CPSC 424:
Parallel Programming Techniques

Lecture 2: Matrix Multiplication, C++, OpenMP

Lecturer: Quanquan C. Liu

Spring 2026

Special thanks to **MIT 6.106** and the FASTCODE Community for part of this lecture's content!

C++ versus C

- C++ contains Object-Oriented Programming (OOP) like classes, objects, inheritance…etc.
  - Great for modularity and reuse
- C++ contains a standard library of ready-to-use data structures and algorithms
- Contains constructors and destructors for memory management
- Stricter type checking than C
- Approximately the same performance as C

C++ Standard Library

- Contains data structures, algorithms, classes, concurrency options...etc.
- **Data structures**
  - std::vector, std::map, std::set, std::deque
- **Algorithms**
  - std::sort, std::find, std::for_each
- **Stream operations**
  - std::cin, std::cout
- Use the namespace to the C++ standard library
  - using namespace std;

Last Class: Square Matrix Multiplication

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

C A B

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

Assume for simplicity that $n = 2^k$.

Slide courtesy of MIT 6.106.

AWS c4.8xlarge Machine Specs

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

Slide courtesy of MIT 6.106.

Nested Loops in Python

```
import sys, random
from time import *

n = 4096

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

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

print '%0.6f' % (end - start)
```

Slide courtesy of MIT 6.106.

Nested Loops in Python

Is this fast?

Should we expect more from our machine?

```
import sys, random
from time import *

n = 4096

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

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

print '%0.6f' % (end - start)
```

Running time:

≈ 6 microseconds?

≈ 6 milliseconds?

≈ 6 seconds?

≈ 6 hours?

≈ 6 days?

Slide courtesy of MIT 6.106.

Nested Loops in Python

Back-of-the-envelope calculation

$2n^3 = 2(2^{12})^3 = 2^{37}$ floating-point operations
Running time = 21042 seconds
∴ Python gets $2^{37}/21042 \approx 6.25$ MFLOPS
Peak ≈ 836 GFLOPS
Python gets ≈ 0.00075% of peak

Slide courtesy of MIT 6.106.

How do we make matrix multiplication faster?

- Keep the $\Theta(n^3)$ asymptotic running time
- Consider programming techniques that improves real-life performance
- First, consider a different language
- Second, consider chip architecture
- Third, consider compiler optimizations

How do we make matrix multiplication faster?

- Keep the $\Theta(n^3)$ asymptotic running time
- Consider programming techniques that improves real-life performance
- **First, consider a different language**
- Second, consider chip architecture
- Third, consider compiler optimizations

Changing the Language

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent 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

Java has **8.81x** speedup from Python

C has **18.21x** speedup from Python

Numbers courtesy of MIT 6.106.

Changing the Language

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent 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

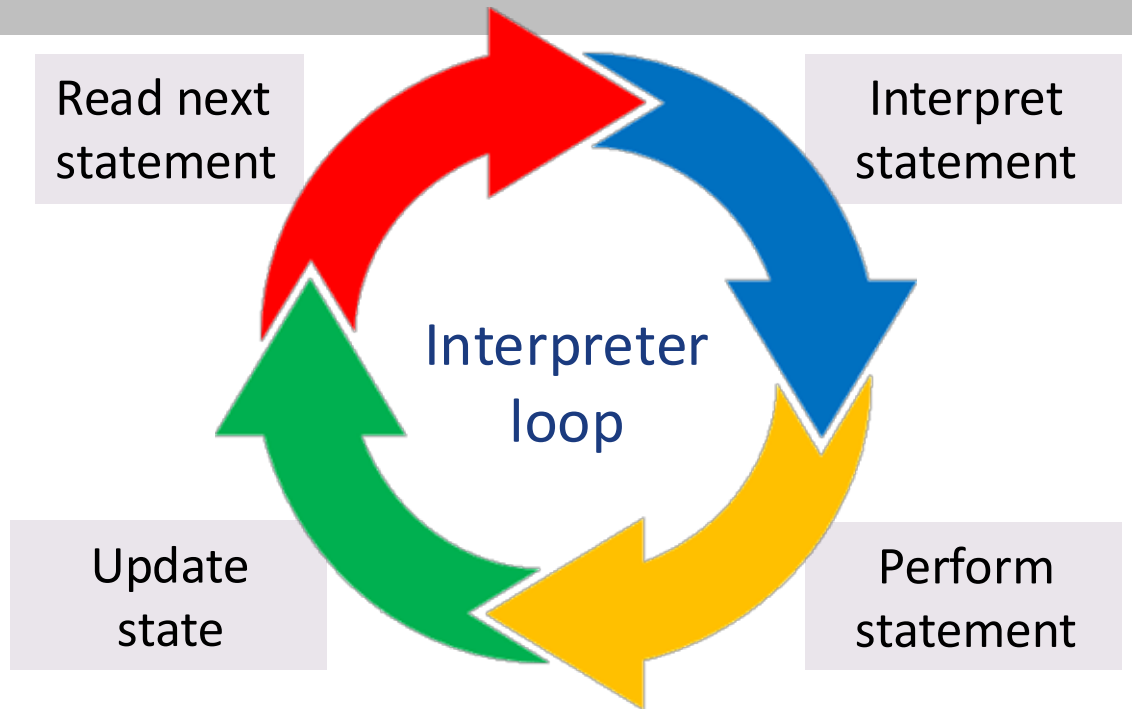
Why is Python so slow and C so fast?

- 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.

Slide courtesy of MIT 6.106

What is an interpreter?

- Runs a program by performing the following per statement:
  - Read each statement
  - Interpret what it means
  - Execute logic to perform the statement
  - Updating program state



Slide courtesy of MIT 6.106.

What is an interpreter?

- At **runtime**, the interpreter spends a lot of time dynamically reading the program
- Much time spent reading and interpreting the program **before even running it**!
- No need to compile the program when running it; **it runs immediately**
  - Can support dynamically generating and changing code at runtime
- **Very slow** because of the reading and interpreting overhead

What is a compiler?

- On the other hand, C is a **compiled language**
- A compiler compiles a C code into **machine code** before running (execution)
- C compilation process consists of four distinct stages:
  - **Pre-processing**: the compiler handles lines starting with # including header files and expanding macros
  - **Compilation**: Translated preprocessed code into assembly
  - **Assembly**: assembler converts assembly code into machine code
  - **Linking**: linker combines different files to create executable

What is a compiler?

- **Advantages**:
  - No runtime interpretation overhead; the machine code is directly run
  - Well-suited for performance
  - Compilers perform extensive optimizations
- **Disadvantages**:
  - Platform dependent: need to recompile for different OS or architectures
  - Long compilation time (sometimes longer than the execution time!)
  - Difficult debugging runtime errors (e.g. segfault…)

Best of Both Worlds: Just-in-Time 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

Slide courtesy of MIT 6.106.

Best of Both Worlds: Just-in-Time Compilation

Java-–first compiles into bytecode

- **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

Slide courtesy of MIT 6.106.

Changing the Language

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent 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

Still takes 19 minutes!

Performance engineering is more than "write your code in C/C++"!

Slide courtesy of MIT 6.106

How do we make matrix multiplication faster?

- Keep the $\Theta(n^3)$ asymptotic running time
- Consider programming techniques that improves real-life performance
- First, consider a different language
- **Second, consider chip architecture**
- Third, consider compiler optimizations

Loop Order

```
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];
    }
  }
}
```

Loops can be executed in any order!

Does the order of loops matter for performance?

Slide courtesy of MIT 6.106

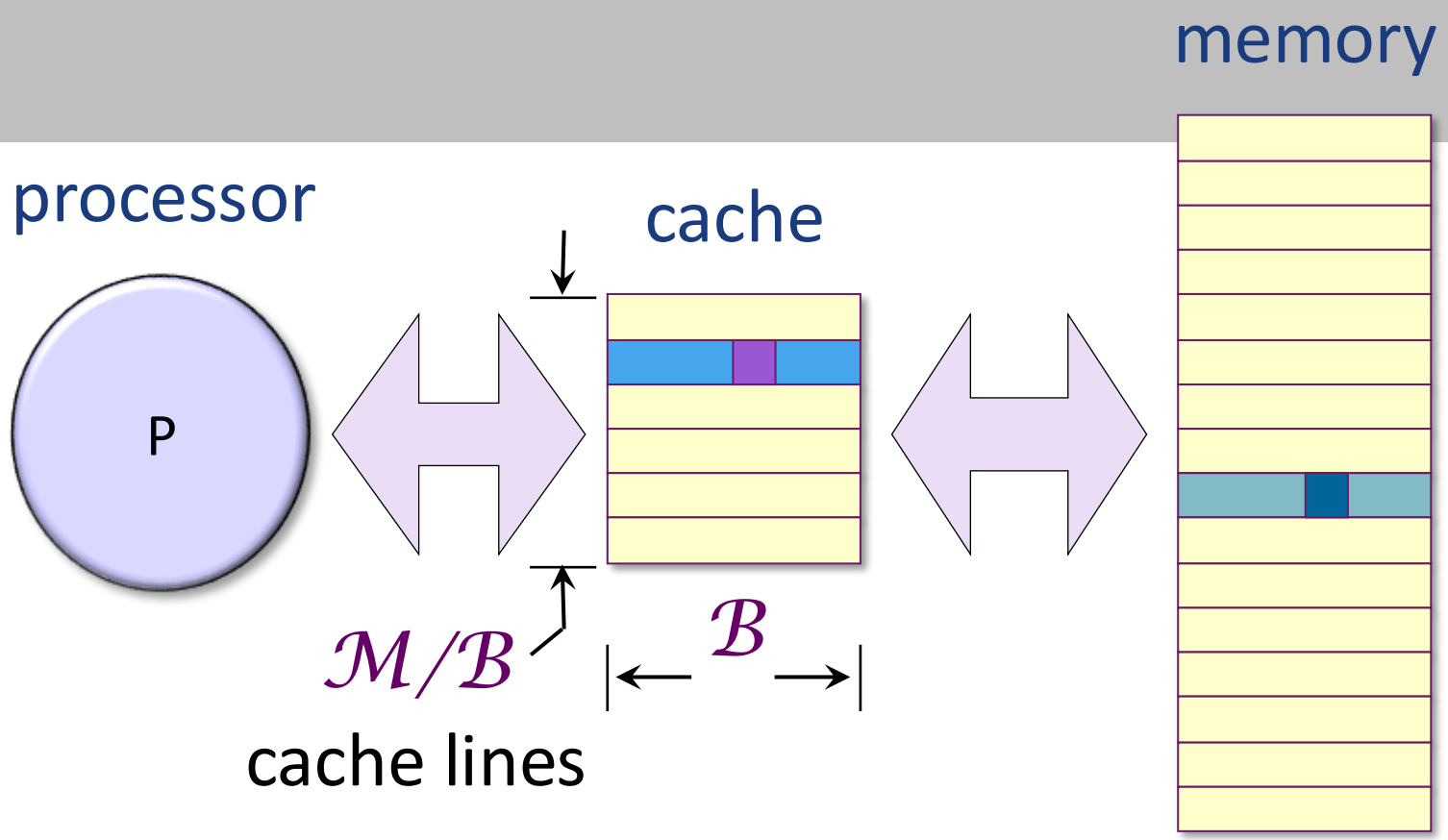
Performance of Different Loop Orders

Loop order (outer to inner)	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

- Loop order affects running time by a factor of **18x**!
- What's going on?

Numbers courtesy of MIT 6.106.

Hardware Caches

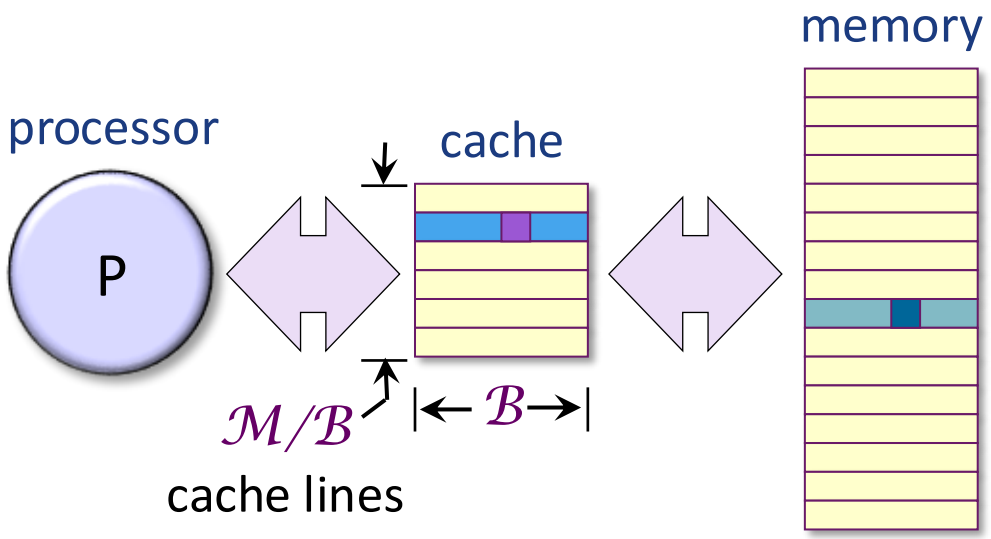


$\mathcal{M}$: memory of the cache
$\mathcal{B}$: words in a cache line

Slide courtesy of MIT 6.106

Hardware 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



Slide courtesy of MIT 6.106.

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

A[][] is a 2-dimensional array

A[1][3] is accessing the element in the first row, third column

Memory

Row 1	Row 2	Row 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

What does this layout imply about the performance of different loop orders?

Memory

Row 1	Row 2	Row 3

Memory Layout of Matrices

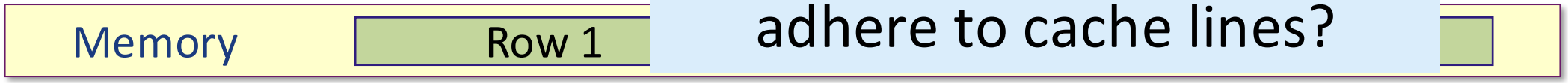
In this matrix-multiplication code, matrices are laid out in memory in ***row-major order***

Matrix

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

What does this layout imply about the performance of different loop orders?

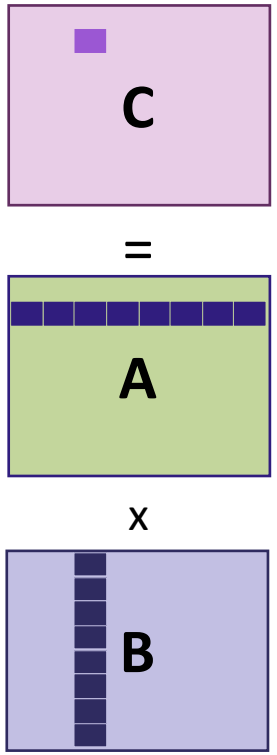
Spatial Locality: do the access pattern of the data adhere to cache lines?



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

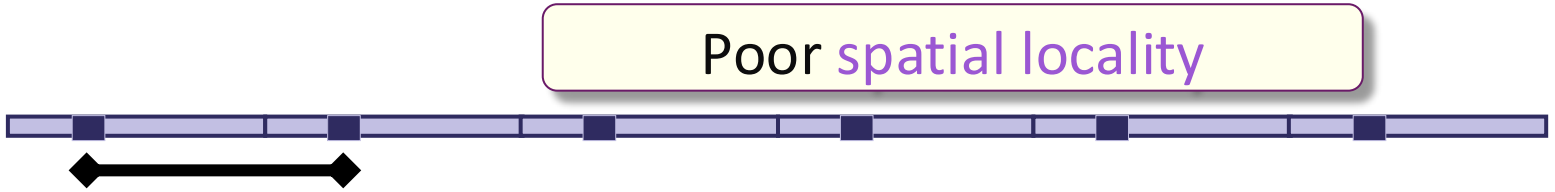


In-memory layout

Excellent spatial locality

Good spatial locality

Poor spatial locality



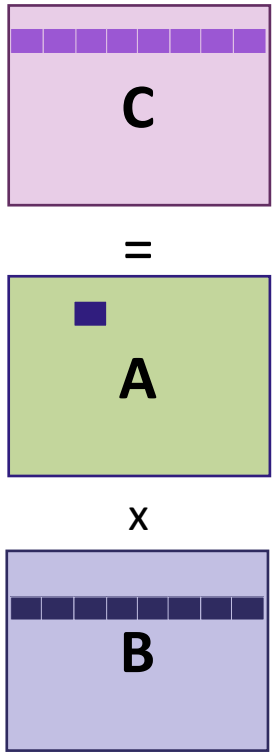
4096 elements apart

Slide courtesy of MIT 6.106

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



In-memory layout

Good spatial locality

Excellent spatial locality

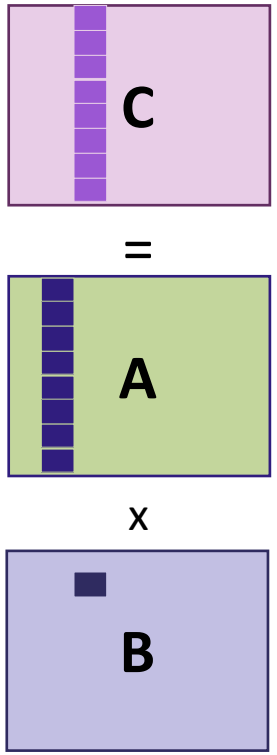
Good spatial locality

Slide courtesy of MIT 6.106

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



In-memory layout

Poor spatial locality

Poor spatial locality

Excellent spatial locality

Slide courtesy of MIT 6.106

Performance of Different Orders

valgrind is an open-source tool for dynamic analysis of programs

Performance of Different Orders

We can measure the effect of different access patterns using the Cachegrind cache simulator:

```
$ valgrind --tool=cachegrind ./mm
```

Loop order (outer to inner)	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%

valgrind is an open-source tool for dynamic analysis of programs

Slide courtesy of MIT 6.106

With Interchange Loops

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093

What else can we try?

Slide courtesy of MIT 6.106

Compiler Optimization

GCC (compiler) provides a collection of optimization switches. You can specify a switch to the compiler to ask it to optimize.

Opt. 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

-O2 performs slightly better than –O3 probably due to normal variance between runs. Generally –O3 will give you the best optimization

Slide courtesy of MIT 6.106

Optimization Flags

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	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 of the machine.

What more can we do?

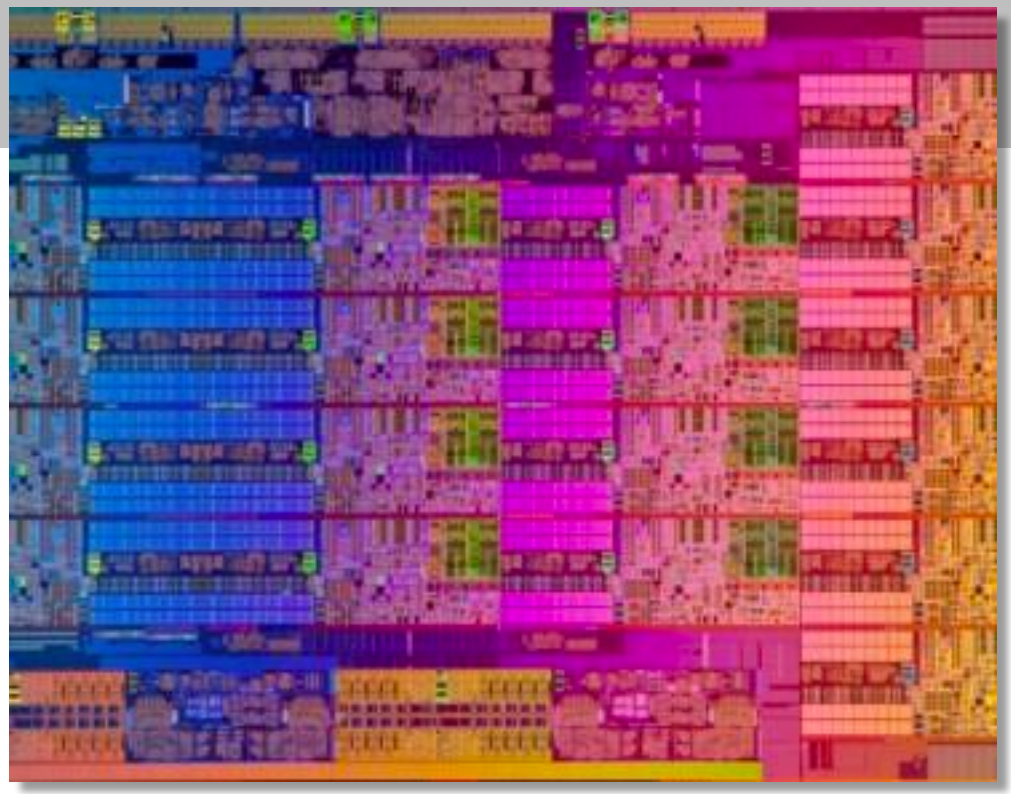
Numbers courtesy of MIT 6.106.

AWS c4.8xlarge Machine Specs

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

Slide courtesy of MIT 6.106.

Use the Power of 4240/5240



Intel Haswell E5:
9 cores per chip

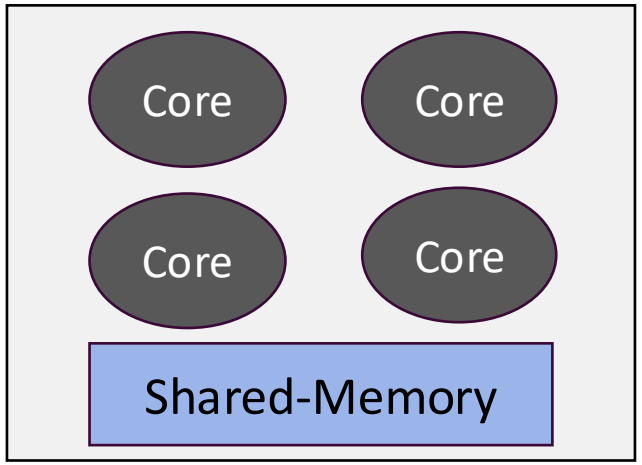
The AWS test machine has **2** of these chips.

We're running on just **1** of the **18** parallel-processing cores on this system. ***Let's use them all!***

Slide courtesy of MIT 6.106

Parallelism

- Parallel algorithms and programs split the computation among **many cores** on the same chip
- Each core can perform computation **simultaneously**, called **“in parallel”**
- Each core reads and writes from the **same shared-memory**



Parallelism

- Some computations can be parallelized while others cannot be!
- **Which of these operations can be done simultaneously and which cannot?**

$$(4 + 3) \cdot (3 + 5) + 6 + 7 \cdot 8 + 10/5$$

In other words, which operations **depend** on other operations before they can be performed?

Parallelism

- **Which of these operations can be done simultaneously and which cannot?**

$$(4+3)\cdot(3+5)+6+7\cdot 8+10/5$$

Parallelism

- **Which of these operations can be done simultaneously and which cannot?**

$$(4+3)\cdot(3+5)+6+7\cdot 8+10/5$$

$$7\cdot 8+6+56+2$$

Parallelism

- **Which of these operations can be done simultaneously and which cannot?**

$$(4+3)\cdot(3+5)+6+7\cdot 8+10/5$$

$$7\cdot 8+6+56+2$$

$$7\cdot 8+6+56+2$$

Parallelism

- **Which of these operations can be done simultaneously and which cannot?**

$$(4+3)\cdot(3+5)+6+7\cdot 8+10/5$$

$$7\cdot 8+6+56+2$$

$$7\cdot 8+6+56+2$$

$$56+62+2$$

Parallelism

- **Which of these operations can be done simultaneously and which cannot?**

$$(4+3)\cdot(3+5)+6+7\cdot 8+10/5$$

$$7\cdot 8+6+56+2$$

$$7\cdot 8+6+56+2$$

$$56+62+2$$

$$118+2$$

Parallelism

- **Which of these operations can be done simultaneously and which cannot?**

$$(4+3)\cdot(3+5)+6+7\cdot 8+10/5$$

$$7\cdot 8+6+56+2$$

$$7\cdot 8+6+56+2$$

$$56+62+2$$

$$118+2$$

$$120$$

OpenMP

- **OpenMP** is a parallel API for C/C++
- Uses compiler directives (instructions for the compiler) to specify which parts of the code should be executed in parallel
- `#pragma omp` is the prefix for the directives
- Fun fact: `#pragma` generally indicates compiler directives
- Compiler directives are not part of the standard C/C++ language but are specific to certain compilers and extensions like OpenMP
- Uses **threads for parallelism**; a **thread** is a lightweight unit of execution within a process

OpenMP: Parallel For

```
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    // Code to be executed in parallel by multiple threads
}
```

Creates a thread for each i and executes in parallel

Note: the code inside must be independent of other iterations!

Parallel Loops

How does parallel for loops help us in matrix multiplication?

```
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];
```

Which of these for loops can be parallelized without issues?

What are the dependencies for each computation?

Parallel Loops

```
parallel_for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    parallel_for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

Both of these loops can theoretically be parallelized

Some packages allow you to parallelize **both** simultaneously easily; others do not

Parallel Loops

```
parallel_for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    parallel_for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

OpenMP does not allow you to parallelize both simultaneously

Parallel Loops

Which is better?

```
#pragma omp parallel for
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];
```

```
for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    #pragma omp parallel for
    for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

Parallel Loops

Explore in your homework!

```
#pragma omp parallel for
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];
```

```
for (int i = 0; i < n; ++i)
  for (int k = 0; k < n; ++k)
    #pragma omp parallel for
    for (int j = 0; j < n; ++j)
      C[i][j] += A[i][k] * B[k][j];
```

Parallel Loops

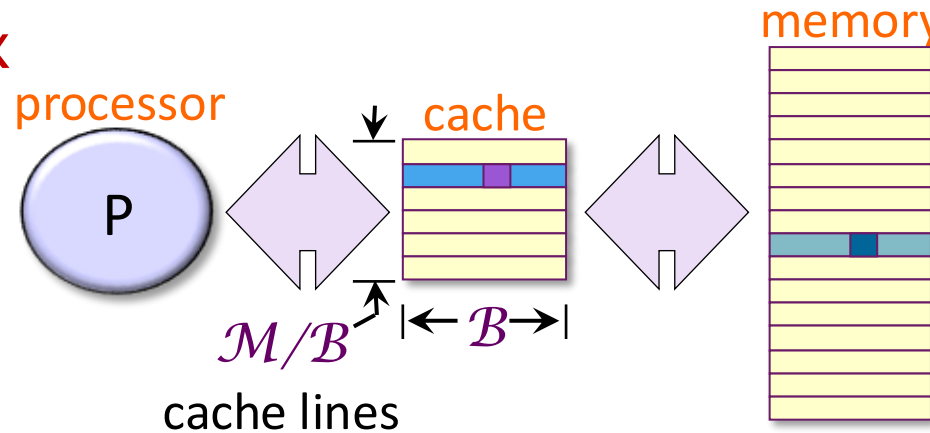
Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	Parallel loops	3.04	17.97	6,921	45.211	5.408

Using parallel loops gets us almost 18× speedup on 18 cores! (Disclaimer: Not all code is so easy to parallelize effectively.)

Why are we still getting less than 5% of peak?

Hardware 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
- Cache Capacity
  - One Row of a matrix = 4096*8bytes = 32KB
  - L1D cache = 32KB → 1 row of **one** matrix
  - Can't fit all rows from all three matrices!



Slide courtesy of MIT 6.106

Divide-and-Conquer Matrix Multiplication

For matrix multiplication, a recursive, parallel, divide-and-conquer algorithm uses caches almost optimally.

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

IDEA: Divide the matrices into $(n/2)\times(n/2)$ submatrices.

Slide courtesy of MIT 6.106

IDEA: For matrix multiplication, a recursive, parallel, divide-and-conquer algorithm uses caches almost optimally.

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

$$= \begin{pmatrix} A_{00}B_{00} & A_{00}B_{01} \\ A_{10}B_{00} & A_{10}B_{01} \end{pmatrix} + \begin{pmatrix} A_{01}B_{10} & A_{01}B_{11} \\ A_{11}B_{10} & A_{11}B_{11} \end{pmatrix}$$

1. Compute $C_{00} \mathrel{+}= A_{00}B_{00}$; $C_{01} \mathrel{+}= A_{00}B_{01}$; $C_{10} \mathrel{+}= A_{10}B_{00}$; and $C_{11} \mathrel{+}= A_{10}B_{01}$ recursively in parallel.
2. Compute $C_{00} \mathrel{+}= A_{01}B_{10}$; $C_{01} \mathrel{+}= A_{01}B_{11}$; $C_{10} \mathrel{+}= A_{11}B_{10}$; and $C_{11} \mathrel{+}= A_{11}B_{11}$ recursively in parallel.

Slide courtesy of MIT 6.106

Parallel Divide-and-Conquer

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	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	Parallel divide-and-conquer	1.30	2.35	16,197	105.722	12.646

Implementation	Cache references × 10^6	L1-d cache misses × 10^6	Last-level cache misses × 10^6
Parallel loops	104,090	17,220	8,600
Parallel divide-and-conquer	58,230	9,407	64

Slide courtesy of MIT 6.106

Parallel Divide-and-Conquer

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	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	Parallel divide-and-conquer	1.30	2.35	16,197	105.722	12.646

Implementation	Cache references × 10^6	L1-d cache misses × 10^6	Last-level cache misses × 10^6
Parallel loops	104,090	17,220	8,600
Parallel divide-and-conquer	58,230	9,407	64

16185.9x improvement

Slide courtesy of MIT 6.106