Matrix Multiply: A Case Study

Saman Amarasinghe
Fall 2011
Outline

Administrivia

Why Performance Engineering

Step-by-step improvement of the performance of matrix multiply
Staff

Prof. Saman Amarasinghe
32-G744

Prof. Charles Leiserson
32-G768

Ekanathan (Eka) Natarajan
Ruben Perez
Phumpong (yod) Watanaprakornkul

Tao B (TB) Schardl

Masters in the Practice of Software Systems Engineering

- 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

E-mail:
- 6172-staff@csail.mit.edu
- Personal issues only
- Do not e-mail individual staff members unless there is a compelling reason

Lecture notes:
- http://nb.mit.edu

© Saman Amarasinghe 2009
Recitations

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

- Hands-on
- Basic tools and techniques

Organization

- Once a week on Friday (10:00 to noon, 1:00 to 3:00, 4:00 to 6:00)
- Mandatory (can miss only one; more than that will be an automatic F)
  - Need to complete a set of tasks and get checked by the TA
  - If you miss a recitation, do the tasks yourself and get it checked by a TA
- Two hours
  - First hour is a TA led hands on exercise (bring your laptop)
  - Second hour is extra time to finish up and ask questions, get help on the projects etc.
Office Hours

Monday, Tuesday, Wednesday

- 4:00PM to 6:00PM
- Stata Gates Tower, 7th floor lounge

Bring your coding bugs, conceptual questions etc.

A good place to work on the project with your team

- Answer to questions and debugging support is right there
Exams

Two in-class exams
Closed book, but crib sheet allowed
Late and missing work

Hard to play catch-up in this class
➤ Check the class calendar

No late homework
➤ Hand in what you got by the deadline for partial credit

If there is an issue, talk to a staff member ASAP
## Projects

<table>
<thead>
<tr>
<th>Project</th>
<th>Group Size</th>
<th>Out</th>
<th>Due</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project 0</td>
<td>Individual</td>
<td>Thu Sep 8(^{th})</td>
<td>Final Due Mon Sep 12(^{th})</td>
</tr>
<tr>
<td>Project 1</td>
<td>Group of 2</td>
<td>Tue Sep 13(^{th})</td>
<td>Beta Due Mon Sep 26(^{th})</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Design Review Mon-Fri Oct 3(^{rd}) to 7(^{th})</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Final Due Mon Oct 17(^{th})</td>
</tr>
<tr>
<td>Project 2</td>
<td>Group of 2</td>
<td>Thu Sep 27(^{th})</td>
<td>Part 1 Beta Due Wed Oct 12(^{th})</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Part 2 Beta Due Mon Oct 24(^{th})</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Design Review Mon-Fri Oct 31(^{st}) to Nov 4(^{th})</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Final Due Thu Nov 23(^{rd})</td>
</tr>
<tr>
<td>Project 3</td>
<td>Group of 2</td>
<td>Tue Oct 25(^{th})</td>
<td>Part 1 Beta Due Mon Nov 7(^{th})</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Part 2 Beta Due Wed Nov 16(^{th})</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Design Review Mon-Fri Nov 21(^{st}) to 25(^{th})</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Final Due Tue Dec 7(^{th})</td>
</tr>
<tr>
<td>Problem Set</td>
<td>Individual</td>
<td>Tue Nov 8(^{th})</td>
<td>Due Fri Nov 18(^{th})</td>
</tr>
<tr>
<td>Final Project</td>
<td>Group of 3-4</td>
<td>Thu Nov 15(^{th})</td>
<td>Design Doc Mon Nov 28(^{th})</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Design Review (opt) Tue-Fri Nov 29(^{th}) to Dec 2(^{nd})</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Final Mon Dec 12(^{th})</td>
</tr>
<tr>
<td></td>
<td>Out</td>
<td>Due</td>
<td>% Grade</td>
</tr>
<tr>
<td>--------------------------</td>
<td>----------------------</td>
<td>------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>Project 0 Individual</td>
<td>Thu Sep 8th</td>
<td>Final Due Mon Sep 12th</td>
<td>2%</td>
</tr>
<tr>
<td>Project 1 Group of 2</td>
<td>Tue Sep 13th</td>
<td>Beta Due Mon Sep 26th</td>
<td>8%</td>
</tr>
<tr>
<td></td>
<td>Design Review Mon Fri Oct 3rd to 7th</td>
<td>Mon Oct 17th</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Final Due</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Project 2 Group of 2</td>
<td>Thu Sep 27th</td>
<td>Part 1 Beta Due Wed Oct 12th</td>
<td>12%</td>
</tr>
<tr>
<td></td>
<td>Part 2 Beta Due Mon Oct 24th</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Design Review Mon Fri Oct 31st to Nov 4th</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Final Due Thu Nov 23rd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Project 3 Group of 2</td>
<td>Tue Oct 25th</td>
<td>Part 1 Beta Due Mon Nov 7th</td>
<td>12%</td>
</tr>
<tr>
<td></td>
<td>Part 2 Beta Due Wed Nov 16th</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Design Review Mon Fri Nov 21st to 25th</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Final Due Tue Dec 7th</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem Set Individual</td>
<td>Tue Nov 8th</td>
<td>Due Fri Nov 18th</td>
<td>4%</td>
</tr>
<tr>
<td>Final Project Group of 3-4</td>
<td>Thu Nov 15th</td>
<td>Design Doc Mon Nov 28th</td>
<td>24%</td>
</tr>
<tr>
<td></td>
<td>Design Review(opt) Tue-Fri Nov 29th to Dec 2nd</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Final Mon Dec 12th</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quiz 1</td>
<td></td>
<td></td>
<td>16%</td>
</tr>
<tr>
<td>Quiz 2</td>
<td></td>
<td></td>
<td>16%</td>
</tr>
<tr>
<td>Design Review Participation</td>
<td></td>
<td></td>
<td>6%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>100%</td>
</tr>
</tbody>
</table>
Academic Honesty

You may share:
- With your group
- With your master (and any other students attending the session with you)
- With the course staff
AND NO ONE ELSE

Use of material:
- If you use any outside material, properly cite them

Check the course information handout.

If you have any questions, please contact the staff ASAP

We will be using technology to detect cheating.

© Saman Amarasinghe 2009
Projects

Correct but inefficient program is given

Your missions: Make the program run much faster

- Take advantage of algorithmic inefficiencies
  - May be able to change the computation within the given bounds
- Take advantage of the machine resources

There is no right answer!

- A lot of freedom (and little direction)
- But easy to figure out who got the fastest answer!

Journey is as important as the outcome

- You may try many things that will not give a performance improvement
- Failure is as important as the successes
- Tell is everything you did and why

© Saman Amarasinghe 2008
**Project Process**

**Project starts**
- Projects 2 and 3 have two parts (two due dates for the beta)
- Final project requires a design document submission in mid project

**After the beta turn-in**
- We will publish the best expected performance number

**Design Review Week**
- After the project is turned-in you have a week to schedule a 60 to 90 minute design review meeting with your assigned Master.
- Master will provide feedback on your code and design
- Master’s will not grade you, however your attendance is at the design review is mandatory. (Lead to a large deduction if you don’t attend)

**Final turn-in**
- Update the code to reflect the comments provided by the Master
- Update the performance to reach (closer to) the best expected performance
  - If you achieve that number → will get full credit for performance
  - Better than that number → will earn bragging rights
  - Worse than that number → faction relative to the slowdown
Masters in the Practice of Software Systems Engineering

Practicing engineers from local companies
➢ They are volunteering their time to help you!
Project Programming Language

Language: C and C++

Why? → “Closest to the metal”
  • Machine’s memory is directly exposed
    – Malloc and free
    – Pointers
    – Native data types
  • Compiles down to machine’s instruction set
  • No background (garbage collection) or extra (bounds check) work

Resources available on the Stellar class web page
(http://stellar.mit.edu/S/course/6/fa11/6.172/)
  • C for Java programmer
  • Gdb debugger
  • Makefiles
  • (x86 assembly)
Project Machine Resources

GUI Machines
- Cagfarm-xx.csail.mit.edu to cagfarm-yy.csail.mit.edu
  - You will login to these machines
  - You will develop code and run vtune gui on these machines

Performance Evaluation Machines
- cagnode1.csail.mit.edu to cagnode8.csail.mit.edu
  - 2 x 3.15GHz 4 core Intel processors with 8 GB of memory
  - When you “submit” a job, it will be queued and run on one of these machines
  - These are research machines
    - Will have 2 machines assigned permanently for the class
    - Will add additional machines near the deadline
    - The fastest turnaround will be a couple of days before the deadline
      - don’t wait till the last day, you may have to waste time on the queue!
Why is Performance Important?

Acceptable response time

Ability to add more functionality

Ability to scale

Use less power / resources

© Saman Amarasinghe 2008
Acceptable Response Times

Many systems have stringent requirements

- Anti-lock break system → response time ≤ hydraulic system
- Mpeg decoder → 20 frames a second or will get a jittery movie
- Google Search → results should be available within a second

If the response times are not met, system is not usable
Ability to add more functionality

If the necessary work can be done faster, more room within the required response time lead to:

- added features
- better higher-quality processing
- bigger data sets
Ability to Scale

Successful programs will get pushed hard

- From hundred to millions of users/documents/data
  - Scale the system to handle the increased workload
  - Gracefully deal with unexpected issues due to scaling
Use Less Power / Resources

More instruction executed → more power used

- In 2005 approx. 1.2% US total power going to servers
  - Many supercomputer centers are now power limited
- Viability of cell phones dictated by battery life
  - New iPhone lasts less than a day on one charge
    Cannot add more functionality without improving the battery life or making the existing applications more efficient

Cost of scaling an inefficient system is very high

- Keep adding servers can be expensive

© Saman Amarasinghe 2008
Improving performance is hard

Knowing that there is a performance problem
Identifying the performance bottlenecks
Establishing the leading cause of the problem
Eliminating the performance problem
Knowing that there is a problem

We know how to find incorrect programs → testing, verification and validation

But, how close is your program to the maximum achievable performance?

- Hard to know if the program performance can improve a lot.

How do you know if you the program performs ok?

- Back of the envelope calculations
- Performance debugging
- Scalability testing
- Comparisons to similar programs
- Experience!
Identifying Performance Bottlenecks

Profile the programs
- Figure out where most of the execution time is at.
  - Is that expected? Or is there a problem?
- Look at machine characteristics
  - Instruction, cache, memory, IO behavior normal?
    - A very complex system with a small window to look into.

Scalability testing
- Pushing the program to the limit
- Has to do this with limited resources
- Need to understand what will scale and what will not

Measure without Perturbing
- If profiling change the performance too much, results are not valid

© Saman Amarasinghe 2008
Establish the Leading Cause

**Study the algorithm**
- Is the algorithm too costly
- Can any computation be eliminated from the critical path
  - Preprocessing, caching etc.

**Study data structures and data layout**
- Is the layout affecting the memory behavior

**Study the program structure**
- Is the program structure leading to bad instruction selection or stalls

**Trial-and-Error**
- Many hunches will not work out
  - Or the impact may be hidden by a bigger first-order effect
  - Or the performance is max of A, B and C. All need to be improved before you see a benefit.
Eliminating Performance Problems

Problem-Free Design
- Much better to design without any performance problems

Cuts-through Abstraction Boundaries
- Performance has be done end-to-end

Need to Understand All the Layers and their Impact
- All the software layers
- The compiler
- The processor
- The system

Adhere to Software Engineering Principles
- Simplicity, modularity, portability etc.
- Cannot compromise the correctness!
Performance Possibilities: an Example

Matrix Multiply
Matrix Multiply

Matrix multiple is a fundamental operation in many computations

- Example: video encoding, weather simulation, computer graphics

```
for(int i =0; i < x; i++)
    for(int j =0; j < y; j++)
        for(int k=0; k < z; k++)
            A[i][j] += B[i][k]*C[k][j]
```
Matrix Multiply using PHP

A server-side scripting language
 Popular in creating web backends

Mainly an interpreted language

```php
function mult($mat1, $mat2, $m, $o, $n) {
    $res = zero($m, $n);
    for ($i = 0; $i < $m; $i++) {
        for ($j = 0; $j < $n; $j++) {
            for ($k = 0; $k < $o; $k++) {
                $res[$i][$j] += $mat1[$i][$k] * $mat2[$k][$j];
            }
        }
    }
    return $res;
}
```

© Saman Amarasinghe 2009
Is the performance good?

It took almost 2 1/2 hours to multiply two 2048x2048 matrices

\[ 2048^3 = 8,589,934,592 \text{ operations} \]

Each operation is multiply, add and 3 index updates, and branch check → 6 ops

\[ 8,589,934,592 \times 6 = 51,539,607,552 \]

Operations per second = \[ \frac{51,539,607,552}{9,298} = 5,542,823 = 5.543 \times 10^6 \]

The server machine runs at 1.9 GHz → 1.9 \times 10^9 \text{ cycles / second}

That comes to about 342 cycles per each visible operation

How can we improve performance?
Profiling

Look deeply in to the program execution
Find out where you are spending your time
  ➢ By method
  ➢ By line

Lot of interesting information
  ➢ Time spend
  ➢ Cumulative time spend
  ➢ Number of invocations
  ➢ Etc. etc.

Great way to zero in on what matters – Hotspots
  ➢ If 90% time is in one routine, inefficiencies in the rest of the program don’t matter
  ➢ Also, is the hotspots doing what you expect them to do?
## Profile Data

<table>
<thead>
<tr>
<th>Percentage</th>
<th>Function</th>
<th>File</th>
<th>Memory Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>64.31%</td>
<td>php</td>
<td>/usr/bin/php5</td>
<td>0x000000002a6c40</td>
</tr>
<tr>
<td>25.20%</td>
<td>php</td>
<td>/usr/bin/php5</td>
<td>zend_hash_index_find</td>
</tr>
<tr>
<td>2.49%</td>
<td>php</td>
<td>/usr/bin/php5</td>
<td>compare_function</td>
</tr>
<tr>
<td>2.29%</td>
<td>php</td>
<td>/usr/bin/php5</td>
<td>mul_function</td>
</tr>
<tr>
<td>2.06%</td>
<td>php</td>
<td>/usr/bin/php5</td>
<td>execute</td>
</tr>
<tr>
<td>0.95%</td>
<td>php</td>
<td>/usr/bin/php5</td>
<td>add_function</td>
</tr>
<tr>
<td>0.74%</td>
<td>php</td>
<td>/usr/bin/php5</td>
<td>0x000000002674ec</td>
</tr>
</tbody>
</table>
How about Python instead?

A high-level programming language

def mult(mat1, mat2, m, o, n):
    # Multiply two matrices
    result = [[0 for row in range(m)] for col in range(n)]
    for i in range(m):
        for j in range(n):
            for k in range(o):
                result[i][j] += mat1[i][k]*mat2[k][j]
    return result
## Performance

<table>
<thead>
<tr>
<th></th>
<th>PhP</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms</td>
<td>9,298,440</td>
<td>6,145,070</td>
</tr>
<tr>
<td>Cycles/OP</td>
<td>343</td>
<td>227</td>
</tr>
</tbody>
</table>

1.6x
## Profile Data

<table>
<thead>
<tr>
<th>Percentage</th>
<th>Module</th>
<th>Source Path</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>80.05%</td>
<td>matmul.py</td>
<td>/usr/bin/python2.5</td>
<td>PyEval_EvalFrameEx</td>
</tr>
<tr>
<td>10.53%</td>
<td>matmul.py</td>
<td>/usr/bin/python2.5</td>
<td>0x0000000003abc0</td>
</tr>
<tr>
<td>2.57%</td>
<td>matmul.py</td>
<td>/usr/bin/python2.5</td>
<td>PyInt_AsSsize_t</td>
</tr>
<tr>
<td>1.39%</td>
<td>matmul.py</td>
<td>/usr/bin/python2.5</td>
<td>PyObject_SetItem</td>
</tr>
<tr>
<td>1.02%</td>
<td>matmul.py</td>
<td>/usr/bin/python2.5</td>
<td>PyNumber_AsSsize_t</td>
</tr>
<tr>
<td>0.92%</td>
<td>matmul.py</td>
<td>/usr/bin/python2.5</td>
<td>PyNumber_InPlaceAdd</td>
</tr>
<tr>
<td>0.79%</td>
<td>matmul.py</td>
<td>/usr/bin/python2.5</td>
<td>0x0000000000dc50e</td>
</tr>
<tr>
<td>0.71%</td>
<td>matmul.py</td>
<td>/usr/bin/python2.5</td>
<td>PyInt_FromLong</td>
</tr>
<tr>
<td>0.66%</td>
<td>matmul.py</td>
<td>/usr/bin/python2.5</td>
<td>PyNumber_Multiply</td>
</tr>
<tr>
<td>0.56%</td>
<td>matmul.py</td>
<td>/usr/bin/python2.5</td>
<td>PyNumber_Index</td>
</tr>
</tbody>
</table>
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
## Performance

<table>
<thead>
<tr>
<th></th>
<th>PhP</th>
<th>Python</th>
<th>Python (Jitted)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms</td>
<td>9,298,440</td>
<td>6,145,070</td>
<td>540,891</td>
</tr>
<tr>
<td>Cycles/OP</td>
<td>343</td>
<td>227</td>
<td>20</td>
</tr>
</tbody>
</table>

- 1.6x
- 11.4x
- 18x
## Profile Data

<table>
<thead>
<tr>
<th>Time (%)</th>
<th>Module</th>
<th>Source</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>96.75%</td>
<td>pypy</td>
<td>/lib/ld-2.7.so</td>
<td>0x00000000014f96</td>
</tr>
<tr>
<td>2.54%</td>
<td>pypy</td>
<td>/lib/libc-2.7.so</td>
<td>memset</td>
</tr>
<tr>
<td>0.41%</td>
<td>pypy</td>
<td>/afs/csail.mit.edu/u/n/neboat/pypy-1.6/bin/pypy</td>
<td>0x00000000876379</td>
</tr>
<tr>
<td>0.03%</td>
<td>pypy</td>
<td>/lib/libc-2.7.so</td>
<td>memcpy</td>
</tr>
</tbody>
</table>
Lets move to Java

**General purpose object oriented language**

- Good JIT compilers available
- Java manage the memory
  - Safe $\rightarrow$ less bugs
  - Programmer has no control $\rightarrow$ can be slow

```java
 double[][] A = new double[x][y];
double[][] B = new double[x][z];
double[][] C = new double[z][y];

long started = System.nanoTime();

for(int i =0; i < x; i++)
    for(int j =0; j < y; j++)
        for(int k=0; k < z; k++)
            A[i][j] += B[i][k]*C[k][j];

long ended = System.nanoTime();
```
## Performance

<table>
<thead>
<tr>
<th></th>
<th>PhP</th>
<th>Python</th>
<th>Python (Jitted)</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms</td>
<td>9,298,440</td>
<td>6,145,070</td>
<td>540,891</td>
<td>348,749</td>
</tr>
<tr>
<td>Cycles/OP</td>
<td>343</td>
<td>227</td>
<td>20</td>
<td>13</td>
</tr>
</tbody>
</table>

- 1.6x
- 11.4x
- 18x
- 27x
From Java to C

Java

- Memory managed by Java
- Memory bounds check
- Bytecode first interpreted and then JITted (fast compilation, no time to generate the best code)

C

- Memory managed by programmer
- No such thing in C
- Intel C compiler compiles the program directly into x86 assembly

© Saman Amarasinghe 2008
uint64_t testMM(const int x, const int y, const int z)
{
    double **A;
    double **B;
    double **C;
    uint64_t started, ended;
    uint64_t timeTaken;
    int i, j, k;

    A = (double**)malloc(sizeof(double *)*x);
    B = (double**)malloc(sizeof(double *)*x);
    C = (double**)malloc(sizeof(double *)*y);

    for (i = 0; i < x; i++)
        A[i] = (double *) malloc(sizeof(double)*y);

    for (i = 0; i < z; i++)
        B[i] = (double *) malloc(sizeof(double)*z);

    for (i = 0; i < z; i++)
        C[i] = (double *) malloc(sizeof(double)*z);

    ......
started = read_timestamp_counter();

for(i =0; i < x; i++)
    for(j =0; j < y; j++)
        for(k=0; k < z; k++)
            A[i][j] += B[i][k] * C[k][j];

ended = read_timestamp_counter();
timeTaken = (ended - started);
printf("Time: %f ms\n", timeTaken/3158786.0);

return timeTaken;
}
## Performance

<table>
<thead>
<tr>
<th></th>
<th>PhP</th>
<th>Python</th>
<th>Python (Jitted)</th>
<th>Java</th>
<th>In C</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms</td>
<td>9,298,440</td>
<td>6,145,070</td>
<td>540,891</td>
<td>348,749</td>
<td>19,564</td>
</tr>
<tr>
<td>Cycles/OP</td>
<td>343</td>
<td>227</td>
<td>20</td>
<td>13</td>
<td>1/2</td>
</tr>
</tbody>
</table>

- 1.6x
- 11.4x
- 18x
- 27x
- 476x
Profiling with Performance Counters

Modern hardware counts “events”
- Lot more information than just execution time

CPI – Clock cycles Per Instruction
- Measures if instructions are stalling

L1 and L2 Cache Miss Rate
- Are your accesses using the cache well or is the cache misbehaving?

Instructions Retired
- How many instructions got executed

<table>
<thead>
<tr>
<th>CPI</th>
<th>L1 Miss Rate</th>
<th>L2 Miss Rate</th>
<th>Percent SSE</th>
<th>Instructions Retired</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.78</td>
<td>0.24</td>
<td>0.02</td>
<td>43%</td>
<td>13,137,280,000</td>
</tr>
</tbody>
</table>
Issues with Matrix Representation

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
```c
#define IND(A, x, y, d) A[(x)*(d)+(y)]
...

A = (double *)malloc(sizeof(double)*x*y);
B = (double *)malloc(sizeof(double)*x*z);
C = (double *)malloc(sizeof(double)*y*z);
Cx = (double *)malloc(sizeof(double)*y*z);

started = read_timestamp_counter();

for(j =0; j < y; j++)
    for(k=0; k < z; k++)
        IND(Cx,j,k,z) = IND(C, k, j, y);

for(i =0; i < x; i++)
    for(j =0; j < y; j++)
        for(k=0; k < z; k++)
            IND(A, i, j, y) += IND(B, i, k, z)*IND(Cx, j, k, z);

ended = read_timestamp_counter();
timeTaken = (ended - started);
printf("Time: %f ms\n", timeTaken/3158786.0);
```
## Performance

<table>
<thead>
<tr>
<th></th>
<th>PhP (ms)</th>
<th>Python (ms)</th>
<th>Python (Jitted)</th>
<th>Java (ms)</th>
<th>In C (ms)</th>
<th>Transposed (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cycles/OP</strong></td>
<td>343</td>
<td>227</td>
<td>20</td>
<td>13</td>
<td>1/2</td>
<td>1/2</td>
</tr>
<tr>
<td><strong>Speedup</strong></td>
<td>1.6x</td>
<td>1.6x</td>
<td>1.2x</td>
<td>11.4x</td>
<td>17.9x</td>
<td>1.6x</td>
</tr>
<tr>
<td><strong>Speedup</strong></td>
<td>18x</td>
<td>27x</td>
<td>476x</td>
<td>541x</td>
<td>541x</td>
<td>541x</td>
</tr>
</tbody>
</table>

- Performance comparison for different languages and execution environments.
## Profile Data

<table>
<thead>
<tr>
<th></th>
<th>CPI</th>
<th>L1 Miss Rate</th>
<th>L2 Miss Rate</th>
<th>Percent SSE Instructions</th>
<th>Instructions Retired</th>
</tr>
</thead>
<tbody>
<tr>
<td>In C</td>
<td>4.78</td>
<td>0.24</td>
<td>0.02</td>
<td>43%</td>
<td>13,137,280,000</td>
</tr>
<tr>
<td>Transposed</td>
<td>1.13</td>
<td>0.15</td>
<td>0.02</td>
<td>50%</td>
<td>13,001,486,336</td>
</tr>
</tbody>
</table>
The Memory System

The memory system dilemma
- Small amount of memory $\rightarrow$ fast access
- Large amount of memory $\rightarrow$ slow access
- How do you have a lot of memory and access them very fast

Cache Hierarchy
- Store most probable accesses in small amount of memory with fast access
- Hardware heuristics determine what will be in each cache and when

The temperamental cache
- If your access pattern matches heuristics of the hardware $\rightarrow$ blazingly fast
- Otherwise $\rightarrow$ dog slow
Data Reuse

Data reuse

- Change of computation order can reduce the # of loads to cache
- Calculating a row (1024 values of A)
  - A: $1024 \times 1 = 1024$ + B: $384 \times 1 = 394$ + C: $1024 \times 384 = 393,216 = 394,524$
- Blocked Matrix Multiply ($32^2 = 1024$ values of A)
  - A: $32 \times 32 = 1024$ + B: $384 \times 32 = 12,284$ + C: $32 \times 384 = 12,284 = 25,600$
Changing the Program

Many ways to get to the same result

- Change the execution order
- Change the algorithm
- Change the data structures

Some changes can perturb the results

- Select a different but equivalent answer
- Reorder arithmetic operations
  - \((a + b) + c \neq a + (b + c)\)
- Drop/change precision
- Operate within an acceptable error range

© Saman Amarasinghe 2008


```c
started = read_timestamp_counter();

for(j2 = 0; j2 < y; j2 += block_x)
    for(k2 = 0; k2 < z; k2 += block_y)
        for(i = 0; i < x; i++)
            for(j = j2; j < min(j2 + block_x, y); j++)
                for(k=k2; k < min(k2 + block_y, z); k++)
                    IND(A,i,j,y) += IND(B,i,k,z) * IND(C,k,j,z);

ended = read_timestamp_counter();
timeTaken = (ended - started);
printf("Time: %f ms\n", timeTaken/3158786.0);
```
## Performance

<table>
<thead>
<tr>
<th></th>
<th>PhP</th>
<th>Python</th>
<th>Python (Jitted)</th>
<th>Java</th>
<th>In C</th>
<th>Transposed</th>
<th>Tiled</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms</td>
<td>9,298,440</td>
<td>6,145,070</td>
<td>540,891</td>
<td>348,749</td>
<td>19,564</td>
<td>17,206</td>
<td>12,887</td>
</tr>
<tr>
<td>Cycles/OP</td>
<td>343</td>
<td>227</td>
<td>20</td>
<td>13</td>
<td>1/2</td>
<td>1/2</td>
<td>1/3</td>
</tr>
</tbody>
</table>

1.6x

11.4x

17.9x

1.2x

1.6x

1.6x

1.6x

1.4x

1.2x

1.4x

1.2x

1.4x

1.4x

1.4x

1.4x

1.4x

1.4x

1.4x

1.4x

1.4x

1.4x

1.4x

1.4x

1.4x

1.4x

1.4x

1.4x

1.4x
## Profile Data

<table>
<thead>
<tr>
<th></th>
<th>CPI</th>
<th>L1 Miss Rate</th>
<th>L2 Miss Rate</th>
<th>Percent SSE Instructions</th>
<th>Instructions Retired</th>
</tr>
</thead>
<tbody>
<tr>
<td>In C</td>
<td>4.78</td>
<td>0.24</td>
<td>0.02</td>
<td>43%</td>
<td>13,137,280,000</td>
</tr>
<tr>
<td>Transposed</td>
<td>1.13</td>
<td>0.15</td>
<td>0.02</td>
<td>50%</td>
<td>13,001,486,336</td>
</tr>
<tr>
<td>Tiled</td>
<td>0.49</td>
<td>0.02</td>
<td>0</td>
<td>39%</td>
<td>18,044,811,264</td>
</tr>
</tbody>
</table>

© Saman Amarasinghe 2008
Instruction Level Optimizations

**Modern processors have many other performance tricks**

- Instruction Level Parallelism
  - 2 integer, 2 floating point and 1 MMX/SSE
- MMX/SSE Instructions
  - Can do the same operation on multiple contiguous data at the same time
- Cache hierarchy
- Prefetching of data

**Nudge the Compiler**

- Need to nudge the compiler to generate the vector code
  - Removed any perceived dependences
  - Bound most constant variables to the constant
  - Possible use of compiler #pragma’s
  - Use of vector reporting to see why a loop is not vectorizing
- Other options is to write vector assembly code 😞


```
#define N 1024
#define BLOCK_X 256
#define BLOCK_Y 1024
#define IND(A, x, y, d) A[(x)*(d)+(y)]

......

started = read_timestamp_counter();

for(j =0; j < N; j++)
    for(k=0; k < N; k++)
        IND(Cx,j,k,N) = IND(C, k, j, N);

for(j2 = 0; j2 < N; j2 += BLOCK_X)
    for(k2 = 0; k2 < N; k2 += BLOCK_Y)
        for(i = 0; i < N; i++)
            for(j = 0; j < BLOCK_X; j++)
                for(k = 0; k < BLOCK_Y; k++)
                    IND(A,i,j+j2,N) += IND(B,i,k+k2,N) * IND(Cx,j+j2,k+k2,N);

ended = read_timestamp_counter();
timeTaken = (ended - started);
printf("Time: %f ms\n", timeTaken/3158786.0);
```
Play with the compiler flags

- icc –help
- Find the best flags
  - icc -c -O3 -xT -msse3 mxm.c
- Use information from icc
  - icc -vec-report5 ...
- Generate assembly and stare!
  - Icc -S -fsource-asm -fverbose-asm...

Tweaked the program until the compiler is happy 😊
## Performance

<table>
<thead>
<tr>
<th></th>
<th>PhP</th>
<th>Python</th>
<th>Python (Jitted)</th>
<th>Java</th>
<th>In C</th>
<th>Transposed</th>
<th>Tiled</th>
<th>Vectorized</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms</td>
<td>9,298,440</td>
<td>6,145,070</td>
<td>540,891</td>
<td>348,749</td>
<td>19,564</td>
<td>17,206</td>
<td>12,887</td>
<td>6,607</td>
</tr>
<tr>
<td>Cycles/OP</td>
<td>343</td>
<td>227</td>
<td>20</td>
<td>13</td>
<td>1/2</td>
<td>1/2</td>
<td>1/3</td>
<td>1/5</td>
</tr>
</tbody>
</table>

1.6x  
1.6x  
1.2x  
2x

11.4x  
17.9x  
1.4x

1.6x  
18x  
27x  
476x  
541x  
722x  
1408x
## Profile Data

<table>
<thead>
<tr>
<th></th>
<th>CPI</th>
<th>L1 Miss Rate</th>
<th>L2 Miss Rate</th>
<th>Percent SSE Instructions</th>
<th>Instructions Retired</th>
</tr>
</thead>
<tbody>
<tr>
<td>In C</td>
<td>4.78</td>
<td>0.24</td>
<td>0.02</td>
<td>43%</td>
<td>13,137,280,000</td>
</tr>
<tr>
<td>Transposed</td>
<td>1.13</td>
<td>0.15</td>
<td>0.02</td>
<td>50%</td>
<td>13,001,486,336</td>
</tr>
<tr>
<td>Tiled</td>
<td>0.49</td>
<td>0.02</td>
<td>0</td>
<td>39%</td>
<td>18,044,811,264</td>
</tr>
<tr>
<td>Vectorized</td>
<td>0.9</td>
<td>0.07</td>
<td>0</td>
<td>88%</td>
<td>3,698,018,048</td>
</tr>
</tbody>
</table>

© Saman Amarasinghe 2008
Tuned Libraries

**BLAS Library**
- Hand tuned library in C/assembly to take the full advantage of hardware

**Intel® Math Kernel Library**
- Experts at Intel figuring out how to get the maximum performance for commonly used math routines
- They have a specially tuned BLAS library for x86
int main(int argc, char *argv[]) {

double *A, *B, *C;
uint64_t started, ended, timeTaken;

A = (double *)calloc( N*N, sizeof( double ) );
B = (double *)calloc( N*N, sizeof( double ) );
C = (double *)calloc( N*N, sizeof( double ) );

int i, j;
started = read_timestamp_counter();
//enum ORDER {CblasRowMajor=101, CblasColMajorR=102};
//enum TRANSPOSE {CblasNotrans=111, CblasTrans=112, CblasConjtrans=113};
//void gemm(CBLAS_ORDER Order, CBLAS_TRANSPOSE TransB, CBLAS_TRANSPOSE TransC,
//     int M,  int N,  int K,
//     double alpha,//     double B[]),  int strideB,
//     double C[],  int strideC,
//     double beta, //     double A[],  int strideA)
//  A = alpha * B x C + beta * A
cblas_dgemm(CblasColMajor, CblasTrans, CblasTrans, N, N, N, 1,B, N, C, N, 0, A, N);

ended = read_timestamp_counter();
timeTaken = (ended - started);
printf("Time: %f ms\n", timeTaken/3158786.0);
<table>
<thead>
<tr>
<th></th>
<th>PhP</th>
<th>Python</th>
<th>Python (Jitted)</th>
<th>Java</th>
<th>In C</th>
<th>Transposed</th>
<th>Tiled</th>
<th>Vectorized</th>
<th>BLAS MxM</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms</td>
<td>9,298,440</td>
<td>6,145,070</td>
<td>540,891</td>
<td>348,749</td>
<td>19,564</td>
<td>17,206</td>
<td>12,887</td>
<td>6,607</td>
<td>1,680</td>
</tr>
<tr>
<td>Cycles/OP</td>
<td>343</td>
<td>227</td>
<td>20</td>
<td>13</td>
<td>1/2</td>
<td>1/2</td>
<td>1/3</td>
<td>1/5</td>
<td>1/17</td>
</tr>
</tbody>
</table>

- Performance metrics for different programming languages and optimization techniques.
- Comparisons include execution time (ms) and cycles per operation (Cycles/OP).
- Enabling optimizations like vectorization and transposition significantly reduces execution time.
## Profile Data

<table>
<thead>
<tr>
<th></th>
<th>CPI</th>
<th>L1 Miss Rate</th>
<th>L2 Miss Rate</th>
<th>Percent SSE</th>
<th>Instructions Retired</th>
</tr>
</thead>
<tbody>
<tr>
<td>In C</td>
<td>4.78</td>
<td>0.24</td>
<td>0.02</td>
<td>43%</td>
<td>13,137,280,000</td>
</tr>
<tr>
<td>Transposed</td>
<td>1.13</td>
<td>0.15</td>
<td>2x</td>
<td>50%</td>
<td>13,001,486,336</td>
</tr>
<tr>
<td>Tiled</td>
<td>0.49</td>
<td>0.02</td>
<td>8x</td>
<td>39%</td>
<td>18,044,811,264</td>
</tr>
<tr>
<td>Vectorized</td>
<td>0.9</td>
<td>0.07</td>
<td>1/4x</td>
<td>88%</td>
<td>3,698,018,048</td>
</tr>
<tr>
<td>BLAS</td>
<td>0.37</td>
<td>0.02</td>
<td>4x</td>
<td>78%</td>
<td>3,833,811,968</td>
</tr>
</tbody>
</table>

© Saman Amarasinghe 2008
Parallel Execution

**Multicores are here**
- 2 to 6 cores in a processor,
- 1 to 4 processors in a box
- Cloud machines have 4 processors with 6 cores each (total 24 cores)

**Use concurrency for parallel execution**
- Divide the computation into multiple independent/concurrent computations
- Run the computations in parallel
- Synchronize at the end
Issues with Parallelism

Amdhal’s Law

- Any computation can be analyzed in terms of a portion that must be executed sequentially, $T_s$, and a portion that can be executed in parallel, $T_p$. Then for $n$ processors:
  - $T(n) = T_s + \frac{T_p}{n}$
  - $T(\infty) = T_s$, thus maximum speedup $(T_s + T_p) / T_s$

Load Balancing

- The work is distributed among processors so that all processors are kept busy all of the time.

Granularity

- The size of the parallel regions between synchronizations or the ratio of computation (useful work) to communication (overhead).
Parallel Execution of Matrix Multiply

\[
\begin{align*}
A[0] \times B[0] &= C \\
\end{align*}
\]
<table>
<thead>
<tr>
<th></th>
<th>PhP</th>
<th>Python</th>
<th>Python (Jitted)</th>
<th>Java</th>
<th>In C</th>
<th>Transposed</th>
<th>Tiled</th>
<th>Vectorized</th>
<th>BLAS MxM</th>
<th>BLAS Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms</td>
<td>9,298,440</td>
<td>6,145,070</td>
<td>540,891</td>
<td>348,749</td>
<td>19,564</td>
<td>17,206</td>
<td>12,887</td>
<td>6,607</td>
<td>1,680</td>
<td>182</td>
</tr>
<tr>
<td>Cycles/OP</td>
<td>343</td>
<td>227</td>
<td>20</td>
<td>13</td>
<td>1/2</td>
<td>1/2</td>
<td>1/3</td>
<td>1/5</td>
<td>1/17</td>
<td>1/149</td>
</tr>
<tr>
<td>1.6x</td>
<td></td>
<td></td>
<td>1.6x</td>
<td></td>
<td>1.2x</td>
<td></td>
<td>2x</td>
<td></td>
<td>9.3x</td>
<td></td>
</tr>
<tr>
<td>11.4x</td>
<td></td>
<td></td>
<td>17.9x</td>
<td></td>
<td>1.4x</td>
<td></td>
<td>4x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.6x</td>
<td></td>
<td></td>
<td></td>
<td>18x</td>
<td>27x</td>
<td></td>
<td></td>
<td></td>
<td>476x</td>
<td></td>
</tr>
<tr>
<td>1.6x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>541x</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>722x</td>
<td></td>
</tr>
<tr>
<td>1.4x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4x</td>
<td></td>
<td></td>
<td></td>
<td>1408x</td>
<td></td>
</tr>
<tr>
<td>5537x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>51057x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PhP</td>
<td>Python</td>
<td>Python (Jitted)</td>
<td>Java</td>
<td>In C</td>
<td>Transposed</td>
<td>Tiled</td>
<td>Vectorized</td>
<td>BLAS M×M</td>
<td>BLAS Parallel</td>
</tr>
<tr>
<td>---------------</td>
<td>-------</td>
<td>--------</td>
<td>-----------------</td>
<td>--------</td>
<td>--------</td>
<td>------------</td>
<td>-------</td>
<td>------------</td>
<td>----------</td>
<td>---------------</td>
</tr>
<tr>
<td><strong>ms</strong></td>
<td>9,298,440</td>
<td>6,145,070</td>
<td>540,891</td>
<td>348,749</td>
<td>19,564</td>
<td>17,206</td>
<td>12,887</td>
<td>6,607</td>
<td>1,680</td>
<td>182</td>
</tr>
</tbody>
</table>

![Diagram with ratios and cycle counts](attachment:image.png)

<table>
<thead>
<tr>
<th></th>
<th>Cycles/OP</th>
<th>343</th>
<th>227</th>
<th>20</th>
<th>13</th>
<th>1/2</th>
<th>1/2</th>
<th>1/3</th>
<th>1/5</th>
<th>1/17</th>
<th>1/149</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PhP</strong></td>
<td>9,298,440</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Python</strong></td>
<td>6,145,070</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Python (Jitted)</strong></td>
<td>540,891</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Java</strong></td>
<td>348,749</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>In C</strong></td>
<td>19,564</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Transposed</strong></td>
<td>17,206</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Tiled</strong></td>
<td>12,887</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Vectorized</strong></td>
<td>6,607</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>BLAS M×M</strong></td>
<td>1,680</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>BLAS Parallel</strong></td>
<td>182</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary

There is a lot of room for performance improvements!

- Matrix Multiply is an exception, other programs may not yield gains this large
- That said, in Matrix Multiple from PhP to Parallel BLAS 51,057x improvement
- In comparison Miles per Gallon improvement

Need to have a good understanding on what the hardware and underling software is doing