Introduction to Software Performance Engineering

Marc Moreno Maza

University of Western Ontario, London, Ontario (Canada)

CS 433 - CS 9624
Plan

1. Hardware Acceleration Technologies
2. Software Performance Engineering
3. A Case Study: Matrix Multiplication
4. Course Outline
Plan

1. Hardware Acceleration Technologies
2. Software Performance Engineering
3. A Case Study: Matrix Multiplication
4. Course Outline
Hardware Acceleration Technologies

Introduction to Software Performance Engineering
Hardware Acceleration Technologies

Core

L1 inst

L1 data

L1 ins

L1 data

L1 inst

L1 data

L1 ins

L1 data

L2

L2

Main Memory

(Moreno Maza)
### L1 Data Cache

<table>
<thead>
<tr>
<th>Size</th>
<th>Line Size</th>
<th>Latency</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 KB</td>
<td>64 bytes</td>
<td>3 cycles</td>
<td>8-way</td>
</tr>
</tbody>
</table>

### L1 Instruction Cache

<table>
<thead>
<tr>
<th>Size</th>
<th>Line Size</th>
<th>Latency</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 KB</td>
<td>64 bytes</td>
<td>3 cycles</td>
<td>8-way</td>
</tr>
</tbody>
</table>

### L2 Cache

<table>
<thead>
<tr>
<th>Size</th>
<th>Line Size</th>
<th>Latency</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 MB</td>
<td>64 bytes</td>
<td>14 cycles</td>
<td>24-way</td>
</tr>
</tbody>
</table>
### Hardware Acceleration Technologies

**Capacity**

- **CPU Registers**
  - 100s Bytes
  - 300 – 500 ps (0.3-0.5 ns)

- **L1 and L2 Cache**
  - 10s-100s K Bytes
  - ~1 ns - ~10 ns
  - $1000s/ GByte

- **Main Memory**
  - G Bytes
  - 80ns- 200ns
  - ~ $100/ GByte

- **Disk**
  - 10s T Bytes, 10 ms
  - (10,000,000 ns)
  - ~ $1 / GByte

- **Tape**
  - infinite sec-min
  - ~$1 / GByte

**Access Time**

- **CPU Registers**
  - 100s Bytes
  - 300 – 500 ps (0.3-0.5 ns)

- **L1 and L2 Cache**
  - 10s-100s K Bytes
  - ~1 ns - ~10 ns
  - $1000s/ GByte

**Cost**

- **CPU Registers**
  - 100s Bytes

- **L1 and L2 Cache**
  - 10s-100s K Bytes

- **Main Memory**
  - G Bytes

- **Disk**
  - 10s T Bytes, 10 ms

- **Tape**
  - infinite sec-min

**Staging**

- **Xfer Unit**
  - prog./compiler
  - 1-8 bytes

  - cache cntl
  - 32-64 bytes

  - cache cntl
  - 64-128 bytes

- **OS**
  - 4K-8K bytes

- **user/operator**
  - Mbytes

**Upper Level**

- faster

**Lower Level**

- Larger

---

(Moreno Maza)

*Introduction to Software Performance Engineering*
Plan

1. Hardware Acceleration Technologies
2. Software Performance Engineering
3. A Case Study: Matrix Multiplication
4. Course Outline
Why is Performance Important?

- **Acceptable response time** (Anti-lock break system, Mpeg decoder, Google Search, etc.)
- **Ability to scale** (from hundred to millions of users/documents/data)
- **Use less power / resource** (viability of cell phones dictated by battery life, etc.)
Improving Performance is Hard

- **Knowing that there is a performance problem:** complexity estimates, performance analysis software tools, read the generated assembly code, scalability testing, comparisons to similar programs, experience and curiosity!

- **Establishing the leading cause of the problem:** examine the algorithm, the data structures, the data layout; understand the programming environment and architecture.

- **Eliminating the performance problem:** (Re-)design the algorithm, data structures and data layout, write programs close to the metal (C/C++), adhere to software engineering principles (simplicity, modularity, portability)

- **Golden rule:** Be reactive, not proactive!
Remember that Picture!

**Capacity**

- **CPU Registers**
  - 100s Bytes
  - 300 – 500 ps (0.3-0.5 ns)

- **L1 and L2 Cache**
  - 10s-100s K Bytes
  - ~1 ns - ~10 ns
  - $1000s/ GByte

- **Main Memory**
  - G Bytes
  - 80ns- 200ns
  - ~ $100/ GByte

- **Disk**
  - 10s T Bytes, 10 ms
  - (10,000,000 ns)
  - ~ $1 / GByte

- **Tape**
  - infinite sec-min
  - ~$1 / GByte

**Access Time**

- **L1 Cache**
  - Blocks
  - 10s-100s K Bytes
  - ~1 ns - ~10 ns
  - $1000s/ GByte

- **L2 Cache**
  - Blocks
  - cache cntl
  - 32-64 bytes

- **Memory**
  - Pages
  - cache cntl
  - 64-128 bytes

- **Disk**
  - Files
  - OS
  - 4K-8K bytes

- **Tape**
  - user/operator
  - Mbytes

**Cost**

- **Staging Xfer Unit**
  - prog./compiler
  - 1-8 bytes

- **Upper Level**
  - faster

- **Lower Level**
  - Larger

---

(Moreno Maza)
Plan

1. Hardware Acceleration Technologies
2. Software Performance Engineering
3. A Case Study: Matrix Multiplication
4. Course Outline
Matrix-Matrix Multiplication (MMM)

```
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 multiple is a fundamental operation in many computations: video encoding, weather simulation, computer graphics, etc.
- The following case was studied by Prof. Saman Amarasinghe (MIT).
- Application developers in the Industry write that kind of code (two years of personal experience with student internships at the Université of Lille, France).
I’d like my matrix representation to be:

- Object oriented
- Immutable
- Represent both integers and doubles
**Value**: A class for matrix coefficients

**Matrix**: A class of matrices

**MatrixRow**: An abstract class for matrix rows

**DoubleRow**: A concrete class for matrix rows

**MatrixMultiply**: A class for testing matrix multiplication
public class Value {
    final MatrixType type;
    final int iVal;
    final double dVal;
    Value(int i) --
    int getInt() throws Exception --

    Value(double d) {
        type = MatrixType.FLOATING_POINT;
        dVal = d;
        iVal = 0;
    }
    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];
        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;
    }
}
MMM Java Code: Performance

- It took almost 5 hours to multiply two 1024x1024 matrices on a PC with a single-core running at 3.15 GHz, thus providing \(3.15 \times 10^9\) cycles / second.

- Each inner loop iteration performs 1 multiply, 1 add, 3 index updates, and 1 branch check, so 6 ops. Total:

  \[
  6 \times 1024^3 = 6,442,450,944. 
  \]

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

- How can we improve performance?
MMM Java Code: Profiling

- Find out where you are spending your time
- Lot of interesting information (time spend, cumulative time spend, number of calls, etc.)
- If 90% time is in one routine, inefficiencies in the rest of the program don’t matter.
- Are the hot-spots really doing what you expect them to do?
### MMM Java Code: Profiling 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>30,740</td>
<td>63,540</td>
</tr>
<tr>
<td>Matrix.update(int, int, Value)</td>
<td>11,535</td>
<td>31,630</td>
<td>32,610</td>
</tr>
<tr>
<td>MatrixMultiply.testMM(int, int, int)</td>
<td>1</td>
<td>1,790</td>
<td>1,790</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>172,420</td>
</tr>
<tr>
<td>Main.main(String[])</td>
<td>1</td>
<td>-</td>
<td>-</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.toToString()</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>
To update one coefficient in $A$ we copy the whole row!

This implies a huge memory footprint (allocation, de-allocation).

Can we do better?
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);
    }
}

(Moreno Maza)
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]);
    }
}
### MMM Java Code: Performance (Stage 2)

<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
## MMM Java Code: Profiling 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.get(int)</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>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>-</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>
Method call overhead:
- Each method call needs to look-up the object type
- Dynamic dispatch is an address lookup + an indirect branch

Indirect branches are costly:
- Modern microprocessors are deeply pipelined (12 pipeline stages in core 2 duo, 20 in Pentium 4)
- Hence the proc. needs to be able to keep fetching next instructions before executing them!
- For a direct branch, the target address is known
- For an indirect branch, the proc. needs to wait until address fetch completes

Can we do better?
A Case Study: Matrix Multiplication

MMM Java Code: The DoubleMatrix Class

```java
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);
    }
}
```

(Moreno Maza)
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];
    }
}
<table>
<thead>
<tr>
<th></th>
<th>Immutable</th>
<th>Mutable</th>
<th>Double Only</th>
</tr>
</thead>
<tbody>
<tr>
<td>time (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>

**Performance**

- `Immutable`: 219.7x slower than `Mutable`
- `Double Only`: 2.4x slower than `Immutable`
### MMM Java Code: Profiling 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>
### A Case Study: Matrix Multiplication

### MMM Java Code: Performance from Stage 1 to 3

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>java.lang.Double.&lt;init&gt;(double)</code></td>
</tr>
<tr>
<td><code>DoubleRow.&lt;init&gt;(int)</code></td>
</tr>
<tr>
<td><code>DoubleRow.update(int, Value)</code></td>
</tr>
<tr>
<td><code>Matrix.update(int, int, Value)</code></td>
</tr>
<tr>
<td><code>MatrixMultiply.testMM(int, int, int)</code></td>
</tr>
<tr>
<td><code>DoubleRow.get(int)</code></td>
</tr>
<tr>
<td><code>Matrix.get(int, int)</code></td>
</tr>
<tr>
<td><code>Value.getDouble()</code></td>
</tr>
<tr>
<td><code>Value.&lt;init&gt;(double)</code></td>
</tr>
<tr>
<td><code>DoubleRow.&lt;init&gt;(Double[], int)</code></td>
</tr>
<tr>
<td><code>MatrixRow.&lt;init&gt;()</code></td>
</tr>
<tr>
<td><code>Matrix.&lt;init&gt;(MatrixRow[], MatrixType, int, int)</code></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><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, Value)</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>
MMM Java Code: Issues with Objects

- Memory fragmentation:
  - Objects are allocated independently all over the memory
  - If contiguous in memory, then getting to the next is just an index increment!

- Method call overhead:
  - Method calls are expensive!
  - It is hard to optimize the loop body because of the method call.
A Case Study: Matrix Multiplication

MMM Code: No objects!

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();
### MMM Java Code: Performance (Stage 4)

<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>
</tbody>
</table>

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

| Cycles/OP | 8,358 | 38 | 16 | 7 |
MMM Code: from Java to C

- In Java:
  - Memory bound checks
  - bytecode compilation and dynamic compilation (fast compilation, no time to generate the best code)

- In C:
  - No memory bound checks
  - Intel C compiler compiles the program directly into x86 assembly code.
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);
    return timeTaken;
}
<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
- 2.2x
- 2.4x
- 2.1x

<table>
<thead>
<tr>
<th>Cycles/OP</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8,358</td>
<td>38</td>
<td>16</td>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>
Modern hardware counts **events**

- **CPI**  Clock cycles Per Instruction: the number of clock cycles that happen when an instruction is being executed. With pipelining we can improve the CPI by exploiting instruction level parallelism
- **L1 and L2 Cache Miss Rate.**
- **Instructions Retired:** In the event of a misprediction, instructions that were scheduled to execute along the mispredicted path must be canceled.
A Case Study: Matrix Multiplication

Remember that Picture!

**Capacity**

- **CPU Registers**
  - 100s Bytes
  - 300 – 500 ps (0.3-0.5 ns)

- **L1 and L2 Cache**
  - 10s-100s K Bytes
  - ~1 ns - ~10 ns
  - $1000s/ GByte

- **Main Memory**
  - G Bytes
  - 80ns- 200ns
  - ~ $100/ GByte

- **Disk**
  - 10s T Bytes, 10 ms
  - (10,000,000 ns)
  - ~ $1 / GByte

- **Tape**
  - infinite sec-min
  - ~$1 / GByte

**Access Time**

- **Registers**
  - Instr. Operands
  - faster

- **Upper Level**
  - Blocks
  - Blocks

- **Lower Level**
  - Pages
  - Files

**Cost**

- **Staging Xfer Unit**
  - prog./compiler
  - 1-8 bytes

  - cache cntl
  - 32-64 bytes

  - cache cntl
  - 64-128 bytes

  - OS
  - 4K-8K bytes

  - user/operator
  - Mbytes

  - ~$1 / GByte

(Moreno Maza)
Contiguous accesses are better:

- Data fetch as cache line (Core 2 Duo 64 byte L2 Cache line)
- With contiguous data, a single cache fetch supports 8 reads of doubles.
- Transposing the matrix C should reduce L1 cache misses!
#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);
### MMM C Code: Performance (Stage 6)

<table>
<thead>
<tr>
<th></th>
<th>Immutable ms</th>
<th>Mutable ms</th>
<th>Double Only ms</th>
<th>No Objects ms</th>
<th>In C ms</th>
<th>Transposed ms</th>
</tr>
</thead>
<tbody>
<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>
</tr>
</tbody>
</table>

**Cycles/OP**

|               | 8,358 | 38 | 16 | 7 | 4 | 1 |
### MMM C Code: Performance (Stage 6)

<table>
<thead>
<tr>
<th>In C</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></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>

- **CPI**: Cycle Per Instruction
- **L1 Miss Rate**: Miss Rate for Level 1 Cache
- **L2 Miss Rate**: Miss Rate for Level 2 Cache
The memory system dilemma:
- Small amount of memory implies fast access
- Large amount of memory implies slow access

Workarounds:
- Programmer stores most probable accesses in caches
- Hardware heuristics determine what will be in each cache and when

Computing a $32 \times 32$-block of $A$, so computing again 1024 coefficients: 1024 accesses in $A$, $384 \times 32$ in $B$ and $32 \times 384$ in $C$. Total = 25,600.

The iteration space is traversed so as to reduce memory accesses.
A Case Study: Matrix Multiplication

MMM C Code: Blocking

```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);
```

(Moreno Maza)
### A Case Study: Matrix Multiplication

#### MMM C Code: Performance (Stage 7)

<table>
<thead>
<tr>
<th>Immutable ms</th>
<th>Mutable ms</th>
<th>Double Only 32,800</th>
<th>No Objects 15,306</th>
<th>In C 7,530</th>
<th>Transposed 2,275</th>
<th>Tiled 1,388</th>
</tr>
</thead>
</table>

219.7x

522x

1117x

2271x

7514x

12316x

---

(Moreno Maza)
### MMM C Code: Performance (Stage 7)

<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>0.00</td>
<td>39%</td>
<td>18,044,811,264</td>
</tr>
</tbody>
</table>

- In C: 5x in L1, 2x in L2
- Transposed: 3x in L1, 8x in L2
- Tiled: No L1 and L2 misses

(Moreno Maza)
MMM C Code: Instruction Level Optimizations

- Modern processors have many performance tricks:
  - Instruction Level Parallelism
  - MMX/SSE Instructions (do the same operation on multiple contiguous data at the same time)
  - Prefetching of data
- Nudge the Compiler:
  - Removed any perceived dependences
  - Bound most constant variables to the constant
  - Possible use of compiler pragma’s
- Play with the compiler flags (vector reporting, generate assembly) and tweak the program until the compiler is happy!
- Or use tuned libraries (BLAS).
#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);
## MMM C Code: Performance (Stage 8)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Immutable (ms)</th>
<th>Mutable (ms)</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></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
- 2.2x
- 2.4x
- 1.7x
- 3.4x
- 2.1x
- 2.8x
- 2.7x

- 522x
- 1117x
- 2271x
- 7514x
- 12316x
- 33453x
### MMM C Code: Performance (Stage 8)

<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>5x</td>
<td>2x</td>
<td>43%</td>
<td>13,137,280,000</td>
</tr>
<tr>
<td>Transposed</td>
<td>1.13</td>
<td>3x</td>
<td>8x</td>
<td>50%</td>
<td>13,001,486,336</td>
</tr>
<tr>
<td>Tiled</td>
<td>0.49</td>
<td>1/2x</td>
<td>1/4x</td>
<td>39%</td>
<td>18,044,811,264</td>
</tr>
<tr>
<td>Vectorized</td>
<td>0.9</td>
<td>3x</td>
<td>4x</td>
<td>88%</td>
<td>3,698,018,048</td>
</tr>
<tr>
<td>BLAS</td>
<td>0.37</td>
<td>0.24</td>
<td>0.02</td>
<td>78%</td>
<td>3,833,811,968</td>
</tr>
</tbody>
</table>
Plan

1. Hardware Acceleration Technologies
2. Software Performance Engineering
3. A Case Study: Matrix Multiplication

4. Course Outline