**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

«ВЕКТОРИЗАЦИЯ ВЫЧИСЛЕНИЙ»

студентки 2 курса, группы 21207

**Черновской Яны Тихоновны**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

А.Ю. Власенко

Новосибирск 2022

# 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 алгоритм из задания:
   - вариант без ручной 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

следующее:

- Титульный лист.
- Цель лабораторной работы.
- Результаты измерения времени работы трех программ.
- Полный компилируемый листинг реализованных программ и
- команды для их компиляции.
- Вывод по результатам лабораторной работы.

ВАРИАНТЫ ЗАДАНИЙ

1. Алгоритм обращения матрицы $A$ размером $N \times N$ с помощью разложения в ряд: $A^{-1} = (I + R + R^2 + \ldots)B$, 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 (на главной диагонали – единицы, остальные – нули). Параметры алгоритма: N – размер матрицы, M – число членов ряда (число итераций цикла в реализации алгоритма).

# ОПИСАНИЕ РАБОТЫ

1. Был написан алгоритм для обращения матрицы без использования векторизации.
   Правильность работы программы была проверена на маленькой матрице 4x4(N= 4, M = 1000)

```
6 0 3 3
2 0 0 8
3 9 8 6
7 6 3 6
0.0906145 -0.0631067 -0.0776697 0.116504
-0.170442 -0.0598704 0.00323585 0.161811
0.174757 -0.0145631 0.135922 -0.203882
-0.0226536 0.140776 0.0194175 -0.029126
```

2. Далее были подставлены значения из условия задания (N= 2048, M = 10)

```
evmpu@comrade:~/21207/Chernovskaya/lab7$ g++ default.cpp -o default.out
evmpu@comrade:~/21207/Chernovskaya/lab7$ ./default.out
TIME: 400.702
```

3. Была изменена функция умножения матриц - были добавлены ассемблерные вставки.

   Время работы программы уменьшилось почти в 4 раза

```
evmpu@comrade:~/21207/Chernovskaya/lab7$ g++ opt1.cpp -o opt1.out
evmpu@comrade:~/21207/Chernovskaya/lab7$ ./opt1.out
TIME: 96.269
```

4. Далее ускорение работы произведение матриц было осуществлено за счет использования библиотеки BLAS

```
evmpu@comrade:~/21207/Chernovskaya/lab7$ ./withblas1.out
TIME : 9.698176 sec
```

# ЗАКЛЮЧЕНИЕ

В ходе работы были изучены SIMD-расширений архитектуры x86/x86-64, способы использования SIMD-расширений в программах на языке Си, а также были получены навыки использования SIMD-расширений.

По результатам измерения времени, можно прийти к выводу, что лучше всего использовать готовую библиотеку BLAS. Однако ручная векторизация может также уменьшить время работы программы в 4 раза.

# ПРИЛОЖЕНИЕ

**Приложение 1.1 Полный листинг программы без ручной векторизации**

```
1.  #include <iostream>
2.  #include <cmath>
3.  #include <ctime>
4.  #include <limits>
5.
6.  const int N = 2048;
7.  const int M = 10;
8.
9.  ////////////////////////////////////////////////////////////////////
10.
11. float GetMultMaxSums(float *matrix);
12. void DefineB(float *matrix, float *B);
13. void DefineI(float *I);
14.
15. void Mult(float *matrix1, float *matrix2, float *result);
16. void Additional(float *matrix1, float *matrix2, float *result);
17. void Subtraction(float *matrix1, float *matrix2, float *result);
18.
19. void Inverse(float *matrix, float *result);
20.
21. void PrintMatrix(const float *matrix);
22. void CopyMatrix(float *matrixInput, float *matrixOutput);
23.
24. ////////////////////////////////////////////////////////////////////
25.
26. int main()
27. {
28.     srand(time(0));
29.
30.     auto *matrix = new float[N * N];
31.     auto *result = new float[N * N];
32.
33.     for (int i = 0; i < N * N; ++i)
34.     {
35.         matrix[i] = rand() % 10;
36.         result[i] = 0;
37.     }
38.
39.     // PrintMatrix(matrix);
40.     clock_t start = clock();
41.     Inverse(matrix, result);
42.     clock_t end = clock();
43.     // PrintMatrix(result);
44.     std::cout << "TIME: " << (((double)end - start) / CLOCKS_PER_SEC) << std::endl;
45.
46.     return EXIT_SUCCESS;
47. }
48. ////////////////////////////////////////////////////////////////////
49.
50. float GetMultMaxSums(float *matrix)
51. {
52.     float currentSumRow = 0;
53.     float currentSumColumn = 0;
54.
55.     float maxSumRow = std::numeric_limits<float>::min();
56.     float maxSumColumn = std::numeric_limits<float>::min();
57.
58.     for (int i = 0; i < N; ++i)
```

```
59.     {
60.         currentSumColumn = 0;
61.         currentSumRow = 0;
62.
63.         for (int j = 0; j < N; ++j)
64.         {
65.             currentSumRow += matrix[N * i + j];
66.             currentSumColumn += matrix[j * N + i];
67.         }
68.
69.         if (currentSumRow > maxSumRow) maxSumRow = currentSumRow;
70.         if (currentSumColumn > maxSumColumn) maxSumColumn = currentSumColumn;
71.
72.     }
73.
74.     return maxSumColumn * maxSumRow;
75. }
76.
77. void DefineB(float *matrix, float *B)
78. {
79.     float divider = GetMultMaxSums(matrix);
80.     for (int i = 0; i < N; ++i)
81.         for (int j = 0; j < N; ++j)
82.             B[N * i + j] = matrix[j * N + i] / divider;
83. }
84.
85. void DefineI(float *I)
86. {
87.     for (int i = 0; i < N; ++i)
88.         for (int j = 0; j < N; ++j)
89.             (i == j) ? I[N * i + j] = 1 : I[N * i + j] = 0;
90. }
91.
92. ////////////////////////////////////////////////////////////////////////////////
93.
94. void Mult(float *matrix1, float *matrix2, float *result)
95. {
96.     for (int i = 0; i < N * N; ++i)
97.         result[i] = 0;
98.
99.     for (int i = 0; i < N; ++i)
100.         for (int j = 0; j < N; ++j)
101.             for (int k = 0; k < N; ++k)
102.                 result[i * N + k] += matrix1[i * N + j] * matrix2[j * N + k];
103.}
104.
105.////////////////////////////////////////////////////////////////////////////////
106.
107.void Subtraction(float *matrix1, float *matrix2, float *result)
108.{
109.     for (int i = 0; i < N * N; i++)
110.         result[i] = matrix1[i] - matrix2[i];
111.}
112.
113.void Additional(float *matrix1, float *matrix2, float *result)
114.{
115.     for (int i = 0; i < N * N; i++)
116.         result[i] = matrix1[i] + matrix2[i];
117.}
118.
119.void Inverse(float *matrix, float *result)
120.{
121.     auto *R = new float[N * N];
122.     auto *B = new float[N * N];
```

```
123.    auto *I = new float[N * N];
124.    auto *tmp = new float[N * N];
125.
126.    DefineB(matrix, B);
127.    DefineI(I);
128.
129.    Mult(B, matrix, tmp);   // tmp = BA
130.    Subtraction(I, tmp, R); // R = I - BA
131.    Additional(I, R, tmp);  // tmp = I + R
132.    CopyMatrix(R, result);  // result = R
133.
134.    for (int i = 2; i < M; ++i)
135.    {
136.        if (i % 2 == 0)
137.        {
138.            Mult(R, result, I);    // I = result * R
139.            Additional(tmp, I, tmp); // tmp = tmp + result
140.        }
141.        else
142.        {
143.            Mult(R, I, result);    // result = I * R
144.            Additional(tmp, result, tmp); // tmp = tmp + I
145.        }
146.    }
147.
148.    Mult(tmp, B, result); // result = tmp * B
149.
150.    delete[] I;
151.    delete[] R;
152.    delete[] B;
153.    delete[] tmp;
154.}
155.
156.void PrintMatrix(const float *matrix)
157.{
158.    for (int i = 0; i < N; i++)
159.    {
160.        for (int j = 0; j < N; j++)
161.            std::cout << matrix[N * i + j] << " ";
162.
163.        std::cout << std::endl;
164.    }
165.}
166.
167.void CopyMatrix(float *matrixInput, float *matrixOutput)
168.{
169.    for (int i = 0; i < N * N; i++)
170.        matrixOutput[i] = matrixInput[i];
171.}
```

**Приложение 1.2 Полный листинг программы с ручной векторизацией**

```
1.  #include <iostream>
2.  #include <cmath>
3.  #include <ctime>
4.  #include <limits>
5.
6.  const int N = 2048;
7.  const int M = 10;
8.
9.  ////////////////////////////////////////////////////////////////
10.
11. float GetMultMaxSums(float *matrix);
12. void DefineB(float *matrix, float *B);
13. void DefineI(float *I);
14.
15. void Mult(float *matrix1, float *matrix2, float *result);
16. void Additional(float *matrix1, float *matrix2, float *result);
17. void Subtraction(float *matrix1, float *matrix2, float *result);
18.
19. void Inverse(float *matrix, float *result);
20.
21. void PrintMatrix(const float *matrix);
22. void CopyMatrix(float *matrixInput, float *matrixOutput);
23. void ConvertToMatrix(float *matrix1, float *matrix2, float **matrix1_, float **matrix2_);
24.
25. ////////////////////////////////////////////////////////////////
26.
27. struct Vector
28. {
29.     float x, y, z, w;
30. };
31.
32. ////////////////////////////////////////////////////////////////
33.
34. int main()
35. {
36.     srand(time(0));
37.
38.     auto *matrix = new float[N * N];
39.     auto *result = new float[N * N];
40.
41.     for (int i = 0; i < N * N; ++i)
42.     {
43.         matrix[i] = rand() % 10;
44.         result[i] = 0;
45.     }
46.
47.     // PrintMatrix(matrix);
48.     clock_t start = clock();
49.     Inverse(matrix, result);
50.     clock_t end = clock();
51.     // PrintMatrix(result);
52.     std::cout << "TIME: " << (((double)end - start) / CLOCKS_PER_SEC) << std::endl;
53.
54.     return EXIT_SUCCESS;
55. }
56.
57. ////////////////////////////////////////////////////////////////
58.
```

```
59. void Add_SSE(Vector *a, Vector *b, Vector *result)
60. {
61.     __asm__ volatile("mov %0, %%rax" ::"m"(a));
62.     __asm__ volatile("mov %0, %%rbx" ::"m"(b));
63.     __asm__ volatile("movaps (%rax), %xmm0");
64.     __asm__ volatile("movaps (%rbx), %xmm1");
65.     __asm__ volatile("addps %xmm1, %xmm0");
66.     __asm__ volatile("mov %0, %%rax" ::"m"(result));
67.     __asm__ volatile("movaps %xmm0, (%rax)");
68. }
69.
70. void Mul_SSE(Vector *a, Vector *b, Vector *result)
71. {
72.     __asm__ volatile("mov %0, %%rax" ::"m"(a));
73.     __asm__ volatile("mov %0, %%rbx" ::"m"(b));
74.     __asm__ volatile("movaps (%rax), %xmm0");
75.     __asm__ volatile("movaps (%rbx), %xmm1");
76.     __asm__ volatile("mulps %xmm1, %xmm0");
77.     __asm__ volatile("mov %0, %%rax" ::"m"(result));
78.     __asm__ volatile("movaps %xmm0, (%rax)");
79. }
80.
81. float VectorSum(float *arr)
82. {
83.     Vector sumVector = {0, 0, 0, 0};
84.     Vector *vector;
85.     vector = (Vector *)arr;
86.
87.     for (int i = 0; i < N / 4; i++, vector++)
88.         Add_SSE(vector, &sumVector, &sumVector);
89.
90.     return (sumVector.x + sumVector.y + sumVector.z + sumVector.w);
91. }
92.
93. ////////////////////////////////////////////////////////////////////
94.
95. float GetMultMaxSums(float *matrix)
96. {
97.     float currentSumRow = 0;
98.     float currentSumColumn = 0;
99.
100.    float maxSumRow = std::numeric_limits<float>::min();
101.    float maxSumColumn = std::numeric_limits<float>::min();
102.
103.    for (int i = 0; i < N; ++i)
104.    {
105.        currentSumColumn = 0;
106.        currentSumRow = 0;
107.
108.        for (int j = 0; j < N; ++j)
109.        {
110.            currentSumRow += matrix[N * i + j];
111.            currentSumColumn += matrix[j * N + i];
112.        }
113.
114.        if (currentSumRow > maxSumRow) maxSumRow = currentSumRow;
115.        if (currentSumColumn > maxSumColumn) maxSumColumn = currentSumColumn;
116.
117.    }
118.
119.    return maxSumColumn * maxSumRow;
120.}
121.
122.void DefineB(float *matrix, float *B)
```

```
123.{
124.    float divider = GetMultMaxSums(matrix);
125.    for (int i = 0; i < N; ++i)
126.        for (int j = 0; j < N; ++j)
127.            B[N * i + j] = matrix[j * N + i] / divider;
128.}
129.
130.void DefineI(float *I)
131.{
132.    for (int i = 0; i < N; ++i)
133.        for (int j = 0; j < N; ++j)
134.            (i == j) ? I[N * i + j] = 1 : I[N * i + j] = 0;
135.}
136.
137.////////////////////////////////////////////////////////////////////////////////
138.
139.void ConvertToMatrix(float *matrix1, float *matrix2, float **matrix1_, float **matrix2_)
140.{
141.
142.    for (int i = 0; i < N; i++)
143.    {
144.        matrix1_[i] = new float[N];
145.        matrix2_[i] = new float[N];
146.
147.        for (int j = 0; j < N; j++)
148.        {
149.            matrix1_[i][j] = matrix1[N * i + j];
150.            matrix2_[i][j] = matrix2[N * j + i]; // +transpose
151.        }
152.    }
153.}
154.
155.////////////////////////////////////////////////////////////////////////////////
156.
157.void Mult(float *matrix1, float *matrix2, float *result)
158.{
159.    for (int i = 0; i < N * N; ++i)
160.        result[i] = 0;
161.
162.    float **matrix1_ = new float *[N];
163.    float **matrix2_ = new float *[N];
164.    ConvertToMatrix(matrix1, matrix2, matrix1_, matrix2_);
165.
166.    auto *currentResult = new float[N * N];
167.    for (int i = 0; i < N; i++)
168.    {
169.        for (int j = 0; j < N; j++)
170.        {
171.            Vector *x, *y, *z;
172.            x = (Vector *)matrix1_[i];
173.            y = (Vector *)matrix2_[j];
174.            z = (Vector *)currentResult;
175.
176.            Mul_SSE(x, y, z);
177.            result[i * N + j] = VectorSum(currentResult);
178.        }
179.    }
180.
181.    delete[] currentResult;
182.}
183.
184.void Subtraction(float *matrix1, float *matrix2, float *result)
185.{
186.    for (int i = 0; i < N * N; i++)
```

```
187.        result[i] = matrix1[i] - matrix2[i];
188.}
189.
190.void Additional(float *matrix1, float *matrix2, float *result)
191.{
192.    for (int i = 0; i < N * N; i++)
193.        result[i] = matrix1[i] + matrix2[i];
194.}
195.
196.void Inverse(float *matrix, float *result)
197.{
198.    auto *R = new float[N * N];
199.    auto *B = new float[N * N];
200.    auto *I = new float[N * N];
201.    auto *tmp = new float[N * N];
202.
203.    DefineB(matrix, B);
204.    DefineI(I);
205.
206.    Mult(B, matrix, tmp);   // tmp = BA
207.    Subtraction(I, tmp, R); // R = I - BA
208.    Additional(I, R, tmp);  // tmp = I + R
209.    CopyMatrix(R, result);  // result = R
210.
211.    for (int i = 2; i < M; ++i)
212.    {
213.        if (i % 2 == 0)
214.        {
215.            Mult(R, result, I);    // I = result * R
216.            Additional(tmp, I, tmp); // tmp = tmp + result
217.        }
218.        else
219.        {
220.            Mult(R, I, result);    // result = I * R
221.            Additional(tmp, result, tmp); // tmp = tmp + I
222.        }
223.    }
224.
225.    Mult(tmp, B, result); // result = tmp * B
226.
227.    delete[] I;
228.    delete[] R;
229.    delete[] B;
230.    delete[] tmp;
231.}
232.
233.void PrintMatrix(const float *matrix)
234.{
235.    for (int i = 0; i < N; i++)
236.    {
237.        for (int j = 0; j < N; j++)
238.            std::cout << matrix[N * i + j] << " ";
239.
240.        std::cout << std::endl;
241.    }
242.}
243.
244.void CopyMatrix(float *matrixInput, float *matrixOutput)
245.{
246.    for (int i = 0; i < N * N; i++)
247.        matrixOutput[i] = matrixInput[i];
248.}
```

**Приложение 1.3 Полный листинг программы с векторизацией с помощью библиотеки BLAS**

```
#include "mkl.h"
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <float.h>

const int N = 2048;
const int M = 10;

//////////////////////////////////////////////////////////////////////

float GetMultMaxSums(float *matrix);
void DefineB(float *matrix, float *B);
void DefineI(float *I);

void Mult(float *matrix1, float *matrix2, float *result);
void Additional(float *matrix1, float *matrix2, float *result);
void Subtraction(float *matrix1, float *matrix2, float *result);

void Inverse(float *matrix, float *result);

//void PrintMatrix(const float *matrix);
void CopyMatrix(float *matrixInput, float *matrixOutput);

//////////////////////////////////////////////////////////////////////

int main()
{
    srand(time(0));

    float *matrix = (float *)calloc(N*N, sizeof(float));
    float *result = (float *)calloc(N*N, sizeof(float));

    for (int i = 0; i < N * N; ++i)
    {
        matrix[i] = rand() % 10;
        result[i] = 0;
    }

    // PrintMatrix(matrix);
    clock_t start = clock();
    Inverse(matrix, result);
    clock_t end = clock();
    // PrintMatrix(result);
    printf("TIME : %lf sec \n ", (double)(clock() - start) / CLOCKS_PER_SEC);

    return EXIT_SUCCESS;
}
//////////////////////////////////////////////////////////////////////

float GetMultMaxSums(float *matrix)
{
    float currentSumRow = 0;
    float currentSumColumn = 0;

    float maxSumRow = FLT_MIN;
    float maxSumColumn = FLT_MIN;

    for (int i = 0; i < N; ++i)
    {
        currentSumColumn = 0;
```

```
63.         currentSumRow = 0;
64.
65.         for (int j = 0; j < N; ++j)
66.         {
67.             currentSumRow += matrix[N * i + j];
68.             currentSumColumn += matrix[j * N + i];
69.         }
70.
71.         if (currentSumRow > maxSumRow) maxSumRow = currentSumRow;
72.         if (currentSumColumn > maxSumColumn) maxSumColumn = currentSumColumn;
73.
74.     }
75.
76.     return maxSumColumn * maxSumRow;
77. }
78.
79. void DefineB(float *matrix, float *B)
80. {
81.     float divider = GetMultMaxSums(matrix);
82.     for (int i = 0; i < N; ++i)
83.         for (int j = 0; j < N; ++j)
84.             B[N * i + j] = matrix[j * N + i] / divider;
85. }
86.
87. void DefineI(float *I)
88. {
89.     for (int i = 0; i < N; ++i)
90.         for (int j = 0; j < N; ++j)
91.             (i == j) ? I[N * i + j] = 1 : I[N * i + j] = 0;
92. }
93.
94. ////////////////////////////////////////////////////////////////////////////////
95.
96. void Mult(float *matrix1, float *matrix2, float *result)
97. {
98.       cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, N, N, N, 1.0, matrix1, N, matrix2, N, 0.0,
    result, N);
99. }
100.
101.////////////////////////////////////////////////////////////////////////////////
102.
103.void Subtraction(float *matrix1, float *matrix2, float *result)
104.{
105.    cblas_saxpy(N*N, -1.0, matrix2, 1, matrix1, 1);
106.    cblas_scopy(N * N, matrix1, 1, result, 1);
107.}
108.
109.void Additional(float *matrix1, float *matrix2, float *result)
110.{
111.    for (int i = 0; i < N * N; i++)
112.        result[i] = matrix1[i] + matrix2[i];
113.}
114.
115.void Inverse(float *matrix, float *result)
116.{
117.    float *R = (float *)calloc(N*N, sizeof(float));
118.    float *B = (float *)calloc(N*N, sizeof(float));
119.    float *I = (float *)calloc(N*N, sizeof(float));
120.    float *tmp = (float *)calloc(N*N, sizeof(float));
121.
122.    DefineB(matrix, B);
123.    DefineI(I);
124.
125.    Mult(B, matrix, tmp);   // tmp = BA
```

```
126.    Subtraction(I, tmp, R); // R = I - BA
127.    Additional(I, R, tmp);  // tmp = I + R
128.    CopyMatrix(R, result);  // result = R
129.
130.    for (int i = 2; i < M; ++i)
131.    {
132.        if (i % 2 == 0)
133.        {
134.            Mult(R, result, I);    // I = result * R
135.            Additional(tmp, I, tmp); // tmp = tmp + result
136.        }
137.        else
138.        {
139.            Mult(R, I, result);    // result = I * R
140.            Additional(tmp, result, tmp); // tmp = tmp + I
141.        }
142.    }
143.
144.    Mult(tmp, B, result); // result = tmp * B
145.
146.    free(I);
147.    free(R);
148.    free(B);
149.    free(tmp);
150.}
151.
152./*void PrintMatrix(const float *matrix)
153.{
154.    for (int i = 0; i < N; i++)
155.    {
156.        for (int j = 0; j < N; j++)
157.            std::cout << matrix[N * i + j] << " ";
158.
159.        std::cout << std::endl;
160.    }
161.}*/
162.
163.void CopyMatrix(float *matrixInput, float *matrixOutput)
164.{
165.    for (int i = 0; i < N * N; i++)
166.        matrixOutput[i] = matrixInput[i];
167.}
```