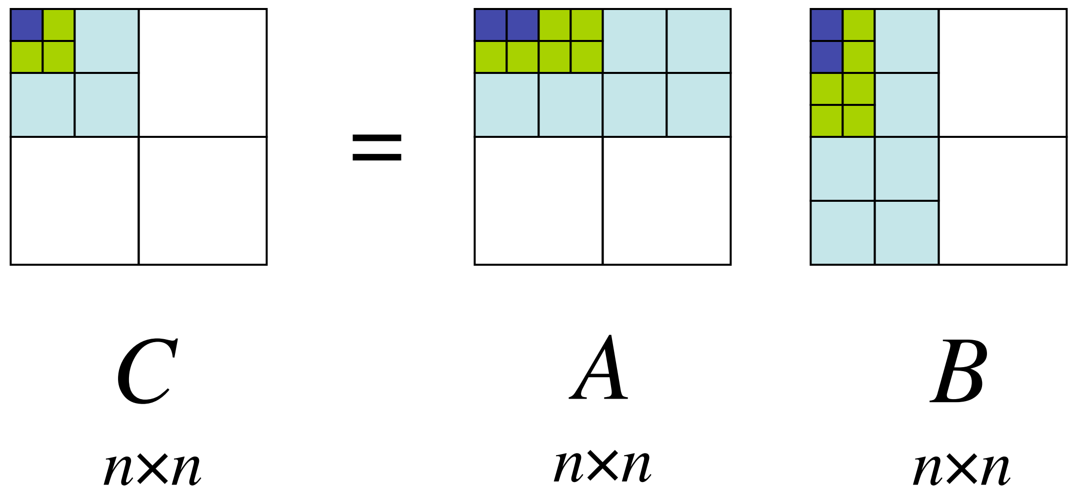
block × block multiplications

cache misses per block

Challenges with blocking

- Programmer/code needs to know the size Z of the cache — different code for every CPU?
- Multiple levels of cache = nested blocking
- Many complications to get near-optimal "constant factor" in Θ
  - optimal block size is non-square to balance load/store cost.
  - lowest level (cache=registers) requires unrolling, SIMD optimizations, lots of tricks…

(optimal) Cache-Oblivious Matrix Multiply



divide and conquer:

divide C into 4 blocks

compute block multiply recursively

achieves optimal $\Theta(n^3/\sqrt{Z})$ cache complexity

without knowing the Z, works for nested caches too

A little C implementation (~25 lines)

```
/* C = C + AB, where A is m x n, B is n x p, and C is m x p, in
   row-major order.  Actually, the physical size of A, B, and C
   are m x fdA, n x fdB, and m x fdC, but only the first n/p/p
   columns are used, respectively. */
void add_matmul_rec(const double *A, const double *B, double *C,
                    int m, int n, int p, int fdA, int fdB, int fdC)
{
     if (m+n+p <= 48) { /* <= 16x16 matrices "on average" */
          int i, j, k;
          for (i = 0; i < m; ++i)
               for (k = 0; k < p; ++k) {
                    double sum = 0;
                    for (j = 0; j < n; ++j)
                         sum += A[i*fdA +j] * B[j*fdB + k];
                    C[i*fdC + k] += sum;
               }
     }
     else { /* divide and conquer */
          int m2 = m/2, n2 = n/2, p2 = p/2;

          add_matmul_rec(A, B, C, m2, n2, p2, fdA, fdB, fdC);
          add_matmul_rec(A+n2, B+n2*fdB, C, m2, n-n2, p2, fdA, fdB, fdC);

          add_matmul_rec(A, B+p2, C+p2, m2, n2, p-p2, fdA, fdB, fdC);
          add_matmul_rec(A+n2, B+p2+n2*fdB, C+p2, m2, n-n2, p-p2, fdA, fdB, fdC);

          add_matmul_rec(A+m2*fdA, B, C+m2*fdC, m-m2, n2, p2, fdA, fdB, fdC);
          add_matmul_rec(A+m2*fdA+n2, B+n2*fdB, C+m2*fdC, m-m2, n-n2, p2, fdA, fdB, fdC);

          add_matmul_rec(A+m2*fdA, B+p2, C+m2*fdC+p2, m-m2, n2, p-p2, fdA, fdB, fdC);
          add_matmul_rec(A+m2*fdA+n2, B+p2+n2*fdB, C+m2*fdC+p2, m-m2, n-n2, p-p2, fdA, fdB, fdC);
     }
}

void matmul_rec(const double *A, const double *B, double *C,
                int m, int n, int p)
{
     memset(C, 0, sizeof(double) * m*p);
     add_matmul_rec(A, B, C, m, n, p, n, p, p);
}
```

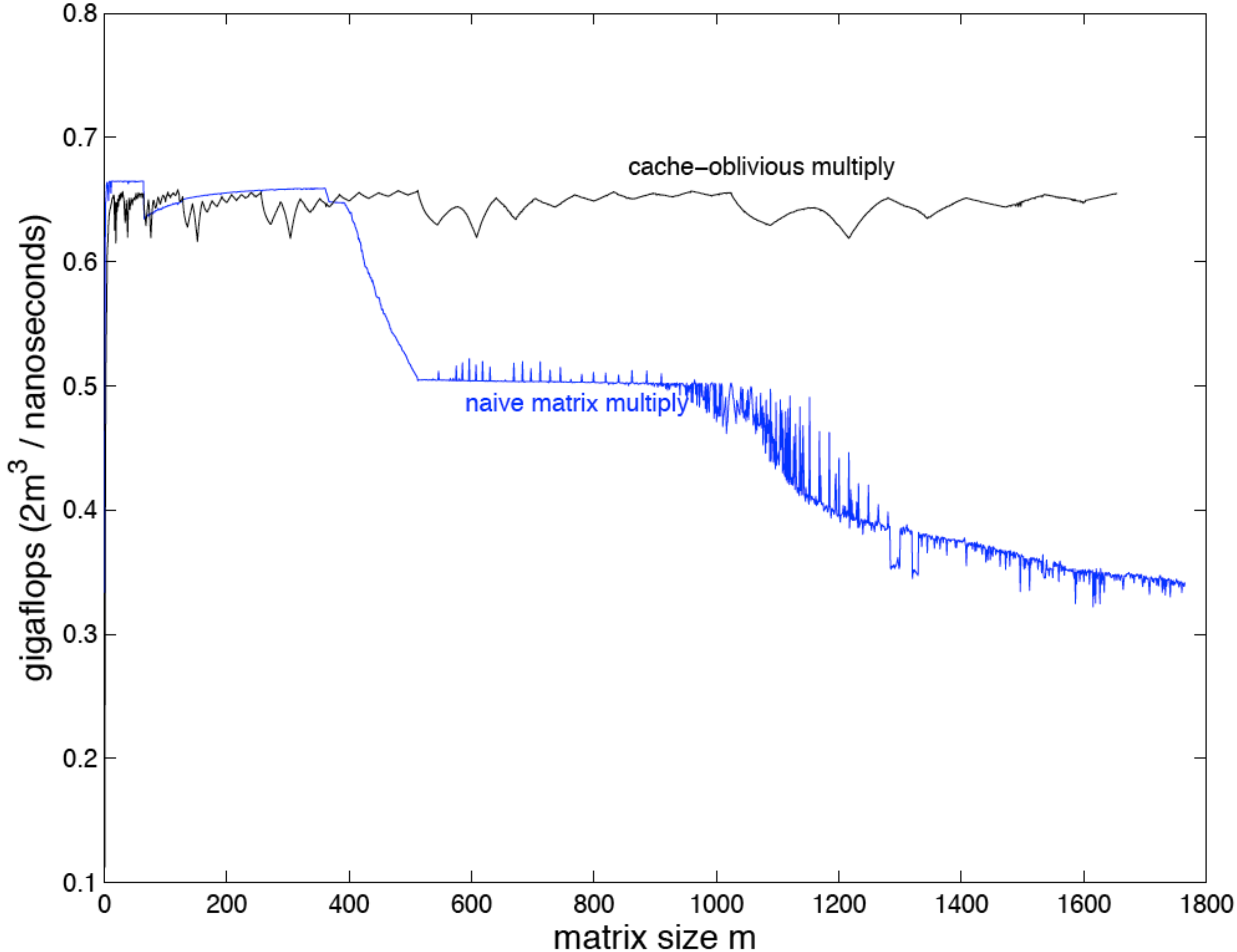
note: base case is ~16×16

recursing down to 1×1 would kill performance

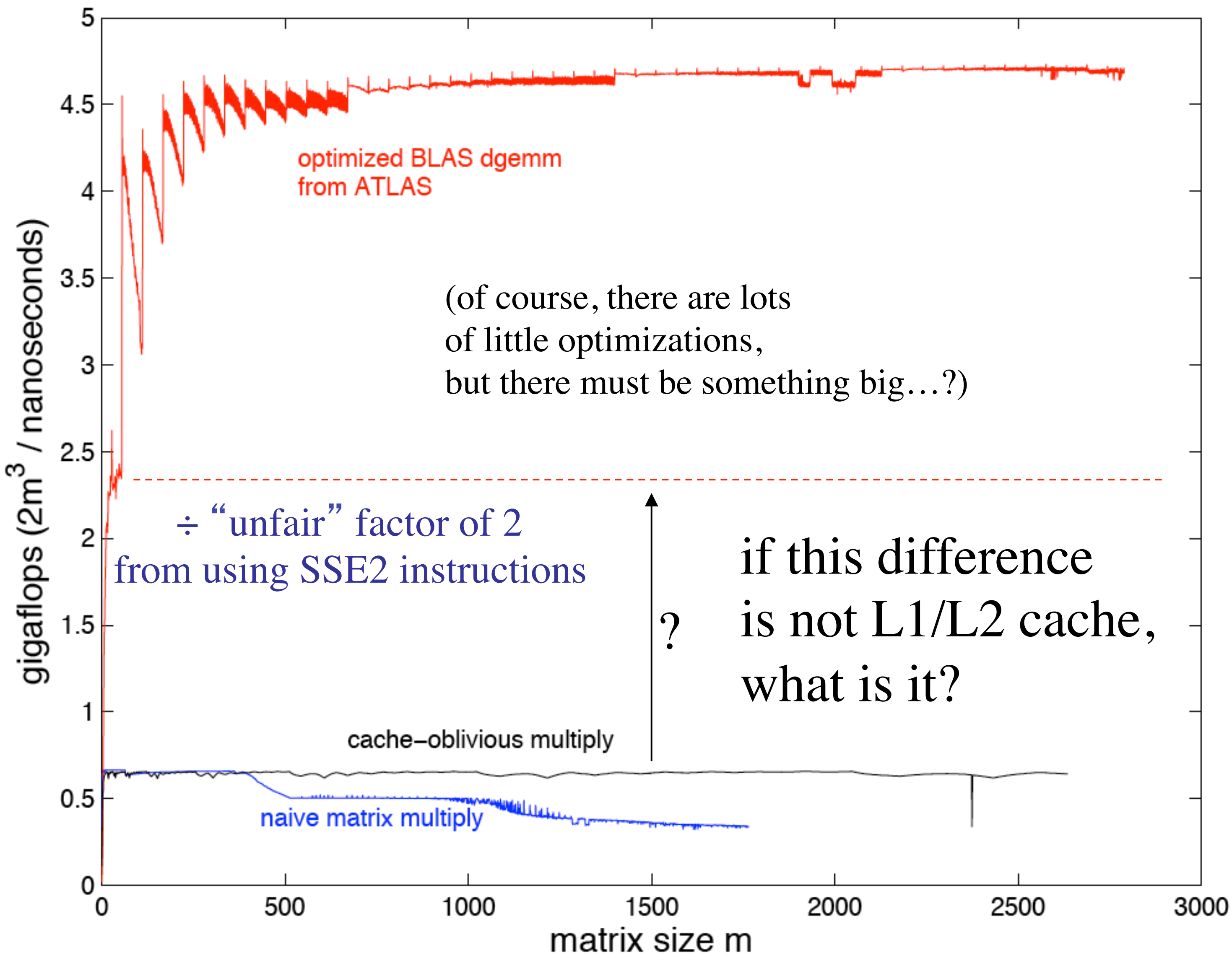
(1 function call per element, no register re-use)

dividing *C* into 4 — note that, instead, for very non-square matrices, we might want to divide *C* in 2 along longest axis

No Cache-based Performance Drops!



…but absolute performance still sucks



Registers == Cache

- The registers (~100) form a very small, almost ideal cache
  - Three nested loops is not the right way to use this “cache” for the same reason as with other caches
- Need long blocks of unrolled code: load blocks of matrix into local variables (= registers), do matrix multiply, write results
  - Loop-free blocks = many optimized hard-coded base cases of recursion for different-sized blocks … often automatically generated (ATLAS)
  - Unrolled $n \times n$ multiply has $(n^3)!$ possible code orderings — compiler cannot find optimal schedule (NP hard) — cache-oblivious scheduling can help (c.f. FFTW), but ultimately requires some experimentation/wizardry (automated in ATLAS)
  - Optimal blocks are non-square to balance load/store cost, and details (e.g. scheduling) turn out to depend on the CPU.

No data re-use = no possibility of temporal locality … what then?

Suppose we are computing the dot product $\mathbf{x}^{*}\mathbf{y}$ of two vectors:

$$x^{*}y = \sum_{i=1}^{n} \overline{x_i} y_i$$

Each element of **x** and **y** is used exactly once.

Does this mean we get no benefit from the caches?

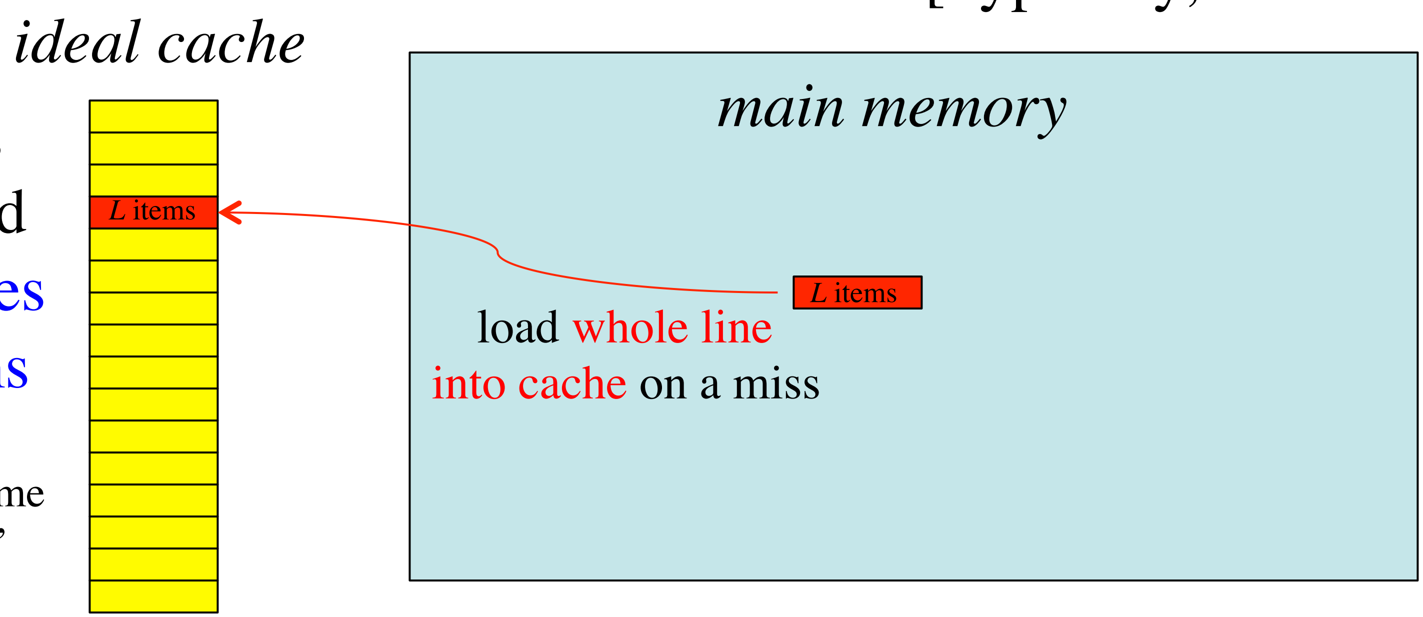
Cache lines and spatial locality

To speed up algorithms with little or no data re-use, caches exploit the fact that memory access is often consecutive by reading in a whole cache line of L items on a miss

[typically, $L \sim 64$ bytes]

Z items organized in Z/L lines of L items

typically assume "tall cache:" $Z/L > L$



Cache-optimization strategy:

when you access data in memory, try to access *nearby data* soon afterwards … maximize "spatial locality"

Example: Matrix addition

$$\underset{m\times n}{C} = \underset{m\times n}{A} + \underset{m\times n}{B}$$

two possible algorithms:

for $i = 1$ to m
  for $j = 1$ to n
    $C_{ij} = A_{ij} + B_{ij}$

or

for $j = 1$ to n
  for $i = 1$ to m
    $C_{ij} = A_{ij} + B_{ij}$

which one to use? depends on how matrices are stored!

Column-major storage

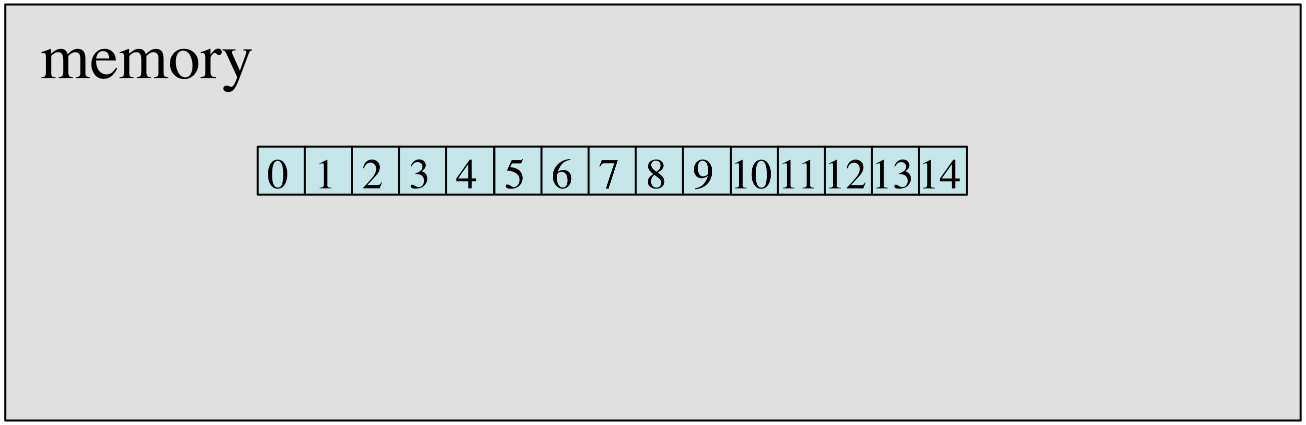
used by Fortran, Matlab, Julia, …

= store columns of A consecutively in memory

0	5	10
1	6	11
2	7	12
3	8	13
4	9	14

A

5×3



access rows consecutively:

for i = 1 to m
  for j = 1 to n
    $C_{ij} = A_{ij} + B_{ij}$

= $\Theta(mn)$ misses

access columns consecutively:

for j = 1 to n
  for i = 1 to m
    $C_{ij} = A_{ij} + B_{ij}$

= $\Theta(mn/L)$ misses