**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра вычислительной техники**

**ОТЧЕТ**

**по лабораторной работе № 2**

**по дисциплине «Параллельные алгоритмы и системы»**

**Тема: Умножение матриц**

|  |  |  |
| --- | --- | --- |
| Студент гр. 9307 |  | Казачук М.О. |
| Преподаватель |  | Пазников А.А. |

Оглавление

[Цель работы 3](#_Toc131493689)

[Основные положения 3](#_Toc131493690)

[1. Без оптимизаций 3](#_Toc131493691)

[2. Оптимизация переменных 3](#_Toc131493692)

[3. Изменение порядка вычислений 4](#_Toc131493693)

[4. Распараллеливание 4](#_Toc131493694)

[5. Векторизация 5](#_Toc131493695)

[6. Использования оптимизаторов компилятора 6](#_Toc131493696)

[Выводы 6](#_Toc131493697)

[Последняя версия кода 7](#_Toc131493698)

# Цель работы

Разработать и оптимизировать алгоритм умножения матриц.

# Основные положения

Используются квадратные матрицы 1024х1024 элементов.

# Без оптимизаций

Наивная реализация умножения «в лоб»:

for (int i = 0; i < n; ++i)

    {

        float \* c = C + i \* n;

        int in = i\*n;

        for (int j = 0; j < n; ++j)

        {

            c[j] = 0;

            for (int k = 0; k < n; ++k)

            {

                c[j] += A[in + k] \* B[k\*n];

            }

        }

    }

|  |  |  |
| --- | --- | --- |
| № | Затраченное время | Ускорение (отн. 1) |
| 1. | 5.495 | 1 |

# Оптимизация переменных

Вынесем некоторые неизменяемые переменные за пределы своих циклов:

for (int i = 0; i < n; ++i)

    {

        float \* c = C + i \* n;

        for (int j = 0; j < n; ++j)

            c[j] = 0;

        for (int k = 0; k < n; ++k)

        {

            const float \* b = B + k \* n;

            float a = A[i\*n + k];

            for (int j = 0; j < n; ++j)

                c[j] += a \* b[j];

        }

}

|  |  |  |
| --- | --- | --- |
| № | Затраченное время | Ускорение (отн. 1) |
| 1. | 5.495 | 1 |
| 2. | 5.124 | 1.003 |

Ожидаемо, это не принесло большого прироста.

# Изменение порядка вычислений

Проходить по столбцам матрицы А не очень эффективно в плане кэш промахов. Данные в соседних ячейках столбца расположены довольно далеко в физической памяти.

    for (int i = 0; i < n; ++i)

    {

        float \* c = C + i \* n;

        for (int j = 0; j < n; ++j)

            c[j] = 0;

        for (int k = 0; k < n; ++k)

        {

            const float \* b = B + k \* n;

            float a = A[i\*n + k];

            for (int j = 0; j < n; ++j)

                c[j] += a \* b[j];

        }

    }

Так мы оптимизируем также и локальность данных, к которым обращаемся.

|  |  |  |
| --- | --- | --- |
| № | Затраченное время | Ускорение (отн. 1) |
| 1. | 5.495 | 1 |
| 2. | 5.124 | 1.003 |
| 3. | 4.804 | 1.143 |

Также значительно сократилось количество кэш промахов.

# Распараллеливание

Умножение матриц очень хорошо поддается параллельным вычислениям.

    #pragma omp parallel for schedule(static)

    for (int i = 0; i < n; ++i)

    {

        float \* c = C + i \* n;

        for (int j = 0; j < n; ++j)

            c[j] = 0;

        for (int k = 0; k < n; ++k)

        {

            const float \* b = B + k \* n;

            float a = A[i\*n + k];

            for (int j = 0; j < n; ++j)

                c[j] += a \* b[j];

        }

    }

}

|  |  |  |
| --- | --- | --- |
| № | Затраченное время | Ускорение (отн. 1) |
| 1. | 5.495 | 1 |
| 2. | 5.124 | 1.003 |
| 3. | 4.804 | 1.143 |
| 4. | 3.149 | 1.744 |

Не очень высокие результаты можно объяснить не очень мощной машиной с двумя ядрами, т.е. 4 логических процессора. Также далеко не кратное повышение производительности может быть из-за не успевающей общей шине данных.

# Векторизация

Так как данные и так расположены последовательно, векторизация поможет обрабатывать их сразу пачкой по (256/sizeof(float))=8. Также применим ручную развертку цикла.

  #pragma omp parallel for schedule(static)

    for (int i = 0; i < n; ++i)

    {

        float \* c = C + i \* n;

        for (int j = 0; j < n; j += 8)

            \_mm256\_storeu\_ps(c + j + 0, \_mm256\_setzero\_ps());

        for (int k = 0; k < n; ++k)

        {

            const float \* b = B + k \* n;

            \_\_m256 a = \_mm256\_set1\_ps(A[i\*n + k]);

            for (int j = 0; j < n; j += 16)

            {

                \_mm256\_storeu\_ps(c + j + 0, \_mm256\_fmadd\_ps(a,

                    \_mm256\_loadu\_ps(b + j + 0), \_mm256\_loadu\_ps(c + j + 0)));

                \_mm256\_storeu\_ps(c + j + 8, \_mm256\_fmadd\_ps(a,

                    \_mm256\_loadu\_ps(b + j + 8), \_mm256\_loadu\_ps(c + j + 8)));

            }

        }

    }

|  |  |  |
| --- | --- | --- |
| № | Затраченное время | Ускорение (отн. 1) |
| 1. | 5.495 | 1 |
| 2. | 5.124 | 1.003 |
| 3. | 4.804 | 1.143 |
| 4. | 3.149 | 1.744 |
| 5. | 2.176 | 2.525 |

# Использования оптимизаторов компилятора

В MinGW встроены некоторые оптимизаторы, которые можно подключить с помощью некоторых флагов. Флаг -O2 применяет набор эффективных оптимизаторов. Флаг -О3 включает О2 и некоторые дополнительные, которые не всегда ведут к увеличению производительности. Флаг -ffast-math ускоряет некоторые математические вычисления путем некоторой потери точности.

|  |  |  |
| --- | --- | --- |
| № | Затраченное время | Ускорение (отн. 1) |
| 1. | 5.495 | 1 |
| 2. | 5.124 | 1.003 |
| 3. | 4.804 | 1.143 |
| 4. | 3.149 | 1.744 |
| 5. | 2.176 | 2.525 |
| 6.1 | 1.398 | 3.931 |
| 6.2 | 1.435 | 3.829 |
| 6.3 | 1.391 | 3.950 |

Флаг -О3 не дал прироста в сравнении с О2, также как и -ffast-math. Скорее всего, последний не применяется в операциях умножения. Или как минимум в связке с векторными инструкциями.

# Выводы

|  |  |  |
| --- | --- | --- |
| № | Затраченное время | Ускорение (отн. 1) |
| 1. | 5.495 | 1 |
| 2. | 5.124 | 1.003 |
| 3. | 4.804 | 1.143 |
| 4. | 3.149 | 1.744 |
| 5. | 2.176 | 2.525 |
| 6.1 | 1.398 | 3.931 |
| 6.2 | 1.435 | 3.829 |
| 6.3 | 1.391 | 3.950 |

1 – наивная реализация

2 – оптимизация переменных

3 – Изменение порядка вычислений

4 – Использование дополнительных потоков

5 – Векторизация

6 – Оптимизация с помощью флагов компилятора

Удалось ускорить наивную реализацию почти в 4 раза. Наибольший прирост, не считая компилятора, дало распараллеливание. На более производительных компьютерах этот метод даст даже более значимое влияние.

Стандартный подход – вычисление с помощью GPU формально и есть использование большого числа потоков.

# Последняя версия кода

struct Matrix

{

    float \*p;

    int n;

    Matrix(int size) : n(size), p((float\*)\_mm\_malloc(size \* 4, 64)) {}

    ~Matrix() { \_mm\_free(p); }

};

void init(Matrix &mx)

{

    for (int i = 0; i < mx.n; ++i)

        mx.p[i] = float(rand());

}

void test(func f, const std::string & desc, const Matrix & a, const Matrix & b)

{

    Matrix res(n\*n);

    double t = 0;

    int n = 0;

    while(t < TIME)

    {

        bench\_start();

        f(a.p, b.p, res.p, n);

        t += bench\_measure();

        n++;

    }

    std::cout << desc << " : " << std::setprecision(3) << std::fixed << " t = " << t/n\*1000.0f << " msec." << std::endl;

}

int main(int argc, char\* argv[])

{

    Matrix a(n\*n), b(n\*n), c(n\*n);

    init(a);

    init(b);

    test(func4, "func4", a, b);

    return 0;

}

void func4(const float \* A, const float \* B, float \* C, int n)

{

    #pragma omp parallel for schedule(static)

    for (int i = 0; i < n; ++i)

    {

        float \* c = C + i \* n;

        for (int j = 0; j < n; j += 8)

            \_mm256\_storeu\_ps(c + j + 0, \_mm256\_setzero\_ps());

        for (int k = 0; k < n; ++k)

        {

            const float \* b = B + k \* n;

            \_\_m256 a = \_mm256\_set1\_ps(A[i\*n + k]);

            for (int j = 0; j < n; j += 16)

            {

                \_mm256\_storeu\_ps(c + j + 0, \_mm256\_fmadd\_ps(a,

                    \_mm256\_loadu\_ps(b + j + 0), \_mm256\_loadu\_ps(c + j + 0)));

                \_mm256\_storeu\_ps(c + j + 8, \_mm256\_fmadd\_ps(a,

                    \_mm256\_loadu\_ps(b + j + 8), \_mm256\_loadu\_ps(c + j + 8)));

            }

        }

    }

}