Lecture 1
Introduction & Matrix Multiplication
Charles E. Leiserson
September 6, 2012
WHY PERFORMANCE ENGINEERING?
Software Properties

What software properties are more important than performance?

- Compatibility
- Correctness
- Clarity
- Debuggability
- 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” these properties with performance.
Technology Scaling

Intel processor chips

Transistors x 1000

“Moore’s Law”
Until 2003

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

- Transistors x 1000
- Clock frequency (MHz)

© 2012 Charles E. Leiserson and I-Ting Angelina Lee
Power density, had scaling of clock frequency continued its trend of 25%–30% increase per year.

Technology Scaling

- **Transistors x 1000**
- **Clock frequency (MHz)**
- **Power (W)**

© 2012 Charles E. Leiserson and I–Ting Angelina Lee
Vendor Solution: Multicore

- To scale performance, put many processing cores on the microprocessor chip.
- Each generation of Moore’s Law potentially doubles the number of cores.

Intel Xeon E7
(10 cores per chip)
Server systems contain 4 chips
One Tiny Fly in the Ointment
One Tiny Fly in the Ointment

Software Problem
Virtually all application software is written for only a single core.
Staff

Lecturers
- Prof. Charles E. Leiserson
- Dr. I–Ting Angelina Lee

Teaching assistants
- Tim Kaler
- Justin Zhang
- Liz Fong–Jones

Administrative support
- Marcia Davidson

Masters in the Practice of Software Systems Engineering (MITPOSSE)
- Expert programmers from industry who will review your code and provide feedback
Communication

Class home page

Correspondence
- http://www.piazza.com/
- All course material, project related, and administrative questions
- Mark personal communications to staff as private, but try to make most communications public

Calendar
- http://goo.gl/FFst7

Lecture notes
- http://nb.mit.edu
- Basis of new textbook
- Significant feedback earns an acknowledgment!
Recitations

“Learning the life skills needed to be a true hacker”

Organization

- Two-hour duration
- Once a week on Friday
- “Lecture–studio” format
- Mandatory
- Must complete a set of tasks and get it checked off by your TA
TA Office Hours

Times
- Monday, Tuesday, Wednesday
- 4:00 P.M. to 6:00 P.M.
- Location announced in lecture

Bring your laptop
- debugging support
- help with tools
- answers to conceptual questions
- good place to work on the projects with your team
Required Work

10% — Homeworks 1–4

20% — Quiz
  • Evening exam on Tuesday, November 6
  • Closed book, but crib sheet allowed
  • Details forthcoming

40% — Projects 1–3
  • Beta, MITPOSSE review, final

30% — Final project
  • Beta–1, MITPOSSE review, Beta2, Final

No final exam
Expectations

**Required work is required!**

- If you fail to do all the assignments, you risk failing the class
- If you miss a recitation assignment, you risk failing the class
- No late homework — hand in what you have by the deadline for partial credit

If an issue arises, please talk to your TA as soon as possible so that arrangements can be made!
Projects

You are given a correct but inefficient program. **Your mission:** Make the program run fast.

Do whatever you can within the rules

- There is no right answer!
- Take advantage of the machine resources.
- Lots of creative freedom to explore many possible directions.
- Hard to be fastest, but easy to test which is fastest!

The journey is as important as the outcome

- You may try many things that will not give a performance improvement.
- Failure is as important as success → feedback!
- Tell us everything you did and why in your write-up.
Project Process

Project starts

Beta submission
- The staff will publish the performance results and a baseline for the final submission.

MITPOSSE design review
- After the Beta, you have just over a week to meet in a 90-minute design-review meeting with your assigned Master.
- Your Master will provide feedback on your code and design.
- Your Master will not grade you, but your attendance at the design review is mandatory.

Final submission
- Update the code to reflect your Master’s comments.
- Enhance the performance to reach the published baseline.
  - Better than baseline $\rightarrow$ full credit for performance
  - Worse than baseline $\rightarrow$ fraction relative to the slowdown
MITPOSSE

Practicing engineers from local companies
- Unpaid volunteers who are contributing their time to help you!
- Senior engineers with lots of experience.
- You can learn a lot from them!

Please accord them proper respect
- Be responsive when they contact you to schedule the design review.
- Thank them for their feedback.
- Be personable.
Programming Languages

Start with C, move to C++

- Close to the metal
- Machine’s memory is directly exposed
  - `malloc()` and `free()`, pointers, native data types
- Code compiles directly to machine language
- No hidden work (garbage collection, bounds checking)

Resources available on the class home page

- Manuals for various tools
- Quick references

Online resources

- www.cprogramming.com
- www.cplusplus.com
- search will find many other resources
Multicore Machine Resources

Cloud machines
- Collection of 12-core machines donated by Dell and Intel (thanks!)
- Apply for CSAIL username
- Log onto cloud#.csail.mit.edu for # = 0, 1, ..., 11
- Use git to pull and push from class repository
- Edit on cloud or locally using AFS to share directory
- For performance measurements, submit batch jobs using CQ, which will run on a dedicated machine

Laptop development
- Many, but not all, tools will run on your laptop
- Recommended, but use at your own risk
- 6.172 staff will not help maintain your software
- Performance measurements will be taken on clouds
Academic Honesty

While a project is active, you may share
- with your group,
- with the course staff, including the MITPOSSE,
- with other students attending your design review (no hardcopies),
- but no one else!

Use of outside materials
- You may use outside materials as long as you properly cite them.

Read the course information handout
- If you have any questions, please talk to your TA.

We will be using technology to detect cheating
CASE STUDY: MATRIX MULTIPLICATION
Square–Matrix Multiplication

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

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

Assume for simplicity that \( n = 2^k \).
# Intel Xeon Computer System

<table>
<thead>
<tr>
<th>Feature</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microarchitecture</td>
<td>Sandy Bridge</td>
</tr>
<tr>
<td>Clock frequency</td>
<td>2.4 GHz</td>
</tr>
<tr>
<td>Processor chips</td>
<td>2</td>
</tr>
<tr>
<td>Processing cores</td>
<td>8 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 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</td>
<td>20 MB shared 20–way set associative</td>
</tr>
<tr>
<td>DRAM</td>
<td>32 GB</td>
</tr>
</tbody>
</table>

Peak = \(2 \times 8 \times 8 \times 2.4 \times 10^9 = 307\) GFLOPS
1. Triply Nested Loops in Python

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

n = 4096

A = [[1.0*random.random()
     for row in xrange(n)]
     for col in xrange(n)]

B = [
    [1.0*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
\[= 34,962 \text{ seconds} \approx 9.75 \text{ hours}\]

Is this fast?

**Back–of–the–envelope calculation**

\[2n^3 = 2^{37} \text{ floating–point operations}\]
Running time \(\approx 2^{15}\) seconds

\[\therefore \text{Python gets } 2^{37}/2^{15} = 2^{22} \approx 4 \text{ MFLOPS}\]

Peak = 307 GFLOPS
Python gets \(\approx 0.0013\%\) of peak
2. Let’s Try 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 = 2,531 seconds
≈ 42 minutes
... about $14\times$ faster than Python!
Still only 0.0177% of peak.

© 2012 Charles E. Leiserson and I–Ting Angelina Lee
# 3. Why Not C?

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <assert.h>

typedef unsigned long long uint64_t;

#define n 4096

double A[n][n];
double 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(int argc, const char *argv[]) {
    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;
}
```

Using the GCC compiler

Running time = 1,463 seconds

\[ \approx 24 \text{ minutes} \]

... about \( 1.7 \times \) faster than Java.
Where We Stand So Far

<table>
<thead>
<tr>
<th>Version</th>
<th>Implementation</th>
<th>Time (s)</th>
<th>GFLOPS</th>
<th>Absolute speedup</th>
<th>Relative speedup</th>
<th>Fraction of peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Python</td>
<td>34,962.21</td>
<td>0.004</td>
<td>1</td>
<td>1</td>
<td>0.00%</td>
</tr>
<tr>
<td>2</td>
<td>Java</td>
<td>2,530.65</td>
<td>0.054</td>
<td>14</td>
<td>13.8</td>
<td>0.00%</td>
</tr>
<tr>
<td>3</td>
<td>C, using GCC</td>
<td>1,462.50</td>
<td>0.094</td>
<td>24</td>
<td>1.7</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

Why is Python so slow and C so fast?

- Python is interpreted.
- Java is compiled to byte-code, which is then interpreted and just-in-time (JIT) compiled.
- C is compiled directly to machine code.
4. Optimization Switches

GCC provides a collection of optimization switches. Without touching the C code, we can just 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>1463</td>
</tr>
<tr>
<td>-01</td>
<td>Optimize</td>
<td>856</td>
</tr>
<tr>
<td>-02</td>
<td>Optimize even more</td>
<td>851</td>
</tr>
<tr>
<td>-03</td>
<td>Optimize yet more</td>
<td>427</td>
</tr>
</tbody>
</table>
GCC is not the only compiler in the world. Let’s try the Intel compiler ICC with the -03 optimization switch.

<table>
<thead>
<tr>
<th>Version</th>
<th>Implementation</th>
<th>Time (s)</th>
<th>GFLOPS</th>
<th>Absolute speedup</th>
<th>Relative speedup</th>
<th>Fraction of peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Python</td>
<td>34,962.21</td>
<td>0.004</td>
<td>1</td>
<td>1</td>
<td>0.00%</td>
</tr>
<tr>
<td>2</td>
<td>Java</td>
<td>2,530.65</td>
<td>0.054</td>
<td>14</td>
<td>13.8</td>
<td>0.00%</td>
</tr>
<tr>
<td>3</td>
<td>C, using GCC</td>
<td>1,462.50</td>
<td>0.094</td>
<td>24</td>
<td>1.7</td>
<td>0.00%</td>
</tr>
<tr>
<td>4</td>
<td>+ switches</td>
<td>426.79</td>
<td>0.322</td>
<td>82</td>
<td>3.4</td>
<td>0.10%</td>
</tr>
<tr>
<td>5</td>
<td>C, using ICC + switches</td>
<td>41.44</td>
<td>3.317</td>
<td>844</td>
<td>10.3</td>
<td>1.10%</td>
</tr>
</tbody>
</table>

Wow! We’re now running in less than a minute and are over 800 times faster than the original Python! Why is ICC so much faster?
Vectorization

Each core of our computer has 8 vector units which can initiate 8 floating-point operations on each cycle using a single vector instruction, as long as the operations are independent. Most compilers can be induced to produce a vectorization report:

```
$ icc -O3 -std=c99 mm_c.c -o mm_c_icc_O3 -vec-report2
...
mm_c.c(42): (col. 5) remark: PERMUTED LOOP WAS VECTORIZED.
mm_c.c(43): (col. 7) remark: loop was not vectorized: not inner loop.
mm_c.c(41): (col. 3) remark: loop was not vectorized: not inner loop.
```

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

Interchange these two loops.
We’re running on only one of our 16 cores, leaving 15 idle. Let’s use all of them!

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

The *cilk_for* keyword, which is supported by both ICC and GCC (not the main branch, however), indicates that all the iterations of the loop may execute in parallel.
Parallel-Loops Performance

Running time

- 18 seconds, the fastest so far!
- But wait, it’s only 2.3 times faster than the previous version, which used only 1 core, and we’re now using 16!
Hardware Performance Counters

Diagnose the problem with hardware performance counters

$ perf stat -e instructions -e cache-misses -- ./mm_c_icc_03
38.946693
  Performance counter stats for './mm_c_icc_03':
  174,398,127,129 instructions # 0.00 insns per cycle
   19,491,135 cache-misses
  39.422211009 seconds time elapsed

$ perf stat -e instructions -e cache-misses -- ./mm_ploops
21.164532
  Performance counter stats for './mm_ploops':
  373,875,833,663 instructions # 0.00 insns per cycle
   9,682,355,938 cache-misses
  21.65352167 seconds time elapsed

The parallel code is executing more than twice the number of instructions and incurring about 500 times the number of cache misses.
Compiler Bug!

Inspection of the assembly language output from the compiler reveals that `icc -O3 tiles` the matrix — an effective optimization — but it fails to do so in the context of `cilk_for`.

What is `tiling`?
“Sandy Bridge” Memory Hierarchy

Latency in clock cycles

1 4 10 26 200

DRAM
Cache Misses for Matrix Multiply

Cache misses

- Consider the code after the two inner loops have been permuted.
- As the code sequences through matrix B it incurs $\Theta(n^2)$ cold cache misses for new data elements it encounters.
- By the time the code sequences through the matrix again, the first elements have been evicted.
Cache Misses for Matrix Multiply

Cache misses
- As the code sequences through matrix B, it incurs $\Theta(n^2)$ cold cache misses for new data elements it encounters.
- By the time the code sequences through the matrix again, the first elements have been evicted.
- $\Theta(n^3)$ cache misses.
7. Tiling

```cilk
#include <cilk>

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

**Cache misses**

- If $s^2$ is sufficiently smaller than the size of the cache, the tiled loops incur only $\Theta(n^3/s)$ cache misses.
Performance of Tiling

<table>
<thead>
<tr>
<th>Tile size</th>
<th>1 core (s)</th>
<th>16 cores (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>859.64</td>
<td>65.30</td>
</tr>
<tr>
<td>2</td>
<td>309.60</td>
<td>26.32</td>
</tr>
<tr>
<td>4</td>
<td>178.17</td>
<td>9.19</td>
</tr>
<tr>
<td>8</td>
<td>93.14</td>
<td>6.56</td>
</tr>
<tr>
<td>16</td>
<td>76.65</td>
<td>5.09</td>
</tr>
<tr>
<td>32</td>
<td>60.85</td>
<td>3.49</td>
</tr>
<tr>
<td>64</td>
<td>54.17</td>
<td>3.24</td>
</tr>
<tr>
<td>128</td>
<td>44.79</td>
<td>2.76</td>
</tr>
<tr>
<td>256</td>
<td>40.87</td>
<td>4.73</td>
</tr>
<tr>
<td>512</td>
<td>42.77</td>
<td>6.79</td>
</tr>
<tr>
<td>1024</td>
<td>69.00</td>
<td>8.39</td>
</tr>
<tr>
<td>2048</td>
<td>65.96</td>
<td>26.13</td>
</tr>
</tbody>
</table>

**Tile size**

- For 16 cores, a $128 \times 128$ tile gives the best performance.
- For 1 core, however, $256 \times 256$ tile works best.
- If tile size is not properly tuned — either too large or too small — the code may perform poorly.
- Tiling is fragile.
## Cache Behavior of Tiling

<table>
<thead>
<tr>
<th>Version</th>
<th>Implementation</th>
<th>Tiling</th>
<th>Cores</th>
<th>Instructions</th>
<th>Cache misses</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Serial optimized loops</td>
<td>no</td>
<td>1</td>
<td>165,416,672,245</td>
<td>1,980,866,303</td>
<td>64.1</td>
</tr>
<tr>
<td>5</td>
<td>Serial optimized loops</td>
<td>compiler</td>
<td>1</td>
<td>174,397,675,927</td>
<td>19,635,564</td>
<td>39.4</td>
</tr>
<tr>
<td>6</td>
<td>Parallel loops</td>
<td>no</td>
<td>1</td>
<td>246,711,574,370</td>
<td>4,363,182,438</td>
<td>82.9</td>
</tr>
<tr>
<td>6</td>
<td>Parallel loops</td>
<td>no</td>
<td>16</td>
<td>307,981,194,383</td>
<td>6,743,511,892</td>
<td>17.9</td>
</tr>
<tr>
<td>7</td>
<td>Parallel loops</td>
<td>128</td>
<td>1</td>
<td>177,072,223,809</td>
<td>69,526,210</td>
<td>44.2</td>
</tr>
<tr>
<td>7</td>
<td>Parallel loops</td>
<td>128</td>
<td>16</td>
<td>179,009,006,797</td>
<td>98,549,525</td>
<td>2.9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Version</th>
<th>Implementation</th>
<th>Time (s)</th>
<th>GFLOPS</th>
<th>Absolute speedup</th>
<th>Relative speedup</th>
<th>Fraction of peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Python</td>
<td>34,962.21</td>
<td>0.004</td>
<td>1</td>
<td>1.0</td>
<td>0.00%</td>
</tr>
<tr>
<td>2</td>
<td>Java</td>
<td>2,530.65</td>
<td>0.054</td>
<td>14</td>
<td>13.8</td>
<td>0.00%</td>
</tr>
<tr>
<td>3</td>
<td>C, using GCC</td>
<td>1,462.50</td>
<td>0.094</td>
<td>24</td>
<td>1.7</td>
<td>0.00%</td>
</tr>
<tr>
<td>4</td>
<td>+ switches</td>
<td>426.79</td>
<td>0.322</td>
<td>82</td>
<td>3.4</td>
<td>0.10%</td>
</tr>
<tr>
<td>5</td>
<td>C, using ICC + switches</td>
<td>41.44</td>
<td>3.317</td>
<td>844</td>
<td>10.3</td>
<td>1.10%</td>
</tr>
<tr>
<td>6</td>
<td>Parallel loops</td>
<td>17.95</td>
<td>7.658</td>
<td>1,948</td>
<td>2.3</td>
<td>2.50%</td>
</tr>
<tr>
<td>7</td>
<td>+ tiling</td>
<td>2.76</td>
<td>49.825</td>
<td>12,675</td>
<td>6.5</td>
<td>16.20%</td>
</tr>
</tbody>
</table>
IDEA: Tile for every power of 2.

\[
\begin{pmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{pmatrix}
= \begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix}
\cdot
\begin{pmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{pmatrix}
\]

\[
= \begin{pmatrix}
A_{11}B_{11} & A_{11}B_{12} \\
A_{21}B_{11} & A_{21}B_{12}
\end{pmatrix}
+ \begin{pmatrix}
A_{12}B_{21} & A_{12}B_{22} \\
A_{22}B_{21} & A_{22}B_{22}
\end{pmatrix}
\]

- 8 multiplications of $n/2 \times n/2$ matrices.
- 1 addition of $n \times n$ matrices.
void mmdac (double *C, double *A, double *B, int size) {
  if (size <= 1) {
    *C += *A * *B;
  } else {
    int s11 = 0;
    int s12 = size/2;
    int s21 = (size/2)*n;
    int s22 = (size/2)*(n+1);
    cilk_spawn mmdac(C+s11, A+s11, B+s11, size/2);
    cilk_spawn mmdac(C+s12, A+s11, B+s12, size/2);
    cilk_spawn mmdac(C+s21, A+s21, B+s11, size/2);
    mmdac(C+s21, A+s21, B+s11, size/2);
    cilk_sync;
    cilk_spawn mmdac(C+s11, A+s12, B+s21, size/2);
    cilk_spawn mmdac(C+s12, A+s12, B+s22, size/2);
    cilk_spawn mmdac(C+s21, A+s22, B+s21, size/2);
    mmdac(C+s22, A+s22, B+s22, size/2);
    cilk_sync;
  }
}
**Performance of D&C**

<table>
<thead>
<tr>
<th>Version</th>
<th>Implementation</th>
<th>Time (s)</th>
<th>GFLOPS</th>
<th>Absolute speedup</th>
<th>Relative speedup</th>
<th>Fraction of peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Python</td>
<td>34,962.21</td>
<td>0.004</td>
<td>1</td>
<td></td>
<td>0.00%</td>
</tr>
<tr>
<td>2</td>
<td>Java</td>
<td>2,530.65</td>
<td>0.054</td>
<td>14</td>
<td>13.8</td>
<td>0.00%</td>
</tr>
<tr>
<td>3</td>
<td>C, using GCC</td>
<td>1,462.50</td>
<td>0.094</td>
<td>24</td>
<td>1.7</td>
<td>0.00%</td>
</tr>
<tr>
<td>4</td>
<td>+ switches</td>
<td>426.79</td>
<td>0.322</td>
<td>82</td>
<td>3.4</td>
<td>0.10%</td>
</tr>
<tr>
<td>5</td>
<td>C, using ICC + switches</td>
<td>41.44</td>
<td>3.317</td>
<td>844</td>
<td>10.3</td>
<td>1.10%</td>
</tr>
<tr>
<td>6</td>
<td>Parallel loops</td>
<td>17.95</td>
<td>7.658</td>
<td>1,948</td>
<td>2.3</td>
<td>2.50%</td>
</tr>
<tr>
<td>7</td>
<td>+ tiling</td>
<td>2.76</td>
<td>49.825</td>
<td>12,675</td>
<td>6.5</td>
<td>16.20%</td>
</tr>
<tr>
<td>8</td>
<td>Parallel divide–and–conquer</td>
<td>142.61</td>
<td>0.964</td>
<td>245</td>
<td>0.0</td>
<td>0.30%</td>
</tr>
</tbody>
</table>

*Uh, oh! A big step backwards!*
Function-Call Overhead

```c
void mmdac (double *C, double *A, double *B, int size) {
  if (size <= 1) {
    *C += *A * *B;
  } else {
    int s11 = 0;
    int s12 = size/2;
    int s21 = (size/2)*n;
    int s22 = (size/2)*(n+1);
    cilk_spawn mmdac(C+s11, A+s11, B+s11, size/2);
    cilk_spawn mmdac(C+s12, A+s11, B+s12, size/2);
    cilk_spawn mmdac(C+s21, A+s21, B+s11, size/2);
    mmdac(C+s22, A+s21, B+s12, size/2);
    cilk_sync;
    cilk_spawn mmdac(C+s11, A+s12, B+s21, size/2);
    cilk_spawn mmdac(C+s12, A+s12, B+s22, size/2);
    cilk_spawn mmdac(C+s21, A+s22, B+s21, size/2);
    mmdac(C+s22, A+s22, B+s22, size/2);
    cilk_sync;
  }
}
```

The base case is too small. We must **coarsen** the recursion to avoid function-call overhead.
Versions 9 & 10

<table>
<thead>
<tr>
<th>Version</th>
<th>Implementation</th>
<th>Time (s)</th>
<th>GFLOPS</th>
<th>Absolute speedup</th>
<th>Relative speedup</th>
<th>Fraction of peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Python</td>
<td>34,962.21</td>
<td>0.004</td>
<td>1</td>
<td>1</td>
<td>0.00%</td>
</tr>
<tr>
<td>2</td>
<td>Java</td>
<td>2,530.65</td>
<td>0.054</td>
<td>14</td>
<td>13.8</td>
<td>0.00%</td>
</tr>
<tr>
<td>3</td>
<td>C, using GCC</td>
<td>1,462.50</td>
<td>0.094</td>
<td>24</td>
<td>1.7</td>
<td>0.00%</td>
</tr>
<tr>
<td>4</td>
<td>+ switches</td>
<td>426.79</td>
<td>0.322</td>
<td>82</td>
<td>3.4</td>
<td>0.10%</td>
</tr>
<tr>
<td>5</td>
<td>C, using ICC + switches</td>
<td>41.44</td>
<td>3.317</td>
<td>844</td>
<td>10.3</td>
<td>1.10%</td>
</tr>
<tr>
<td>6</td>
<td>Parallel loops</td>
<td>17.95</td>
<td>7.658</td>
<td>1,948</td>
<td>2.3</td>
<td>2.50%</td>
</tr>
<tr>
<td>7</td>
<td>+ tiling</td>
<td>2.76</td>
<td>49.825</td>
<td>12,675</td>
<td>6.5</td>
<td>16.2%</td>
</tr>
<tr>
<td>8</td>
<td>Parallel divide-and-conquer</td>
<td>142.61</td>
<td>0.964</td>
<td>245</td>
<td>0</td>
<td>0.30%</td>
</tr>
<tr>
<td>9</td>
<td>+ coarsening</td>
<td>5.44</td>
<td>25.255</td>
<td>6,425</td>
<td>26.2</td>
<td>8.20%</td>
</tr>
<tr>
<td>10</td>
<td>+ transpose</td>
<td>1.72</td>
<td>79.698</td>
<td>20,274</td>
<td>3.2</td>
<td>25.90%</td>
</tr>
</tbody>
</table>

- Coarsening does a good job.
- Transposing the B matrix does a great job!
Unportable Performance

- Use the –xHost ICC compiler switch to generate modern AVX vector instructions, but the code won’t run on older machines.
- Use the –axAVX ICC compiler switch, which generates multiple clones of the code, one of which uses the AVX instructions. A test is made at runtime as to which version to use.
- Portable code, but the performance is not portable.
- Use compiler intrinsics (assembly-language directives) to access the AVX instructions directly. Highly unportable, but great performance!
## Final Reckoning

<table>
<thead>
<tr>
<th>Version</th>
<th>Implementation</th>
<th>Time (s)</th>
<th>GFLOPS</th>
<th>Absolute speedup</th>
<th>Relative speedup</th>
<th>Fraction of peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Python</td>
<td>34,962.21</td>
<td>0.004</td>
<td>1</td>
<td></td>
<td>0.00%</td>
</tr>
<tr>
<td>2</td>
<td>Java</td>
<td>2,530.65</td>
<td>0.054</td>
<td>14</td>
<td>13.8</td>
<td>0.00%</td>
</tr>
<tr>
<td>3</td>
<td>C, using GCC</td>
<td>1,462.50</td>
<td>0.094</td>
<td>24</td>
<td>1.7</td>
<td>0.00%</td>
</tr>
<tr>
<td>4</td>
<td>+ switches</td>
<td>426.79</td>
<td>0.322</td>
<td>82</td>
<td>3.4</td>
<td>0.10%</td>
</tr>
<tr>
<td>5</td>
<td>C, using ICC + switches</td>
<td>41.44</td>
<td>3.317</td>
<td>844</td>
<td>10.3</td>
<td>1.10%</td>
</tr>
<tr>
<td>6</td>
<td>Parallel loops</td>
<td>17.95</td>
<td>7.658</td>
<td>1,948</td>
<td>2.3</td>
<td>2.50%</td>
</tr>
<tr>
<td>7</td>
<td>+ tiling</td>
<td>2.76</td>
<td>49.825</td>
<td>12,675</td>
<td>6.5</td>
<td>16.20%</td>
</tr>
<tr>
<td>8</td>
<td>Parallel divide–and–conquer</td>
<td>142.61</td>
<td>0.964</td>
<td>245</td>
<td>0.0</td>
<td>0.30%</td>
</tr>
<tr>
<td>9</td>
<td>+ coarsening</td>
<td>5.44</td>
<td>25.255</td>
<td>6,425</td>
<td>26.2</td>
<td>8.20%</td>
</tr>
<tr>
<td>10</td>
<td>+ transpose</td>
<td>1.72</td>
<td>79.698</td>
<td>20,274</td>
<td>3.2</td>
<td>25.90%</td>
</tr>
<tr>
<td>11</td>
<td>+ machine–specific compilation</td>
<td>1.58</td>
<td>86.725</td>
<td>22,061</td>
<td>1.1</td>
<td>28.20%</td>
</tr>
<tr>
<td>12</td>
<td>+ AVX intrinsics</td>
<td>0.76</td>
<td>180.741</td>
<td>45,978</td>
<td>2.1</td>
<td>58.80%</td>
</tr>
<tr>
<td>13</td>
<td>Intel MKL</td>
<td>0.63</td>
<td>218.096</td>
<td>55,480</td>
<td>1.2</td>
<td>71.00%</td>
</tr>
</tbody>
</table>

Our sub–1–second Version 12 rivals the experts who coded the Intel Math Kernel Library! We’re over 4.6 orders of magnitude better than the original Python.
You won’t generally see the kind 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.