Lecture 1
Introduction & Matrix Multiplication
Saman Amarasinghe
2014
Outline

Why performance engineering

6.172 administrivia

Case study: Matrix multiplication
WHY PERFORMANCE ENGINEERING?
In the Golden Era of Computing, Performance Engineering Ruled the World

Every programmer was a Performance Engineer

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

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

Any useful program will stretch the machine resources

Program has to be planned around the machine
Many will not ‘fit’ without intense performance hacks

© 2014 Charles E. Leiserson, Saman P. Amarasinghe
Software Properties

What programmers want to do?

- New Functionality
  ... and...
  - Scalability
  - Compatibility
  - Correctness
  - Clarity
  ... and more.

- Debuggability
- Maintainability
- Modularity
- Portability
- Reliability
- Robustness
- Testability
- Usability

Performance is the currency of computing. You can often "buy" needed properties with performance.
In the Dominant Era of Computing, Performance became Free

The currency was free

Only need to wait a few months

- Performance doubled every 2 years

Performance is the currency of computing. You can often "buy" needed properties with performance.
Technology Scaling

Intel processor chips

“Moore’s Law”

Transistors x 1000
Technology Scaling

Transistors x 1000

Clock frequency (MHz)

© 2014 Charles E. Leiserson, Saman P. Amarasinghe
In the Dominant Era, Performance was Free

Moore’s Law and the scaling of clock frequency = printing press for the currency of performance
In the Dominant Era, Performance was Free

The currency was free
Only need to wait a few months
Performance doubled every 18 months

Performance engineering was ‘optional’ at best and ‘irrelevant’ in the eyes of most programmers

Performance is the currency of computing. You can often "buy" needed properties with performance.
Moore’s law is not giving free performance any more.

Performance is the currency of computing. You can often "buy" needed properties with performance.
Technology Scaling

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

The graph shows the exponential growth of transistors and clock frequency over time from 1970 to 2010.
Power Density


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)
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
Technology Scaling

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

© 2014 Charles E. Leiserson, Saman P. Amarasinghe
Moore’s law is not giving free performance any more

Performance Engineering is the only way to get more currency

Performance is the currency of computing. You can often "buy" needed properties with performance.
6.172 ADMINISTRIVIA
Staff

Lecturers
- Prof. Charles E. Leiserson
- Prof. Saman P. Amarasinghe
- Dr. Aparna Chandramowlushwaran

Teaching assistants
- William Mitchell Leiserson
- Tao Benjamin Schardl
- Cong Yan
- Michael J Xu
- Damon Doucet

Administrative support
- Mary McDavitt
- Cree Bruins

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
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
**Required Work**

12% — Weekly Homeworks
30% — 2 Quizzes
  - In class
  - Closed book, but crib sheet allowed
  - Details forthcoming
36% — Projects 1–3
  - Beta, MITPOSSE review, final
22% — project 4
  - Beta–1, MITPOSSE review, Beta2, Final

No final exam
Homework 1 & Recitation 1

Homework 1 is out today
- Part 1 is due by today @ 8:00PM!!
- Rest due on Monday

There will be a recitation tomorrow
- Go to any one of 3 recitations
TA Office Hours

**Times**
- Sunday, Monday, Tuesday, Wednesday
- 4:00 P.M. to 7:00 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 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.
• Code from other projects will be available to all
  • Study and understand what they did, get inspired….but don’t copy!

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 → full credit for performance
  ■ Worse than baseline → fraction relative to the slowdown
Practicing engineers from the industry

- 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.

What to expect

- Input on your code style
- Advice on deployment, testing debugging process
- What programmers do in industry
- Even career advice and mentoring

NOT how to make your program faster
Academic Honesty

Homework
- No sharing

While a project is active, you may share
- with your group

After beta, you will have access to everyone's code
- But... do not copy, just learn from them
- Practically, after looking at someone else's code, wait sometime before you touch your code (ex: one hour)

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
We will be using 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
- C primer on Monday the 8th @ 7:00PM

Online resources

- [www.cprogramming.com](http://www.cprogramming.com)
- search will find many other resources
Multicore Machine Resources

Cloud machines
- Collection of 12-core machines donated by Dell and Intel (thanks!)
- Log onto cloud#.csail.mit.edu for # = 0, 1, ..., 4
  - Need a CSAIL account to login
- Use by everyone (you need to load balance)
- For editing, compiling the code and running the tool frontends

Lanka machines
- Place to run the executables
- Only one job will be run on a machine at a time via a queue
- For performance number gathering

Intel E5–2695 V2 @ 2.4Ghz
12-cores (2 per node)
128 GB of DDR3 memory

Laptop development
- Recommended, but use at your own risk
- 6.172 staff will not help maintain your software
CASE STUDY: MATRIX MULTIPLICATION
Square-Matrix Multiplication

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

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

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

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

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 = \( 2 \times 8 \times 8 \times 2.4 \times 10^9 = 307 \text{ GFLOPS} \)
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 seconds
≈ 9.75 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 \text{ 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× faster than Python!
Still only 0.0177% of peak.
3. Why Not C?

Using the GCC compiler
Running time = 1,463 seconds
≈ 24 minutes
... about 1.7× faster than Java.

```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];

double 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;
}
```
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></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.
Interpreter overhead

Lot of extra work

- Repeated again and again

JIT Compiler

- Do a lot of work once, and reuse
- Granularity of basic blocks or methods
  - basic block = instruction sequences without control flow
- When a new instruction/method, check if it is already jitted
- If already jitted, directly execute it
- If not, run the read/interpret/evaluate/update cycle and create a executable list of machine instructions
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>
How is our processor doing?

- We know only 0.1% of the peak
- But, why?
- What may be contributing to this slowdown?
Performance Counters

Modern hardware counts “events”

- Lot more information than just execution time

<table>
<thead>
<tr>
<th>Version</th>
<th>Implementation</th>
<th>Run Time(s)</th>
<th>Cache References</th>
<th>Cache misses</th>
<th>Hit rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>+ switches</td>
<td>426.79</td>
<td>34,320,418,733</td>
<td>34,042,409,392</td>
<td>0.81%</td>
</tr>
</tbody>
</table>

Reference and misses for L3 cache
“Sandy Bridge” Memory Hierarchy

Latency in clock cycles

1 4 10 26 200

DRAM
Modern hardware counts “events”

- Lot more information than just execution time

<table>
<thead>
<tr>
<th>Version</th>
<th>Implementation</th>
<th>Run Time(s)</th>
<th>Cache References</th>
<th>Cache misses</th>
<th>Hit rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>+ switches</td>
<td>426.79</td>
<td>34,320,418,733</td>
<td>34,042,409,392</td>
<td>0.81%</td>
</tr>
</tbody>
</table>

Reference and misses for L3 cache

Really bad hit rate

- should be in the 90’s
5. Data Transpose

Scanning the memory

Contiguous accesses are better

- Data fetch as cache line (Core 2 Duo 64 byte L2 Cache line)
- Contiguous data $\rightarrow$ Single cache fetch supports 8 reads of doubles
Preprocessing of Data

In Matrix Multiply
- $n^3$ computation
- $n^2$ data

Possibility of preprocessing data before computation
- $n^2$ data $\rightarrow$ $n^2$ processing
- Can make the $n^3$ happens faster

One matrix don’t have good cache behavior
Transpose that matrix
- $n^2$ operations
- Will make the main matrix multiply loop run faster
5. Data Transpose

After transposing the C matrix

<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>+ transpose</td>
<td>79.84</td>
<td>1.722</td>
<td>438</td>
<td>5.4</td>
<td>0.56%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Version</th>
<th>Implementation</th>
<th>Run Time(s)</th>
<th>Cache References</th>
<th>Cache misses</th>
<th>Hit rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>+ switches</td>
<td>426.79</td>
<td>34,320,418,733</td>
<td>34,042,409,392</td>
<td>0.81%</td>
</tr>
<tr>
<td>5</td>
<td>+ transpose</td>
<td>79.84</td>
<td>4,699,663,602</td>
<td>1,863,550,316</td>
<td>60.35%</td>
</tr>
</tbody>
</table>

~8x reduction of cache references!

- Why?

Improved hit rate
Data Reuse

Data reuse

- Change of computation order can reduce the # of loads to cache

- Calculating a row (1024 values of C)
  - C: \(1024 \times 1 = 1024\) + A: \(384 \times 1 = 394\) + B: \(1024 \times 384 = 393,216\) = 394,524

- Blocked Matrix Multiply \((32^2 = 1024\) values of C)
  - C: \(32 \times 32 = 1024\) + A: \(384 \times 32 = 12,284\) + B: \(32 \times 384 = 12,284\) = 25,600
Data Reuse

Data reuse

- Change of computation order can reduce the # of loads to cache

- Calculating a row (1024 values of C)
  - C: $1024 \times 1 = 1024$ + A: $384 \times 1 = 394$ + B: $1024 \times 384 = 393,216$
  - $= 394,524$

- Blocked Matrix Multiply ($32^2 = 1024$ values of C)
  - C: $32 \times 32 = 1024$ + A: $384 \times 32 = 12,284$ + B: $32 \times 384 = 12,284$
  - $= 25,600$

- If the data needed does not fit in the cache, has to be fetched each time needed from the next level
  - In calculating a Raw, each element used from B is reused only after 394,525 accesses to data (for the next raw)
  - In Blocked matrix multiply, each element used from B is reused after 65 accesses (and reused 32 times)
    - If the cache has more then 65 elements $\Rightarrow$ cache hit!
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.

- Optimal size depends on many factors such as memory size and processor type
- Need to retune to keep the edge
IDEA: Tile for every power of 2.

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

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

- 8 multiplications of \( \frac{n}{2} \times \frac{n}{2} \) 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);
        mmdac(C+s11, A+s11, B+s11, size/2);
        mmdac(C+s12, A+s11, B+s12, size/2);
        mmdac(C+s21, A+s21, B+s11, size/2);
        mmdac(C+s22, A+s21, B+s12, size/2);
        mmdac(C+s11, A+s12, B+s21, size/2);
        mmdac(C+s12, A+s12, B+s22, size/2);
        mmdac(C+s21, A+s22, B+s21, size/2);
        mmdac(C+s22, A+s22, B+s22, size/2);
    }
}
performance of D&C

uh, oh! a big step backwards!

- cache references nearly doubled
  - why?
- but... we improved the hit rate
  - why?
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);
        mmdac(C+s11, A+s11, B+s11, size/2);
        mmdac(C+s12, A+s11, B+s12, size/2);
        mmdac(C+s21, A+s21, B+s11, size/2);
        mmdac(C+s22, A+s21, B+s12, size/2);
        mmdac(C+s11, A+s12, B+s21, size/2);
        mmdac(C+s12, A+s12, B+s22, size/2);
        mmdac(C+s21, A+s22, B+s21, size/2);
        mmdac(C+s22, A+s22, B+s22, size/2);
    }
}
```

The base case is too small. We must coarsen the recursion to avoid function–call overhead.
7. Coarsening

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

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

What is the CUTOFF value?

It depends…

Later we will learn about autotuning.
Performance of Coarsening + Transpose

<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>+ transpose</td>
<td>79.84</td>
<td>1.722</td>
<td>438</td>
<td>5.4</td>
<td>0.56%</td>
</tr>
<tr>
<td>6</td>
<td>Dvide-and-conquer</td>
<td>314.73</td>
<td>0.437</td>
<td>111</td>
<td>0.3</td>
<td>0.14%</td>
</tr>
<tr>
<td>7</td>
<td>+ coarsending, transpose</td>
<td>79.04</td>
<td>1.739</td>
<td>442</td>
<td>4.0</td>
<td>0.57%</td>
</tr>
</tbody>
</table>

OK, we are back!

Cache references are down by ~7x

- Why?
8. 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:

```
$ gcc -O3 -std=c99 mm_c.c -o mm_c_gcc_03 -ftree-vectorizer-verbose=2
... Vectorizing loop at mm_c.c:14
mm_c.c:14: note: LOOP VECTORIZED.
... mm_c.c:3: note: vectorized 1 loops in function.
mm_c.c:3: note: Completely unroll loop 15 times
```

```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.
## 8. Vectorization

<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>+ transpose</td>
<td>79.84</td>
<td>1.722</td>
<td>438</td>
<td>5.4</td>
<td>0.56%</td>
</tr>
<tr>
<td>6</td>
<td>Dvidie–and–conquer</td>
<td>314.73</td>
<td>0.437</td>
<td>111</td>
<td>0.3</td>
<td>0.14%</td>
</tr>
<tr>
<td>7</td>
<td>+ coarsending, transpose</td>
<td>79.04</td>
<td>1.739</td>
<td>442</td>
<td>4.0</td>
<td>0.57%</td>
</tr>
<tr>
<td>8</td>
<td>+ vectorization</td>
<td>20.55</td>
<td>6.689</td>
<td>1,702</td>
<td>3.9</td>
<td>2.18%</td>
</tr>
</tbody>
</table>

Close to 4x speedup
9. Parallel Loops

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

```c
void matrix_multiply()
{
    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 latest version of GCC, indicates that all the iterations of the loop may execute in parallel.
8. Recursive Parallel Matrix Multiply

```c
void mmdac (double *C, double *A, double *B, int size) {
    if (size <= CUTOFF) {
        *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 named *child* function may execute in parallel with the *parent* caller.

Control may not pass this point until all spawned children have returned.
Parallel-Loops Performance

<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>+ transpose</td>
<td>79.84</td>
<td>1.722</td>
<td>438</td>
<td>5.4</td>
<td>0.56%</td>
</tr>
<tr>
<td>6</td>
<td>Dvidie-and-conquer</td>
<td>314.73</td>
<td>0.437</td>
<td>111</td>
<td>0.3</td>
<td>0.14%</td>
</tr>
<tr>
<td>7</td>
<td>+ coarsending, transpose</td>
<td>79.04</td>
<td>1.739</td>
<td>442</td>
<td>4.0</td>
<td>0.57%</td>
</tr>
<tr>
<td>8</td>
<td>+ vectorization</td>
<td>20.55</td>
<td>6.689</td>
<td>1,702</td>
<td>3.9</td>
<td>2.18%</td>
</tr>
<tr>
<td>9</td>
<td>Parallel divide-andconquer</td>
<td>1.42</td>
<td>96.666</td>
<td>24,590</td>
<td>14.5</td>
<td>31.47%</td>
</tr>
</tbody>
</table>

Running time

- **14.5x** performance improvement when using **16** cores!
- Now up to 1/3 of the peak performance!
• Use the `–march=corei7-avx` GCC compiler switch to generate modern AVX vector instructions, but the code won’t run on older machines.

• Use the `–ffast-math -mavx` GCC 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 non-portable, 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>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>40</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>+ transpose</td>
<td>79.84</td>
<td>1.72</td>
<td>438</td>
<td>5.4</td>
<td>0.56%</td>
</tr>
<tr>
<td>6</td>
<td>Dvide-and-conquer</td>
<td>314.73</td>
<td>6.437</td>
<td>111</td>
<td>0.3</td>
<td>0.14%</td>
</tr>
<tr>
<td>7</td>
<td>+ coarsending, transpose</td>
<td>79.04</td>
<td>0.739</td>
<td>442</td>
<td>4.0</td>
<td>0.57%</td>
</tr>
<tr>
<td>8</td>
<td>+ vectorization</td>
<td>20.55</td>
<td>6.689</td>
<td>1,702</td>
<td>3.9</td>
<td>2.18%</td>
</tr>
<tr>
<td>9</td>
<td>Parallel divide-and-conquer</td>
<td>1.02</td>
<td>96.666</td>
<td>24,590</td>
<td>14.5</td>
<td>31.47%</td>
</tr>
<tr>
<td>10</td>
<td>+ machine-specific compilation</td>
<td>1.05</td>
<td>131.465</td>
<td>33,442</td>
<td>1.4</td>
<td>42.80%</td>
</tr>
<tr>
<td>11</td>
<td>+ AVX intrinsics</td>
<td>0.67</td>
<td>206.288</td>
<td>52,479</td>
<td>1.6</td>
<td>67.15%</td>
</tr>
</tbody>
</table>

Our sub-1-second Version 11 rivals the experts who coded the Intel Math Kernel Library!
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.