Matrix Multiply: A Case Study

Saman Amarasinghe
Fall 2010
Matrix multiple is a fundamental operation in many computations

- Example: video encoding, weather simulation, computer graphics

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

    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];

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

<table>
<thead>
<tr>
<th>Class</th>
<th>Relationship</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>DoubleRow</td>
<td>extends</td>
<td>MatrixRow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Matrix</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IntegerRow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DoubleRow</td>
</tr>
</tbody>
</table>
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;
    }
}
Is the performance good?

It took almost 5 hours to multiply two 1024x1024 matrices

\[ 1024^3 = 1,073,741,824 \text{ operations} \]

Each operation is multiply, add and 3 index updates, and branch check \( \rightarrow 6 \text{ ops} \)

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

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

My PC runs at 3.15 GHz \( \rightarrow 3.15 \times 10^9 \) cycles / second

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

How can we improve performance?
Profiling

Look deeply 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?
<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>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>Main.main(String[])</td>
<td>1</td>
<td>1</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>
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/OP</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>1,958,974</td>
<td>36,791</td>
<td>36,791</td>
</tr>
<tr>
<td>Matrix.get(int, int)</td>
<td>1,469,230</td>
<td>27,725</td>
<td>64,624</td>
</tr>
<tr>
<td>DoubleRow.getRow()</td>
<td>1,692,307</td>
<td>25,343</td>
<td>36,900</td>
</tr>
<tr>
<td>Value.&lt;init&gt;(double)</td>
<td>1,958,974</td>
<td>15,501</td>
<td>15,501</td>
</tr>
<tr>
<td>Matrix.set(int, int, Value)</td>
<td>489,743</td>
<td>13,032</td>
<td>35,220</td>
</tr>
<tr>
<td>DoubleRow.set(int, Value)</td>
<td>489,743</td>
<td>12,932</td>
<td>22,188</td>
</tr>
<tr>
<td>DoubleRow.&lt;init&gt;(int)</td>
<td>372</td>
<td>21</td>
<td>23</td>
</tr>
<tr>
<td>MatrixRow.&lt;init&gt;()</td>
<td>372</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>-</td>
<td>1</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>-</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>-</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>
</tbody>
</table>

- 219.7x
- 2.4x
- 219.7x
- 522x

| Cycles/OP | 8,358 | 38 | 16 |
# 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)</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)</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>
### Profile Data

#### Immutable

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

#### Mutable

<table>
<thead>
<tr>
<th>Method</th>
<th>Num Calls</th>
<th>Method Time</th>
<th>Cumulative Times</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MatrixMultiply.testMM(int, int, int)</code></td>
<td>1</td>
<td>40,076</td>
<td>171,425</td>
</tr>
<tr>
<td><code>Value.getDouble()</code></td>
<td>1,958,974</td>
<td>36,791</td>
<td>36,791</td>
</tr>
<tr>
<td><code>Matrix.get(int, int)</code></td>
<td>1,469,230</td>
<td>27,725</td>
<td>64,624</td>
</tr>
<tr>
<td><code>DoubleRow.get(int)</code></td>
<td>1,469,230</td>
<td>25,343</td>
<td>36,900</td>
</tr>
<tr>
<td><code>Value.&lt;init&gt;(double)</code></td>
<td>1,958,974</td>
<td>15,501</td>
<td>15,501</td>
</tr>
<tr>
<td><code>Matrix.set(int, int)</code></td>
<td>489,743</td>
<td>13,032</td>
<td>35,220</td>
</tr>
<tr>
<td><code>DoubleRow.set(int, Value)</code></td>
<td>489,743</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><code>Matrix.get(int, int)</code></td>
<td>1,943,313</td>
<td>66,120</td>
<td>100,310</td>
</tr>
<tr>
<td><code>MatrixMultiply.testMM(int, int, int)</code></td>
<td>1</td>
<td>44,590</td>
<td>179,960</td>
</tr>
<tr>
<td><code>DoubleRow.get(int)</code></td>
<td>1,943,313</td>
<td>34,190</td>
<td>34,190</td>
</tr>
<tr>
<td><code>Matrix.set(int, int, double)</code></td>
<td>647,770</td>
<td>22,950</td>
<td>34,940</td>
</tr>
<tr>
<td><code>DoubleRow.set(int, double)</code></td>
<td>647,770</td>
<td>11,990</td>
<td>11,990</td>
</tr>
<tr>
<td><code>DoubleRow.&lt;init&gt;(int)</code></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
```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>Immutable</th>
<th>Mutable</th>
<th>Double Only</th>
<th>No Objects</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>
</tr>
<tr>
<td><strong>Cycles/OP</strong></td>
<td>8,358</td>
<td>38</td>
<td>16</td>
<td>7</td>
</tr>
</tbody>
</table>

- 219.7x
- 2.4x
- 219.7x
- 522x
- 1117x
- 2.2x
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

© 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>Immutable</th>
<th>Mutable</th>
<th>Double Only</th>
<th>No Objects</th>
<th>In C</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>
</tr>
</tbody>
</table>

- 219.7x
- 2.4x
- 2.1x
- 522x
- 1117x
- 2271x

| Cycles/OP | 8,358 | 38 | 16 | 7 | 1 |
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 Instructions</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 → 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++)

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

<table>
<thead>
<tr>
<th>Ms</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>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></td>
</tr>
</tbody>
</table>

- 219.7x
- 2.2x
- 3.4x
- 2.4x
- 2.1x
- 522x
- 1117x
- 2271x
- 7514x

| Cycles/OP | 8,358 | 38 | 16 | 7 | 4 | 1 |
# 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 5x</td>
<td>0.02 2x</td>
<td>43%</td>
<td>13,137,280,000</td>
</tr>
<tr>
<td>Transposed</td>
<td>1.13</td>
<td>0.15 1x</td>
<td>0.02 2x</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
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>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>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>
</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>
</tr>
</tbody>
</table>

- 219.7x
- 2.2x
- 3.4x
- 2.4x
- 2.1x
- 1.7x
- 522x
- 1117x
- 2271x
- 7514x
- 12316x
## 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>
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 😞
```c
#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>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>219.7x</td>
<td>2.2x</td>
<td>3.4x</td>
<td>2.8x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>522x</td>
<td></td>
<td>2.4x</td>
<td>2.1x</td>
<td>1.7x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1117x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2271x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7514x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12316x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33453x</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>
</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>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>0</td>
<td>88%</td>
<td>3,698,018,048</td>
</tr>
</tbody>
</table>
Tuned Libraries

BLAS Library
- Hand tuned library in C/assembly to take the full advantage of hardware
- See http://www.netlib.org/blas/

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>
<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>
</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>
<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>
Parallel Execution

**Multicores are here**
- 2 to 6 cores in a processor,
- 1 to 4 processors in a box
- Cloud machines have 2 processors with 6 cores each (total 12 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

\[ C = x A[0] B[0] \]
<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>
<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>
</tbody>
</table>

- 219.7x
- 2.2x
- 3.4x
- 2.8x
- 3.5x

- 2.4x
- 2.1x
- 1.7x
- 2.7x

- 219.7x

- 522x

- 1117x

- 2271x

- 7514x

- 12316x

- 33453x

- 87042x

- 296260x

<p>| Cycles/OP     | 8,358     | 38       | 16          | 7          | 4    | 1          | 1/2   | 1/5        | 1/11     | 1/36          |</p>
<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>
<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>219.7x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.4x</td>
<td>2.2x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>522x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1117x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2271x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7514x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12316x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33453x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>87042x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>296260x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Cycles/OP        | 8,358     | 38       | 16          | 7          | 4    | 1          | 1/2   | 1/5        | 1/11    | 1/36          |
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 underling software is doing