Lecture 1
Introduction & Matrix Multiplication
Charles E. Leiserson
WHY PERFORMANCE ENGINEERING?
Software Properties

What software properties are more important than performance?

- Compatibility
- Correctness
- Clarity
- Deuggability

- Functionality
- Maintainability
- Modularity
- Portability

- Reliability
- Robustness
- Testability
- Usability

... and more.

If programmers are willing to sacrifice performance for these properties, why study performance?

Performance is the currency of computing. You can often “buy” needed properties with performance.
Software performance engineering was common, because machine resources were limited.

IBM System/360
- Launched: 1964
- Clock rate: 33 KHz
- Data path: 32 bits
- Memory: 524 Kbytes
- Cost: $5,000/month

DEC PDP–11
- Launched: 1970
- Clock rate: 1.25 MHz
- Data path: 16 bits
- Memory: 56 Kbytes
- Cost: $20,000

Apple II
- Launched: 1977
- Clock rate: 1 MHz
- Data path: 8 bits
- Memory: 48 Kbytes
- Cost: $1,395

Many programs strained the machine’s resources.
- Programs had to be planned around the machine.
- Many programs would not “fit” without intense performance engineering.
Lessons Learned from the 70’s and 80’s

Premature optimization is the root of all evil. [K79]

Donald Knuth

More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason — including blind stupidity. [W79]

William Wulf

The First Rule of Program Optimization: Don’t do it.
The Second Rule of Program Optimization — For experts only: Don’t do it yet. [J88]

Michael Jackson
Technology Scaling Until 2004

Processor data from Stanford’s CPU DB [DKM12].
Technology Scaling Until 2004

Normalized transistor count

Clock speed (MHz)

"Dennard scaling"

Processor data from Stanford’s CPU DB [DKM12].

© 2008–2018 by the MIT 6.172 Lecturers
## Advances in Hardware

### Apple computers with similar prices from 1977 to 2004

<table>
<thead>
<tr>
<th>Model</th>
<th>Launched</th>
<th>Clock rate</th>
<th>Data path</th>
<th>Memory</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Apple II</strong></td>
<td>1977</td>
<td>1 MHz</td>
<td>8 bits</td>
<td>48 KB</td>
<td>$1,395</td>
</tr>
<tr>
<td><strong>Power Macintosh G4</strong></td>
<td>2000</td>
<td>400 MHz</td>
<td>32 bits</td>
<td>64 MB</td>
<td>$1,599</td>
</tr>
<tr>
<td><strong>Power Macintosh G5</strong></td>
<td>2004</td>
<td>1.8 GHz</td>
<td>64 bits</td>
<td>256 MB</td>
<td>$1,499</td>
</tr>
</tbody>
</table>

© 2008–2018 by the MIT 6.172 Lecturers
Until 2004

Moore’s Law and the scaling of clock frequency = printing press for the currency of performance.
Technology Scaling After 2004

Normalized transistor count

Clock speed (MHz)

Processor data from Stanford’s CPU DB [DKM12].

© 2008–2018 by the MIT 6.172 Lecturers
The growth of power density, as seen in 2004, if the scaling of clock frequency had continued its trend of 25\%-30\% increase per year.
Vendor Solution: Multicore

Intel Core i7 3960X (Sandy Bridge E), 2011
- 6 cores
- 3.3 GHz
- 15–MB L3 cache

To scale performance, processor manufacturers put many processing cores on the microprocessor chip.

Each generation of Moore’s Law potentially doubles the number of cores.
Technology Scaling

Processor data from Stanford’s CPU DB [DKM12].

© 2008–2018 by the MIT 6.172 Lecturers
Moore’s Law continues to increase computer performance. But now that performance looks like big multicore processors with complex cache hierarchies, wide vector units, GPU’s, FPGA’s, etc. Generally, software must be adapted to utilize this hardware efficiently!
Software Bugs Mentioning “Performance”

**Bug reports for Mozilla “Core”**

**Commit messages for MySQL**

**Commit messages for OpenSSL**

**Bug reports for the Eclipse IDE**
Software Developer Jobs

Mentioning “performance”

Mentioning “optimization”

Mentioning “parallel”

Mentioning “concurrency”

Source: Monster.com

© 2008–2018 by the MIT 6.172 Lecturers
A modern multicore desktop processor contains parallel-processing cores, vector units, caches, prefetchers, GPU’s, hyperthreading, dynamic frequency scaling, etc., etc.

How can we write software to utilize modern hardware efficiently?
CASE STUDY
MATRIX MULTIPLICATION
Square–Matrix Multiplication

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

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

Assume for simplicity that \( n = 2^k \).
# AWS c4.8xlarge Machine Specs

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

Peak = \((2.9 \times 10^9) \times 2 \times 9 \times 16 = 836\) GFLOPS

© 2008–2018 by the MIT 6.172 Lecturers
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
= 21042 seconds
≈ 6 hours

Is this fast?

Should we expect more from our machine?
Version 1: Nested Loops in Python

```python
import sys, random
from time import *

n = 4096

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

B =

C =

start =
for
del
end

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

Running time
\[= 21042 \text{ seconds} \approx 6 \text{ hours} \]

Is this fast?

Back-of-the-envelope calculation

\[
2n^3 = 2(2^{12})^3 = 2^{37} \text{ floating-point operations}
\]

Running time \[= 21042 \text{ seconds} \]

\[
\therefore \text{Python gets } 2^{37}/21042 \approx 6.25 \text{ MFLOPS}
\]

Peak \[\approx 836 \text{ GFLOPS} \]

Python gets \[\approx 0.00075\% \text{ of peak} \]
Version 2: Java

```java
import java.util.Random;

public class mm_java {
    static int n = 4096;
    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);
    }
}
```

Running time = 2,738 seconds
≈ 46 minutes
... about \(8.8 \times\) faster than Python.
Using the Clang/LLVM 5.0 compiler

Running time = 1,156 seconds 
≈ 19 minutes, 
or about $2 \times$ faster than Java and about $18 \times$ faster than Python.

```
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        C[i][j] += A[i][k] * B[k][j];
    }
}
```
### Where We Stand So Far

<table>
<thead>
<tr>
<th>Version</th>
<th>Implementation</th>
<th>Running time (s)</th>
<th>Relative speedup</th>
<th>Absolute Speedup</th>
<th>GFLOPS</th>
<th>Percent of peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Python</td>
<td>21041.67</td>
<td>1.00</td>
<td>1</td>
<td>0.007</td>
<td>0.001</td>
</tr>
<tr>
<td>2</td>
<td>Java</td>
<td>2387.32</td>
<td>8.81</td>
<td>9</td>
<td>0.058</td>
<td>0.007</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>1155.77</td>
<td>2.07</td>
<td>18</td>
<td>0.119</td>
<td>0.014</td>
</tr>
</tbody>
</table>

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

• 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.
Loop Order

We can change the order of the loops in this program without affecting its correctness.

```java
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

We can change the order of the loops in this program without affecting its correctness.

```c
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];
        }
    }
}
```

Does the order of loops matter for performance?
## Performance of Different Orders

<table>
<thead>
<tr>
<th>Loop order (outer to inner)</th>
<th>Running time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i, j, k</td>
<td>1155.77</td>
</tr>
<tr>
<td>i, k, j</td>
<td>177.68</td>
</tr>
<tr>
<td>j, i, k</td>
<td>1080.61</td>
</tr>
<tr>
<td>j, k, i</td>
<td>3056.63</td>
</tr>
<tr>
<td>k, i, j</td>
<td>179.21</td>
</tr>
<tr>
<td>k, j, i</td>
<td>3032.82</td>
</tr>
</tbody>
</table>

Loop order affects running time by a factor of 18!

What’s going on!?
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.
In this matrix-multiplication code, matrices are laid out in memory in *row-major order*.

What does this layout 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

In-memory layout

- Excellent spatial locality
- Good spatial locality
- Poor spatial locality

4096 elements apart
```
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];
```

In-memory layout

**Running time:** 177.68s
Access Pattern for Order j, k, i

```c
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

© 2008-2018 by the MIT 6.172 Lecturers
Performance of Different Orders

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

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

<table>
<thead>
<tr>
<th>Loop order (outer to inner)</th>
<th>Running time (s)</th>
<th>Last-level-cache miss rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>i, j, k</td>
<td>1155.77</td>
<td>7.7%</td>
</tr>
<tr>
<td>i, k, j</td>
<td>177.68</td>
<td>1.0%</td>
</tr>
<tr>
<td>j, i, k</td>
<td>1080.61</td>
<td>8.6%</td>
</tr>
<tr>
<td>j, k, i</td>
<td>3056.63</td>
<td>15.4%</td>
</tr>
<tr>
<td>k, i, j</td>
<td>179.21</td>
<td>1.0%</td>
</tr>
<tr>
<td>k, j, i</td>
<td>3032.82</td>
<td>15.4%</td>
</tr>
</tbody>
</table>
### Version 4: Interchange Loops

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

What other simple changes we can try?
Clang provides a collection of optimization switches. You can specify a switch to the compiler to ask it to optimize.

<table>
<thead>
<tr>
<th>Opt. level</th>
<th>Meaning</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-00</td>
<td>Do not optimize</td>
<td>177.54</td>
</tr>
<tr>
<td>-01</td>
<td>Optimize</td>
<td>66.24</td>
</tr>
<tr>
<td>-02</td>
<td>Optimize even more</td>
<td>54.63</td>
</tr>
<tr>
<td>-03</td>
<td>Optimize yet more</td>
<td>55.58</td>
</tr>
</tbody>
</table>

Clang also supports optimization levels for special purposes, such as –Os, which aims to limit code size, and –Og, for debugging purposes.
## Version 5: Optimization Flags

<table>
<thead>
<tr>
<th>Version</th>
<th>Implementation</th>
<th>Running time (s)</th>
<th>Relative speedup</th>
<th>Absolute Speedup</th>
<th>GFLOPS</th>
<th>Percent of peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Python</td>
<td>21041.67</td>
<td>1.00</td>
<td>1</td>
<td>0.006</td>
<td>0.001</td>
</tr>
<tr>
<td>2</td>
<td>Java</td>
<td>2387.32</td>
<td>8.81</td>
<td>9</td>
<td>0.058</td>
<td>0.007</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>1155.77</td>
<td>2.07</td>
<td>18</td>
<td>0.118</td>
<td>0.014</td>
</tr>
<tr>
<td>4</td>
<td>+ interchange loops</td>
<td>177.68</td>
<td>6.50</td>
<td>118</td>
<td>0.774</td>
<td>0.093</td>
</tr>
<tr>
<td>5</td>
<td>+ optimization flags</td>
<td>54.63</td>
<td>3.25</td>
<td>385</td>
<td>2.516</td>
<td>0.301</td>
</tr>
</tbody>
</table>

With simple code and compiler technology, we can achieve 0.3% of the peak performance of the machine.

What’s causing the low performance?
Multicore Parallelism

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!

© Intel. All rights reserved. This content is excluded from our Creative Commons license.
For more information, see https://ocw.mit.edu/help/faq-fair-use/

© 2008–2018 by the MIT 6.172 Lecturers
Parallel Loops

The *cilk_for* loop allows all iterations of the loop to execute in parallel.

```
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];
```

These loops can be (easily) parallelized.

Which parallel version works best?
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

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

Rule of Thumb
Parallelize outer loops rather than inner loops.

Running time: 531.71s

Parallel i and j

```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: 10.64s
## Version 6: Parallel Loops

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

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 just 5% of peak?
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: Loops

How many memory accesses must the looping code perform to fully compute 1 row of \( C \)?

- \( 4096 \times 1 = 4096 \) writes to \( C \),
- \( 4096 \times 1 = 4096 \) reads from \( A \), and
- \( 4096 \times 4096 = 16,777,216 \) reads from \( B \), which is
- \( 16,785,408 \) memory accesses total.
How about to compute a $64 \times 64$ block of $C$?

- $64 \cdot 64 = 4096$ writes to $C$,
- $64 \cdot 4096 = 262,144$ reads from $A$, and
- $4096 \cdot 64 = 262,144$ reads from $B$, or
- $528,384$ memory accesses total.
Tiled Matrix Multiplication

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)
          C[ih+il][jh+jl] += A[ih+il][kh+kl] * B[kh+kl][jn+jl];

<table>
<thead>
<tr>
<th>Tile size</th>
<th>Running time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6.74</td>
</tr>
<tr>
<td>8</td>
<td>2.76</td>
</tr>
<tr>
<td>16</td>
<td>2.49</td>
</tr>
<tr>
<td>32</td>
<td>1.74</td>
</tr>
<tr>
<td>64</td>
<td>2.33</td>
</tr>
<tr>
<td>128</td>
<td>2.13</td>
</tr>
</tbody>
</table>

© 2008–2018 by the MIT 6.172 Lecturers
# Version 7: Tiling

<table>
<thead>
<tr>
<th>Version</th>
<th>Implementation</th>
<th>Running time (s)</th>
<th>Relative speedup</th>
<th>Absolute Speedup</th>
<th>GFLOPS</th>
<th>Percent of peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Python</td>
<td>21041.67</td>
<td>1.00</td>
<td>1</td>
<td>0.006</td>
<td>0.001</td>
</tr>
<tr>
<td>2</td>
<td>Java</td>
<td>2387.32</td>
<td>8.81</td>
<td>9</td>
<td>0.058</td>
<td>0.007</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>1155.77</td>
<td>2.07</td>
<td>18</td>
<td>0.118</td>
<td>0.014</td>
</tr>
<tr>
<td>4</td>
<td>+ interchange loops</td>
<td>177.68</td>
<td>6.50</td>
<td>118</td>
<td>0.774</td>
<td>0.093</td>
</tr>
<tr>
<td>5</td>
<td>+ optimization flags</td>
<td>54.63</td>
<td>3.25</td>
<td>385</td>
<td>2.516</td>
<td>0.301</td>
</tr>
<tr>
<td>6</td>
<td>Parallel loops</td>
<td>3.04</td>
<td>17.97</td>
<td>6,921</td>
<td>45.211</td>
<td>5.408</td>
</tr>
<tr>
<td>7</td>
<td>+ tiling</td>
<td>1.79</td>
<td>1.70</td>
<td>11,772</td>
<td>76.782</td>
<td>9.184</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Cache references (millions)</th>
<th>L1–d cache misses (millions)</th>
<th>Last–level cache misses (millions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel loops</td>
<td>104,090</td>
<td>17,220</td>
<td>8,600</td>
</tr>
<tr>
<td>+ tiling</td>
<td>64,690</td>
<td>11,777</td>
<td>416</td>
</tr>
</tbody>
</table>

The tiled implementation performs about 62% fewer cache references and incurs 68% fewer cache misses.

© 2008–2018 by the MIT 6.172 Lecturers
Multicore Cache Hierarchy

Processor chip

Memory Controller

Net-work

LLC (L3)

L2

L1 data

L1 inst

L1 data

L1 inst

L1 data

L1 inst

L1 data

L1 inst

L1 data

L1 inst

P

P

P

Level | Size  | Assoc. | Latency (ns)
-----|-------|--------|-------------
Main  | 60 GB |        | 50          
LLC   | 25 MB | 20     | 12          
L2    | 256 KB| 8      | 4           
L1–d  | 32 KB | 8      | 2           
L1–i  | 32 KB | 8      | 2           

64-byte cache lines

© 2008–2018 by the MIT 6.172 Lecturers
Tiling for a Two-Level Cache

- Two tuning parameters, $s$ and $t$.
- Multidimensional tuning optimization cannot be done with binary search.
Two tuning parameters, \( s \) and \( t \).

Multidimensional tuning optimization cannot be done with binary search.

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 im = 0; im < s; im += t)  
            for (int jm = 0; jm < s; jm += t)  
                for (int km = 0; km < s; km += t)  
                    for (int il = 0; il < t; ++il)  
                        for (int kl = 0; kl < t; ++kl)  
                            for (int jl = 0; jl < t; ++jl)  
                                C[ih+im+il][jh+jm+jl] +=  
                                A[ih+im+il][kh+km+kl] * B[kh+km+kl][jh+jm+jl];
IDEA: Tile for every power of 2 simultaneously.

\[
\begin{bmatrix}
C_{00} & C_{01} \\
C_{10} & C_{11}
\end{bmatrix}
= \begin{bmatrix}
A_{00} & A_{01} \\
A_{10} & A_{11}
\end{bmatrix}
\cdot
\begin{bmatrix}
B_{00} & B_{01} \\
B_{10} & B_{11}
\end{bmatrix}
\]

\[
= \begin{bmatrix}
A_{00}B_{00} & A_{00}B_{01} \\
A_{10}B_{00} & A_{10}B_{01}
\end{bmatrix}
+ \begin{bmatrix}
A_{01}B_{10} & A_{01}B_{11} \\
A_{11}B_{10} & A_{11}B_{11}
\end{bmatrix}
\]

8 multiplications of $n/2 \times n/2$ matrices.
1 addition of $n \times n$ matrices.
Recursive Parallel Matrix Multiply

```c
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{
    // C += A * B
    assert((n & (-n)) == n);
    if (n <= 1) {
        *C += *A * *B;
    } else {
        #define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
        #if defined(cilk_parallel)
            cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
            cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
            cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
            cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
        #endif
    }
}
```

The child function call is **spawned**, meaning it may execute in parallel with the parent caller.

Control may not pass this point until all spawned children have returned.
Recursive Parallel Matrix Multiply

```c
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{
    // C += A * B
    assert((n & (-n)) == n);
    if (n <= 1)
        *C += *A * *B;
    else {
        #define X(M,r,c) (M + (r*(n_ ## M) + c)
        cilk.spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
        cilk.spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
        cilk.spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
        mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
        cilk.sync;
        cilk.spawn mm_dac(X(C,0,0), n_C, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
        cilk.spawn mm_dac(X(C,0,1), n_C, X(A,0,1), X(B,1,0), n_B, n/2);
        cilk.spawn mm_dac(X(C,1,0), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
        mm_dac(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
        cilk.sync;
    }
}
```

The base case is too small. We must **coarsen** the recursion to overcome function-call overheads.

Running time: **93.93s**

... about **50× slower** than the last version!
Coarsening The Recursion

```c
void mm_dac(double *restrict C, int n_C,
             double *restrict A, int n_A,
             double *restrict B, int n_B,
             int n)
{
    /* C += A * B
       assert((n & (-n)) == n);
       if (n <= THRESHOLD)
           mm_base(C, n_C, A, n_A, B, n_B, n);
    } else {
        #define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
        cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
        cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
        cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
        cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
        cilk_sync;
        cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
        cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
        cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
        cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
        cilk_sync;
    }
}
```

Just one tuning parameter, for the size of the base case.
Coarsening The Recursion

```c
void mm_dac(double *restrict C, int n_C,
    double *restrict A, int n_A,
    double *restrict B, int n_B,
    int n)
{
    // C += A * B
    assert((n & (-n)) == n);
    if (n <= THRESHOLD) {
        mm_base(C, n_C, A, n_A, B, n_B, n);
    } else {
        #define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
        cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
        cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
        cilk_spawn mm_dac(X(C,1,0), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
        cilk_sync;
    }
}
```

```c
void mm_base(double *restrict C, int n_C,
    double *restrict A, int n_A,
    double *restrict B, int n_B,
    int n)
{
    // C = A * B
    for (int i = 0; i < n; ++i)
        for (int k = 0; k < n; ++k)
            for (int j = 0; j < n; ++j)
                C[i*n_C+j] += A[i*n_A+k] * B[k*n_B+j];
}
```
void mm_dac(double *restrict C, int n_C,
        double *restrict A, int n_A,
        double *restrict B, int n_B,
        int n)
{
    // C += A * B
    assert((n & (-n)) == n);
    if (n <= THRESHOLD) {
        mm_base(C, n_C, A, n_A, B, n_B, n);
    } else {
        #define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
        cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
        cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(A,1,0), n_A,
                mm_dac(X(C,1,1), n_C, X(A,1,0), n_A,
                cilk_sync;
        cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,1), n_A, X(A,0,1), n_A,
        cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,1), n_A, X(A,1,1), n_A,
        cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,1), n_A,
                mm_dac(X(C,1,1), n_C, X(A,1,1), n_A,
                cilk_sync;
    }
}
## 8. Divide–and–Conquer

<table>
<thead>
<tr>
<th>Version</th>
<th>Implementation</th>
<th>Running time (s)</th>
<th>Relative speedup</th>
<th>Absolute Speedup</th>
<th>GFLOPS</th>
<th>Percent of peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Python</td>
<td>21041.67</td>
<td>1.00</td>
<td>1</td>
<td>0.006</td>
<td>0.001</td>
</tr>
<tr>
<td>2</td>
<td>Java</td>
<td>2387.32</td>
<td>8.81</td>
<td>9</td>
<td>0.058</td>
<td>0.007</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>1155.77</td>
<td>2.07</td>
<td>18</td>
<td>0.118</td>
<td>0.014</td>
</tr>
<tr>
<td>4</td>
<td>+ interchange loops</td>
<td>177.68</td>
<td>6.50</td>
<td>118</td>
<td>0.774</td>
<td>0.093</td>
</tr>
<tr>
<td>5</td>
<td>+ optimization flags</td>
<td>54.63</td>
<td>3.25</td>
<td>385</td>
<td>2.516</td>
<td>0.301</td>
</tr>
<tr>
<td>6</td>
<td>Parallel loops</td>
<td>3.04</td>
<td>17.97</td>
<td>6,921</td>
<td>45.211</td>
<td>5.408</td>
</tr>
<tr>
<td>7</td>
<td>+ tiling</td>
<td>1.79</td>
<td>1.70</td>
<td>11,772</td>
<td>76.782</td>
<td>9.184</td>
</tr>
<tr>
<td>8</td>
<td>Parallel divide–and–conquer</td>
<td>1.30</td>
<td>1.38</td>
<td>16,197</td>
<td>105.722</td>
<td>12.646</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Cache references (millions)</th>
<th>L1–d cache misses (millions)</th>
<th>Last–level cache misses (millions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel loops</td>
<td>104,090</td>
<td>17,220</td>
<td>8,600</td>
</tr>
<tr>
<td>+ tiling</td>
<td>64,690</td>
<td>11,777</td>
<td>416</td>
</tr>
<tr>
<td>Parallel divide–and–conquer</td>
<td>58,230</td>
<td>9,407</td>
<td>64</td>
</tr>
</tbody>
</table>
Modern microprocessors incorporate vector hardware to process data in single-instruction stream, multiple-data stream (SIMD) fashion.

Parallel vector lanes operate synchronously on the words in a vector register.

Each vector register holds multiple words of data.
Clang/LLVM uses vector instructions automatically when compiling at optimization level `-O2` or higher. Clang/LLVM can be induced to produce a `vectorization report` as follows:

```
$ clang -O3 -std=c99 mm.c -o mm -Rpass=vector
mm.c:42:7: remark: vectorized loop (vectorization width: 2, interleaved count: 2) [-Rpass=loop-vectorize]
    for (int j = 0; j < n; ++j) {
      ^
```

Many machines don’t support the newest set of vector instructions, however, so the compiler uses vector instructions conservatively by default.
Programmers can direct the compiler to use modern vector instructions using compiler flags such as the following:

- `-mavx`: Use Intel AVX vector instructions.
- `-mavx2`: Use Intel AVX2 vector instructions.
- `-mfma`: Use fused multiply–add vector instructions.
- `-march=<string>`: Use whatever instructions are available on the specified architecture.
- `-march=native`: Use whatever instructions are available on the architecture of the machine doing compilation.

Due to restrictions on floating-point arithmetic, additional flags, such as `-ffast-math`, might be needed for these vectorization flags to have an effect.
Using the flags `-march=native -ffast-math` nearly doubles the program's performance!

Can we be smarter than the compiler?
Intel provides C–style functions, called *intrinsic instructions*, that provide direct access to hardware vector operations:

We can apply several more insights and performance-engineering tricks to make this code run faster, including:

- Preprocessing
- Matrix transposition
- Data alignment
- Memory-management optimizations
- A clever algorithm for the base case that uses AVX intrinsic instructions explicitly
Think, run, run, run...

...to test and measure many different implementations
## Version 10: AVX Intrinsics

<table>
<thead>
<tr>
<th>Version</th>
<th>Implementation</th>
<th>Running time (s)</th>
<th>Relative speedup</th>
<th>Absolute Speedup</th>
<th>GFLOPS</th>
<th>Percent of peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Python</td>
<td>21041.67</td>
<td>1.00</td>
<td>1</td>
<td>0.006</td>
<td>0.001</td>
</tr>
<tr>
<td>2</td>
<td>Java</td>
<td>2387.32</td>
<td>8.81</td>
<td>9</td>
<td>0.058</td>
<td>0.007</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>1155.77</td>
<td>2.07</td>
<td>18</td>
<td>0.118</td>
<td>0.014</td>
</tr>
<tr>
<td>4</td>
<td>+ interchange loops</td>
<td>177.68</td>
<td>6.50</td>
<td>118</td>
<td>0.774</td>
<td>0.093</td>
</tr>
<tr>
<td>5</td>
<td>+ optimization flags</td>
<td>54.63</td>
<td>3.25</td>
<td>385</td>
<td>2.516</td>
<td>0.301</td>
</tr>
<tr>
<td>6</td>
<td>Parallel loops</td>
<td>3.04</td>
<td>17.97</td>
<td>6,921</td>
<td>45.211</td>
<td>5.408</td>
</tr>
<tr>
<td>7</td>
<td>+ tiling</td>
<td>1.79</td>
<td>1.70</td>
<td>11,772</td>
<td>76.782</td>
<td>9.184</td>
</tr>
<tr>
<td>8</td>
<td>Parallel divide–and–conquer</td>
<td>1.30</td>
<td>1.38</td>
<td>16,197</td>
<td>105.722</td>
<td>12.646</td>
</tr>
<tr>
<td>9</td>
<td>+ compiler vectorization</td>
<td>0.70</td>
<td>1.87</td>
<td>30,272</td>
<td>196.341</td>
<td>23.486</td>
</tr>
<tr>
<td>10</td>
<td>+ AVX intrinsics</td>
<td>0.39</td>
<td>1.76</td>
<td>53,292</td>
<td>352.408</td>
<td>41.677</td>
</tr>
</tbody>
</table>
## Version 11: Final Reckoning

<table>
<thead>
<tr>
<th>Version</th>
<th>Implementation</th>
<th>Running time (s)</th>
<th>Relative speedup</th>
<th>Absolute Speedup</th>
<th>GFLOPS</th>
<th>Percent of peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Python</td>
<td>21041.67</td>
<td>1.00</td>
<td>1</td>
<td>0.006</td>
<td>0.001</td>
</tr>
<tr>
<td>2</td>
<td>Java</td>
<td>2387.32</td>
<td>8.81</td>
<td>9</td>
<td>0.058</td>
<td>0.007</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>1155.77</td>
<td>2.07</td>
<td>18</td>
<td>0.118</td>
<td>0.014</td>
</tr>
<tr>
<td>4</td>
<td>+ interchange loops</td>
<td>177.68</td>
<td>6.50</td>
<td>118</td>
<td>0.774</td>
<td>0.093</td>
</tr>
<tr>
<td>5</td>
<td>+ optimization flags</td>
<td>54.63</td>
<td>3.25</td>
<td>385</td>
<td>2.516</td>
<td>0.301</td>
</tr>
<tr>
<td>6</td>
<td>Parallel loops</td>
<td>3.04</td>
<td>17.97</td>
<td>6,921</td>
<td>45.211</td>
<td>5.408</td>
</tr>
<tr>
<td>7</td>
<td>+ tiling</td>
<td>1.79</td>
<td>1.70</td>
<td>11,772</td>
<td>76.782</td>
<td>9.184</td>
</tr>
<tr>
<td>8</td>
<td>Parallel divide–and–conquer</td>
<td>1.30</td>
<td>1.38</td>
<td>16,197</td>
<td>105.722</td>
<td>12.646</td>
</tr>
<tr>
<td>9</td>
<td>+ compiler vectorization</td>
<td>0.70</td>
<td>1.87</td>
<td>30,272</td>
<td>196.341</td>
<td>23.486</td>
</tr>
<tr>
<td>10</td>
<td>+ AVX intrinsics</td>
<td>0.39</td>
<td>1.76</td>
<td>53,292</td>
<td>352.408</td>
<td>41.677</td>
</tr>
<tr>
<td>11</td>
<td>Intel MKL</td>
<td>0.41</td>
<td>0.97</td>
<td>51,497</td>
<td>335.217</td>
<td>40.098</td>
</tr>
</tbody>
</table>

Version 10 is competitive with Intel’s professionally engineered Math Kernel Library!
You won’t generally see the magnitude of performance improvement we obtained for matrix multiplication.

But in 6.172, you will learn how to print the currency of performance all by yourself.