Matrix Multiply: A Case Study

Saman Amarasinghe
Fall 2008
Outline

Administrivia

Why Performance Engineering

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

Prof. Saman Amarasinghe

- [saman@mit.edu](mailto:saman@mit.edu) 32-G778

Prof. Martin Rinard

- [rinard@csail.mit.edu](mailto:rinard@csail.mit.edu) 32-G744

Prof. Charles Leiserson

- [cel@mit.edu](mailto:cel@mit.edu) 32-G768

Marek Olszewski

- [mareko@mit.edu](mailto:mareko@mit.edu) 32-G776

Ceryen Tan

- [ceryen@mit.edu](mailto:ceryen@mit.edu)
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
Projects

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/fa08/6.197/)
- C for Java programmer
- Gdb debugger
- Makefiles
- (x86 assembly)

© Saman Amarasinghe 2008
Projects

Machines

- cagnode1.csail.mit.edu to cagnode8.csail.mit.edu
  - 2 × 3.15 GHz 4 core Intel processors with 8 GB of memory
- Everyone should use the same machines
  - uniform performance across the class
- You will have to coordinate, so performance runs get the full machine
  - May use other machines to prototype, but get the performance numbers on cagnodes
# Projects

<table>
<thead>
<tr>
<th>Project</th>
<th>Type</th>
<th>Out Date</th>
<th>Due</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project 1</td>
<td>Individual</td>
<td>Wed Sep 3rd</td>
<td>Mon Sep 15th</td>
</tr>
<tr>
<td>Project 2</td>
<td>Individual</td>
<td>Mon Sep 15th</td>
<td>Part 1 Wed Sep 24th, Part 2 Mon Oct 6th</td>
</tr>
<tr>
<td>Project 3</td>
<td>Group of 2</td>
<td>Mon Oct 6th</td>
<td>Mon Oct 20th</td>
</tr>
<tr>
<td>Project 5</td>
<td>Individual</td>
<td>Wed Nov 5th</td>
<td>Mon Nov 17th</td>
</tr>
<tr>
<td>Project 6</td>
<td>Group of 3-4</td>
<td>Wed Nov 12th</td>
<td>Design Mon Nov 24th, Final Wed Dec 10th</td>
</tr>
</tbody>
</table>

© Saman Amarasinghe 2008
## Grades

<table>
<thead>
<tr>
<th></th>
<th>Out</th>
<th>Due</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Project 1</strong> Individual</td>
<td>Wed Sep 3(^{rd})</td>
<td>Mon Sep 15(^{th})</td>
<td>4%</td>
</tr>
<tr>
<td><strong>Project 2</strong> Individual</td>
<td>Mon Sep 15(^{th})</td>
<td>Part 1 Wed Sep 24(^{th})</td>
<td>10%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Part 2 Mon Oct 6(^{th})</td>
<td></td>
</tr>
<tr>
<td><strong>Project 3</strong> Group of 2</td>
<td>Mon Oct 6(^{th})</td>
<td>Mon Oct 20(^{th})</td>
<td>10%</td>
</tr>
<tr>
<td><strong>Project 4</strong> Group of 2</td>
<td>Mon Oct 20(^{th})</td>
<td>Part 1 Wed Oct 29(^{th})</td>
<td>12%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Part 2 Wed Nov 12(^{th})</td>
<td></td>
</tr>
<tr>
<td><strong>Project 5</strong> Individual</td>
<td>Wed Nov 5(^{th})</td>
<td>Mon Nov 17(^{th})</td>
<td>4%</td>
</tr>
<tr>
<td><strong>Project 6</strong> Group of 3-4</td>
<td>Wed Nov 12(^{th})</td>
<td>Design Mon Nov 24(^{th})</td>
<td>25%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Final Wed Dec 10(^{th})</td>
<td></td>
</tr>
<tr>
<td><strong>Quiz 1</strong></td>
<td>Wed Oct 8(^{th})</td>
<td></td>
<td>15%</td>
</tr>
<tr>
<td><strong>Quiz 2</strong></td>
<td>Wed Dec 3(^{rd})</td>
<td></td>
<td>15%</td>
</tr>
<tr>
<td><strong>Class Participation</strong></td>
<td></td>
<td></td>
<td>5%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>100%</strong></td>
</tr>
</tbody>
</table>
Performance Engineering
Why Performance is Important?

Acceptable response time
Ability to add more functionality
Ability to scale
Use less power / resources
Acceptable Response Times

Many systems have stringent requirements

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

If the response times are not met, system is not usable

© Saman Amarasinghe 2008
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
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 effect may be hidden by a bigger first-order effect
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

\[
\begin{align*}
\text{for (int } i = 0; i < x; i++) \\
\quad \text{for (int } j = 0; j < y; j++) \\
\quad \quad \text{for (int } k = 0; k < z; k++ ) \\
\quad \quad \quad A[i][j] += B[i][k]*C[k][j]
\end{align*}
\]
Matrix Representation

I’d like my matrix representation to be

- Object oriented
- Immutable
- Represent both integers and doubles
public class Value {
    final MatrixType type;
    final int iVal;
    final double dVal;

    Value(int i) ......

    Value(double d) {
        type = MatrixType.FLOATING_POINT;
        dVal = d;
        iVal = 0;
    }

    int getInt() throws Exception ......

    double getDouble() throws Exception {
        if(type == MatrixType.FLOATING_POINT)
            return dVal;
        else
            throw new Exception();
    }
}
public class Matrix {
    final MatrixRow[] rows;
    final int nRows, nColumns;
    final MatrixType type;

    Matrix(int rows, int cols, MatrixType type) {
        this.type = type;
        this.nRows = rows;
        this.nColumns = cols;
        this.rows = new MatrixRow[this.nRows];
        for(int i=0; i<this.nRows; i++)
            this.rows[i] = (type == MatrixType.INTEGER) ?
                new IntegerRow(this.nColumns) : new DoubleRow(this.nColumns);
    }
}

......
......

}
public class Matrix {
    
    private Matrix(MatrixRow[] rows, MatrixType type, int nRows, int nCols) {
        this.rows = rows;
        this.nRows = nRows;
        this.nColumns = nCols;
        this.type = type;
    }

    public Matrix update(int row, int col, Value val) throws Exception {
        MatrixRow[] newRows = new MatrixRow[nRows];
        for(int i=0; i<nRows; i++)
            newRows[i] = (i == row)?rows[i].update(col, val):rows[i];
        return new Matrix(newRows, type, nRows, nColumns);
    }

    public Matrix update(int row, int col, Value val) throws Exception {
        MatrixRow[] newRows = new MatrixRow[nRows];
        for(int i=0; i<nRows; i++)
            newRows[i] = (i == row)?rows[i].update(col, val):rows[i];
        return new Matrix(newRows, type, nRows, nColumns);
    }

    Value get(int row, int col) throws Exception {
        return rows[row].get(col);
    }
}

private Matrix(MatrixRow[] rows, MatrixType type, int nRows, int nCols) {
    this.rows = rows;
    this.nRows = nRows;
    this.nColumns = nCols;
    this.type = type;
}

public Matrix update(int row, int col, Value val) throws Exception {
    MatrixRow[] newRows = new MatrixRow[nRows];
    for(int i=0; i<nRows; i++)
        newRows[i] = (i == row)?rows[i].update(col, val):rows[i];
    return new Matrix(newRows, type, nRows, nColumns);
}

Value get(int row, int col) throws Exception {
    return rows[row].get(col);
}
public abstract class MatrixRow {
    abstract Value get(int col) throws Exception;
    abstract public MatrixRow update(int col, Value val) throws Exception;
}
public class DoubleRow extends MatrixRow {
    final Double[] theRow;
    public final int numColumns;

    DoubleRow(int ncols) {
        this.numColumns = ncols;
        theRow = new Double[ncols];
        for(int i=0; i < ncols; i++)
            theRow[i] = new Double(0);
    }

    private DoubleRow(Double[] row, int cols) {
        this.theRow = row;
        this.numColumns = cols;
    }

    public MatrixRow update(int col, Value val) throws Exception {
        Double[] row = new Double[numColumns];
        for(int i=0; i< numColumns; i++)
            row[i] = (i==col)?(new Double(val.getDouble())):theRow[i];
        return new DoubleRow(row, numColumns);
    }

    public Value get(int col) {
        return new Value(theRow[col]);
    }
}
public class MatrixMultiply {

    public static long testMM(int x, int y, int z) {
        Matrix A = new Matrix(x, y, MatrixType.FLOATING_POINT);
        Matrix B = new Matrix(y, z, MatrixType.FLOATING_POINT);
        Matrix C = new Matrix(x, z, MatrixType.FLOATING_POINT);

        long started = System.nanoTime();
        try {
            for(int i =0; i < x; i++)
                for(int j =0; j < y; j++)
                    for(int k=0; k < z; k++)
                        A = A.update(i, j, new Value(A.get(i, j).getDouble() +
                            B.get(i, k).getDouble() *
                            C.get(k, j).getDouble()));
        } catch(Exception e) {
        }
        long time = System.nanoTime();
        long timeTaken = (time - started);
        System.out.println("Time:" + timeTaken/1000000 + "ms");
        return timeTaken;
    }
}
Performance

<table>
<thead>
<tr>
<th></th>
<th>Immutable</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms</td>
<td>17,094.152</td>
</tr>
<tr>
<td>Cycles/OP</td>
<td>8,358</td>
</tr>
</tbody>
</table>

1024x1024 matrix multiply

Is the performance good?

It took almost 5 hours to multiply two 1024x1024 matrices

$1024^3 = 1,073,741,824$ operations

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

$1,073,741,824 \times 6 = 6,442,450,944$

Operations per second = $6,442,450,944 / 17,094 = 376,880 = 3.77 \times 10^5$

My PC runs at 3.15 GHz → $3.15 \times 10^9$ cycles / second

That comes to about 8,358 cycles per each visible operation

How can we improve performance?
Profiling

Look deeply into 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>Method</th>
<th>Num Calls</th>
<th>Method Time</th>
<th>Cumulative Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.Double.&lt;init&gt;(double)</td>
<td>3,157,263</td>
<td>52,100</td>
<td>52,100</td>
</tr>
<tr>
<td>DoubleRow.&lt;init&gt;(int)</td>
<td>3,072</td>
<td>51,120</td>
<td>102,980</td>
</tr>
<tr>
<td>DoubleRow.update(int, Value)</td>
<td>11,535</td>
<td>31,630</td>
<td>32,610</td>
</tr>
<tr>
<td>Matrix.update(int, int, Value)</td>
<td>11,535</td>
<td>30,740</td>
<td>63,540</td>
</tr>
<tr>
<td>MatrixMultiply.testMM(int, int, int)</td>
<td>1</td>
<td>1,790</td>
<td>172,410</td>
</tr>
<tr>
<td>DoubleRow.get(int)</td>
<td>34,605</td>
<td>1,290</td>
<td>1,870</td>
</tr>
<tr>
<td>Matrix.get(int, int)</td>
<td>34,605</td>
<td>1,170</td>
<td>3,040</td>
</tr>
<tr>
<td>Value.getDouble()</td>
<td>46,140</td>
<td>1,000</td>
<td>1,000</td>
</tr>
<tr>
<td>Value.&lt;init&gt;(double)</td>
<td>46,140</td>
<td>810</td>
<td>810</td>
</tr>
<tr>
<td>DoubleRow.&lt;init&gt;(Double[], int)</td>
<td>11,535</td>
<td>310</td>
<td>480</td>
</tr>
<tr>
<td>MatrixRow.&lt;init&gt;()</td>
<td>14,607</td>
<td>220</td>
<td>220</td>
</tr>
<tr>
<td>Matrix.&lt;init&gt;(MatrixRow[], MatrixType, int, int)</td>
<td>11,534</td>
<td>190</td>
<td>190</td>
</tr>
<tr>
<td>Matrix.&lt;init&gt;(int, int, MatrixType)</td>
<td>3</td>
<td>40</td>
<td>103,020</td>
</tr>
<tr>
<td>Main.&lt;init&gt;()</td>
<td>1</td>
<td>10</td>
<td>172,420</td>
</tr>
<tr>
<td>&lt;ROOT&gt;.&lt;ROOT&gt;</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Main.main(String[])</td>
<td>1</td>
<td>-</td>
<td>172,420</td>
</tr>
<tr>
<td>java.lang.Object.&lt;init&gt;()</td>
<td>72,285</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.System.nanoTime()</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.StringBuilder.append(int)</td>
<td>7</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>MatrixType.&lt;clinit&gt;()</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.StringBuilder.append(String)</td>
<td>7</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.StringBuilder.&lt;init&gt;()</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.StringBuilder.toString()</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>java.io.PrintStream.println(String)</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>MatrixType.&lt;init&gt;(String, int)</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.Double.doubleValue()</td>
<td>34,605</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.Enum.&lt;init&gt;(String, int)</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

© Saman Amarasinghe 2008
Issues with Immutability

Updating one location $\rightarrow$ copy of the matrix

$2^N$ copies for each update

$N^3$ updates $\rightarrow$ $N^4$ copies are made.

Copying is costly

- Cost of making duplicates
- Cost of garbage collecting the freed objects
- Huge memory footprint

Can we do better?
Matrix Representation

I’d like my matrix representation to be

- Object oriented
- Immutable
- Represent both integers and doubles
public class Matrix {
    MatrixRow[] rows;
    final int nRows, nColumns;
    final MatrixType type;

    Matrix(int rows, int cols, MatrixType type) {
        this.type = type;
        this.nRows = rows;
        this.nColumns = cols;
        this.rows = new MatrixRow[this.nRows];
        for(int i=0; i<this.nRows; i++)
            this.rows[i] = (type == MatrixType.INTEGER)?
                            new IntegerRow(this.nColumns):new DoubleRow(this.nColumns);
    }

    void set(int row, int col, Value v) throws Exception  {
        rows[row].set(col, v);
    }

    Value get(int row, int col) throws Exception {
        return rows[row].get(col);
    }
}
public class DoubleRow extends MatrixRow {
    double[] theRow;
    public final int numColumns;

    DoubleRow(int ncols) {
        this.numColumns = ncols;
        theRow = new double[ncols];
    }

    public void set(int col, Value val) throws Exception {
        theRow[col] = val.getDouble();
    }

    public Value get(int col) {
        return new Value(theRow[col]);
    }
}

How much do you think the performance will improve?
Performance

<table>
<thead>
<tr>
<th></th>
<th>Immutable</th>
<th>Mutable</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms</td>
<td>17,094,152</td>
<td>77,826</td>
</tr>
<tr>
<td>Cycles/OI</td>
<td>8,358</td>
<td>38</td>
</tr>
</tbody>
</table>

219.7x
Profile Data

<table>
<thead>
<tr>
<th>Method</th>
<th>Num Calls</th>
<th>Method Time</th>
<th>Cumulative Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>MatrixMultiply.testMM(int, int, int)</td>
<td>1</td>
<td>40,076</td>
<td>171,425</td>
</tr>
<tr>
<td>Value.getDouble()</td>
<td>19,589,742</td>
<td>36,791</td>
<td>36,791</td>
</tr>
<tr>
<td>Matrix.get(int, int)</td>
<td>14,692,308</td>
<td>27,725</td>
<td>64,624</td>
</tr>
<tr>
<td>DoubleRow.get(int)</td>
<td>14,692,307</td>
<td>25,343</td>
<td>36,900</td>
</tr>
<tr>
<td>Value.&lt;init&gt;(double)</td>
<td>19,589,742</td>
<td>15,501</td>
<td>15,501</td>
</tr>
<tr>
<td>Matrix.set(int, int, Value)</td>
<td>4,897,435</td>
<td>13,032</td>
<td>35,220</td>
</tr>
<tr>
<td>DoubleRow.set(int, Value)</td>
<td>4,897,435</td>
<td>12,932</td>
<td>22,188</td>
</tr>
<tr>
<td>DoubleRow.&lt;init&gt;(int)</td>
<td>3,072</td>
<td>21</td>
<td>23</td>
</tr>
<tr>
<td>MatrixRow.&lt;init&gt;()</td>
<td>3,072</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Matrix.&lt;init&gt;(int, int, MatrixType)</td>
<td>3</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>Main.&lt;init&gt;()</td>
<td>1</td>
<td>1</td>
<td>171,426</td>
</tr>
<tr>
<td>java.io.PrintStream.println(String)</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.StringBuilder.append(int)</td>
<td>7</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.System.nanoTime()</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Main.main(String[])</td>
<td>1</td>
<td>-</td>
<td>171,426</td>
</tr>
<tr>
<td>MatrixType.&lt;clinit&gt;()</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.StringBuilder.append(String)</td>
<td>7</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.StringBuilder.&lt;init&gt;()</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>MatrixType.&lt;init&gt;(String, int)</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.StringBuilder.toString()</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.Enum.&lt;init&gt;(String, int)</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>&lt;ROOT&gt;.&lt;ROOT&gt;</td>
<td>19,592,818</td>
<td>-</td>
<td>171,426</td>
</tr>
<tr>
<td>java.lang.Object.&lt;init&gt;()</td>
<td>19,592,818</td>
<td>-</td>
<td>171,426</td>
</tr>
</tbody>
</table>
Issues with Dynamic Dispatch

**Method call overhead**
- Multiple subtypes → what method to call depends on the object
- Each method call needs to loop-up the object type in a dispatch table
- Dynamic dispatch is an address lookup + indirect branch

**Indirect branches are costly**
- Modern microprocessors are deeply pipelined
  - 12 pipeline stages in core 2 duo, 20 in Pentium 4
  - i.e. hundreds of instructions in flight
- Need to be able to keep fetching next instructions before executing them
- Normal instructions → keep fetching the next instructions
- Direct branch → target address known, can fetch ahead from target
  → works for conditional branches by predicting the branch
- Indirect branch → target unknown, need to wait until address fetch completes
  → pipeline stall
Matrix Representation

I’d like my matrix representation to be

- Object oriented
- Immutable
- Represent both integers and doubles
public class DoubleMatrix {
    final DoubleRow[] rows;
    final int nRows, nColumns;

    Matrix(int rows, int cols) {
        this.nRows = rows;
        this.nColumns = cols;
        this.rows = new DoubleRow[this.nRows];
        for(int i=0; i<this.nRows; i++)
            this.rows[i] = new DoubleRow(this.nColumns);
    }

    void set(int row, int col, double v)  {
        rows[row].set(col, v);
    }

    double get(int row, int col) {
        return rows[row].get(col);
    }
}
public final class DoubleRow {
    double[] theRow;
    public final int numColumns;

    DoubleRow(int ncols) {
        this.numColumns = ncols;
        theRow = new double[ncols];
    }

    public void set(int col, double val) throws Exception {
        theRow[col] = val;
    }

    public double get(int col) throws Exception {
        return theRow[col];
    }
}

## Performance

<table>
<thead>
<tr>
<th></th>
<th>Immutable</th>
<th>Mutable</th>
<th>Double Only</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms</td>
<td>17,094,152</td>
<td>77,826</td>
<td>32,800</td>
</tr>
<tr>
<td>Cycles/OP</td>
<td>8,358</td>
<td>38</td>
<td>16</td>
</tr>
</tbody>
</table>

- 219.7x
- 2.4x
- 522x
## Profile Data

<table>
<thead>
<tr>
<th>Method</th>
<th>Num Calls</th>
<th>Method Time</th>
<th>Cumulative Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix.get(int, int)</td>
<td>1,943,313</td>
<td>66,120</td>
<td>100,310</td>
</tr>
<tr>
<td>MatrixMultiply.testMM(int, int, int)</td>
<td>1</td>
<td>44,590</td>
<td>179,960</td>
</tr>
<tr>
<td>DoubleRow.get(int)</td>
<td>1,943,313</td>
<td>34,190</td>
<td>34,190</td>
</tr>
<tr>
<td>Matrix.set(int, int, double)</td>
<td>647,770</td>
<td>22,950</td>
<td>34,940</td>
</tr>
<tr>
<td>DoubleRow.set(int, double)</td>
<td>647,770</td>
<td>11,990</td>
<td>11,990</td>
</tr>
<tr>
<td>DoubleRow.&lt;init&gt;(int)</td>
<td>3,072</td>
<td>70</td>
<td>70</td>
</tr>
<tr>
<td>Matrix.&lt;init&gt;(int, int)</td>
<td>3</td>
<td>50</td>
<td>120</td>
</tr>
<tr>
<td>&lt;ROOT&gt;.&lt;ROOT&gt;</td>
<td>-</td>
<td>-</td>
<td>179,960</td>
</tr>
<tr>
<td>Main.main(String[])</td>
<td>1</td>
<td>-</td>
<td>179,960</td>
</tr>
<tr>
<td>Main.&lt;init&gt;()</td>
<td>1</td>
<td>-</td>
<td>179,960</td>
</tr>
<tr>
<td>java.lang.Object.&lt;init&gt;()</td>
<td>3,076</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.System.nanoTime()</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.StringBuilder.toString()</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.StringBuilder.&lt;init&gt;()</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.StringBuilder.append(int)</td>
<td>7</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.StringBuilder.append(String)</td>
<td>7</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>java.io.PrintStream.println(String)</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

© Saman Amarasinghe 2008
### Profile Data

<table>
<thead>
<tr>
<th>Method</th>
<th>Num Calls</th>
<th>Method Time</th>
<th>Cumulative Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.Double.&lt;init&gt;(double)</td>
<td>3,157,263</td>
<td>52,100</td>
<td>52,100</td>
</tr>
<tr>
<td>DoubleRow.&lt;init&gt;(int)</td>
<td>3,072</td>
<td>51,120</td>
<td>102,980</td>
</tr>
<tr>
<td>DoubleRow.update(int, Value)</td>
<td>11,535</td>
<td>31,630</td>
<td>32,610</td>
</tr>
<tr>
<td>Matrix.update(int, int, Value)</td>
<td>11,535</td>
<td>30,740</td>
<td>63,540</td>
</tr>
<tr>
<td>MatrixMultiply.testMM(int, int, int)</td>
<td>1</td>
<td>1,790</td>
<td>172,410</td>
</tr>
<tr>
<td>DoubleRow.get(int)</td>
<td>34,605</td>
<td>1,290</td>
<td>1,870</td>
</tr>
<tr>
<td>Matrix.get(int, int)</td>
<td>34,605</td>
<td>1,170</td>
<td>3,040</td>
</tr>
<tr>
<td>Value.getDouble()</td>
<td>46,140</td>
<td>1,000</td>
<td>1,000</td>
</tr>
<tr>
<td>Value.&lt;init&gt;(double)</td>
<td>46,140</td>
<td>810</td>
<td>810</td>
</tr>
<tr>
<td>DoubleRow.&lt;init&gt;(Double[], int)</td>
<td>11,535</td>
<td>310</td>
<td>480</td>
</tr>
<tr>
<td>MatrixRow.&lt;init&gt;()</td>
<td>14,607</td>
<td>220</td>
<td>220</td>
</tr>
<tr>
<td>Matrix.&lt;init&gt;(MatrixRow[], MatrixType, int, int)</td>
<td>11,534</td>
<td>190</td>
<td>190</td>
</tr>
</tbody>
</table>

### Immutable

<table>
<thead>
<tr>
<th>Method</th>
<th>Num Calls</th>
<th>Method Time</th>
<th>Cumulative Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>MatrixMultiply.testMM(int, int, int)</td>
<td>1</td>
<td>40,076</td>
<td>171,425</td>
</tr>
<tr>
<td>Value.getDouble()</td>
<td>19,589,742</td>
<td>36,791</td>
<td>36,791</td>
</tr>
<tr>
<td>Matrix.get(int, int)</td>
<td>14,692,308</td>
<td>27,725</td>
<td>64,624</td>
</tr>
<tr>
<td>DoubleRow.get(int)</td>
<td>14,692,307</td>
<td>25,343</td>
<td>36,900</td>
</tr>
<tr>
<td>Value.&lt;init&gt;(double)</td>
<td>19,589,742</td>
<td>15,501</td>
<td>15,501</td>
</tr>
<tr>
<td>Matrix.set(int, int, Value)</td>
<td>4,897,435</td>
<td>13,032</td>
<td>35,220</td>
</tr>
<tr>
<td>DoubleRow.set(int, Value)</td>
<td>4,897,435</td>
<td>12,932</td>
<td>22,188</td>
</tr>
</tbody>
</table>

### Double Only

<table>
<thead>
<tr>
<th>Method</th>
<th>Num Calls</th>
<th>Method Time</th>
<th>Cumulative Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix.get(int, int)</td>
<td>1,943,313</td>
<td>66,120</td>
<td>100,310</td>
</tr>
<tr>
<td>MatrixMultiply.testMM(int, int, int)</td>
<td>1</td>
<td>44,590</td>
<td>179,960</td>
</tr>
<tr>
<td>DoubleRow.get(int)</td>
<td>1,943,313</td>
<td>34,190</td>
<td>34,190</td>
</tr>
<tr>
<td>Matrix.set(int, int, double)</td>
<td>647,770</td>
<td>22,950</td>
<td>34,940</td>
</tr>
<tr>
<td>DoubleRow.set(int, double)</td>
<td>647,770</td>
<td>11,990</td>
<td>11,990</td>
</tr>
<tr>
<td>DoubleRow.&lt;init&gt;(int)</td>
<td>3,072</td>
<td>70</td>
<td>70</td>
</tr>
</tbody>
</table>

© Saman Amarasinghe 2008
Issues with Object Oriented

**Memory fragmentation**
- Objects are allocated independently
- All over memory
- If contiguous in memory → getting to the next is just an index increment

**Method call overhead**
- Method calls are expensive
- Cannot optimize the loop body because of the method call
Matrix Representation

I’d like my matrix representation to be

- Object oriented
- Immutable
- Represent both integers and doubles
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>Immutable</th>
<th>Mutable</th>
<th>Double Only</th>
<th>No Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms</td>
<td>17,094,152</td>
<td>77,826</td>
<td>32,800</td>
<td>15,306</td>
</tr>
<tr>
<td>Cycles/OP</td>
<td>8,358</td>
<td>38</td>
<td>16</td>
<td>7</td>
</tr>
</tbody>
</table>

### Performance Ratios
- 219.7x
- 2.2x
- 2.4x
- 219.7x
- 522x
- 1117x
From Java to C

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

C
- No such thing in C
- Intel C compiler compiles the program directly into x86 assembly
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>Immutable</th>
<th>Mutable</th>
<th>Double Only</th>
<th>No Objects</th>
<th>In C</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ms</strong></td>
<td>17,094,152</td>
<td>77,826</td>
<td>32,800</td>
<td>15,306</td>
<td>7,530</td>
</tr>
<tr>
<td><strong>Cycles/OP</strong></td>
<td>8,358</td>
<td>38</td>
<td>16</td>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>
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>

© Saman Amarasinghe 2008
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

© Saman Amarasinghe 2008
#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>Immutable</th>
<th>Mutable</th>
<th>Double Only</th>
<th>No Objects</th>
<th>In C</th>
<th>Transposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms</td>
<td>17,094,152</td>
<td>77,826</td>
<td>32,800</td>
<td>15,306</td>
<td>7,530</td>
<td>2,275</td>
</tr>
<tr>
<td></td>
<td>219.7x</td>
<td>2.2x</td>
<td>3.4x</td>
<td>2.4x</td>
<td>2.1x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.197x</td>
<td>522x</td>
<td></td>
<td>1117x</td>
<td>2271x</td>
<td>7514x</td>
</tr>
<tr>
<td>Cycles/OP</td>
<td>8,358</td>
<td>38</td>
<td>16</td>
<td>7</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>
## 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>

© Saman Amarasinghe 2008
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
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);
<table>
<thead>
<tr>
<th></th>
<th>Immutable</th>
<th>Mutable</th>
<th>Double Only</th>
<th>No Objects</th>
<th>In C</th>
<th>Transposed</th>
<th>Tiled</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ms</strong></td>
<td>17,994,152</td>
<td>77,826</td>
<td>32,800</td>
<td>15,306</td>
<td>7,530</td>
<td>2,275</td>
<td>1,388</td>
</tr>
<tr>
<td><strong>219.7x</strong></td>
<td></td>
<td></td>
<td>2.2x</td>
<td>3.4x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>2.4x</strong></td>
<td></td>
<td></td>
<td>2.1x</td>
<td>1.7x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>219.7x</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>522x</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>1117x</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>2271x</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>7514x</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>12316x</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Cycles/OP</strong></td>
<td>8,358</td>
<td>38</td>
<td>16</td>
<td>7</td>
<td>4</td>
<td>1</td>
<td>1/2</td>
</tr>
</tbody>
</table>
## 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>0.02</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 😊

Inner loop: SSE instructions

© Saman Amarasinghe 2008
## Performance

<table>
<thead>
<tr>
<th></th>
<th>Immutable</th>
<th>Mutable</th>
<th>Double Only</th>
<th>No Objects</th>
<th>In C</th>
<th>Transposed</th>
<th>Tiled</th>
<th>vectorized</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms</td>
<td>17,094,152</td>
<td>77,826</td>
<td>32,800</td>
<td>15,306</td>
<td>7,530</td>
<td>2,275</td>
<td>1,388</td>
<td>511</td>
</tr>
<tr>
<td>219.7x</td>
<td>522x</td>
<td>1117x</td>
<td>2271x</td>
<td>7514x</td>
<td>12316x</td>
<td>33453x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>219.7x</td>
<td>2.4x</td>
<td>2.2x</td>
<td>3.4x</td>
<td>2.8x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>219.7x</td>
<td>1.7x</td>
<td>2.1x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycles/OP</td>
<td>8,358</td>
<td>38</td>
<td>16</td>
<td>7</td>
<td>4</td>
<td>1</td>
<td>1/2</td>
<td>1/5</td>
</tr>
</tbody>
</table>
## 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);
## Performance

<table>
<thead>
<tr>
<th></th>
<th>Immutable</th>
<th>Mutable</th>
<th>Double Only</th>
<th>No Objects</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>17,094,152</td>
<td>77,826</td>
<td>32,800</td>
<td>15,306</td>
<td>7,530</td>
<td>2,275</td>
<td>1,388</td>
<td>511</td>
<td>196</td>
</tr>
</tbody>
</table>

- 219.7x
- 522x
- 1117x
- 2271x
- 7514x
- 12316x
- 33453x
- 87042x

| Cycles/OP | 8,358 | 38 | 16 | 7 | 4 | 1 | 1/2 | 1/5 | 1/11 |
## 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>
<tr>
<td>BLAS</td>
<td>0.37</td>
<td>0.02</td>
<td>0</td>
<td>78%</td>
<td>3,833,811,968</td>
</tr>
</tbody>
</table>

© Saman Amarasinghe 2008
Parallel Execution

Multicores are here
- 2 to 4 cores in a processor,
- 1 to 4 processors in a box
- Cagnodes have 2 processors with 4 cores each (total 8 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:
  - $T(n) = T_s + T_p/n$
  - $T(\infty) = T_s$, 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>Performance</th>
<th>Immutable</th>
<th>Mutable</th>
<th>Double Only</th>
<th>No Objects</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>17,094,152</td>
<td>77,826</td>
<td>32,800</td>
<td>15,306</td>
<td>7,530</td>
<td>2,275</td>
<td>1,388</td>
<td>511</td>
<td>196</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>219.7x</td>
<td>2.2x</td>
<td>3.4x</td>
<td>2.8x</td>
<td></td>
<td>3.5x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.4x</td>
<td>2.1x</td>
<td>1.7x</td>
<td>2.7x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>522x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1117x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2271x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7514x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>12316x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>33453x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>87042x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>296260x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycles/OP</td>
<td>8,358</td>
<td>38</td>
<td>16</td>
<td>7</td>
<td>4</td>
<td>1</td>
<td>1/2</td>
<td>1/5</td>
<td>1/11</td>
<td>1/36</td>
</tr>
<tr>
<td></td>
<td>Immutable</td>
<td>Mutable</td>
<td>Double Only</td>
<td>No Objects</td>
<td>In C</td>
<td>Transposed</td>
<td>Tiled</td>
<td>vectorized</td>
<td>BLAS MxM</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></td>
<td>ms</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>511</td>
<td>196</td>
</tr>
<tr>
<td></td>
<td>17,094,152</td>
<td>77,826</td>
<td>32,800</td>
<td>15,306</td>
<td>7,530</td>
<td>2,275</td>
<td>1,388</td>
<td>511</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>219.7x</td>
<td></td>
<td>2.2x</td>
<td></td>
<td></td>
<td></td>
<td>2.8x</td>
<td>3.5x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.4x</td>
<td></td>
<td>2.1x</td>
<td></td>
<td></td>
<td></td>
<td>2.7x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>219.7x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1349x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>527x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>569x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1117x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>266x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2271x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>131x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7514x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>40x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>12316x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>25x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>33453x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>87042x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>296260x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycles/OP</td>
<td>8,358</td>
<td>38</td>
<td>16</td>
<td>7</td>
<td>4</td>
<td>1</td>
<td>1/2</td>
<td>1/5</td>
<td>1/11</td>
<td>1/36</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 Immutable to Parallel BLAS 296,260x improvement
- In comparison Miles per Gallon improvement

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