**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**Факультет информационных технологий**

**Кафедра параллельных вычислений**

**ОТЧЕТ**

**О ВЫПОЛНЕНИИ ПРАКТИЧЕСКОЙ РАБОТЫ**

«Изучение векторизации вычислений в программах на языке C/C++»

студента 2 курса, группы 20203

**Синюкова Валерия Константиновича**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
доцент кафедры параллельных вычислений
Власенко Андрей Юрьевич

Новосибирск 2021

# СОДЕРЖАНИЕ

# ЦЕЛЬ

1. Изучение SIMD-расширений архитектуры x86/x86-64.

2. Изучение способов использования SIMD-расширений в программах на языке Си.

3. Получение навыков использования SIMD-расширений.

# ЗАДАНИЕ

1. Написать три варианта программы, реализующей следующий алгоритм обращения матрицы A размером N×N с помощью разложения в ряд:
   $A^{-1} = (I + R + R^2 + R^3 + \cdots)B$, где R = I – BA , $B = \frac{A^T}{\|A\|_1 \cdot \|A\|_\infty}$ , $\|A\|_1 = \max_j \sum_i |A|_{ij}$, $\|A\|_\infty = \max_i \sum_j |A|_{ij}$ I – единичная матрица. Параметры алгоритма: N – размер матрицы, M – число членов ряда:
   - вариант без векторизации,
   - вариант с ручной векторизацией (выбрать любой вариант из возможных трех: ассемблерная вставка, встроенные функции компилятора, расширение GCC),
   - вариант с матричными операциями, выполненными с использованием оптимизированной библиотеки BLAS. Для элементов матриц использовать тип данных float.
2. Проверить правильность работы программ на нескольких небольших тестовых наборах входных данных.
3. Каждый вариант программы оптимизировать по скорости, насколько это возможно.
4. Сравнить время работы трех вариантов программы для N=2048, M=10.
5. Составить отчет по лабораторной работе. Отчет должен содержать все обязательные элементы из шаблона и к тому же следующее:
   - Результаты измерения времени работы трех программ.
   - Полный компилируемый листинг реализованных программ и команды для их компиляции.
   - Вывод по результатам лабораторной работы

# ОПИСАНИЕ РАБОТЫ

1) Была написана программа на языке C, реализующая алгоритм из задания (см. листинг данной программы в соответствующем приложении).
2) Корректность работы данной программы была проверена на нескольких набор тестовых входных данных.
3) Было произведено пять замеров времени работы программы с помощью утилиты time для следующего набора тестовых входных данных: N = 4, M = $10^6$,

$$A = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & 1 & 1 \end{pmatrix},$$

где N – размер матрицы, M – количество членов ряда, A – матрица, для которой алгоритм находит обратную.

| № попытки | время user (c) |
|---|---|
| 1 | 3,086 |
| 2 | 2,987 |
| 3 | 2,990 |
| 4 | 2,987 |
| 5 | 3,026 |
| **среднее арифметическое** | **3,015** |

4) Данная программа была оптимизирована следующим образом (см. код оптимизированной программы в соответствующем приложении):
   - вместо того, чтобы при создании очередной матрицы динамически выделять память под массив указателей на float размера N и потом выделять память под N массивов float размера N, выделяется память под один массив float размером $N^2$. Так сокращаются затраты времени на доступ к очередному элементу массива: вместо того, чтобы дважды обращаться в память, обращение происходит лишь единожды. Так же уменьшается объем памяти, используемый нашей программой, так как каждый раз при выделении памяти под динамический массив, некоторый объем памяти выделяется под служебную информацию.
   - *Самая затратная в плане времени часть программы - умножение матриц*, так как в ней происходит больше всего обращений к памяти и арифметических операций, а сама операция умножения матриц используется большее количество раз, чем остальные операции, так как программе необходимо посчитать M степеней матрицы R. Логично предположить, что оптимизация операции умножения матриц будет лучше всего отражаться на времени работы программы. Были выполнены следующие преобразования (см. тело функции `matrixMultiplication` в изначальной и оптимизированной программе):

1. Упрощается вычисление адресов элементов массива: постоянная часть выносится из внутреннего цикла.
2. Когда процессор считывает один элемент из памяти, на самом деле, он загружает в свой кэш последовательность из нескольких десятков байт, лежащих подряд в памяти, к которой мы обращаемся. Таким образом, невыгодно на каждой итерации цикла обращаться к элементу массива, который находится в не в той же строке, в который находился элемент, к которому происходило обращение на предыдущей итерации. Поэтому последовательность циклов в функции `matrixMultiplication` была изменена: внешним стал цикл по индексу k, внутренним - цикл по индексу j.

5) Было произведено пять замеров времени работы оптимизированной программы с помощью утилиты time для набора входных данных из пункта 3).

| № попытки | время user (с) |
|---|---|
| 1 | 2,777 |
| 2 | 2,775 |
| 3 | 2,779 |
| 4 | 2,774 |
| 5 | 2,778 |
| **среднее арифметичекое** | **2,7766** |

*Благодаря сделанным преобразованиям программа стала работать в среднем на 8% быстрее.*

6) Был написан вариант программы с векторизованным с помощью встроенных SIMD-функций компилятора умножением матриц (используются регистры из SSE2). *В данном варианте программы подразумевается, что размер матрицы, для которой находится обратная, кратен 4* (см. реализацию функции `matrixMultiplication`).
7) Было произведено пять замеров времени работы программы с помощью утилиты time для набора входных данных из пункта 3).

| № попытки | время user (с) |
|---|---|
| 1 | 0,91 |
| 2 | 0,907 |
| 3 | 0,912 |
| 4 | 0,908 |
| 5 | 0,906 |
| **среднее арифметическое** | **0,9086** |

*По сравнению с изначальным вариантом программа стала работать в среднем на 70% быстрее.*

8) Так же как и операция умножения матриц, операция сложения матриц используется M раз, и хоть эта операция гораздо менее затратная, нежели умножение, она тоже была векторизована (см. реализацию функции `matrixAddition`).
9) Было произведено пять замеров времени работы программы с помощью утилиты time для набора входных данных из пункта 3).

| № попытки | время user (с) |
|---|---|
| 1 | 0,876 |
| 2 | 0,878 |
| 3 | 0,883 |
| 4 | 0,879 |
| 5 | 0,878 |
| **среднее арифметическое** | **0,8788** |

*По сравнению с предыдущим вариантом программа стала работать в среднем на 3% быстрее.*

10) На основе версии оптимизированной программы из пункта 4) был написан вариант программы с использованием библиотеки BLAS, с помощью функций из данной библиотеки были реализованы операции: транспонирование матриц, вычитание матриц, сложение матриц, умножение матриц (см. код программы в соответствующем приложении).
11) Было произведено пять замеров времени работы программы с помощью утилиты time для набора входных данных из пункта 3).

| № попытки | время user (с) |
|---|---|
| 1 | 3,79 |
| 2 | 3,79 |
| 3 | 3,792 |
| 4 | 3,797 |
| 5 | 3,791 |
| **среднее арифметическое** | **3,792** |

12) Заключительная таблица для тестовых данных из пункта 3)

| | изначальная версия программы | оптимизированная версия программы | программа с векторизацией вычислений | программа, написанная с использованием BLAS |
|---|---|---|---|---|
| **№ попытки** | **время user (с)** | | | |
| 1 | 3,086 | 2,777 | 0,876 | 3,79 |
| 2 | 2,987 | 2,775 | 0,878 | 3,79 |
| 3 | 2,99 | 2,779 | 0,883 | 3,792 |
| 4 | 2,987 | 2,774 | 0,879 | 3,797 |
| 5 | 3,026 | 2,778 | 0,878 | 3,791 |
| **сред. арифм.** | **3,015** | **2,7766** | **0,8788** | **3,792** |

13) Для каждого из трех вариантов программы (без векторизации, с векторизацией, с использованием BLAS) было измерено время их работы на тестовых данных: N = 2048, M = 10 (где N – размер матрицы, для которой нужно найти обратную, M – число членов ряда, участвующих в разложении).

| оптимизированная версия программы | программа с векторизацией вычислений | программа, написанная с использованием BLAS |
|---|---|---|
| **время user (с)** | | |
| **282,415** | **157,85** | **20,502** |

14) Проведем еще по три замера времени работы каждой из трех версий программ для нескольких наборов входных данных

1. N = 32; M = $10^5$

| | оптимизированная версия программы | программа с векторизацией вычислений | программа, написанная с использованием BLAS |
|---|---|---|---|
| **№ попытки** | **время user (с)** | | |
| 1 | 10,407 | 6,111 | 2,626 |
| 2 | 10,393 | 5,938 | 2,542 |
| 3 | 11,021 | 6,071 | 2,543 |
| **сред. арифм.** | **10,607** | **6,04** | **2,570333333** |

2. N = 256; M = $10^3$

| | оптимизированная версия программы | программа с векторизацией вычислений | программа, написанная с использованием BLAS |
|---|---|---|---|
| **№ попытки** | **время user (с)** | | |
| 1 | 52,141 | 27,889 | 3,384 |
| 2 | 51,417 | 28,91 | 3,379 |
| 3 | 51,162 | 27,871 | 3,375 |
| **сред. арифм.** | **51,57333333** | **28,22333333** | **3,379333333** |

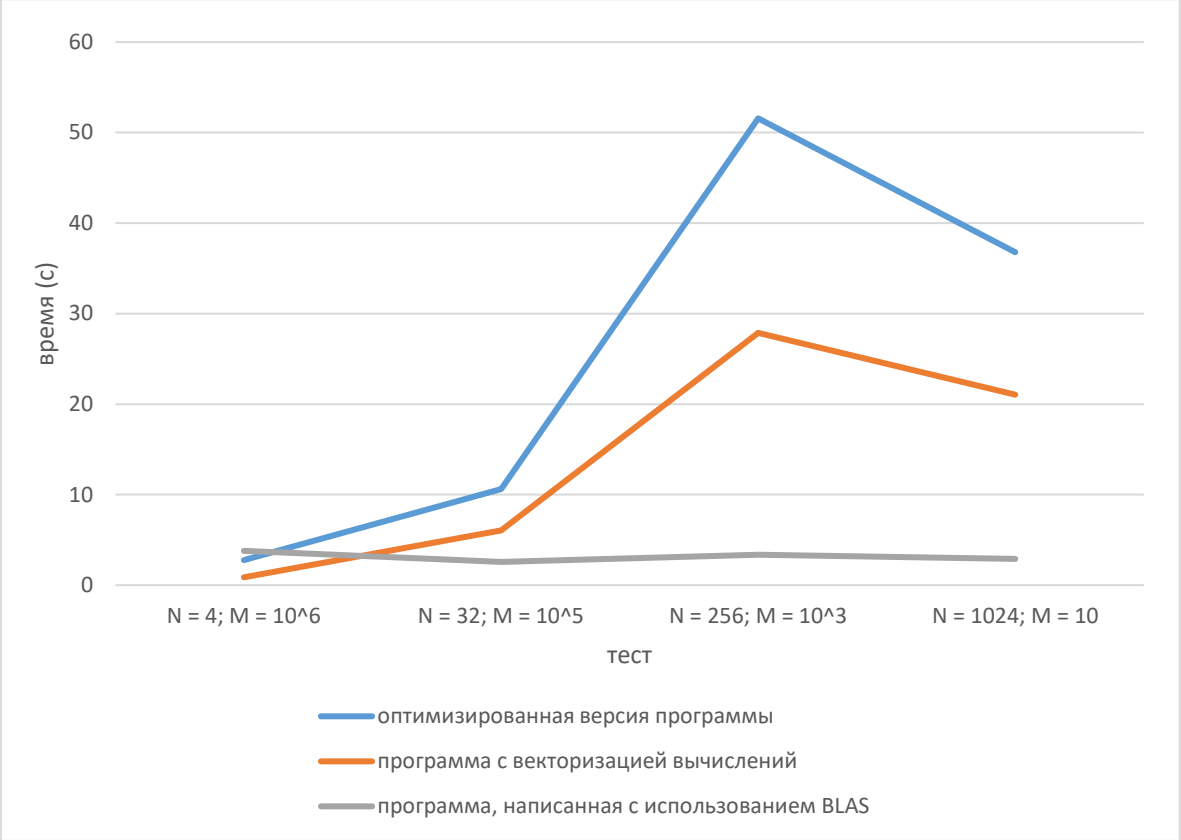
3. N = 1024; M = 10

| | оптимизированная версия программы | программа с векторизацией вычислений | программа, написанная с использованием BLAS |
|---|---|---|---|
| **№ попытки** | **время user (с)** | | |
| 1 | 35,788 | 21,022 | 2,874 |
| 2 | 37,387 | 21,085 | 2,904 |
| 3 | 37,168 | 21,039 | 2,906 |
| **сред. арифм.** | **36,781** | **21,04866667** | **2,894666667** |

4. Итоговая таблица по этим трем наборам тестовых данных и набору из пункта 3)

| | оптимизированная версия программы | программа с векторизацией вычислений | программа, написанная с использованием BLAS |
|---|---|---|---|
| **тест** | **среднее арифметическое время user (с)** | | |
| N = 4; M = 10^6 | 2,77 | 0,87 | 3,792 |
| N = 32; M = 10^5 | 10,607 | 6,04 | 2,57 |
| N = 256; M = 10^3 | 51,57 | 27,871 | 3,375 |
| N = 1024; M = 10 | 36,781 | 21,048 | 2,894 |

15) Заключительный график, построенный по таблице из пункта 14)4.



# ЗАКЛЮЧЕНИЕ

По результатам лабораторной работы была написана программа на языке C++ для обращения матрицы путем разложения в ряд. Данная программа была оптимизирована. Был написан вариант данной программы с полуавтоматической векторизацией умножения и сложения матриц. Был написан вариант данной программы с использованием библиотеки BLAS. По результатам измерения времени работы разных версий данной программы можно сделать вывод, что при больших размерах изначальной матрицы гораздо эффективнее (на порядок) использовать программу, написанную с использованием библиотеки BLAS. Если же размер матрицы малый, а количество членов ряда, участвующих в разложении, большое, то эффективнее использовать программу с ручной векторизацией.

**Приложение 1.** *Листинг изначальной программы, реализующей алгоритм из задания*

```
#include <iostream>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <fstream>

using namespace std;

float** createNewMatrix(int N)
{
	float** matrix = new float* [N];
	int i;
	for (i = 0; i < N; ++i)
		matrix[i] = new float[N];
	return matrix;
}

float** getA(int N, ifstream& in)
{
	float** A = createNewMatrix(N);
	int i,j;
	for (i = 0; i < N; ++i)
		for (j = 0; j < N; ++j)
			in >> A[i][j];
	return A;
}

float ** getAT(float** A, int N)
{
	float** AT = createNewMatrix(N);
	int i, j;
	for (i = 0; i < N; ++i)
		for (j = 0; j < N; ++j)
			AT[i][j] = A[j][i];
	return AT;
}

float getAone(float ** A, int N)
{
	float max = 0, current;
	int i, j;
	for (i = 0; i < N; ++i)
	{
		current = 0;
		for (j = 0; j < N; ++j)
			current += abs(A[j][i]);
		if (current > max)
			max = current;
	}
	return max;
}
```

```
float getAinfinity(float** A, int N)
{
	float max = 0, current;
	int i, j;
	for (i = 0; i < N; ++i)
	{
		current = 0;
		for (j = 0; j < N; ++j)
			current += abs(A[i][j]);
		if (current > max)
			max = current;
	}
	return max;
}


float** getB(float** AT, int N, float Aone, float Ainfinity)
{
	float** B = createNewMatrix(N), AoneAndAinfinityMult = Aone*Ainfinity;
	int i, j;
	for (i = 0; i < N; ++i)
		for (j = 0; j < N; ++j)
			B[i][j] = AT[i][j] / AoneAndAinfinityMult;
	return B;
}


float** matrixMultiplication(float** A, float** B, int N)
{
	float** result = createNewMatrix(N);
	int i, j, k;
	for (i = 0; i < N; ++i)
		for (j = 0; j < N; ++j)
		{
			result[i][j] = 0;
			for (k = 0; k < N; ++k)
				result[i][j] += A[i][k] * B[k][j];
		}
	return result;
}

void matrixAddition(float** A, float** B, int N)
{
	int i, j;
	for (i = 0; i < N; ++i)
		for (j = 0; j < N; ++j)
				A[i][j] += B[i][j];
}

float** matrixSubtraction(float** decreasing, float** subtrahend, int N)
{
	float** result = createNewMatrix(N);
	int i, j;
	for (i = 0; i < N; ++i)
		for (j = 0; j < N; ++j)
```

```
                result[i][j] = decreasing[i][j] - subtrahend[i][j];
        return result;
}

float** getI(int N)
{
        float** I = createNewMatrix(N);
        int i, j;
        for (i = 0; i < N; ++i)
                for (j = 0; j < N; ++j)
                        I[i][j] = 0;
        for (i = 0; i < N; ++i)
                I[i][i] = 1;
        return I;
}

float** getR(float** I, float** B, float** A, int N)
{
        float** ABMult = matrixMultiplication(B, A, N);
        return matrixSubtraction(I, ABMult,N);
}

void cleanUpMatrix(float** matrix, int N)
{
        if (!matrix)
                return;
        int i;
        for (i = 0; i < N; ++i)
                if (matrix[i])
                        delete(matrix[i]);
        delete(matrix);
}

float** getInvertedA(float ** I,float** R, float** B, int N, int M)
{
        float** prevRDeg, ** curRDeg = NULL, ** summ = I;
        matrixAddition(summ, R, N);
        int i;
        curRDeg = matrixMultiplication(R, R, N);
        matrixAddition(summ, curRDeg, N);
        prevRDeg = curRDeg;
        for (i = 2; i < M; ++i)
        {
                curRDeg = matrixMultiplication(R, prevRDeg, N);
                matrixAddition(summ, curRDeg, N);
                cleanUpMatrix(prevRDeg, N);
                prevRDeg = curRDeg;
        }
        cleanUpMatrix(curRDeg,N);
        float** AInverted = matrixMultiplication(summ, B, N);
        return AInverted;
}

void showAInverted(float ** AInverted,int N)
{
        int i, j;
```

```
	for (i = 0; i < N; ++i)
	{
		for (j = 0; j < N; ++j)
			printf("%.4f ", AInverted[i][j]);
		printf("\n");
	}
}

int main(int argc, char ** argv)
{
	if (argc != 4)
	{
		cout << "wrong number of arguments" << endl;
		return -1;
	}
	int N = atoi(argv[1]), M = atoi(argv[2]);
	ifstream in;
	in.open(argv[3]);
	if (!in)
	{
		cout << "couldn't open file" << endl;
		return -1;
	}
	float** A = getA(N,in);
	float** AT = getAT(A, N);
	float Aone = getAone(A, N);
	float Ainfinity = getAinfinity(A, N);
	float** B = getB(AT, N, Aone, Ainfinity);
	float** I = getI(N);
	float** R = getR(I, B, A, N);
	float** AInverted = getInvertedA(I, R, B, N, M);
	showAInverted(AInverted, N);
	cleanUpMatrix(A, N);
	cleanUpMatrix(AT, N);
	cleanUpMatrix(B, N);
	cleanUpMatrix(I, N);
	cleanUpMatrix(R, N);
	cleanUpMatrix(AInverted, N);
	return 0;
}
```

**Приложение 2.** *Листинг оптимизированной программы, реализующей алгоритм из задания*

```
#include <iostream>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <fstream>

using namespace std;

float* getA(int N, ifstream& in)
{
	float* A = new float[N * N];
```

```
	int i;
	for (i = 0; i < N * N; ++i)
		in >> A[i];
	return A;
}

float* getAT(float* A, int N)
{
	float* AT = new float[N * N];
	int i, j;
	for (i = 0; i < N; ++i)
		for (j = 0; j < N; ++j)
			AT[i * N + j] = A[j * N + i];
	return AT;
}

float getAone(float* A, int N)
{
	float max = 0, current = 0;
	int i;
	for (i = 0; i < N * N; ++i)
	{
		current += abs(A[i]);
		if (i % N == N - 1)
		{
			if (current > max)
				max = current;
			current = 0;
		}
	}
	return max;
}

float getAinfinity(float* A, int N)
{
	float max = 0, current = 0;
	int i;
	for (i = 0; i < N * N; ++i)
	{
		current += abs(A[i]);
		if (i % N == N - 1)
		{
			if (current > max)
				max = current;
			current = 0;
		}
	}
	return max;
}

float* getB(float* AT, int N, float Aone, float Ainfinity)
{
	float* B = new float[N * N], AoneAndAinfinityMult = Aone * Ainfinity;
	int i;
	for (i = 0; i < N * N; ++i)
		B[i] = AT[i] / AoneAndAinfinityMult;
```

```
	return B;
}


float* matrixMultiplication(float* A, float* B, int N)
{
	float* result = new float[N * N], * c, * b, a;
	int i, j, k;
	for (i = 0; i < N; ++i)
	{
		c = result + i * N;
		for (j = 0; j < N; ++j)
			c[j] = 0;
		for (k = 0; k < N; ++k)
		{
			a = A[i * N + k];
			b = B + k * N;
			for (j = 0; j < N; ++j)
				c[j] += a * b[j];
		}
	}
	return result;
}

void matrixAddition(float* A, float* B, int N)
{
	int i;
	for (i = 0; i < N * N; ++i)
		A[i] += B[i];
}

float* matrixSubtraction(float* decreasing, float* subtrahend, int N)
{
	float* result = new float[N * N];
	int i;
	for (i = 0; i < N * N; ++i)
		result[i] = decreasing[i] - subtrahend[i];
	return result;
}

float* getI(int N)
{
	float* I = new float[N * N];
	int i;
	for (i = 0; i < N * N; ++i)
		I[i] = 0;
	for (i = 0; i < N; ++i)
		I[i * N + i] = 1;
	return I;
}

float* getR(float* I, float* B, float* A, int N)
{
	float* ABMult = matrixMultiplication(B, A, N);
	return matrixSubtraction(I, ABMult, N);
}
```

```
float* getInvertedA(float* I, float* R, float* B, int N, int M)
{
	float* prevRDeg, * curRDeg = NULL, * summ = I;
	matrixAddition(summ, R, N);
	int i;
	curRDeg = matrixMultiplication(R, R, N);
	matrixAddition(summ, curRDeg, N);
	prevRDeg = curRDeg;
	for (i = 2; i < M; ++i)
	{
		curRDeg = matrixMultiplication(R, prevRDeg, N);
		matrixAddition(summ, curRDeg, N);
		free(prevRDeg);
		prevRDeg = curRDeg;
	}
	free(curRDeg);
	float* AInverted = matrixMultiplication(summ, B, N);
	return AInverted;
}

void showAInverted(float* AInverted, int N)
{
	int i;
	for (i = 0; i < N * N; ++i)
	{
		printf("%.4f ", AInverted[i]);
		if (i % N == N - 1)
			printf("\n");
	}
}

int main(int argc, char** argv)
{
	if (argc != 4)
	{
		cout << "wrong number of arguments" << endl;
		return -1;
	}
	int N = atoi(argv[1]), M = atoi(argv[2]);
	ifstream in;
	in.open(argv[3]);
	if (!in)
	{
		cout << "couldn't open file" << endl;
		return -1;
	}
	float* A = getA(N, in);
	float* AT = getAT(A, N);
	float Aone = getAone(A, N);
	float Ainfinity = getAinfinity(A, N);
	float* B = getB(AT, N, Aone, Ainfinity);
	float* I = getI(N);
	float* R = getR(I, B, A, N);
	float* AInverted = getInvertedA(I, R, B, N, M);
	showAInverted(AInverted, N);
```

```
	free(A);
	free(AT);
	free(B);
	free(I);
	free(R);
	free(AInverted);
	return 0;
}
```

**Приложение 3.** Функция `matrixMultiplication` *с полуавтоматической векторизацией*

```
float* matrixMultiplication(float* A, float* B, int N)
{
	float* result = new float[N * N], * c, * b;
	int i, j, k;
	__m128 p, a;
	for (i = 0; i < N; ++i)
	{
		c = result + i * N;
		for (j = 0; j < N; j += 4)
			_mm_storeu_ps(c + j, _mm_setzero_ps());
		for (k = 0; k < N; ++k)
		{
			a = _mm_set1_ps(A[i * N + k]);
			b = B + k * N;
			for (j = 0; j < N; j += 4)
			{
				p = _mm_mul_ps(_mm_loadu_ps(b + j), a);
				_mm_storeu_ps(c + j, _mm_add_ps(_mm_loadu_ps(c + j),
p));
			}
		}
	}
	return result;
}
```

**Приложение 4.** Функция `matrixAddition` *с полуавтоматической векторизацией*

```
void matrixAddition(float* A, float* B, int N)
{
	int i;
	for (i = 0; i < N * N; i += 4)
		_mm_storeu_ps(A +
i,_mm_add_ps(_mm_loadu_ps(A+i),_mm_loadu_ps(B+i)));
}
```

**Приложение 5.** *Листинг оптимизированной программы с полуавтоматической векторизацией*

```
#include <iostream>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <fstream>
#include <xmmintrin.h>

using namespace std;

float* getA(int N, ifstream& in)
{
	float* A = new float[N * N];
	int i;
	for (i = 0; i < N * N; ++i)
		in >> A[i];
	return A;
}

float* getAT(float* A, int N)
{
	float* AT = new float[N * N];
	int i, j;
	for (i = 0; i < N; ++i)
		for (j = 0; j < N; ++j)
			AT[i * N + j] = A[j * N + i];
	return AT;
}

float getAone(float* A, int N)
{
	float max = 0, current = 0;
	int i;
	for (i = 0; i < N * N; ++i)
	{
		current += abs(A[i]);
		if (i % N == N - 1)
		{
			if (current > max)
				max = current;
			current = 0;
		}
	}
	return max;
}

float getAinfinity(float* A, int N)
{
	float max = 0, current = 0;
	int i;
	for (i = 0; i < N * N; ++i)
	{
		current += abs(A[i]);
```

```
            if (i % N == N - 1)
            {
                  if (current > max)
                        max = current;
                  current = 0;
            }
      }
      return max;
}

float* getB(float* AT, int N, float Aone, float Ainfinity)
{
      float* B = new float[N * N], AoneAndAinfinityMult = Aone * Ainfinity;
      int i;
      for (i = 0; i < N * N; ++i)
            B[i] = AT[i] / AoneAndAinfinityMult;
      return B;
}

float* matrixMultiplication(float* A, float* B, int N)
{
      float* result = new float[N * N], * c, * b;
      int i, j, k;
      __m128 p, a;
      for (i = 0; i < N; ++i)
      {
            c = result + i * N;
            for (j = 0; j < N; j +=4)
                  _mm_storeu_ps(c + j, _mm_setzero_ps());
            for (k = 0; k < N; ++k)
            {
                  a = _mm_set1_ps(A[i * N + k]);
                  b = B + k * N;
                  for (j = 0; j < N; j += 4)
                  {
                        p = _mm_mul_ps(_mm_loadu_ps(b + j),a);
                        _mm_storeu_ps(c + j, _mm_add_ps(_mm_loadu_ps(c +
j),p));
                  }
            }
      }
      return result;
}

void matrixAddition(float* A, float* B, int N)
{
      int i;
      for (i = 0; i < N * N; i += 4)
            _mm_storeu_ps(A +
i,_mm_add_ps(_mm_loadu_ps(A+i),_mm_loadu_ps(B+i)));
}

float* matrixSubtraction(float* decreasing, float* subtrahend, int N)
{
      float* result = new float[N * N];
      int i;
```

```
	for (i = 0; i < N * N; ++i)
		result[i] = decreasing[i] - subtrahend[i];
	return result;
}

float* getI(int N)
{
	float* I = new float[N * N];
	int i;
	for (i = 0; i < N * N; ++i)
		I[i] = 0;
	for (i = 0; i < N; ++i)
		I[i * N + i] = 1;
	return I;
}

float* getR(float* I, float* B, float* A, int N)
{
	float* ABMult = matrixMultiplication(B, A, N);
	return matrixSubtraction(I, ABMult, N);
}

float* getInvertedA(float* I, float* R, float* B, int N, int M)
{
	float* prevRDeg, * curRDeg = NULL, * summ = I;
	matrixAddition(summ, R, N);
	int i;
	curRDeg = matrixMultiplication(R, R, N);
	matrixAddition(summ, curRDeg, N);
	prevRDeg = curRDeg;
	for (i = 2; i < M; ++i)
	{
		curRDeg = matrixMultiplication(R, prevRDeg, N);
		matrixAddition(summ, curRDeg, N);
		free(prevRDeg);
		prevRDeg = curRDeg;
	}
	free(curRDeg);
	float* AInverted = matrixMultiplication(summ, B, N);
	return AInverted;
}

void showAInverted(float* AInverted, int N)
{
	int i;
	for (i = 0; i < N * N; ++i)
	{
		printf("%.4f ", AInverted[i]);
		if (i % N == N - 1)
			printf("\n");
	}
}

int main(int argc, char** argv)
{
	if (argc != 4)
```

```
	{
		cout << "wrong number of arguments" << endl;
		return -1;
	}
	int N = atoi(argv[1]), M = atoi(argv[2]);
	ifstream in;
	in.open(argv[3]);
	if (!in)
	{
		cout << "couldn't open file" << endl;
		return -1;
	}
	float* A = getA(N, in);
	float* AT = getAT(A, N);
	float Aone = getAone(A, N);
	float Ainfinity = getAinfinity(A, N);
	float* B = getB(AT, N, Aone, Ainfinity);
	float* I = getI(N);
	float* R = getR(I, B, A, N);
	float* AInverted = getInvertedA(I, R, B, N, M);
	showAInverted(AInverted, N);
	free(A);
	free(AT);
	free(B);
	free(I);
	free(R);
	free(AInverted);
	return 0;
}
```

## **Приложение 6.** *Листинг программы с матричными операциями, выполненными с помощью библиотеки BLAS*

```
#include <math.h>
#include <iostream>
#include <fstream>
#include <stdio.h>
#include <stdlib.h>
#include <cblas.h>

using namespace std;

void showAInverted(float* AInverted, int N);

float* getA(int N, ifstream& in)
{
	float* A = new float[N * N];
	int i;
	for (i = 0; i < N * N; ++i)
		in >> A[i];
	return A;
}

float getAone(float* A, int N)
```

```
{
	float max = 0, current = 0;
	int i;
	for (i = 0; i < N * N; ++i)
	{
		current += abs(A[i]);
		if (i % N == N - 1)
		{
			if (current > max)
				max = current;
			current = 0;
		}
	}
	return max;
}

float getAinfinity(float* A, int N)
{
	float max = 0, current = 0;
	int i;
	for (i = 0; i < N * N; ++i)
	{
		current += abs(A[i]);
		if (i % N == N - 1)
		{
			if (current > max)
				max = current;
			current = 0;
		}
	}
	return max;
}

float* getB(float* A, float* I, int N, float Aone, float Ainfinity)
{
	float* B = new float[N * N], AoneAndAinfinityMult = Aone * Ainfinity;
	cblas_sgemm(CblasRowMajor, CblasTrans, CblasNoTrans, N, N, N, (float)(1
/ AoneAndAinfinityMult), A, N, I, N, 0, B, N);
	return B;
}

float* getI(int N)
{
	float* I = new float[N * N];
	int i;
	for (i = 0; i < N * N; ++i)
		I[i] = 0;
	for (i = 0; i < N; ++i)
		I[i * N + i] = 1;
	return I;
}

float* getR(float* I, float* B, float* A, int N)
{
	float* R = getI(N);
```

```
	cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, N, N, N, -1, B,
N, A, N, 1, R, N);
	return R;
}

float* getInvertedA(float* I, float* R, float* B, int N, int M)
{
	float* nextRDeg = new float[N * N], * prevRDeg = new float[N * N], *
summ = getI(N);
	cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, N, N, N, 1, R,
N, I, N, 1, summ, N);
	int i;
	cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, N, N, N, 1, R,
N, R, N, 0, prevRDeg, N);
	cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, N, N, N, 1,
prevRDeg, N, I, N, 1, summ, N);
	for (i = 2; i < M; ++i)
	{
		cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, N, N, N,
1, R, N, prevRDeg, N, 0, nextRDeg, N);
		cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, N, N, N,
1, nextRDeg, N, I, N, 1, summ, N);
		free(prevRDeg);
		prevRDeg = nextRDeg;
		nextRDeg = new float[N * N];
	}
	free(prevRDeg);
	free(nextRDeg);
	float* AInverted = new float[N * N];
	cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, N, N, N, 1,
summ, N, B, N, 0, AInverted, N);
	free(summ);
	return AInverted;
}

void showAInverted(float* AInverted, int N)
{
	int i;
	for (i = 0; i < N * N; ++i)
	{
		printf("%.4f ", AInverted[i]);
		if (i % N == N - 1)
			printf("\n");
	}
}

int main(int argc, char** argv)
{
	if (argc != 4)
	{
		cout << "wrong number of arguments" << endl;
		return -1;
	}
	int N = atoi(argv[1]), M = atoi(argv[2]);
	ifstream in;
	in.open(argv[3]);
```

```
	if (!in)
	{
		cout << "couldn't open file" << endl;
		return -1;
	}
	float* A = getA(N, in);
	float Aone = getAone(A, N);
	float Ainfinity = getAinfinity(A, N);
	float* I = getI(N);
	float* B = getB(A, I, N, Aone, Ainfinity);
	float* R = getR(I, B, A, N);
	float* AInverted = getInvertedA(I, R, B, N, M);
	showAInverted(AInverted, N);
	free(A);
	free(B);
	free(I);
	free(R);
	free(AInverted);
	return 0;
}
```