Introduction to Software Performannce Engineering

Marc Moreno Maza

University of Western Ontario, London, Ontario (Canada)

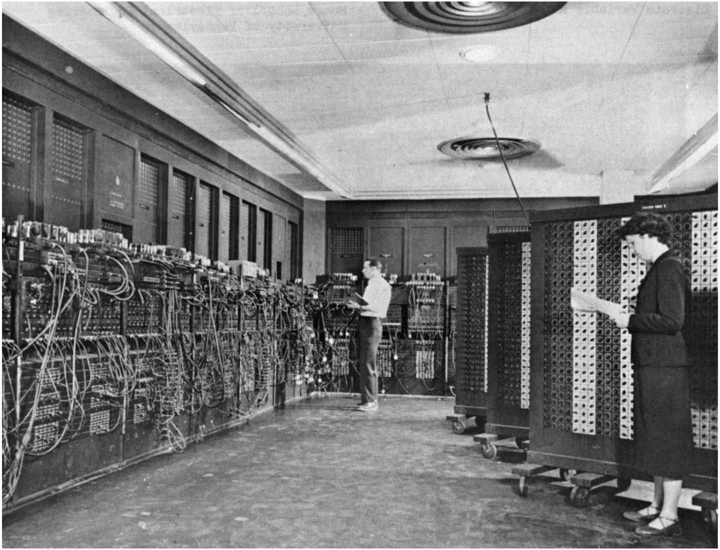
CS 433 - CS 9624

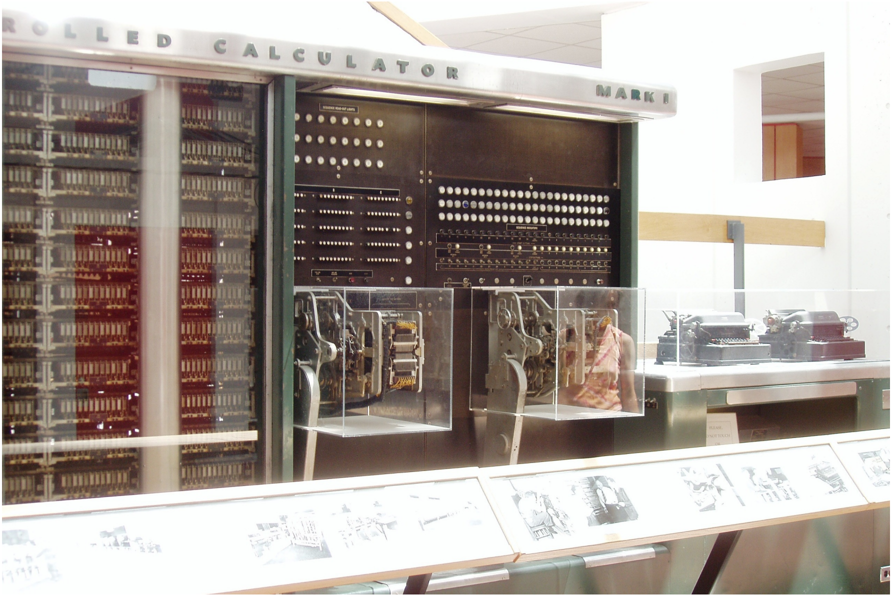
Plan

Plan

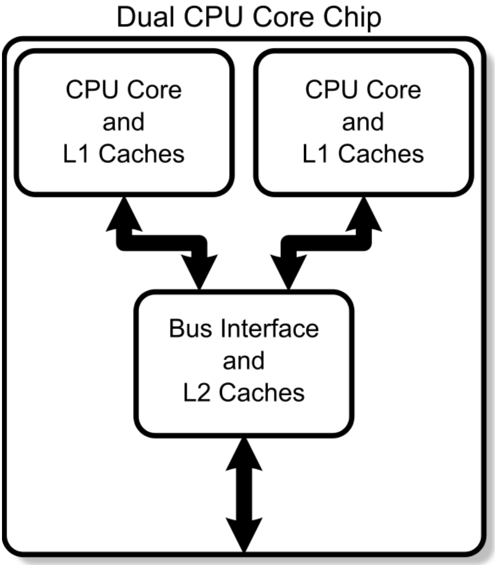

intel®
pentium® 4
2.4 GHZ/512/400/1.5V
SL65R COSTA RICA
3212A185-0282


intel pentium
intel pentium PROCESSOR
intel pentium PROCESSOR
intel pentium
intel pentium
intel pentium
intel pentium
intel pentium
intel pentium
intel pentium w/ MMX™ tech

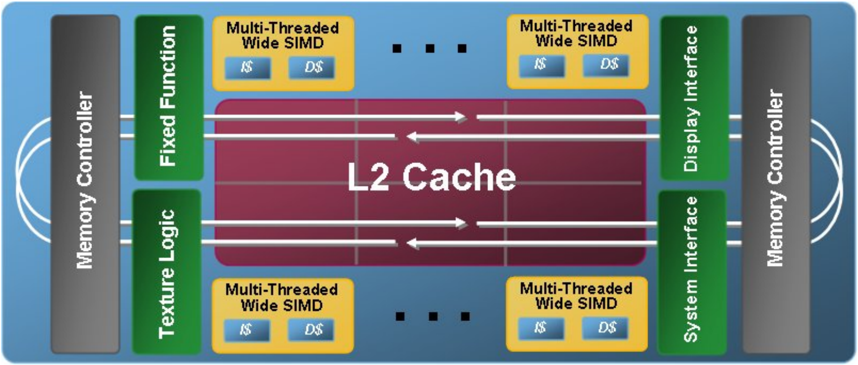


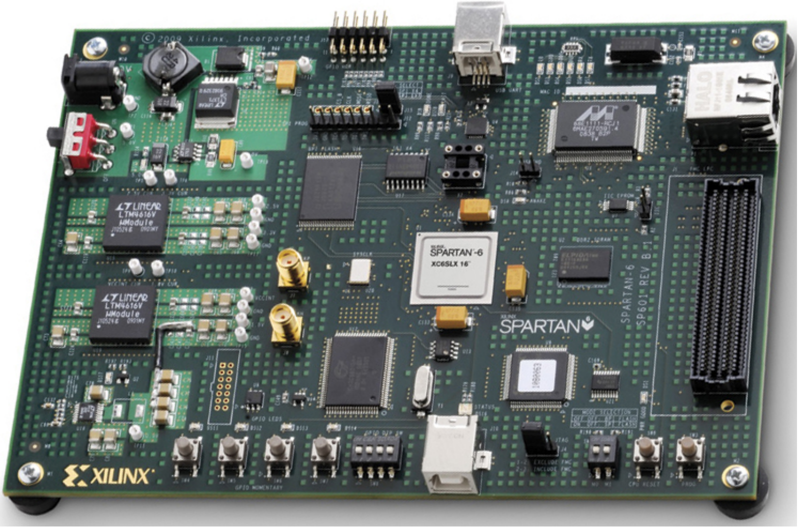

ROLLED CALCULATOR
MARK I


INTEL©'05 E6750
INTEL® CORE™2 DUO
SLA9V MALAY
2.66GHZ/4M/1333/06
L726A863

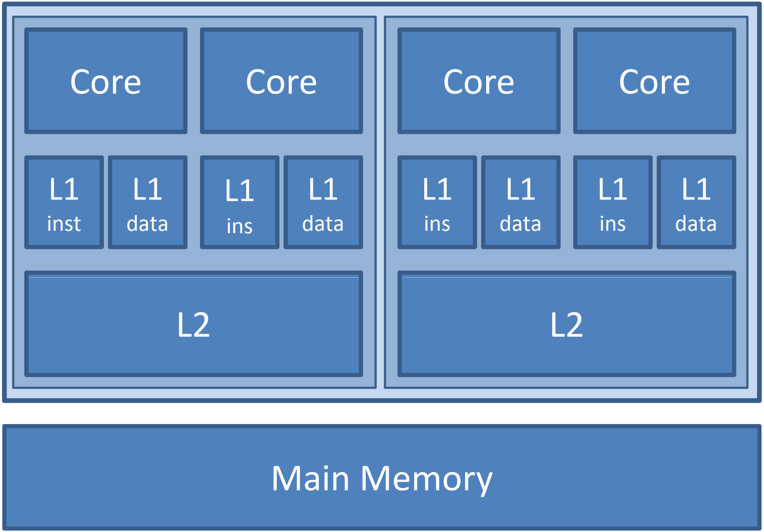

Dual CPU Core Chip
CPU Core
and
L1 Caches
CPU Core
and
L1 Caches
Bus Interface
and
L2 Caches


94V-0
MADE IN CHINA
NVIDIA
RoHS
COMPLIANT


Memory Controller
Fixed Function
Texture Logic
Multi-Threaded Wide SIMD
I$
D$
Multi-Threaded Wide SIMD
I$
D$
L2 Cache
Multi-Threaded Wide SIMD
I$
D$
Multi-Threaded Wide SIMD
I$
D$
Display Interface
System Interface
Memory Controller


© 2009 Xilinx, Incorporated
LINEAR
LTM4616V
μModule
SPARTAN-6
XC6SLX 16
SPARTAN
SPARTAN-6
SP601 REV B 1
XILINX


INTEL © '05 Q6600
INTEL® CORE™2 QUAD
SLACR MALAY
2.40GHZ/8M/1066/05A
L738B526


Core
Core
Core
Core
L1
inst
L1
data
L1
ins
L1
data
L1
ins
L1
data
L1
ins
L1
data
L2
L2
Main Memory

L1 Data Cache			
Size	Line Size	Latency	Associativty
32 KB	64 bytes	3 cycles	8-way
L1 Instruction Cache			
Size	Line Size	Latency	Associativty
32 KB	64 bytes	3 cycles	8-way
L2 Cache			
Size	Line Size	Latency	Associativty
6 MB	64 bytes	14 cycles	24-way

Capacity *Access Time* *Cost*	Level	Transfer	*Staging* *Xfer Unit*
CPU Registers 100s Bytes 300 – 500 ps (0.3-0.5 ns)	**Registers**	Instr. Operands	prog./compiler 1-8 bytes
L1 and L2 Cache 10s-100s K Bytes ~1 ns - ~10 ns $1000s/ GByte	**L1 Cache**	Blocks	cache cntl 32-64 bytes
	L2 Cache	Blocks	cache cntl 64-128 bytes
Main Memory G Bytes 80ns- 200ns ~ $100/ GByte	**Memory**	Pages	OS 4K-8K bytes
Disk 10s T Bytes, 10 ms (10,000,000 ns) ~ $1 / GByte	**Disk**	Files	user/operator Mbytes
Tape infinite sec-min ~$1 / GByte	**Tape**		

Upper Level

faster ↑ ↓ Larger

Lower Level

Plan

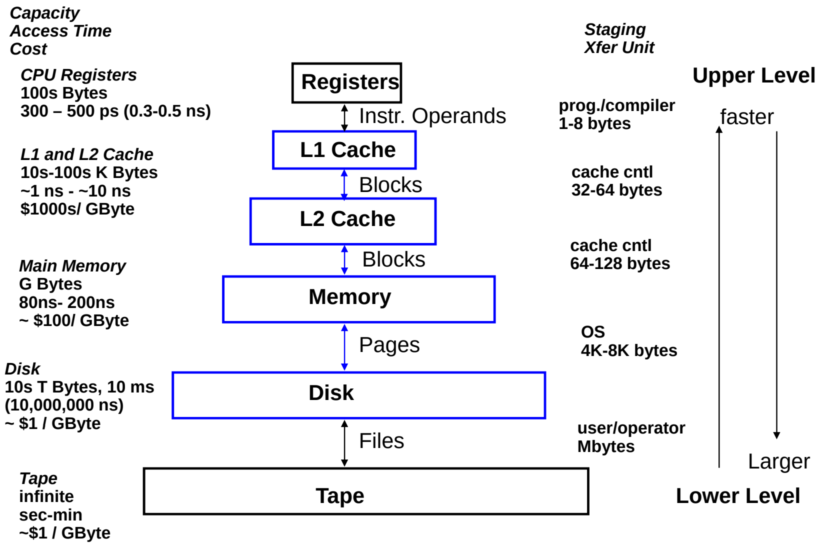
Why is Performance Important?

- 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!

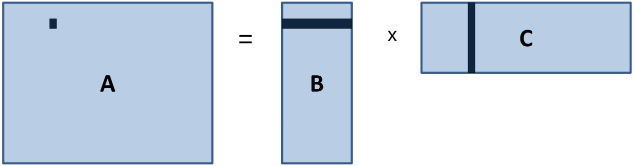
Remember that Picture!



Plan

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

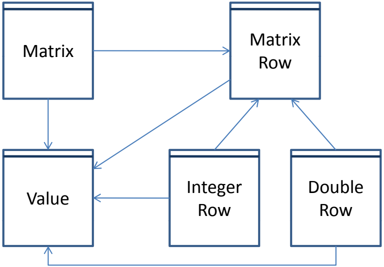
Matrix-Matrix Multiplication (MMM)




```
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).

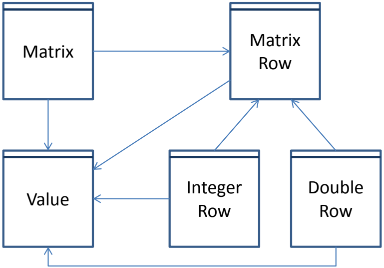
MMM Java Code (1/2)



I'd like my matrix representation to be:

- Object oriented
- Immutable
- Represent both integers and doubles

MMM Java Code (2/2)



`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

MMM Java Code: Value Class

```
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();
                }
    }
```

MMM Java Code: Matrix Class (1/2)

```
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);
                 }
        }
        --
        --
}
```

MMM Java Code: Matrix Class (2/2)

```
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);
        }
}
```

MMM Java Code: Abstract Class `MatrixRow`

```
public abstract class MatrixRow {
       abstract Value get(int col) throws Exception;
       abstract public MatrixRow update(int col, Value val) throws
                                                        Exception;
}
```

MMM Java Code: DoubleRow Class

```
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]);
       }
```

MMM Java Code: MatrixMultiply Class

```
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 3.15×10^9 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?

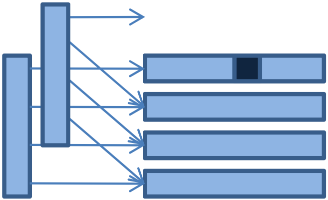
MMM Java Code: Profiling

- 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

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

MMM Java Code: Issues with Immutability



- To update one coefficient in `A` we copy the whole row!
- This implies ah huge memory footprint (allocation, de-allocation).
- Can we do better?

MMM Java Code: Matrix Class Revisited

```
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);
        }
}
```

MMM Java Code: DoubleRow Class Revisited

```
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)

	Immutable	Mutable
ms	17,094,152	77,826
	219.7x	
Cycles/OP	8,358	38

MMM Java Code: Profiling Data

Method	Num Calls	Method Time	Cumulative Times
MatrixMultiply.testMM(int, int, int)	1	40,076	171,425
Value.getDouble()	1,958,974	36,791	36,791
Matrix.get(int, int)	1,469,230	27,725	64,624
DoubleRow.get(int)	1,692,307	25,343	36,900
Value.<init>(double)	1,958,974	15,501	15,501
Matrix.set(int, int, Value)	489,743	13,032	35,220
DoubleRow.set(int, Value)	489,743	12,932	22,188
DoubleRow.<init>(int)	372	21	23
MatrixRow.<init>()	372	2	2
Matrix.<init>(int, int, MatrixType)	3	2	25
Main.<init>()	1	1	171,426
java.io.PrintStream.println(String)	1	-	-
java.lang.StringBuilder.append(int)	7	-	-
java.lang.System.nanoTime()	1	-	-
Main.main(String[ ])	1	-	171,426
MatrixType.<clinit>()	1	-	-
java.lang.StringBuilder.append(String)	7	-	-
java.lang.StringBuilder.<init>()	1	-	-
MatrixType.<init>(String, int)	2	-	-
java.lang.StringBuilder.toString()	1	-	-
java.lang.Enum.<init>(String, int)	2	-	-
<ROOT>.<ROOT>	-	-	171,426
java.lang.Object.<init>()	19,592,818	-	-

MMM Java Code: Issues with Dynamic Dispatch

- 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?

MMM Java Code: The DoubleMatrix Class

```
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);
        }
}
```

MMM Java Code: The Class DoubleRow Revisited

```
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];
        }
}
```

MMM Java Code: Performance (Stage 3)

	Immutable	Mutable	Double Only	
ms	17,094,152	77,826	32,800	
	219.7x			
			2.4x	
	219.7x			
		522x		
Cycles/OP	8,358	38	16	

MMM Java Code: Profiling Data

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

MMM Java Code: Performance from Stage 1 to 3

Immutable

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

Mutable

Method	Num Calls	Method Time	Cumulative Times
MatrixMultiply.testMM(int, int, int)	1	40,076	171,425
Value.getDouble()	1,958,974	36,791	36,791
Matrix.get(int, int)	1,469,230	27,725	64,624
DoubleRow.get(int)	1,469,230	25,343	36,900
Value.<init>(double)	1,958,974	15,501	15,501
Matrix.set(int, int, Value)	489,743	13,032	35,220
DoubleRow.set(int, Value)	489,743	12,932	22,188

Double Only

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

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.

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)

	Immutable	Mutable	Double Only	No Objects
ms	17,094,152	77,826	32,800	15,306
	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.

MMM C Code: Typical (or Naive?) C 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;
```

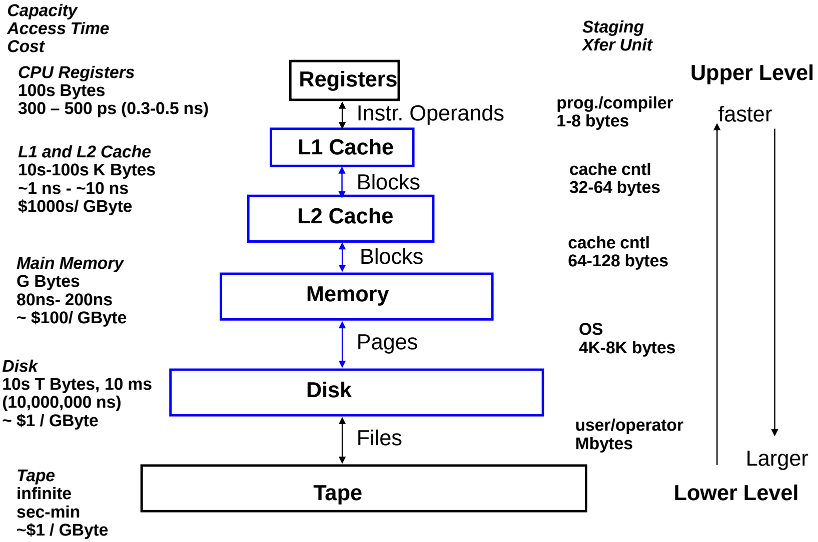
MMM C Code: Performance (Stage 5)

	Immutable	Mutable	Double Only	No Objects	In C
ms	17,094,152	77,826	32,800	15,306	7,530
		219.7x		2.2x	
			2.4x		2.1x
		219.7x			
			522x		
				1117x	
					2271x
Cycles/OP	8,358	38	16	7	4

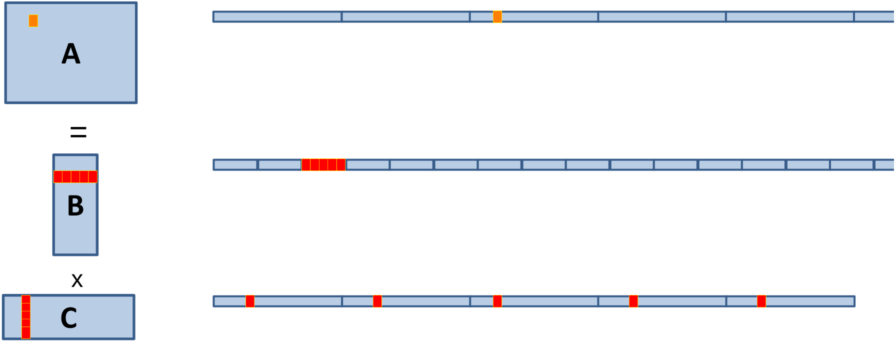
MMM C Code: Profiling with Performance Counters

- 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.

Remember that Picture!



MMM C Code: Issues with Matrix Representation



- Contiguous accesses are better:
  - Data fetch as cache line (Core 2 Duo 64 byte L2 Cache li)
  - With contiguous data, a single cache fetch supports 8 reads of doubles.
  - Transposing the matrix C should reduce L1 cache misses!

MMM C Code: Preprocessing the Data

```
#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);
```

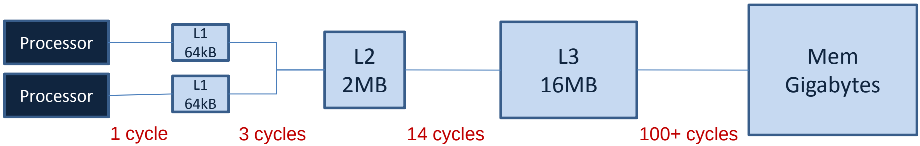
MMM C Code: Performance (Stage 6)

	Immutable	Mutable	Double Only	No Objects	In C	Transposed
ms	17,094,152	77,826	32,800	15,306	7,530	2,275
	219.7x		2.2x		3.4x	
		2.4x		2.1x		
	219.7x					
	522x					
	1117x					
	2271x					
	7514x					
Cycles/OP	8,358	38	16	7	4	1

MMM C Code: Performance (Stage 6)

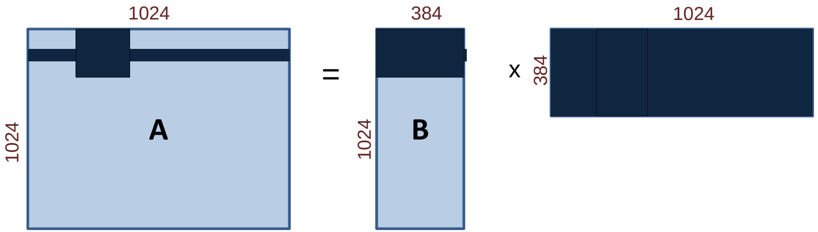
	CPI		L1 Miss Rate		L2 Miss Rate	Percent SSE Instructions	Instructions Retired
In C	4.78		0.24		0.02	43%	13,137,280,000
		5x		2x			
Transposed	1.13		0.15		0.02	50%	13,001,486,336

MMM C Code: Issues with the Memory Hierarchy



- 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

MMM C Code: Blocking for Optimizing Data Reuse



- Naive calculation of a row of `A`, so computing 1024 coefficients: 1024 accesses in `A`, 394 in `B` and $1024 \times 394 = 393,216$ in `C`. Total $= 394,524$.
- Computing a 32×32-block of `A`, so computing again 1024 coefficients: 1024 accesses in `A`, 384×32 in `B` and 32×384 in `C`. Total $= 25,600$.
- The iteration space is traversed so as to reduce memory accesses.

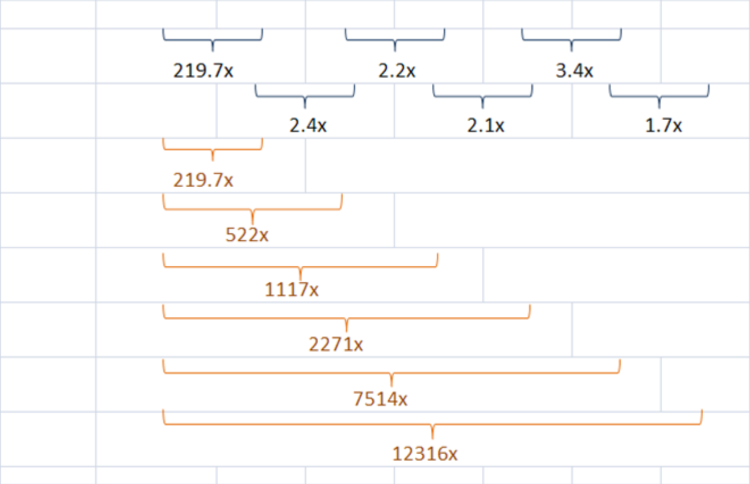
MMM C Code: Blocking

```
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);
```

MMM C Code: Performance (Stage 7)

	Immutable	Mutable	Double Only	No Objects	In C	Transposed	Tiled
ms	17,094,152	77,826	32,800	15,306	7,530	2,275	1,388



MMM C Code: Performance (Stage 7)

	CPI		L1 Miss Rate		L2 Miss Rate		Percent SSE Instructions		Instructions Retired
In C	4.78		0.24		0.02		43%		13,137,280,000
		5x		2x					
Transposed	1.13		0.15		0.02		50%		13,001,486,336
		3x		8x					
Tiled	0.49		0.02		0		39%		18,044,811,264

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).

MMM C Code: Nudge the Compiler

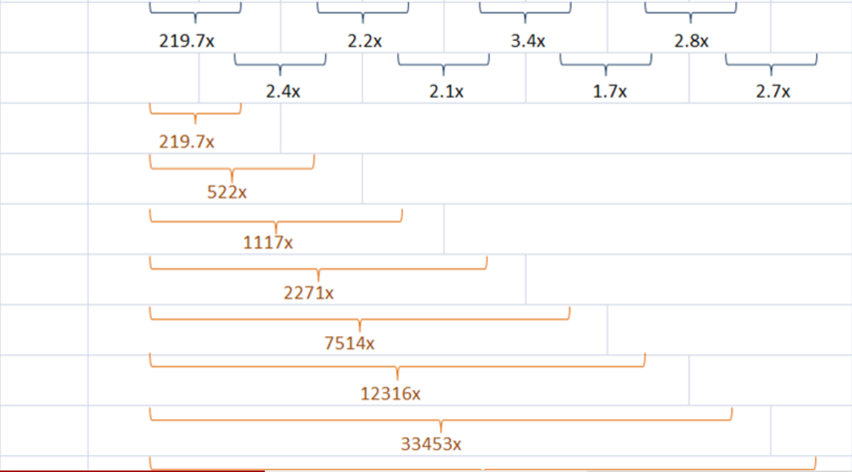
```
#define N 1024
#define BLOCK_X 256
#define BLOCK_Y 1024
#define IND(A, x, y, d) A[(x)*(d)+(y)]

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)

	Immutable	Mutable	Double Only	No Objects	In C	Transposed	Tiled	vectorized	BLAS MxM
ms	17,094,152	77,826	32,800	15,306	7,530	2,275	1,388	511	196



MMM C Code: Performance (Stage 8)

	CPI		L1 Miss Rate		L2 Miss Rate	Percent SSE Instructions	Instructions Retired	
In C	4.78		0.24		0.02	43%	13,137,280,000	
		5x		2x				1x
Transposed	1.13		0.15		0.02	50%	13,001,486,336	
		3x		8x				0.8x
Tiled	0.49		0.02		0	39%	18,044,811,264	
		1/2x		1/4x				5x
Vectorized	0.9		0.07		0	88%	3,698,018,048	
		3x		4x				1x
BLAS	0.37		0.02		0	78%	3,833,811,968	

Plan

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