There is Plenty of Room at the Top

High Performance Computing, Summer 2021

Biagio Cosenza
Department of Computer Science
University of Salerno
bcosenza@unisa.it

There is a Plenty of Room at the Bottom

I would like to describe a field, in which little has been done, but in which an enormous amount can be done in principle (…) (which) would have an enormous number of technical applications.

What I want to talk about is the problem of manipulating and controlling things on a small scale.

(current progress in) miniaturization is nothing.
It is a staggeringly small world that is below.

Richard Feynman

- Feynman foresaw the progress in miniaturization that backed up the Moore's law

source: https://web.archive.org/web/20100211190050/http://www.its.caltech.edu/~feynman/plenty.html#

There is a Plenty of Room at the Top

As miniaturization wanes, the silicon fabrication improvements at the Bottom will no longer provide the predictable, broad-based gains in computer performance that society has enjoyed for more than 50 years.

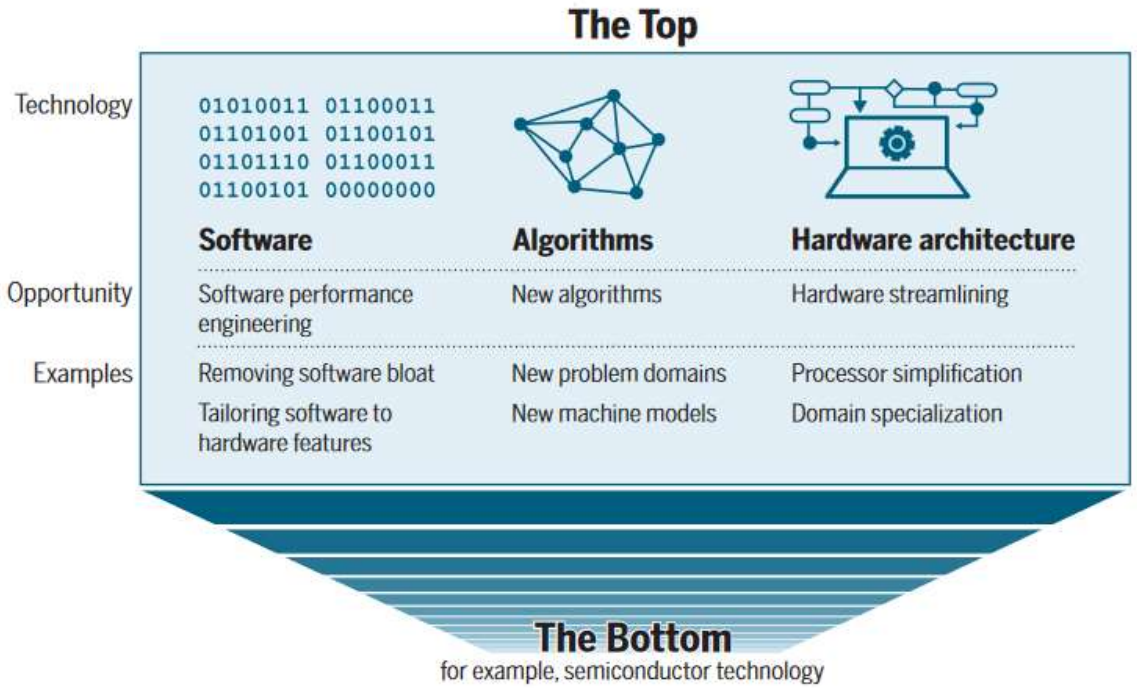
Software performance engineering, development of algorithms, and hardware streamlining at the Top can continue to make computer applications faster in the post-Moore era.

Butler Sampson

Charles Leiserson

Charles E. Leiserson, Neil C. Thompson, Joel S. Emer, Bradley C. Kuszmaul,Butler W. Lampson, Daniel Sanchez1, Tao B. Schardl
There's plenty of room at the Top: What will drive computer performance after Moore's law?
Science 05 Jun 2020: Vol. 368, Issue 6495, eaam9744

The Top vs The Bottom



Performance gains after Moore's law ends. In the post-Moore era, improvements in computing power will increasingly come from technologies at the "Top" of the computing stack, not from those at the "Bottom", reversing the historical trend.

Outline

- Matrix multiplication

- Adapted materials from MIT 6.172 by Charles Leiserson
- Reference paper

Charles E. Leiserson, Neil C. Thompson, Joel S. Emer, Bradley C. Kuszmaul,Butler W. Lampson, Daniel Sanchez1, Tao B. Schardl
There's plenty of room at the Top: What will drive computer performance after Moore's law?
Science 05 Jun 2020: Vol. 368, Issue 6495, eaam9744

Matrix Multiplication

- Assume for simplicity $n = 2^k$

$$\begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}$$

$$C \qquad A \qquad B$$

$$c_{ij} = \sum_{k=1}^{n} a_{ik}\, b_{kj}$$

Machine Specification used for this Exercise

- Reference multicore

Feature	Specification
Microarchitecture	Haswell (Intel Xeon E5-2666 v3)
Clock frequency	2.9 GHz
Processor chips	2
Processing cores	9 per processor chip
Hyperthreading	2 way
Floating-point unit	8 double-precision operations, including fused-multiply-add, per core per cycle
Cache-line size	64 B
L1-icache	32 KB private 8-way set associative
L1-dcache	32 KB private 8-way set associative
L2-cache	256 KB private 8-way set associative
L3-cache (LLC)	25 MB shared 20-way set associative
DRAM	60 GB

$$\text{Peak} = (2.9 \times 10^9) \times 2 \times 9 \times 16 = 836 \text{ GFLOPS}$$

Optimizing Matmul - Version 1: Python3

- Run on Python3

```
> python3 mm_v1.py
```

- How long does it take?
  - running time: 21 042
  - ~6 hours

```
import random
from time import *

n = 4096
A = [[random.random()
      for row in range(n)]
     for col in range(n)]
B = [[random.random()
      for row in range(n)]
     for col in range(n)]
C = [[0 for row in range(n)]
     for col in range(n)]

start = time()
for i in range(n):
    for j in range(n):
        for k in range(n):
            C[i][j] += A[i][k] * B[k][j]
end = time()
print(end - start)
```

Is it fast?

- Back-of-the-envelope calculation
  - floating-point operations: $2n^3 = 2(2^{12})^3 = 2^{37}$
  - running time = 21042 seconds
  - Python gets $2^{37}/21042 \sim$ 6.25 MFLOPS
- However: Peak 836 GFLOPS
  - Python gets 0.00075% of peak

Optimizing Matmul - Version 2: Java

- Running time 2 738 seconds
  - ~46 minutes
  - about 8,8x faster than Python

```
import java.util.Random;
public class Matmul {
    static int n = 40;
    static double [][] A = new double[n][n];
    static double [][] B = new double[n][n];
    static double [][] C = new double[n][n];
    public static void main(String []args){
        Random r = new Random();
        for (int i=0; i<n; i++) {
            for (int j=0; j<n; j++) {
                A[i][j] = r.nextDouble();
                B[i][j] = r.nextDouble();
                C[i][j] = 0;
            }
        }
        long start = System.nanoTime();
        for (int i=0; i<n; i++) {
            for (int j=0; j<n; j++) {
                for (int k=0; k<n; k++) {
                    C[i][j] += A[i][k] * B[k][j];
                }
            }
        }
        long stop = System.nanoTime();
        double tdiff = (stop-start) * 1e-9;
        System.out.println(tdiff);
    }
}
```

Optimizing Matmul - Version 3: C

- Using the Clang/LLVM 5.0 compiler
  - Running time 1 156 seconds
  - ~ 19 minutes
  - 2x faster than Java
  - 18 faster than Python

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#define N 4096
double A[N][N], B[N][N], double C[N][N];
float tdiff(struct timeval *start, struct timeval *end){
    return (end->tv_sec - start->tv_sec) +
    1e-6 * (end->tv_usec - start->tv_usec);
}
int main() {
    for (int i=0; i<N; i++) {
        for (int j=0; j<N; j++) {
            A[i][j] = (double)rand()/(double)RAND_MAX;
            B[i][j] = (double)rand()/(double)RAND_MAX;
            C[i][j] = 0;
        }
    }
    struct timeval start, end;
    gettimeofday(&start, NULL);
    for (int i=0; i<N; i++) {
        for (int j=0; j<N; j++) {
            for (int k=0; k<N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    gettimeofday(&end, NULL);
    printf("%0.6f\n", tdiff(&start, &end));
    return 0;
}
```

Matmul Performance, so far

Version	Implementation	Running time	Relative speedup	Absolute speedup	GLOPS	% of peak
1	Python	21041	1.00	1	0.007	0.001
2	Java	2387	8.81	9	0.058	0.007
3	C	1155	2.07	18	0.119	0.014

Interpreters

- Why is Python so slow and C so fast?
  - Python is interpreted
  - C is compiled directly to machine code
  - Java is compiled to byte-code, which is then interpreted and just-in-time (JIT) compiled to machine code
- Interpreters are versatile, but slow
  - the interpreter reads, interprets, and performs each program statement and updates the machine state
  - interpreters can easily support high-level programming features — such as dynamic code alteration — at the cost of performance

JIT Compilation

- Just in Time (JIT) compilation
- JIT compilers can recover some of the performance lost by interpretation
  - when code is first executed, it is interpreted
  - the runtime system keeps track of how often the various pieces of code are executed
  - whenever some piece of code executes sufficiently frequently, it gets compiled to machine code in real-time
  - future executions of that code use the more-efficient compiled version

What do we optimize next? Loop Ordering

- Observation: we can change the order of the loops in this program without affecting its correctness
- Question: Does it matter for performance?

```
for (int i=0; i<N; i++){
  for (int j=0; j<N; j++) {
    for (int k=0; k<N; k++) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

Loop Order: Loop Interchange Examples

```
for (int i=0; i<N; i++){
  for (int j=0; j<N; j++){
    for (int k=0; k<N; k++) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

```
for (int k=0; k<N; k++) {
  for (int j=0; j<N; j++){
    for (int i=0; i<N; i++){
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

```
for (int i=0; i<N; i++){
  for (int k=0; k<N; k++){
    for (int j=0; j<N; j++){
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

Performance of Different Loop Orders

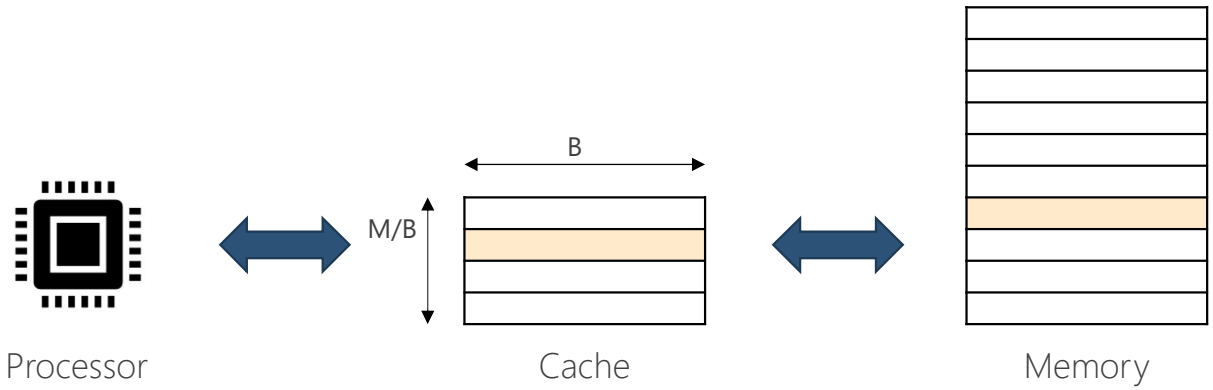
- Loop order affects running time by a factor of 18!
- Why?

Loop order (outer to inned loop)	Running time (s)
`i, j, k`	1155.77
`i, k, j`	177.68
`j, i, k`	1080.61
`j, k, i`	3056.63
`k, i, j`	179.21
`k, j, i`	3032.82

Caches!

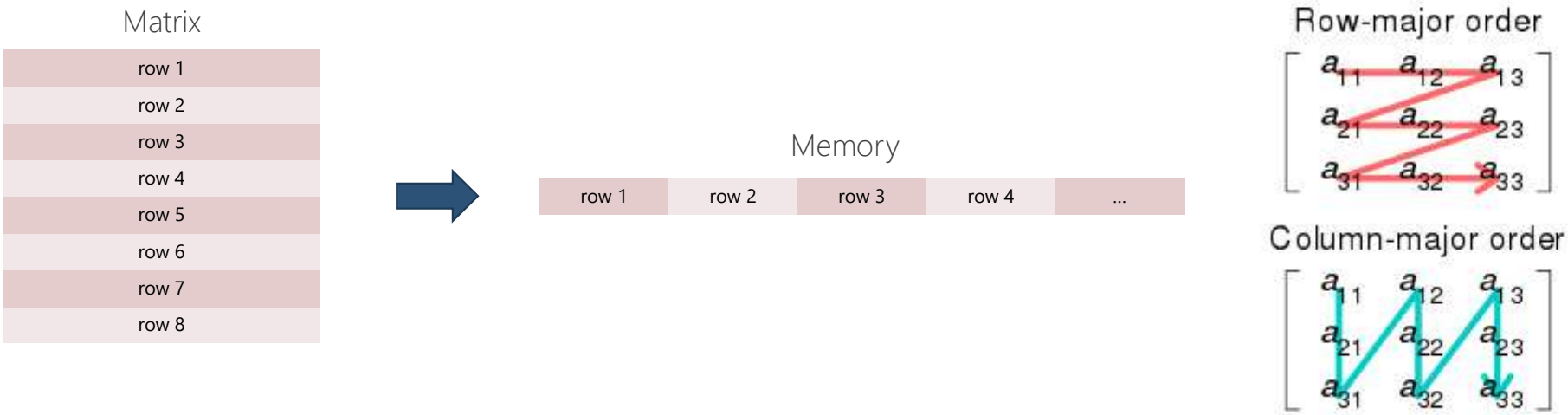
Each processor reads and writes main memory in contiguous blocks called cache lines

- previously accessed cache lines are stored in a smaller memory, called a cache, that sits near the processor
- Cache hits: accesses to data in cache are fast!
- Cache misses: accesses to data not in cache are slow!
  - It goes to the next cache level in the hierarchy or the main memory if LLC



Memory Layout

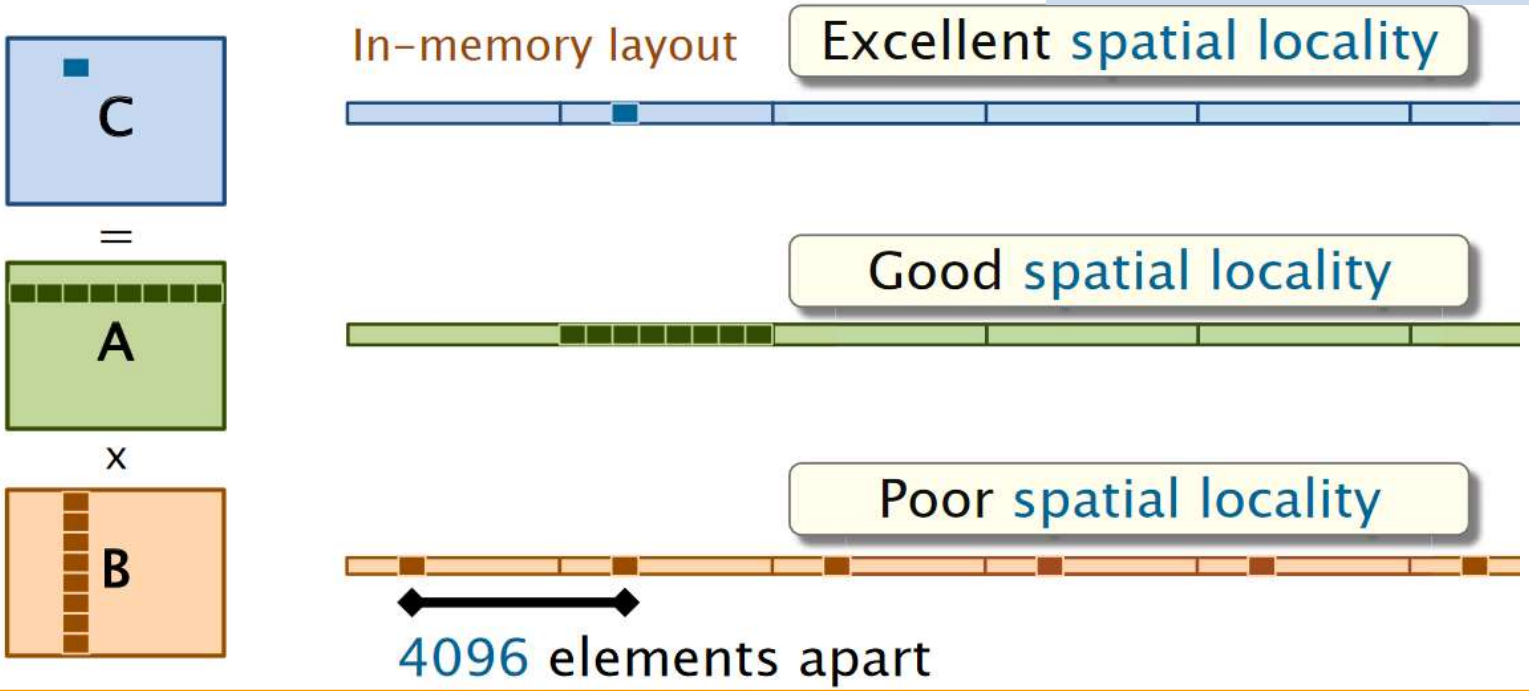
- In C, matrices are laid out in memory in row-major order
- What does this imply about the performance of different loop orders?



Access Pattern for Order i, j, k

```
for (int i=0; i<N; i++){
  for (int j=0; j<N; j++){
    for (int k=0; k<N; k++) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

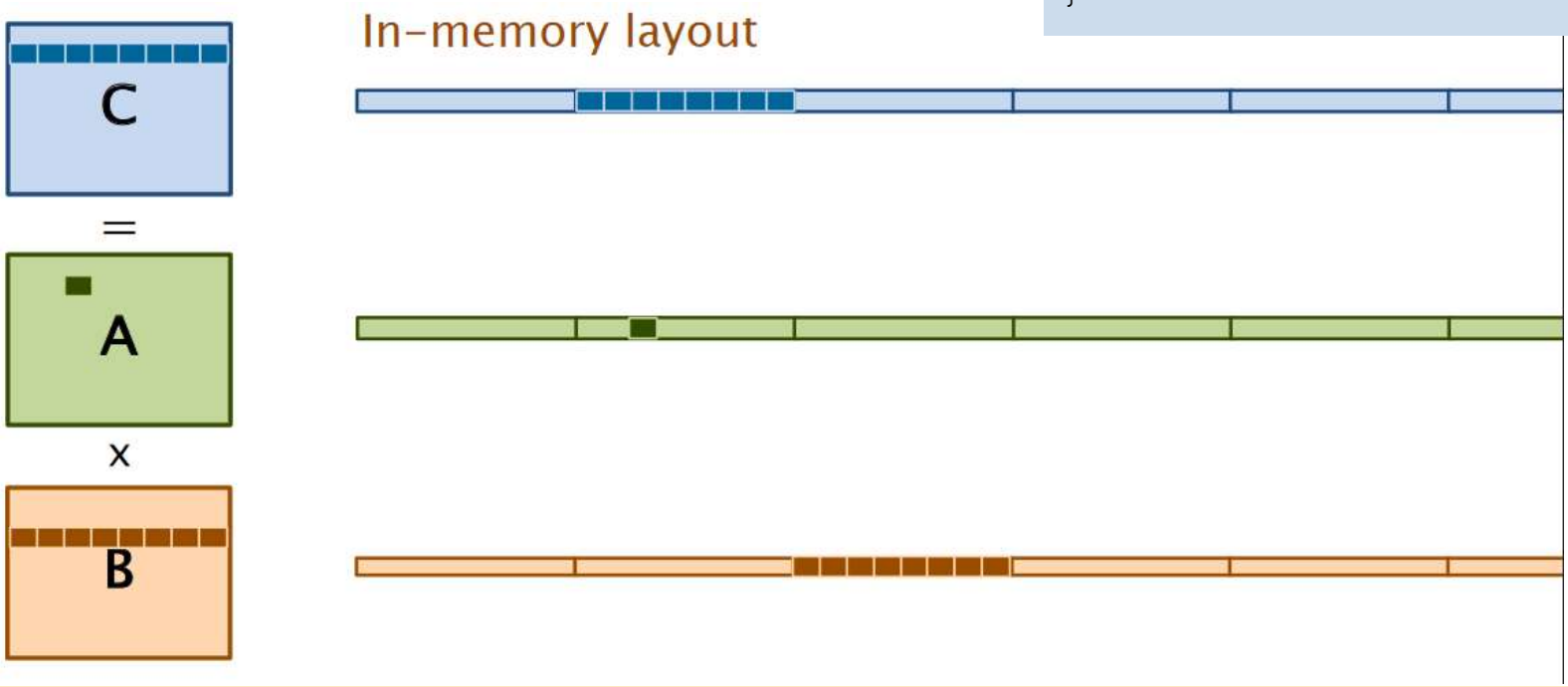
- Running time: 1155.77s



Access Pattern for Order i, k, j

```
for (int i=0; i<N; i++){
  for (int k=0; k<N; k++){
    for (int j=0; j<N; j++){
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

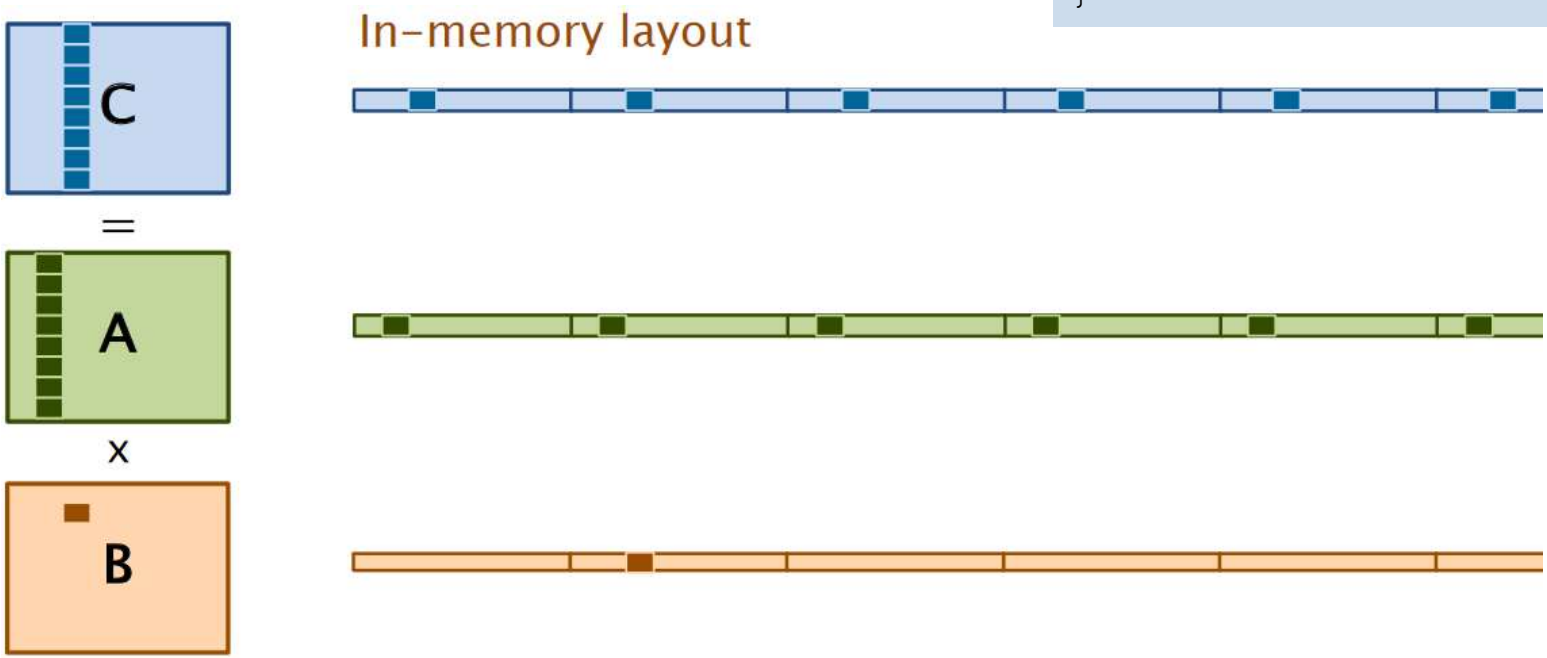
- Running time: 177.68s



Access Pattern for Order j, k, i

```
for (int j=0; j<N; j++){
  for (int k=0; k<N; k++){
    for (int i=0; i<N; i++){
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

- Running time: 3056.63s



Performance of Different Orders

Loop order (outer to inned loop)	Running time (s)	Last-level-cache miss rate
`i, j, k`	1155.77	7.7%
`i, k, j`	177.68	1.0%
`j, i, k`	1080.61	8.6%
`j, k, i`	3056.63	15.4%
`k, i, j`	179.21	1.0%
`k, j, i`	3032.82	15.4%

- Miss rate can be measured with cache simulator, e.g., Cachegrind

```
> valgrind –tool=cachegrind ./mm
```

Matmul – Version 4

Version	Implementation	Running time	Relative speedup	Absolute speedup	GLOPS	% of peak
1	Python	21041.67	1.00	1	0.007	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.119	0.014
4	+ interchange	177.68	6.50	118	0.774	0.093

- What else can we try?

Compiler Optimizations in Clang/LLVM

- Clang provides a collection of optimization switches
  - you can specify a switch to the compiler to ask it to optimize

Optimization level	Meaning	Time (s)
`-O0`	Do not optimize	177.54
`-O1`	Optimize	66.24
`-O2`	Optimize even more	54.63
`-O3`	Optimize yet more	55.58

Matmul with Optimizations Flags – Version 5

Version	Implementation	Running time	Relative speedup	Absolute speedup	GLOPS	% of peak
1	Python	21041.67	1.00	1	0.007	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.119	0.014
4	+ interchange	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301

- With simple code and compiler technology, we can achieve 0.3% of the peak performance
- What's causing the low performance?

Multicore Parallelism

- Intel Core Haswell E5
  - 9 core per chip
  - AWS text machine has 2 chips
  - Overall, 18 parallel processing cores
- Let's use more cores!

Feature	Specification
Microarchitecture	Haswell (Intel Xeon E5-2666 v3)
Clock frequency	2.9 GHz
Processor chips	2
Processing cores	9 per processor chip
Hyperthreading	2 way
Floating-point unit	8 double-precision operations, including fused-multiply-add, per core per cycle
Cache-line size	64 B
L1-icache	32 KB private 8-way set associative
L1-dcache	32 KB private 8-way set associative
L2-cache	256 KB private 8-way set associative
L3-cache (LLC)	25 MB shared 20-way set associative
DRAM	60 GB

$$\text{Peak} = (2.9 \times 10^9) \times 2 \times 9 \times 16 = 836 \text{ GFLOPS}$$

Parallel Loops

- Loop parallelism: allow all iterations of the loop to execute in parallel
  - multiple threads on multiple cores
- Two loops can be easily parallelized

```
for (int i=0; i<N; i++){
  for (int k=0; k<N; k++){
    for (int j=0; j<N; j++){
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

- How can we make a loop parallel?
  - In OpenMP:

```
#pragma omp parallel for
for (int i=0; i<N; i++){…}
```

  - In Cilk:

```
cilk_for (int i=0; i<N; i++){…}
```

Experimenting with Parallel Loops

Parallel i loop

```
cilk_for (int i=0; i<N; i++){
  for (int k=0; k<N; k++){
    for (int j=0; j<N; j++){
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

Running time: 3.18s

Parallel j loop

```
for (int i=0; i<N; i++){
  for (int k=0; k<N; k++){
    cilk_for (int j=0; j<N; j++){
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

Running time: 531.71s

Parallel i and j

```
cilk_for (int i=0; i<N; i++){
  for (int k=0; k<N; k++){
    cilk_for (int j=0; j<N; j++){
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```

Running time: 10.64s

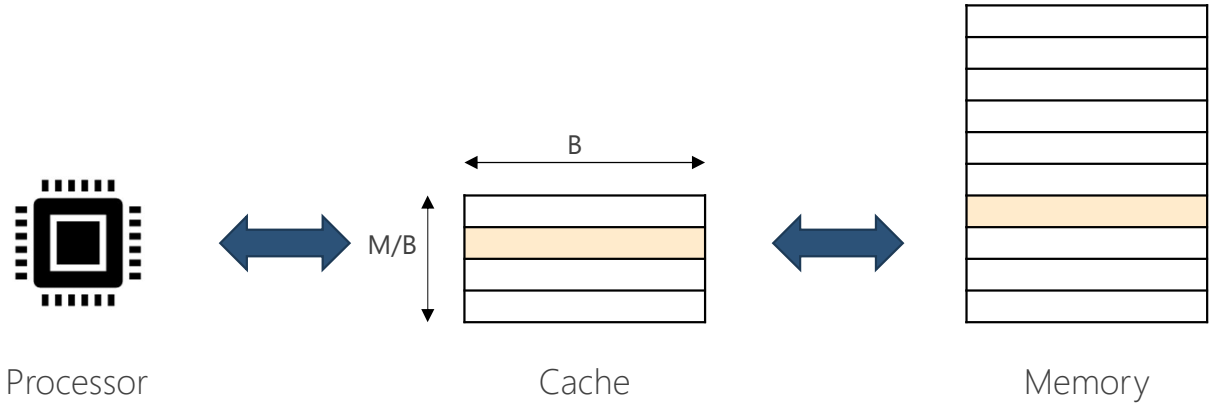
Matmul with Parallel Loops – Version 6

Version	Implementation	Running time	Relative speedup	Absolute speedup	GLOPS	% of peak
1	Python	21041.67	1.00	1	0.007	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.119	0.014
4	+ interchange	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	+ parallel loops	3.04	17.97	6.921	45.211	5.408

- Using parallel loops gets us almost 18x speedup on 18 cores!
  - This is not always the case
- Why are we still just getting 5% of the peak?

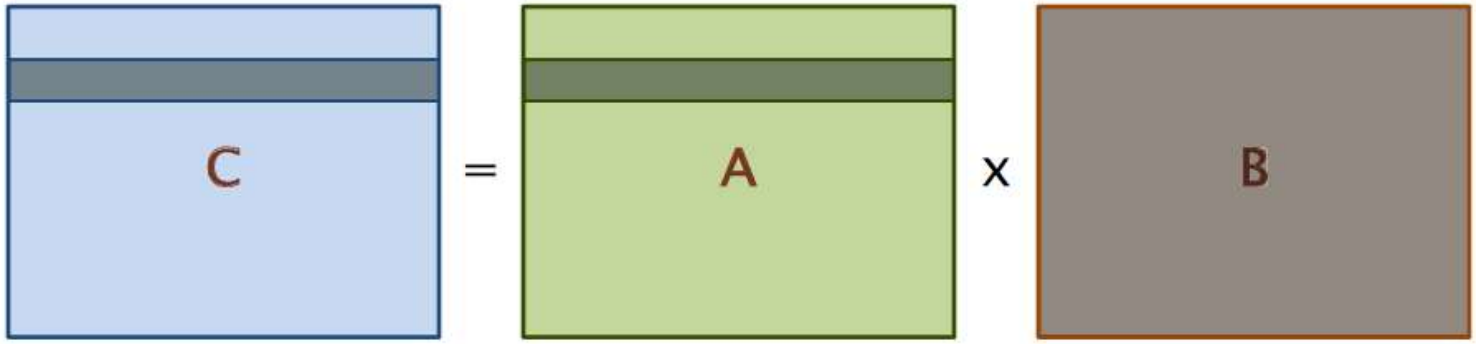
Caches, Revisited

- Idea: Restructure the computation to reuse data in the cache as much as possible
  - Cache misses are slow, and cache hits are fast
  - Try to make the most of the cache by reusing the data that's already there



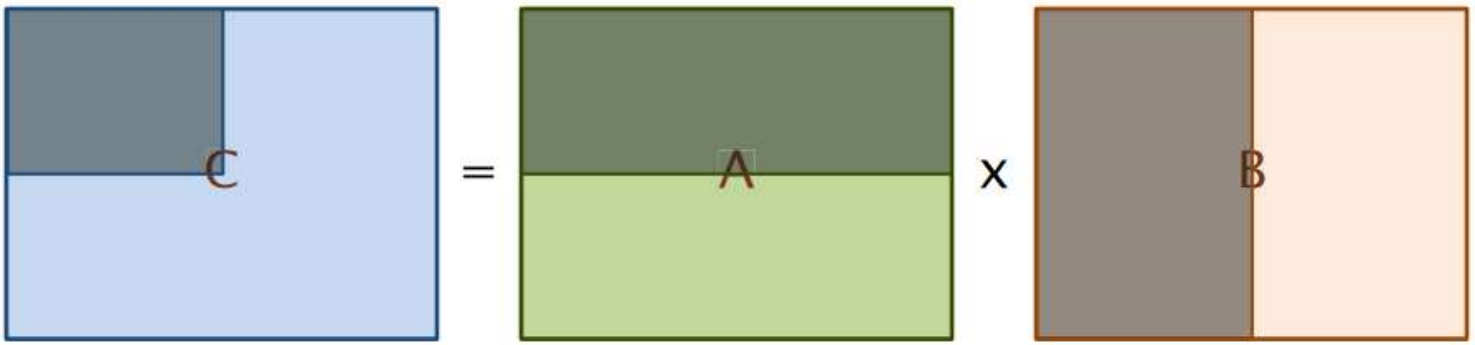
Data Reuse in Loops

- How many memory accesses must the looping code perform to fully compute 1 row of C?
  - 4096 * 1 = 4096 writes to C
  - 4096 * 1 = 4096 reads from A
  - 4096 * 4096 = 16,777,216 reads from B
  - which is 16,785,408 memory accesses total

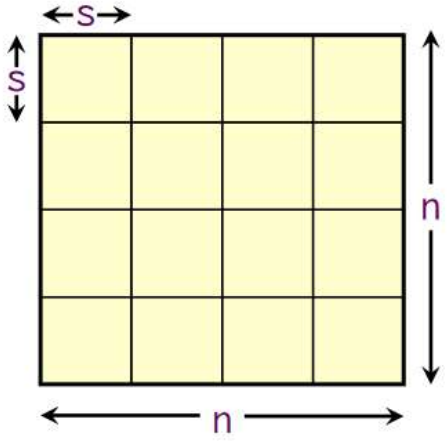


Data Reuse with Blocking (or Tiling)

- How about to compute a 64 x 64 block of C?
  - 64 · 64 = 4096 writes to C
  - 64 · 4096 = 262,144 reads from A,
  - 4096 · 64 = 262,144 reads from B,
  - 528,384 memory accesses total



Tiled Matmul - Version 7




```
cilk_for (int ih=0; ih<N; ih += s)
  cilk_for (int jh=0; jh<N; jh += s)
    for (int kh=0; kh<N; kh +=s)
      for (int il=0; il<s; il++)
        for (int kl=0; kl<s; kl++)
          for (int jl=0; jl<s; jl++)
            C[ih+il][jh+jl] += A[ih+il][kh+kl]*B[kh+kl][jh+jl];
```

- Iteration space organized in tiles
  - Two-level of matmul
    - from 3 to 6 loops
  - Tuning parameter: the tile size s

Tile size	Running time (s)
4	6.74
8	2.76
16	2.49
32	1.74
64	2.33
128	2.13

Matmul with Tiling – Version 7

Version	Implementation	Running time	Relative speedup	Absolute speedup	GLOPS	% of peak
1	Python	21041.67	1.00	1	0.007	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.119	0.014
4	+ interchange	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	+ parallel loops	3.04	17.97	6 921	45.211	5.408
7	+ tiling	1.79	1.70	11 772	76.782	9.184

- The tiled implementation performs about 62% fewer cache references and incurs 68% fewer cache misses
  - so far, we have optimized the L1 cache
- Can we still make better?

The Optimization Journey is not Finished!

- So far, we have seen
  - Loop Interchange (Compiler Module)
  - Loop parallelization (Multicore Module)
  - Tiling and L1 cache optimization (Compiler Module)
- We still have potential to
  - Optimized LLC cache and task parallelism (Multicore Module)
  - Pragma-based vectorization (Vectorization Module)
  - Intrinsic-based vectorization (Vectorization Module)
- Follow this course to know more!

Lab

- Experiment all code versions of matrix multiplication
  - Versions 3, 4, 5, 6, 7
  - Report your times in a table