



Анализ и оптимизация на софтуерни приложения

Александър Пенев

Васил Василев

Слоеве при трансляция

Съдържание

1. Производителност
2. Оптимизация
3. Методи за оптимизация



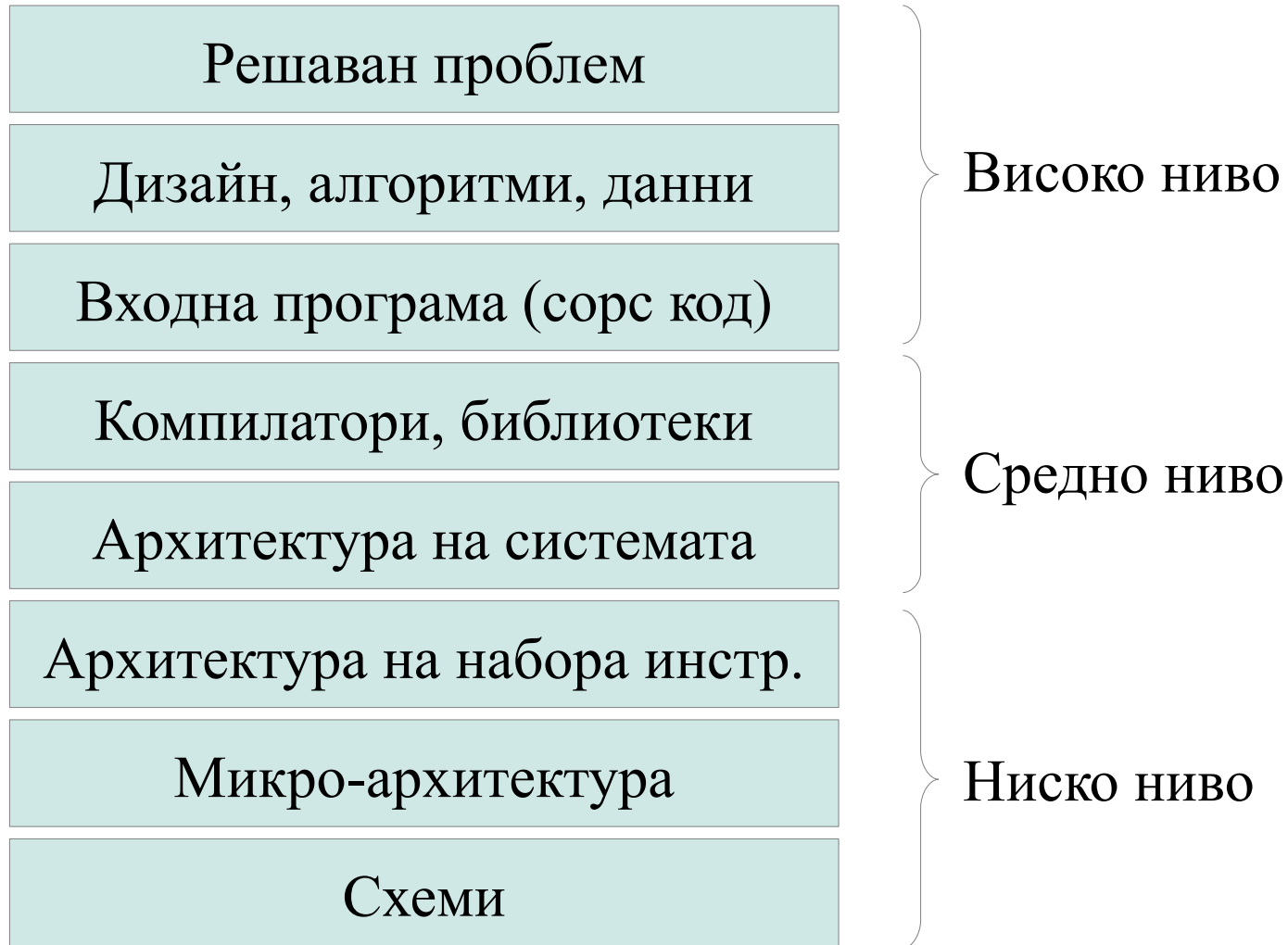
Решаване на проблеми в КИ

- ❖ Започване с конкретен проблем от реалния живот, който трябва да се реши
Например: умножение на 2 матрици $N \times N$
- ❖ Създаваме програма на някакъв език за програмиране от високо ниво
Например: C++, C#, Java, PHP и т.н
- ❖ Компилятор (или интерпретатор) преобразува кода, написан на високо ниво до машинен код
- ❖ Използваме външни библиотеки
- ❖ Сложен процесор със сложна архитектура и по-сложна микро-архитектура изпълнява кода

Много често нямаме дори идея колко ефективен е процесът на трансляция



Обща схема



Възможности за оптимизация

Решаван проблем	}	~10x-1000x
Дизайн, алгоритми, данни		
Входна програма (сорс код)	}	~2x-10x
Компилатори, библиотеки		
Архитектура на системата	}	~10%-20% (без векторизация)
Архитектура на набора инстр.		
Микро-архитектура		
Схеми	}	~5%-20%
	}	~10%-30%



Възможности на оптимизация

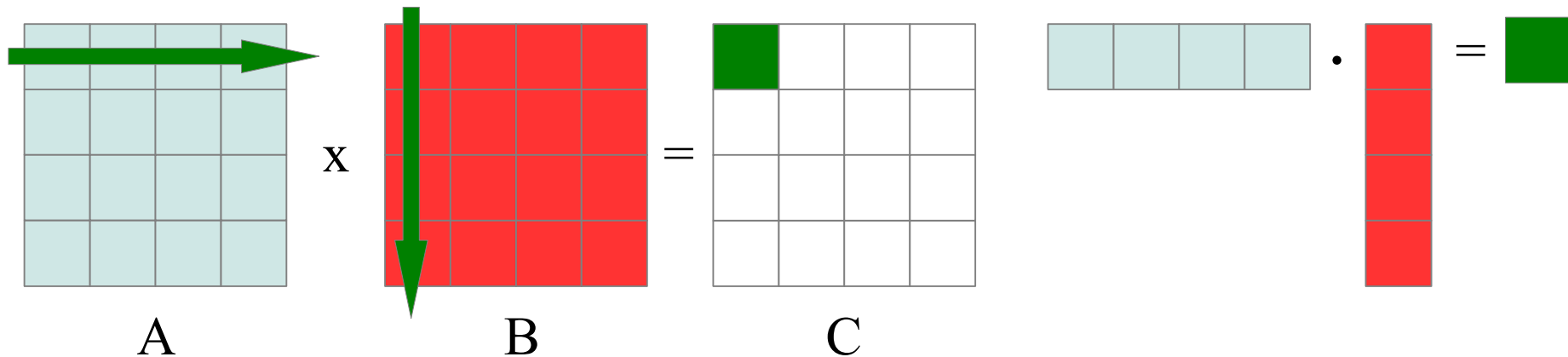
Пример: Изучаване на възможностите за оптимизация на дадено приложение – Умножение на матрици



Умножение на матрици

Умножението на матрици е много често срещано в реалния свят

Например: В компютърната графика; В симулаторите; Във видео-кодирането

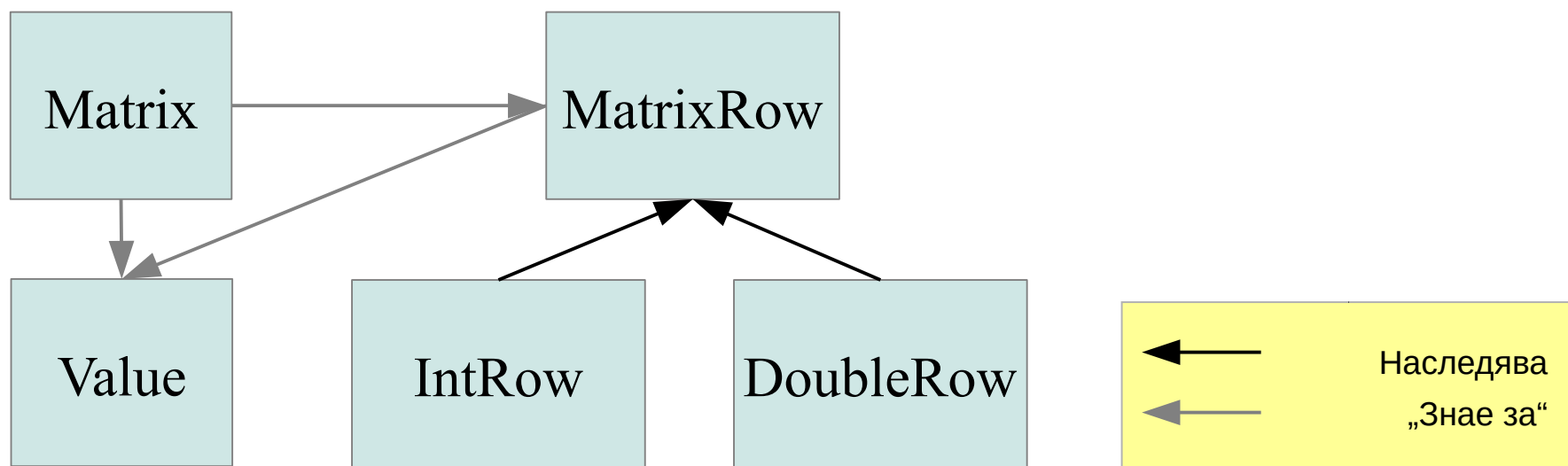


```
for i = 1 to n
  for j = 1 to n
    for k = 1 to n
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
    end for
  end for
end for
```

Представяне на матриците

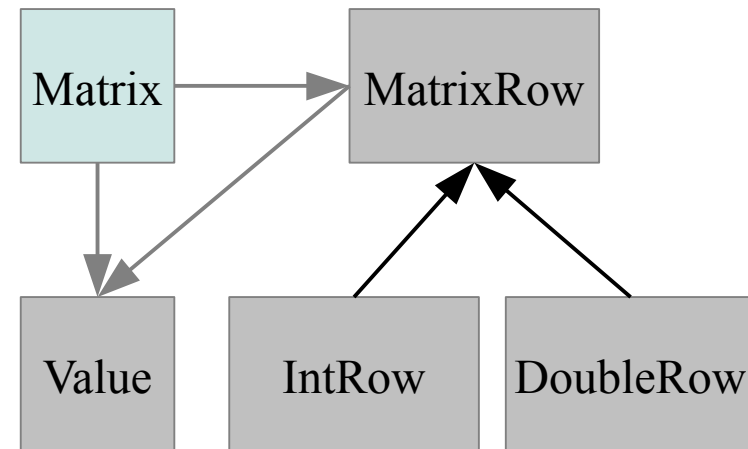
Бихме искали представянето на матрицата да:

- ❖ Се реализира като се използва обектно ориентиран подход
- ❖ Се използват неизменими (immutable) обекти
Незаменим обект е този, чийто състояние не може да бъде променено след като е обектът е създаден
- ❖ Представя цели числа и числа с плаваща запетая



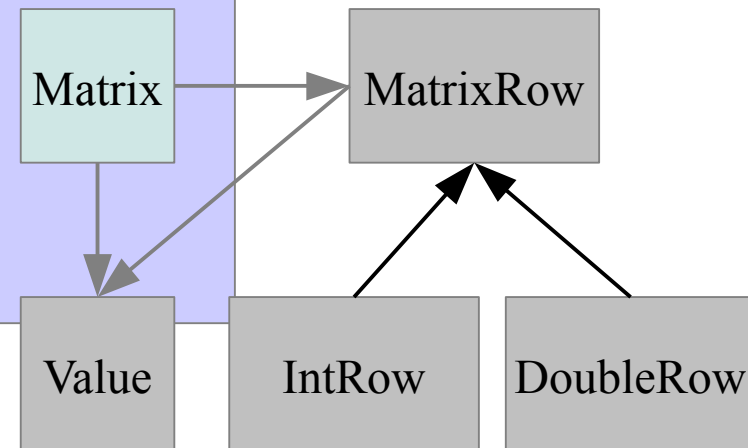
Реализация с неизменными объектами

```
public class Matrix {  
    readonly MatrixRow[] rows;  
    readonly int nRows, nColumns;  
    readonly Type Ty;  
  
    public Matrix(int rows, int cols, Type Typ) {  
        Ty = Typ;  
        nRows = rows;  
        nColumns = cols;  
        this.rows = new MatrixRow[nRows];  
        for (int i = 0; i < nRows; i++) {  
            if (Ty == typeof(int))  
                this.rows[i] = new IntRow(this.nColumns);  
            else  
                this.rows[i] = new DoubleRow(this.nColumns);  
        }  
    }  
    ...  
}
```



Реализация с неизменными объектами

```
...  
private Matrix(MatrixRow[] rows, Type Typ, int nRows, int nCols) {  
    this.rows = rows;  
    this.nRows = nRows;  
    nColumns = nCols;  
    Ty = Typ;  
}  
  
public Matrix update(int row, int col, Value val) {  
    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, Ty, nRows, nColumns);  
}  
  
public Value get(int row, int col) {  
    return rows[row].get(col);  
}  
}
```



Реализация с неизменными объектами

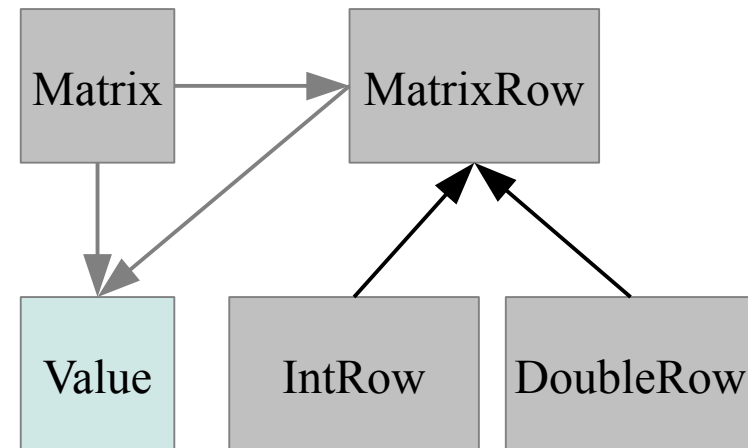
```
public class Value {
    readonly Type Ty;
    readonly int iVal;
    readonly double dVal;

    public Value(double d) {
        Ty = typeof(double);
        dVal = d; iVal = 0;
    }

    public Value(int I) {
        Ty = typeof(int);
        dVal = 0; iVal = I;
    }

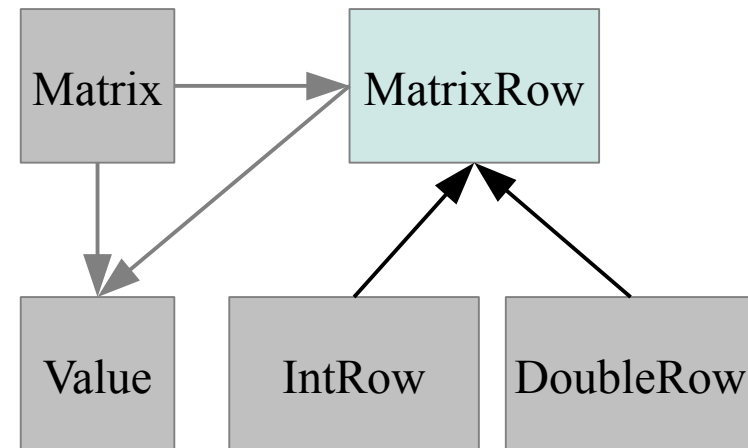
    public double getDouble() {
        if (Ty == typeof(double))
            return dVal;
        else
            throw new Exception();
    }

    public int getInt() {
        if (Ty == typeof(int))
            return iVal;
        else
            throw new Exception();
    }
}
```



Реализация с неизменными объектами

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



Реализация с неизменными объектами

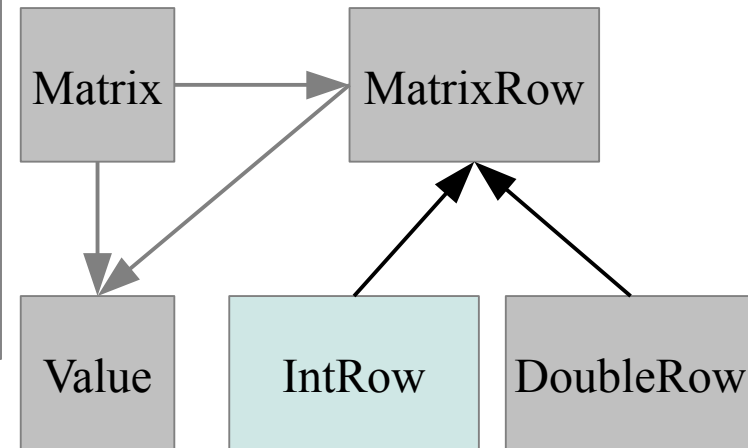
```
public class IntRow : MatrixRow {
    readonly int[] theRow;
    public readonly int numColumns;

    public IntRow(int ncols) {
        numColumns = ncols;
        theRow = new int[ncols];
        for (int i = 0; i < ncols; i++)
            theRow[i] = new int();
    }

    private IntRow(int[] row, int cols) {
        theRow = row;
        numColumns = cols;
    }

    public override MatrixRow update(int col, Value v) {
        int[] row = new int[numColumns];
        for (int i = 0; i < numColumns; i++)
            row[i] = (i == col) ? (v.GetInt()) : theRow[i];
        return new IntRow(row, numColumns);
    }

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



Реализация с неизменными объектами

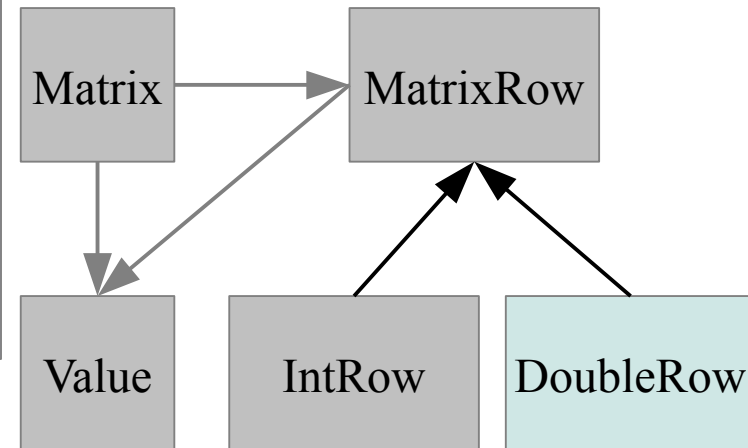
```
public class DoubleRow : MatrixRow {
    readonly Double[] theRow;
    public readonly int numColumns;

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

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

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

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



Реализация с неизменными объектами

```
public class MatrixMultiply {
    Matrix A = null;
    Matrix B = null;
    Matrix C = null;

    public double Multiply(int rA, int cA, int rB, int cB) {
        if (A == null || B == null || C == null) {
            A = new Matrix(rA, cA, typeof(double));
            B = new Matrix(rB, cB, typeof(double));
            C = new Matrix(rA, cB, typeof(double));
        }

        Stopwatch sw = new Stopwatch();
        sw.Start();

        for (int i = 0; i < rA; i++)
            for (int j = 0; j < cB; j++) {
                C = C.update(i, j, new Value(0.0));
                for (int k = 0; k < rB; k++)
                    C = C.update(i, j, new Value(
                        C.get(i, j).getDouble() + A.get(i, k).getDouble() * B.get(k, j).getDouble()));
            }

        sw.Stop();
        Console.WriteLine(sw.Elapsed.ToString() + "s");
        return sw.ElapsedMilliseconds;
    }
}
```



Резултати при изпълнение

1024 x 1024	Immutable
ms	54 000 000

~15 часа!

Дали производителността е задоволителна?

Резултати при изпълнение. Анализ

1024 x 1024	Immutable
ms	54 000 000

22 352 цикъла на операция!?

- ❖ $1024^3 = 1\,073\,741\,824$ итерации
Имаме три вложени цикъла по размера на матриците
- ❖ $6 * 1\,073\,741\,824 = 6\,442\,450\,944$ операции
Всяка операция съдържа умножение, събиране, обновяване на три индекса и преход, т.е. 6 очаквани операции за итерация
- ❖ $1.19 * 10^5$ операции в секунда
Операции в секунда: $6\,442\,450\,944 / 54\,000 \approx 119\,304$
- ❖ Максимално възможни или $2.66 * 10^9$ цикъла/секунда при 2.66 GHz
- ❖ 22 352 цикъла за операция
Цикли на операция: $2.66 * 10^9 / 1.19 * 10^5 = 22\,352$

Можем ли по-добре? Как?



Профайлинг

Дава възможност да се изследва изпълнението на програмата. Може да анализира:

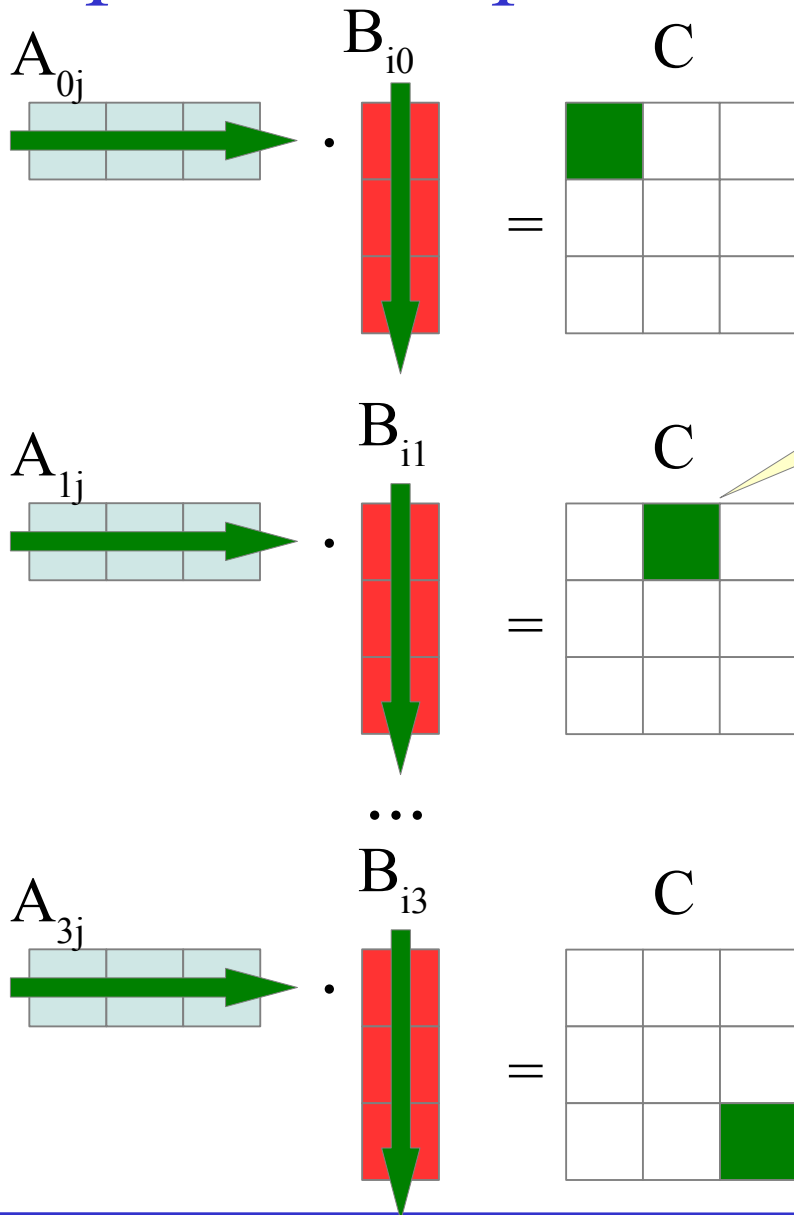
- ❖ Къде се губи най-много време
- ❖ В кой метод
- ❖ На кой ред
- ❖ Представи много различна и полезна информация
- ❖ Общо време загубено
- ❖ Брой извиквания на функциите
- ❖ И много други.

Данни от профайлинг на примера

Метод	Брой извиквания
MatrixMultiply.Multiply()	1
Matrix.update()	1 074 790 400
DoubleRow.update()	1 074 790 400
DoubleRow.DoubleRow()	1 074 790 400
DoubleRow.get()	3 221 225 472
Matrix.Matrix()	1 074 790 400
Matrix.get()	3 221 225 472
Value.Value()	4 296 015 872
Value.getDouble()	4 294 967 296



Проблеми при неизменяемите обекти



Модифицирането на един елемент води до копиране на цялата матрица

N^3 копия на матрицата

Колкото е и броя на итерациите

При копиране на матрица има още $N-1$ други копия

Копира се редът, съдържащ променената стойност, като се копират всички стойности, които са непроменени и се добавя новата стойност

Копиране на обекти

Копирането е скъпо:

- ❖ Създаването на дубликати е скъпо
- ❖ Събирането на боклука от неизползваните обекти е скъпо
- ❖ Създава излишно много обекти, които заемат много памет

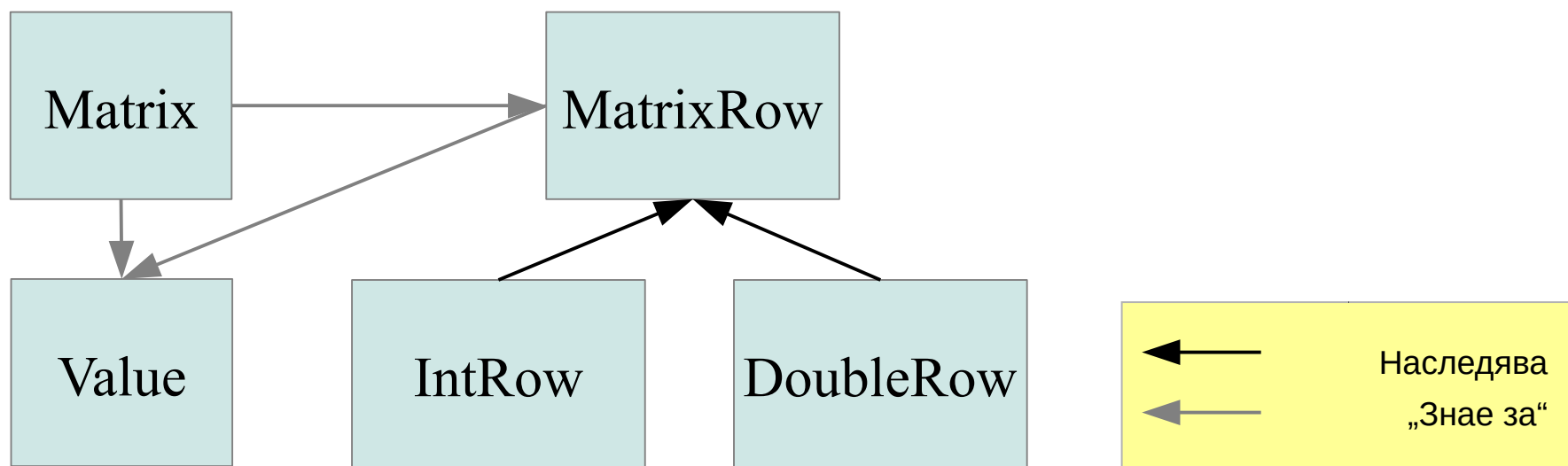
Можем ли по-добре? Как?



Представяне на матриците

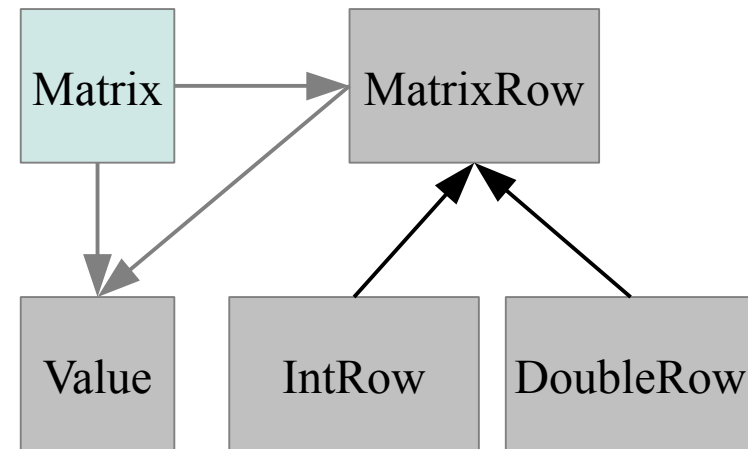
Бихме искали представянето на матрицата да:

- ❖ Се реализира като се използва обектно ориентиран подход
- ❖ ~~Се използват неизменими (immutable) обекти~~
Незаменим обект е този, чийто състояние не може да бъде променено след като е обектът е създаден
- ❖ Представя цели числа и числа с плаваща запетая



Реализация с изменими обекти

```
public class Matrix {  
    readonly MatrixRow[] rows;  
    readonly int nRows, nColumns;  
    readonly Type Ty;  
  
    public Matrix(int rows, int cols, Type Typ) {  
        Ty = Typ; nRows = rows; nColumns = cols;  
        this.rows = new MatrixRow[nRows];  
        for (int i = 0; i < nRows; i++) {  
            if (Ty == typeof(int))  
                this.rows[i] = new IntRow(this.nColumns);  
            else  
                this.rows[i] = new DoubleRow(this.nColumns);  
        }  
    }  
  
    private Matrix(MatrixRow[] rows, Type Typ,  
        int nRows, int nCols) {  
        this.rows = rows; this.nRows = nRows;  
        nColumns = nCols; Ty = Typ;  
    }  
  
    public void set(int row, int col, Value v) {  
        rows[row].set(col, v);  
    }  
  
    public Value get(int row, int col) {  
        return rows[row].get(col);  
    }  
}
```



Реализация с изменими обекти

```
public abstract class MatrixRow {
    public abstract Value get(int col);
    public abstract void set(int col, Value val);
}

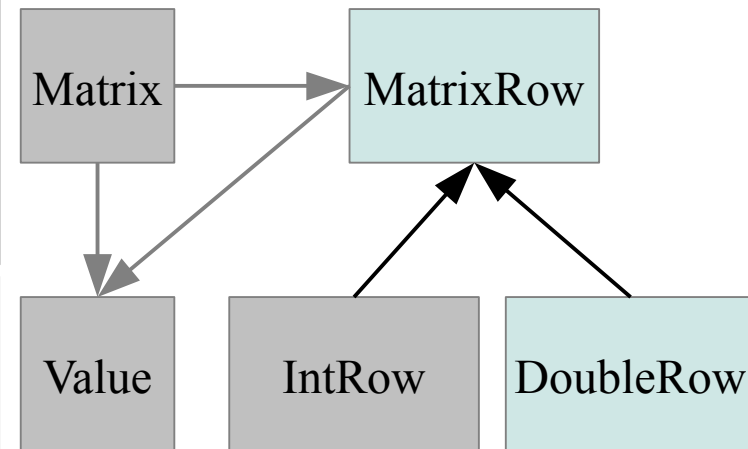
public class DoubleRow : MatrixRow {
    Double[] theRow;
    public readonly int numColumns;

    public DoubleRow(int ncols) {
        numColumns = ncols;
        theRow = new Double[ncols];
    }


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

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

Колко ще се подобри
производителността?



Резултати при изпълнение

1024 x 1024	Immutable	Mutable
ms	54 000 000	303 687
	 118x	
Цикли/Операция	22 352	126

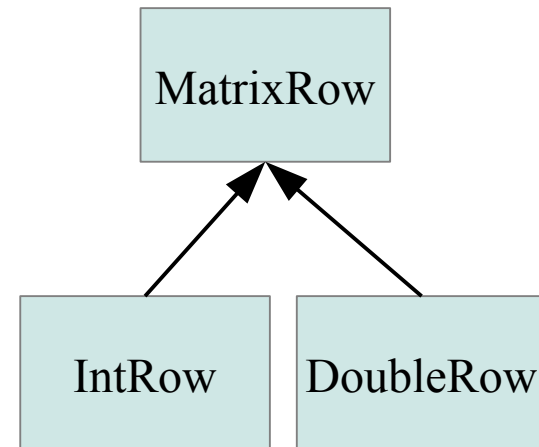


Данни от профайлинг на примера

Метод	Брой извиквания
MatrixMultiply.Multiply ()	1
Matrix.get()	3 221 225 472
DoubleRow.get()	3 221 225 472
Value.Value()	3 221 225 472
Matrix.set()	1 073 741 824
DoubleRow.set()	1 073 741 824
Value.getDouble()	$1.099511628 \times 10^{12}$



Проблеми при късното свързване



Виртуални извиквания:

- ❖ Множество наследници

Кой метод да бъде извикан зависи от обекта

- ❖ Всяко извикване трябва да намери адреса на метода във виртуалната таблица на обекта

Всеки обект, имащ виртуални методи, поддържа структура съдържаща адресите им

- ❖ Всяко извикване има индиректен преход

Вместо да се посочи адреса на следващата инструкция, се посочва адресът, където тя се намира

Индиректни преходи

Скъпи за съвременния процесор. Той

- ❖ Има дълъг конвейер

От 12 (Core 2 Duo) до 20 (Pentium 4) стъпки изпълняващи стотици микрооперации едновременно

- ❖ Извлича следващата инструкция преди да я изпълни (prefetching). Работи при:

- ❖ Нормалните инструкции

- ❖ Директни преходи

Целевия адрес е известен и може да бъде предварително извлечен.

- ❖ Условни преходи

Използва се предсказване на прехода (branch prediction) и се извлича съответното разклонение

- ❖ Индиректни преходи – предварителното извличане не работи

Целевия адрес е неизвестен и трябва да се изчака извличането му. Това води до мехурчета в конвейера (pipeline stalls)



Представяне на матриците

Бихме искали представянето на матрицата да:

- ❖ Се реализира като се използва обектно ориентиран подход
- ❖ ~~Се използват неизменими (immutable) обекти~~
Незаменим обект е този, чийто състояние не може да бъде променено след като е обектът е създаден
- ❖ ~~Представя цели числа и числа с плаваща запетая~~



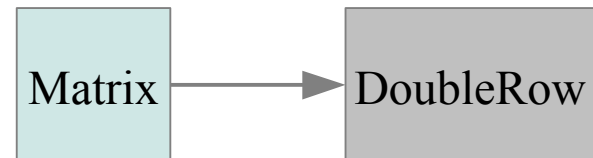
Реализация за числа с плаваща запетая

```
public class Matrix {
    readonly DoubleRow[] rows;
    readonly int nRows, nColumns;

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

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

    public double get(int row, int col) {
        return rows[row].get(col);
    }
}
```



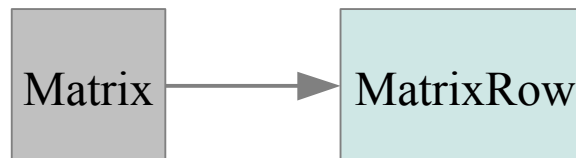
Реализация за числа с плаваща запетая

```
public sealed class DoubleRow {
    Double[] theRow;
    public readonly int numColumns;

    public DoubleRow(int ncols) {
        numColumns = ncols;
        theRow = new Double[ncols];
    }

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

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



Колко ще се подобри
производителността?

Резултати при изпълнение

1024 x 1024	Immutable	Mutable	Double Only
ms	54 000 000	303 687	40 000
	118x		
		7.6x	
	1350x		
Цикли/Операция	22 352	126	17



Данни от профайлинг на примера

Метод	Брой извиквания
MatrixMultiply.Multiply ()	1
DoubleMatrix.get()	3 221 225 472
DoubleRow.get()	3 221 225 472
DoubleMatrix.set()	1 073 741 824
DoubleRow.set()	1 073 741 824



Данни от профайлинг

Immutable

Метод	Брой извиквания
MatrixMultiply.Multiply ()	1
Matrix.update()	1 074 790 400
DoubleRow.update()	1 074 790 400
DoubleRow.DoubleRow()	1 074 790 400
DoubleRow.get()	3 221 225 472
Matrix.Matrix()	1 074 790 400
Matrix.get()	3 221 225 472
Value.Value()	4 296 015 872
Value.getDouble()	4 294 967 296

Mutable

Метод	Брой извиквания
MatrixMultiply.Multiply ()	1
Matrix.get()	3 221 225 472
DoubleRow.get()	3 221 225 472
Value.Value()	3 221 225 472
Matrix.set()	1 073 741 824
DoubleRow.set()	1 073 741 824
Value.getDouble()	1.099511628×10 ¹²

Double Only

Метод	Брой извиквания
MatrixMultiply.Multiply ()	1
DoubleMatrix.get()	3 221 225 472
DoubleRow.get()	3 221 225 472
DoubleMatrix.set()	1 073 741 824
DoubleRow.set()	1 073 741 824



Представяне на матриците

Бихме искали представянето на матрицата да:

- ❖ ~~Се реализира като се използва обектно ориентиран подход~~
- ❖ ~~Се използват неизменими (immutable) обекти~~
Незаменим обект е този, чийто състояние не може да бъде променено след като е обектът е създаден
- ❖ ~~Представя цели числа и числа с плаваща запетая~~



Реализация без OO подход

```
A = new double[rA, cA];  
B = new double[rB, cB];  
C = new double[rA, cB];  
  
Stopwatch sw = new Stopwatch();  
sw.Start();  
  
for (int i = 0; i < rA; i++) {  
    for (int j = 0; j < cB; j++) {  
        C[i, j] = 0;  
        for (int k = 0; k < rB; k++)  
            C[i, j] += A[i, k] * B[k, j];  
    }  
}  
  
sw.Stop();
```



Резултати при изпълнение

1024 x 1024	Immutable	Mutable	Double Only	No OOP
ms	54 000 000	303 687	40 000	27 586
	118x			
		7.6x		
	1350x			
			1.4x	
	1958x			
Цикли/Операция	22 352	126	17	11



От .NET/Mono към C

.NET/Mono/Java

- ❖ **Екстра проверки на границите**

Това не позволява проблеми от рода на buffer underflow/overflow

- ❖ **Проверка на индекс**

Например, при масиви се проверява дали индекса не излиза от границата на масива

- ❖ **Проверка на размера**

Например, има проверка дали се присвоява число по-голямо от това което типа позволява

- ❖ **Бавно изпълнение на bytecode**

Тези езици използват хибридни компилатори, генериращи междинен, код който се интерпретира или компилира от JIT

- ❖ **Интерпретацията е бавна по дефиниция**

- ❖ **JIT компилацията не генерира оптимални инструкции**

Извършва се по време на стартиране и затова компилаторът не може да си позволи да прилага времееотнемащи оптимизации



От .NET/Моно към C

C

- ❖ Няма тези проблеми
- ❖ Компилаторите на Интел за C генерират директно асемблер (машинен код)

Реализация на C

```
double **A = 0;
double **B = 0;
double **C = 0;

double Multiply(const int rA, const int cA, const int rB, const int cB) {
    clock_t start,end;
    double dif;

    if (!A || !B || !C) {
        A = malloc(sizeof(double*)*rA);
        for (int i = 0; i < rA; i++)
            A[i] = (double *) malloc(sizeof(double)*cA);

        B = malloc(sizeof(double*)*rB);
        for (int i = 0; i < rB; i++)
            B[i] = (double *) malloc(sizeof(double)*cB);

        C = malloc(sizeof(double*)*rA);
        for (int i = 0; i < rA; i++)
            C[i] = (double *) malloc(sizeof(double)*cB);
    }

    ...
}
```



Реализация на C

```
...  
  
start = clock();  
  
for (int i=0; i < rA; i++)  
    for (int j=0; j < cB; j++) {  
        C[i][j] = 0.0;  
        for (int k=0; k < rB; k++)  
            C[i][j] += A[i][k] * B[k][j];  
    }  
  
end = clock();  
dif = (end - start)/1000;  
  
printf("Time: %f ms\n", dif);  
return dif;  
}
```



Резултати при изпълнение

1024 x 1024	Immutable	Mutable	Double Only	No OOP	In C
ms	54 000 000	303 687	40 000	27 586	5 360
	118x				
		7.6x			
	1350x				
				5.14x	
	1958x				
	10075x				
Ц/Оп	22 352	126	17	11	2.2



Профайлинг с броячи на произв.

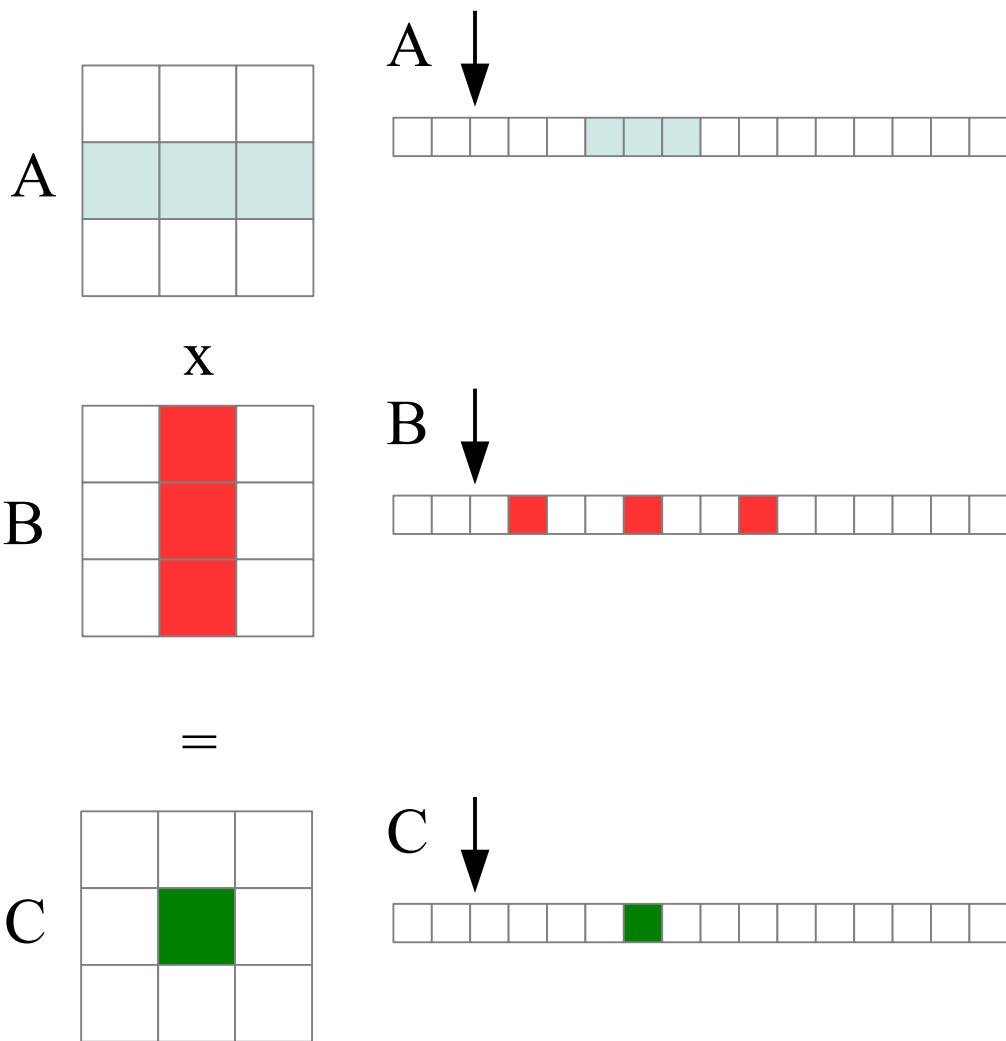
Модерният хардуер може да брои „събития“

- ❖ Много повече информация отколкото брой извиквания и време на изпълнение

Измерва:

- ❖ Цикли на инструкция (CPI – Clock cycles per instruction)
Измерва дали инструкциите са поставени в режим на изчакване (stalling)
- ❖ Пропуски в кеша от L1 или L2
Измерва дали достъпът до данни или инструкции се извършва в кеша
- ❖ Инструкции завършили работа
- ❖ Много други

Проблеми с представянето на матрицата



Последователният достъп е за предпочитане:

- ❖ Извлича се цял ред в кеша (64 bits L2 Cache line)
- ❖ С една операция се извличат до 8 числа от тип double

Предварителна подготовка на данните

При умножението на матрици

- ❖ N^3 изчисления
- ❖ N^2 данни

Може така да се подготвят данните преди изчисленията, че:

- ❖ N^2 изчисления върху N^2 данни
- ❖ Това може да накара алгоритъма да работи по-бързо.

Матрицата по-принцип няма добро поведение на кеша. Добре е да се транспонира:

- ❖ N^2 операции
- ❖ Основният цикъл за умножение ще работи по-бързо. Защо?

Реализация на C. Транспониране

```
#define IND(A, i, j, cols) A[(i)*(cols)+(j)]

double *A = 0; double *B = 0; double *Bx = 0; double *C = 0;

double Multiply(const int rA, const int cA, const int rB, const int cB)
  if (!A || !B || !Bx || !C) {
    A = (double *) malloc(sizeof(double)*rA*cA);
    B = (double *) malloc(sizeof(double)*rB*cB);
    Bx = (double *) malloc(sizeof(double)*cB*rB);
    C = (double *) malloc(sizeof(double)*rA*cB);
  }

  for (int i = 0; i < rB; i++)
    for (int j = 0; j < cB; j++) {
      IND(Bx, j, i, rB) = IND(B, i, j, cB);
    }

  for (int i = 0; i < rA; i++)
    for (int j = 0; j < cB; j++)
      for (int k = 0; k < rB; k++)
        IND(C, i, j, cB) += IND(A, i, k, cA) * IND(Bx, j, k, rB);
}
```



Резултати при изпълнение

1024 x 1024	Immutable	Mutable	Double Only	No OOP	In C	Transposed
ms	54 000 000	303 687	40 000	27 586	5 360	2 450
	118x					
		7.6x				
	1350x					
			1.4x			
	1958x					
	10075x					
	22041x					
Ц/Оп	22 352	126	17	11	2.2	1.01



Памет в компютърната система

Основния проблем:

- ❖ Малък обем памет – бърз достъп
- ❖ Голям обем памет – бавен достъп
- ❖ Как програмата може да използва много памет и да има бърз достъп до нея?



Йерархия на паметта

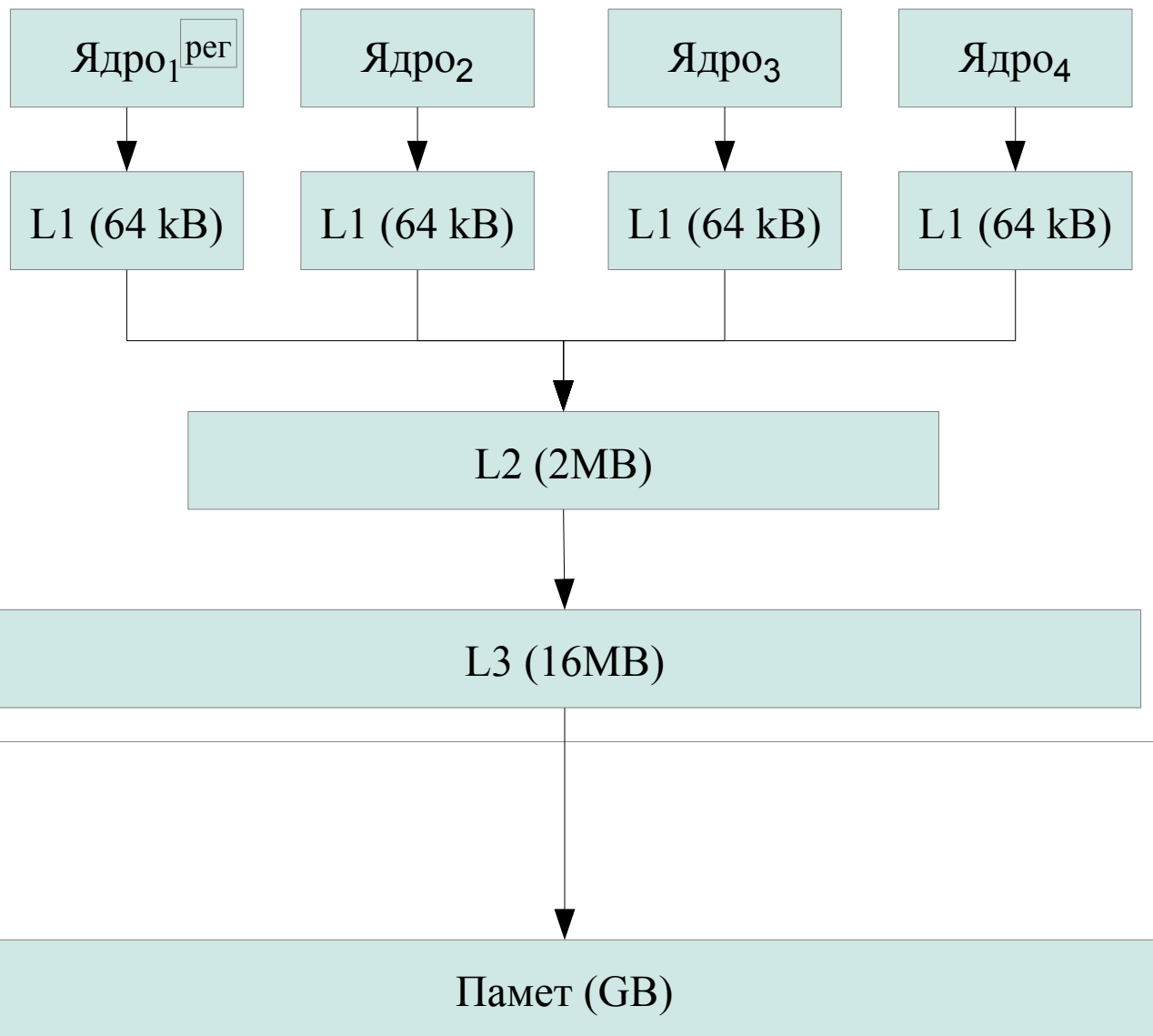
Задача на кеш системата:

- ❖ Да съхранява най-често и най-вероятно използваните стойности в памет с много ниски времена на достъп
- ❖ Хардуерни евристики определят какво и кога да бъде включено в кеш системата

Какво става, ако начинът на достъп в програмата се различава от хардуерната евристика?



Йерархия на паметта



1 цикъл

3 цикъла

14 цикъла

100+ цикъла



Преизползване на данните в кеша

- ❖ Изчисленията могат да бъдат променени така че да намалят броя на зарежданията в кеша
- ❖ Изчисляване на ред (1024 стойности)
- ❖ Изчисляване на крайната стойност на блокове (например с размер 128x128)



Промяна на програмата

Има много различни начини да се получи същия резултат:

- ❖ Промяна на реда на изпълнение
- ❖ Промяна на алгоритъма
- ❖ Промяна на структурите от данни

Някои промени могат да повлияят върху резултата:

- ❖ Избиране на различен но еквивалентен отговор
- ❖ Променяне на реда на операциите
 - ❖ $(a + b) + c \neq a + (b + c)$
- ❖ Загуба на прецизност

Реализация на C. Умножение на блокове

```
#define IND(A, i, j, cols) A[(i)*(cols)+(j)]
int min(int a, int b) { return a<b ? a : b; }
double *A = 0; double *B = 0; double *C = 0;

double Multiply(const int rA, const int cA, const int rB, const int cB) {
    int block_x = 128;
    int block_y = 128;

    If (!A || !B || !C) {
        A = (double *) malloc(sizeof(double)*rA*cA);
        B = (double *) malloc(sizeof(double)*rB*cB);
        C = (double *) malloc(sizeof(double)*rA*cB);
    }

    for (int i = 0; i < rA; i++)
        for (int j = 0; j < cB; j++)
            IND(C, i, j, cB) = 0.0;

    for (int j2 = 0; j2 < cB; j2 += block_x)
        for (int k2 = 0; k2 < cA; k2 += block_y)
            for (int i = 0; i < rA; i++)
                for (int j = j2; j < min(j2 + block_x, cB); j++)
                    for (int k = k2; k < min(k2 + block_y, cA); k++)
                        IND(C, i, j, cB) += IND(A, i, k, cA) * IND(B, k, j, cB);
}
```



Резултати при изпълнение

1024 x 1024	Immutable	Mutable	Double Only	No OOP	In C	Transposed	Tiled
ms	54 000 000	303 687	40 000	27 586	5 360	2 450	740
	118x						
		7.6x					
	1350x						
			1.4x				
	1958x						
	10075x						
	22041x						
	72973x						
Ц/Оп	22 352	126	17	11	2.2	1.01	0.31



Оптимизация на ниво инструкции

Модерните процесори имат много трикове, подобряващи производителността

- ❖ **Instruction Level Parallelism**

Представява мярка доколко операциите в програмата могат да се изпълняват едновременно

- ❖ **Single Instruction Multiple Data**

SSE (Streaming SIMD Extensions) е набор SIMD инструкции, разширяващи x86 архитектурата.

- ❖ **Йерархия на кеш паметта**

- ❖ **Предварително извличане на данни**

Оптимизация на ниво инструкции

Компилаторите могат да се възползват от тези особености на архитектурата

- ❖ В много от тях изрично трябва да се зададе опция за векторизация
- ❖ Възможно е да се използват специални конструкции
- ❖ Възможно е да се използва допълнителна диагностика за анализ на проблемите с векторизацията

Използват се #pragma, подсказваща къде да се векторизира

В компилаторите има подсистеми подпомагащи писането на векторен код



Реализация на C. Векторизиране

```
#define IND(A, i, j, cols) A[(i)*(cols)+(j)]
#define N 1024
#define BLOCK_X 128
#define BLOCK_Y 1024

double *A = 0; double *B = 0; double *Bx = 0; double *C = 0;

double Multiply() {
    A = (double *) malloc(sizeof(double)*N*N);
    B = (double *) malloc(sizeof(double)*N*N);
    Bx = (double *) malloc(sizeof(double)*N*N);
    C = (double *) malloc(sizeof(double)*N*N);

    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            IND(Bx, j, i, N) = IND(B, i, j, N);
    for (int j2 = 0; j2 < N; j2 += BLOCK_X)
        for (int k2 = 0; k2 < N; k2 += BLOCK_Y)
            for (int i = 0; i < N; i++)
                for (int j = 0; j < BLOCK_X; j++) {
                    IND(C, i, j+j2, N) = 0.0;
                    for (int k = 0; k < N; k++)
                        IND(C, i, j+j2, N) += IND(A, i, k+k2, N) * IND(Bx, j+j2, k+k2, N);
                }
}
```



Резултати при изпълнение

1024 x 1024	Immutable	Mutable	Double Only	No OOP	In C	Transposed	Tiled	Vectorized
ms	54 000 000	303 687	40 000	27 586	5 360	2 450	740	680
	118x							
	1350x							
			1.4x					
	1958x							
	10075x							
	22041x							
	72973x							
	79412x							
Ц/Оп	22 352	126	17	11	2.2	1.01	0.31	0.28

Равносметка

Постигнати ускорения:

- ❖ Във високото ниво ~1900 пъти
- ❖ В средното ниво ~7 пъти
- ❖ В ниското ниво ~6 пъти

- ❖ Общо **80 000 пъти !!!**