PERFORMANCE ENGINEERING
OF SOFTWARE SYSTEMS

Matrix Multiply: A Case Study

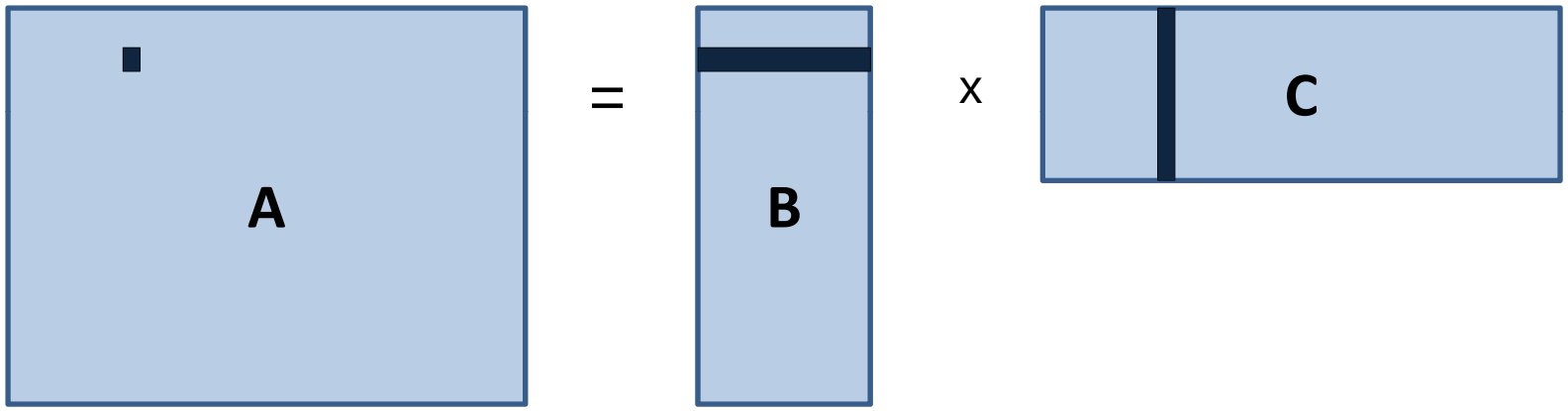
Saman Amarasinghe

Fall 2010

Matrix Multiply

Matrix multiple is a fundamental operation in many computations

- Example: video encoding, weather simulation, computer graphics

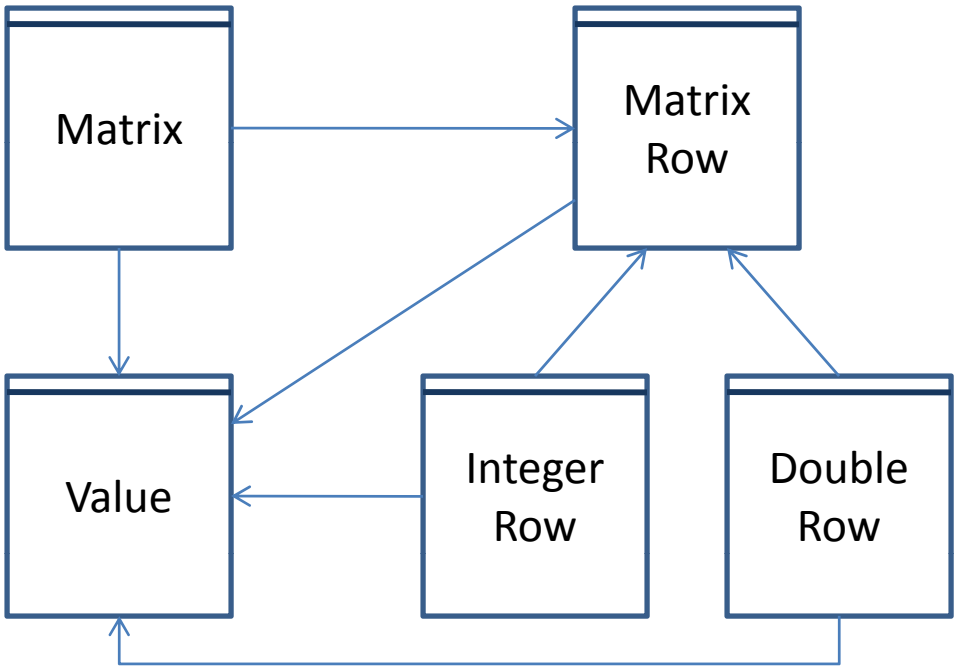



```
for(int i =0; i < x; i++)
   for(int j =0; j < y; j++)
     for(int k=0; k < z; k++)
       A[i][j] += B[i][k]*C[k][j]
```

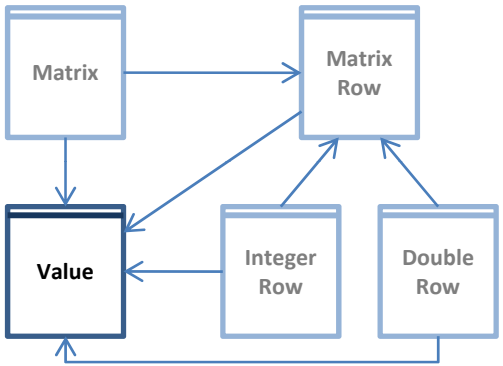
Matrix Representation

I'd like my matrix representation to be

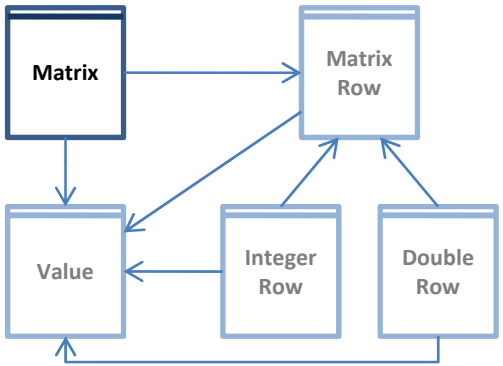
- Object oriented
- Immutable
- Represent both integers and doubles



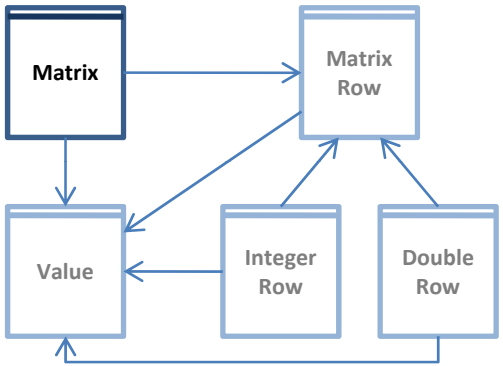
```
public class Value {
  final MatrixType type;
  final int iVal;
  final double dVal;

  Value(int i) ……

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

  int getInt() throws Exception ……

  double getDouble() throws Exception {
      if(type == MatrixType.FLOATING_POINT)
          return dVal;
      else
          throw new Exception();
  }
}
```



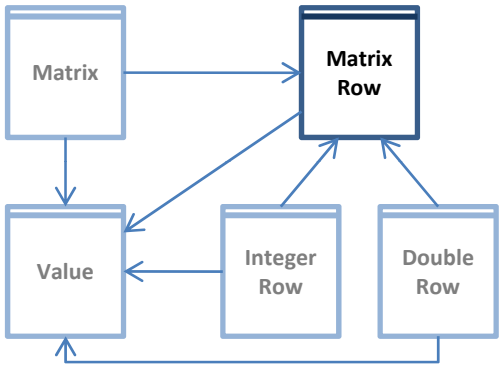
```
public class Matrix {
  final MatrixRow[] rows;
  final int nRows, nColumns;
  final MatrixType type;

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

  ……
  ……

}
```



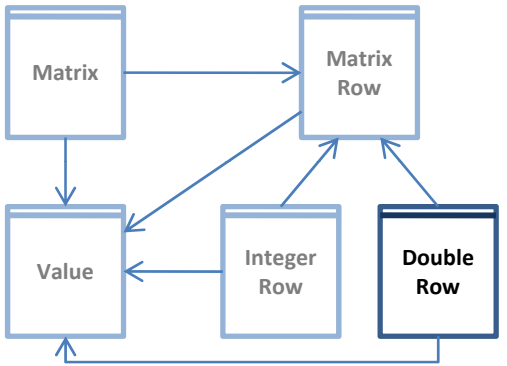
```
public class Matrix {
  ……
  ……


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

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

  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]);
  }
}
```



```
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()*

     } catch(Exception e) {

     }
     long time = System.nanoTime();
     long timeTaken = (time - started);
     System.out.println ("Time:" + timeTaken/1000000 + "ms");
     return timeTaken;
  }
}
```

Performance

	Immutable
ms	17,094,152

1024x1024 matrix multiply

Is the performance good?

It took almost 5 hours to multiply two 1024x1024 matrices

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

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

1,073,741,824 * 6 = 6,442,450,944

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

My PC runs at 3.15 GHz → 3.15×10^9 cycles / second

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

Back of the Envelope Calculation

How can we improve performance?

Profiling

Look deeply in to the program execution

Find out where you are spending your time

- By method
- By line

Lot of interesting information

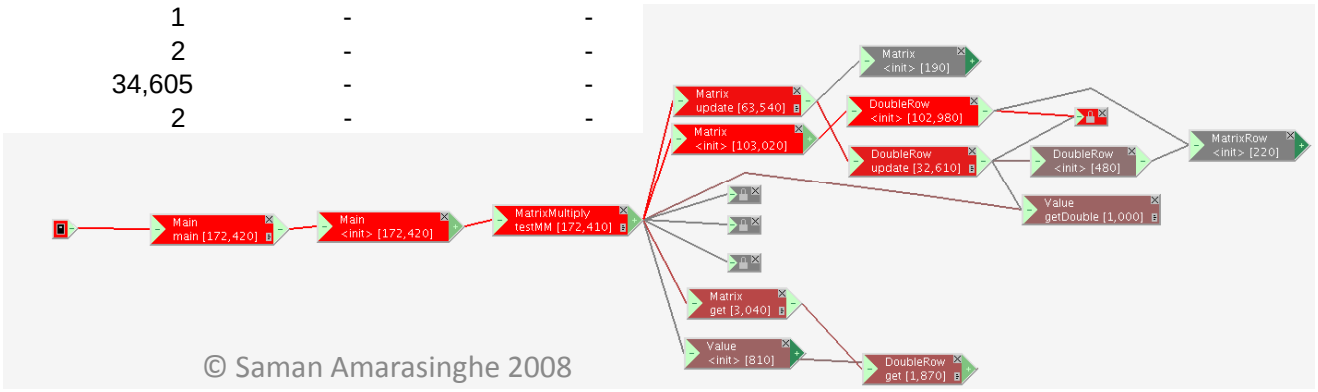
- Time spend
- Cumulative time spend
- Number of invocations
- Etc. etc.

Great way to zero in on what matters – Hotspots

- If 90% time is in one routine, inefficiencies in the rest of the program don't matter
- Also, is the hotspots doing what you expect them to do?

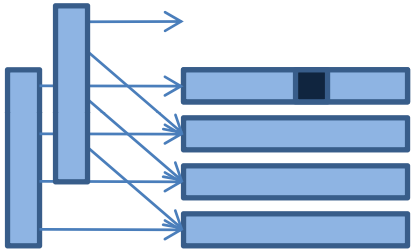
Profile Data

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	-	-



Issues with Immutability

Updating one location → copy of the matrix



2*N copies for each update

N^3 updates → N^4 copies are made.

Copying is costly

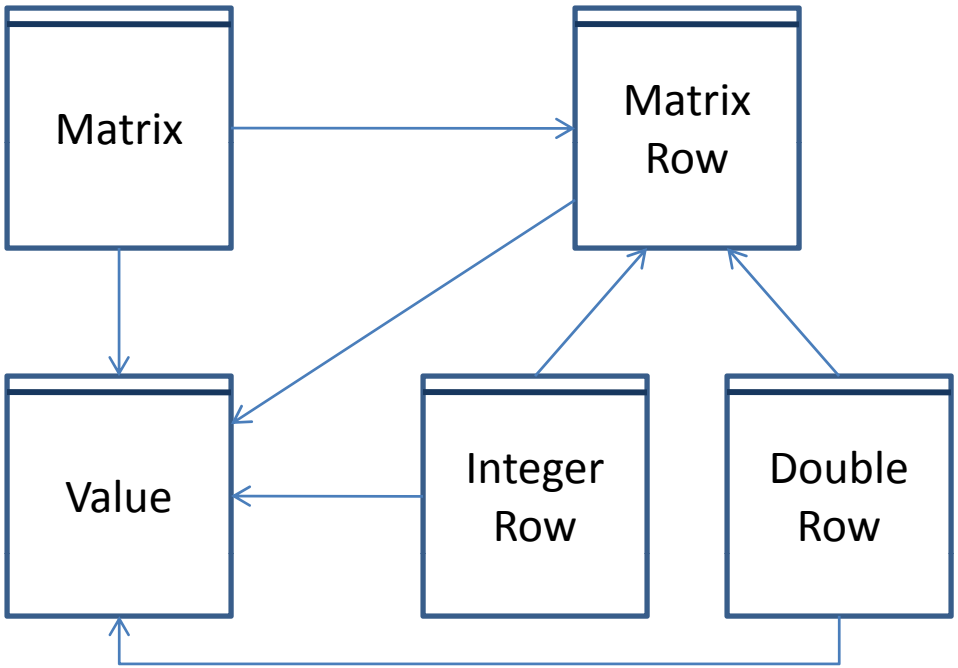
- Cost of making duplicates
- Cost of garbage collecting the freed objects
- Huge memory footprint

Can we do better?

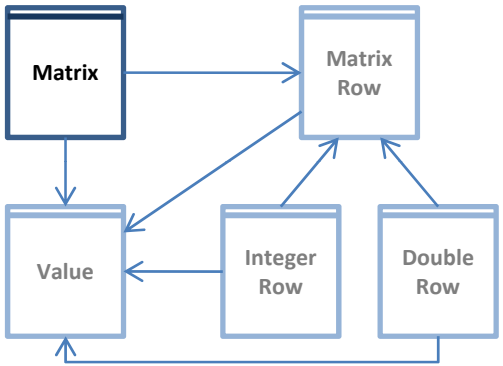
Matrix Representation

I'd like my matrix representation to be

- Object oriented
- ~~Immutable~~
- Represent both integers and doubles

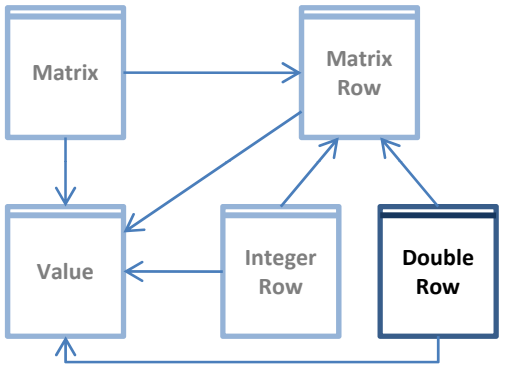


```
public class Matrix {
  MatrixRow[] rows;
  final int nRows, nColumns;
  final MatrixType type;

  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?

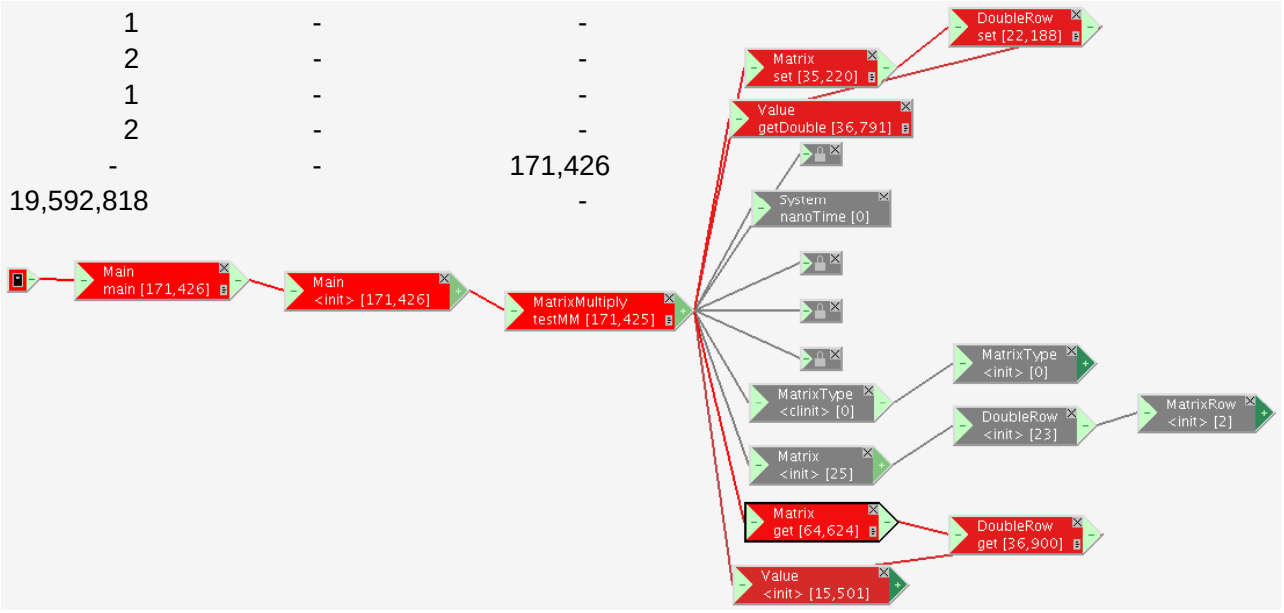


Performance

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

Profile Data

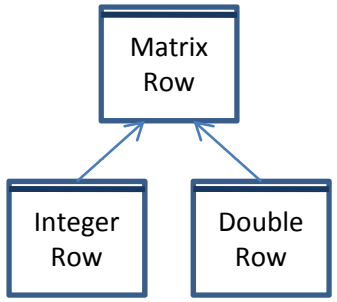
Method	Num Calls	Method Time	Cumulative Times
MatrixMultiply.testMM(int, int, int)	1	40,076	171,425
Value.getDouble()	1,958,974	36,791	36,791
Matrix.get(int, int)	1,469,230	27,725	64,624
DoubleRow.get(int)	1,692,307	25,343	36,900
Value.<init>(double)	1,958,974	15,501	15,501
Matrix.set(int, int, Value)	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		-



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);
   }
 }
```

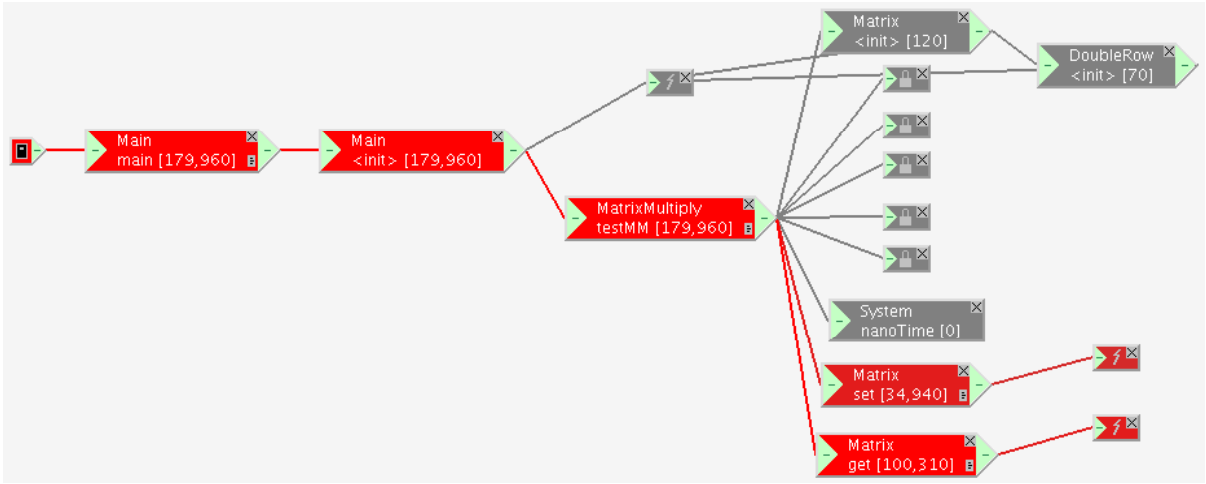
```
public final class DoubleRow  {
  double[] theRow;
  public final int numColumns;

  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

	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	

Profile 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	-	-



Profile Data

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

Issues with Object Oriented

Memory fragmentation

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

Method call overhead

- Method calls are expensive
- Cannot optimize the loop body because of the method call

Matrix Representation

I'd like my matrix representation to be

- ~~Object oriented~~
- ~~Immutable~~
- ~~Represent both integers and~~ doubles

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

long started = System.nanoTime();

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

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

Performance

	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

From Java to C

Java

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

C

- No such thing in C
- Intel C compiler compiles the program directly into x86 assembly

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

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

  for (i = 0; i < x; i++)


  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

	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

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?

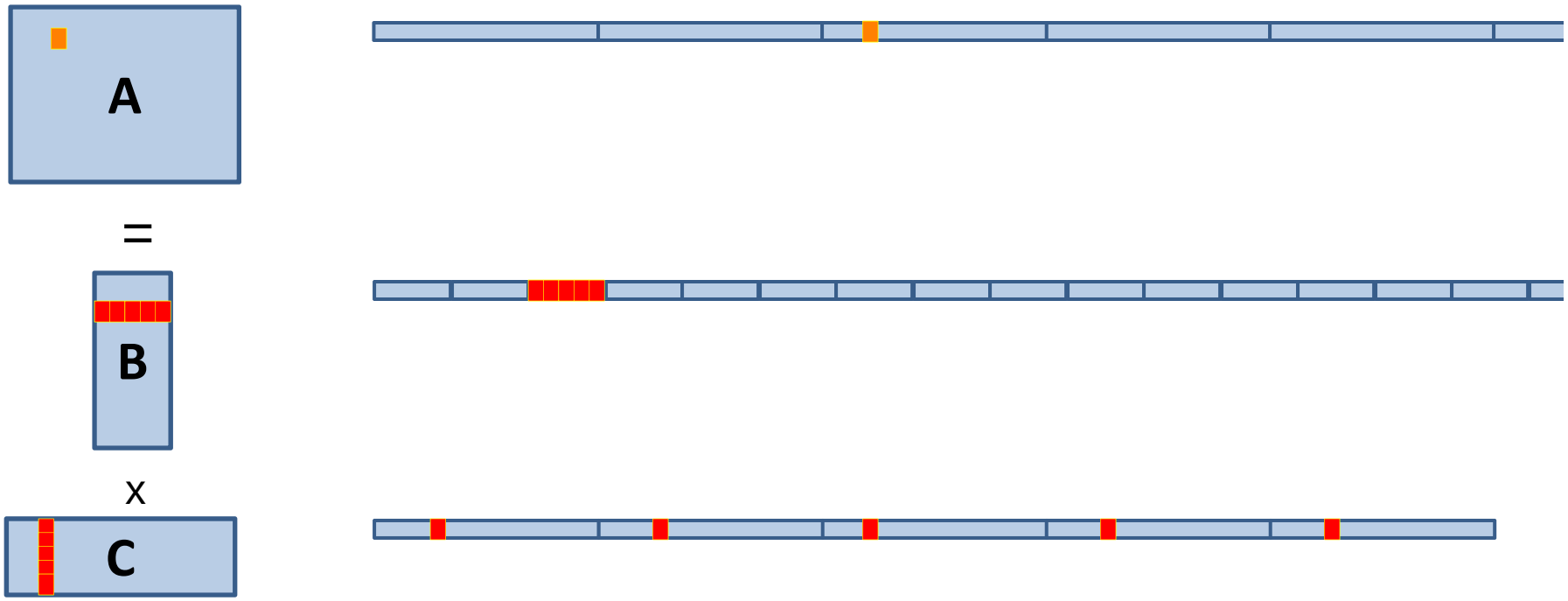
Instructions Retired

- How many instructions got executed

CPI	L1 Miss Rate	L2 Miss Rate	Percent SSE Instructions	Instructions Retired
4.78	0.24	0.02	43%	13,137,280,000

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 → 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

```
  #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

	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

Profile Data

	CPI		L1 Miss Rate		L2 Miss Rate		Percent SSE Instructions		Instructions Retired	
In C	4.78	5x	0.24	2x	0.02		43%		13,137,280,000	1x
Transposed	1.13		0.15		0.02		50%		13,001,486,336	

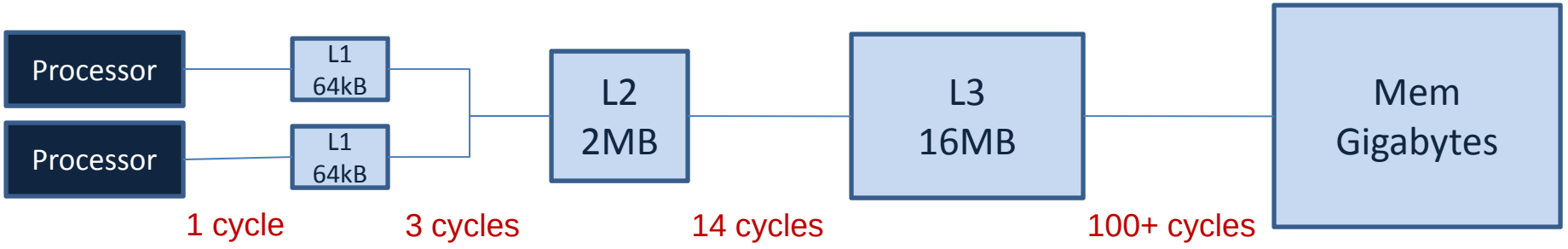
The Memory System

The memory system dilemma

- Small amount of memory → fast access
- Large amount of memory → 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



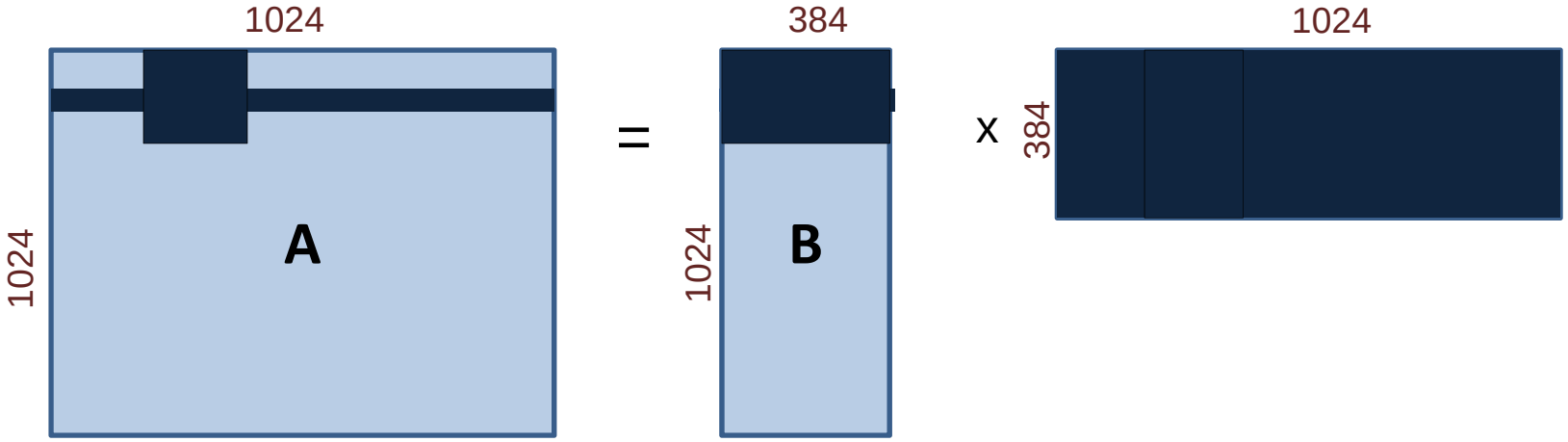
The temperamental cache

- If your access pattern matches heuristics of the hardware → blazingly fast
- Otherwise → 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*1=1024 + B: 384*1=394 + C: 1024*384=393,216 = 394,524
- Blocked Matrix Multiply (32^2 = 1024 values of A)
  - A: 32*32=1024 + B: 384*32 =12,284 + C: 32*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 ≠ a + (b + c)
- Drop/change precision
- Operate within an acceptable error range

…

```
started = read_timestamp_counter();

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

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

Performance

	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
		219.7x		2.2x		3.4x	
			2.4x		2.1x		1.7x
		219.7x					
			522x				
				1117x			
					2271x		
						7514x	
							12316x
Cycles/OP	8,358	38	16	7	4	1	1/2

Profile Data

	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	

Instruction Level Optimizations

Modern processors have many other performance tricks

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

Nudge the Compiler

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

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

……

        started = read_timestamp_counter();

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

        for(j2 = 0; j2 < N; j2 += BLOCK_X)

                    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 ☹

```
;;; for(j2 = 0; j2 < N; j2 += BLOCK_X)
        xorl      %edx, %edx
        xorl      %eax, %eax
        xorps     %xmm0, %xmm0
;;; for(k2 = 0; k2 < N; k2 += BLOCK_Y)
;;; for(i = 0; i < N; i++)
        xorl      %ebx, %ebx
        xorl      %ecx, %ecx
;;; for(j = 0; j < BLOCK_X; j++)
        xorl      %r9d, %r9d
;;; 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);
        movslq    %ecx, %r8
        lea       (%rdx,%rcx), %esi
        movslq    %esi, %rdi
        shlq      $3, %rdi
        movslq    %eax, %rsi
        shlq      $3, %rsi
..B1.13:
        movaps    %xmm0, %xmm2
        movsd     A(%rdi), %xmm1
        xorl      %r10d, %r10d
..B1.14:
        movaps    B(%r10,%r8,8), %xmm3
        mulpd     Cx(%r10,%rsi), %xmm3
        addpd     %xmm3, %xmm1
        movaps    16+B(%r10,%r8,8), %xmm4
        mulpd     16+Cx(%r10,%rsi), %xmm4
        addpd     %xmm4, %xmm2
        movaps    32+B(%r10,%r8,8), %xmm5
        mulpd     32+Cx(%r10,%rsi), %xmm5
        addpd     %xmm5, %xmm1
        movaps    48+B(%r10,%r8,8), %xmm6
        mulpd     48+Cx(%r10,%rsi), %xmm6
        addpd     %xmm6, %xmm2
        movaps    64+B(%r10,%r8,8), %xmm7
        mulpd     64+Cx(%r10,%rsi), %xmm7
        addpd     %xmm7, %xmm1
        movaps    80+B(%r10,%r8,8), %xmm8
        mulpd     80+Cx(%r10,%rsi), %xmm8
        addpd     %xmm8, %xmm2
        movaps    96+B(%r10,%r8,8), %xmm9
        mulpd     96+Cx(%r10,%rsi), %xmm9
        addpd     %xmm9, %xmm1
        movaps    112+B(%r10,%r8,8), %xmm10
        mulpd     112+Cx(%r10,%rsi), %xmm10
        addpd     %xmm10, %xmm2
        addq      $128, %r10
        cmpq      $8192, %r10
        jl        ..B1.14       # Prob 99%
```

Inner loop: SSE instructions

Performance

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

Profile Data

	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	

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

	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
		219.7x		2.2x		3.4x		2.8x	
			2.4x		2.1x		1.7x		2.7x
	219.7x								
	522x								
	1117x								
	2271x								
	7514x								
	12316x								
	33453x								
	87042x								
Cycles/OP	8,358	38	16	7	4	1	1/2	1/5	1/11

Profile Data

	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	

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, Ts, and a portion that can be executed in parallel, Tp. Then for n processors:
- $T(n) = Ts + Tp/n$
- $T(\infty) = Ts$, thus maximum speedup $(Ts + Tp) / Ts$

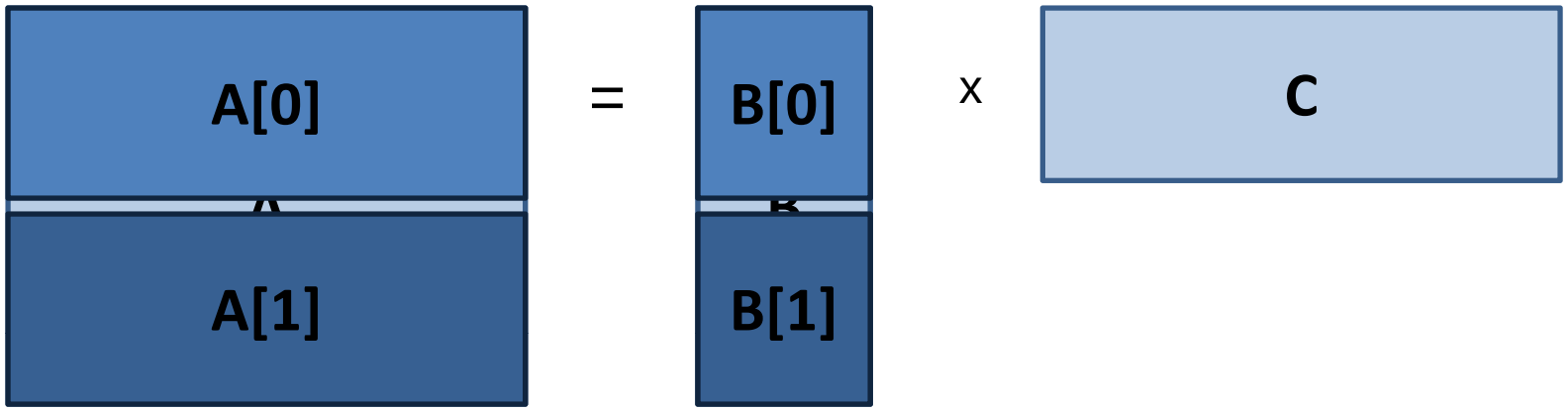
Load Balancing

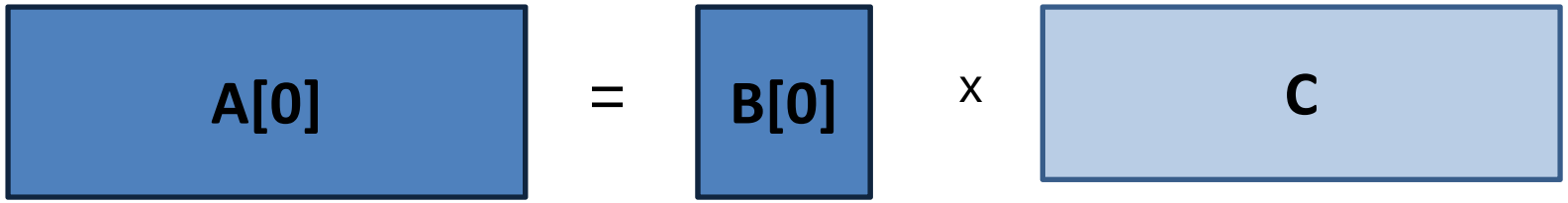
- The work is distributed among processors so that ***all*** processors are kept busy ***all*** of the time.

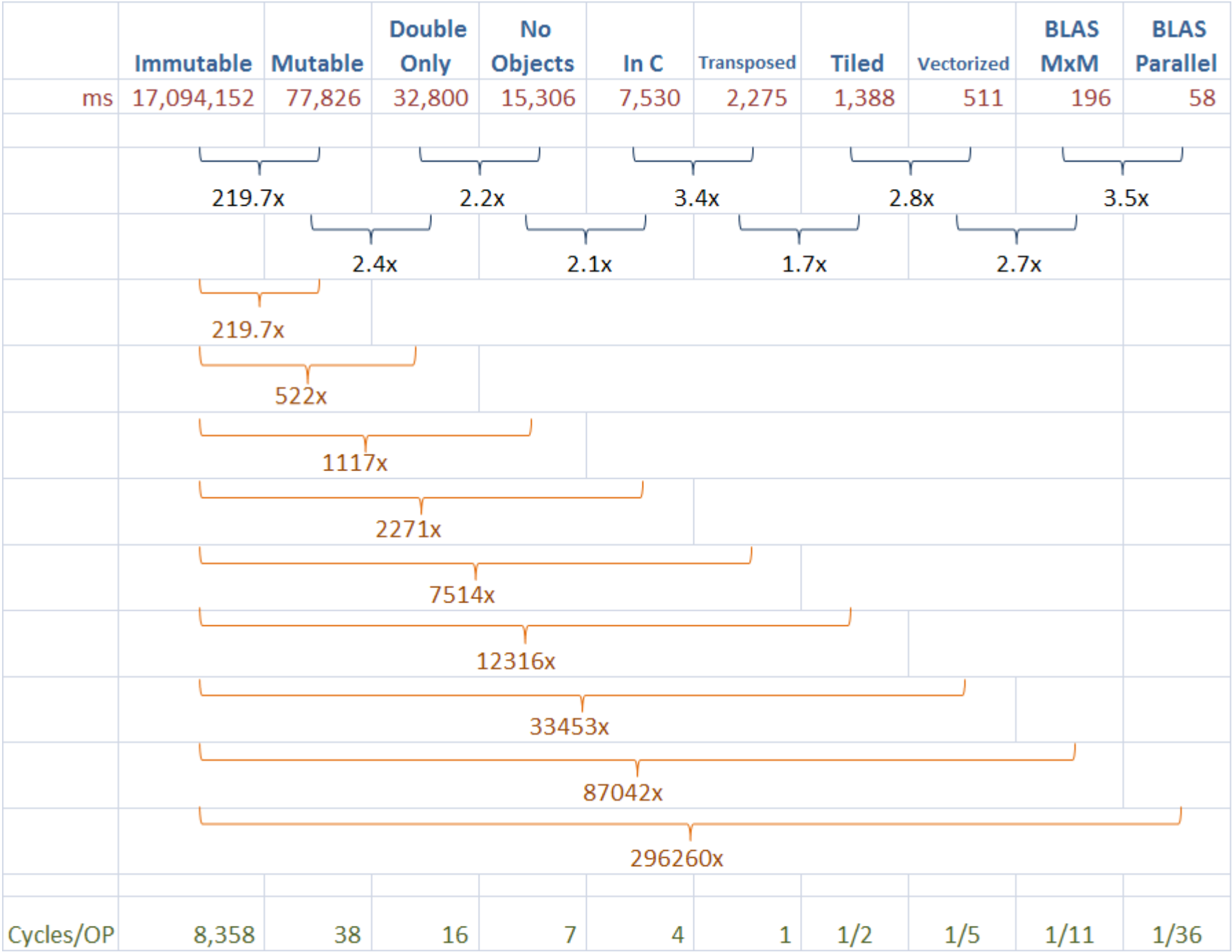
Granularity

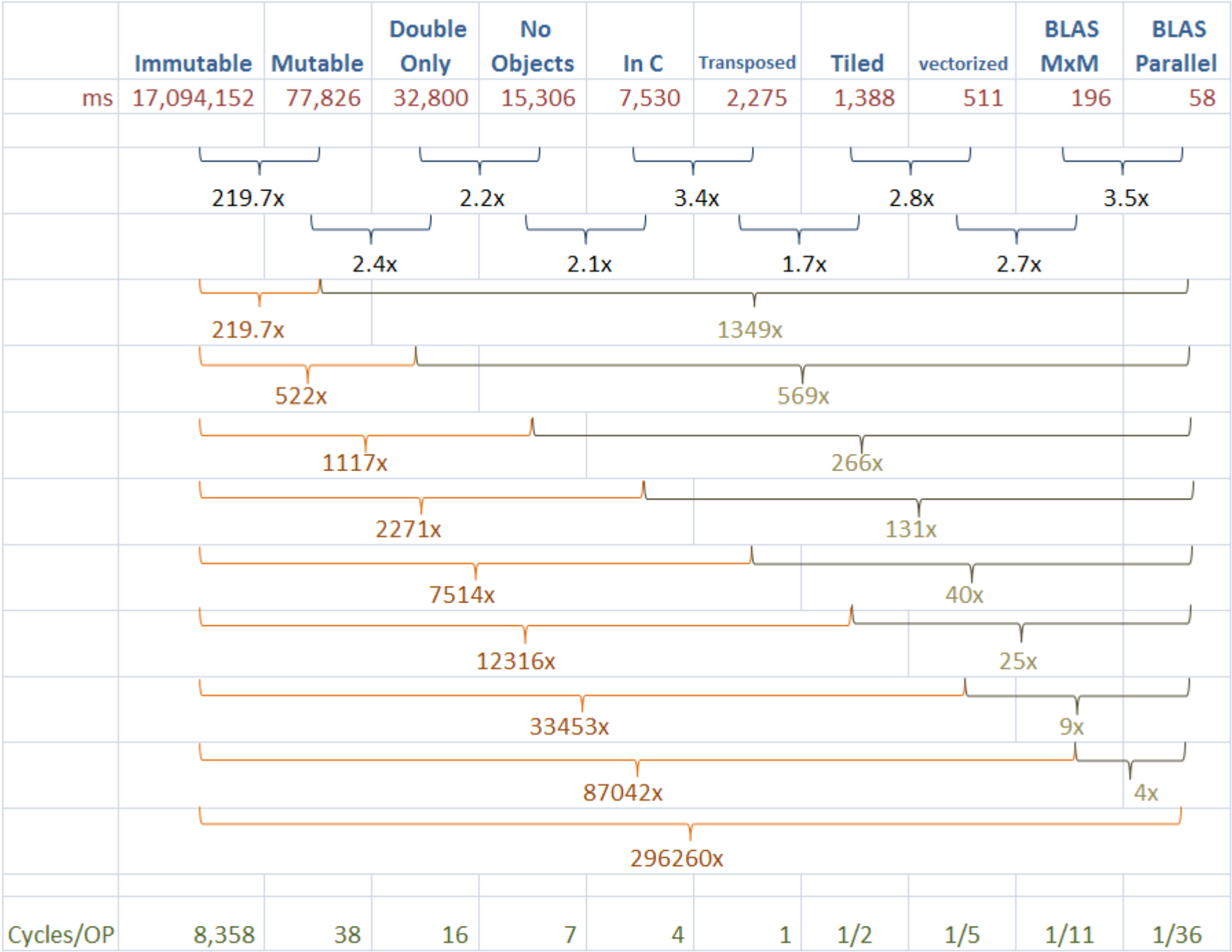
- The size of the parallel regions between synchronizations or the ratio of computation (useful work) to communication (overhead).

Parallel Execution of Matrix Multiply






Immutable
Mutable
Double Only
No Objects
In C
Transposed
Tiled
Vectorized
BLAS MxM
BLAS Parallel
ms
17,094,152
77,826
32,800
15,306
7,530
2,275
1,388
511
196
58
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
Cycles/OP
8,358
38
16
7
4
1
1/2
1/5
1/11
1/36


Immutable
Mutable
Double Only
No Objects
In C
Transposed
Tiled
vectorized
BLAS MxM
BLAS Parallel
ms
17,094,152
77,826
32,800
15,306
7,530
2,275
1,388
511
196
58
219.7x
2.2x
3.4x
2.8x
3.5x
2.4x
2.1x
1.7x
2.7x
219.7x
1349x
522x
569x
1117x
266x
2271x
131x
7514x
40x
12316x
25x
33453x
9x
87042x
4x
296260x
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

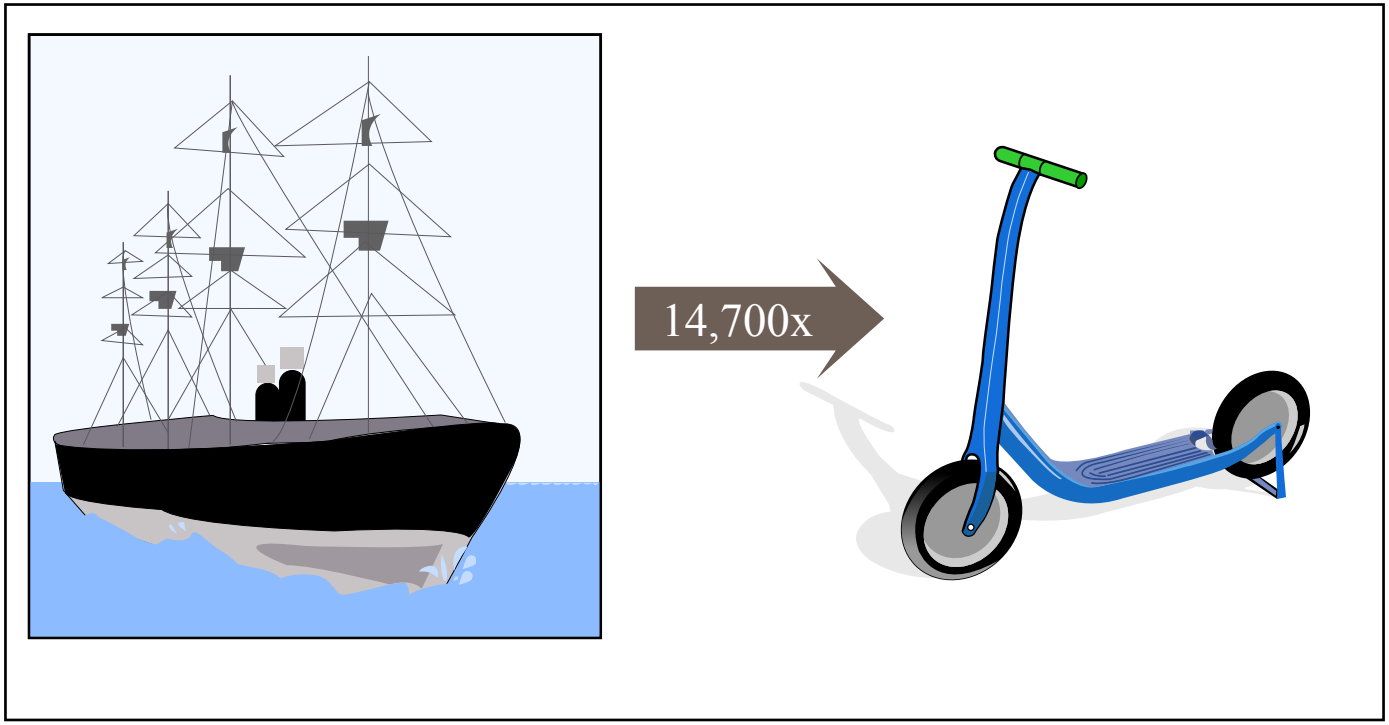


Image by MIT OpenCourseWare.

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

6.172 Performance Engineering of Software Systems

Fall 2010