Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Лабораторная работа № 2
по дисциплине ”Анализ алгоритмов”

Тема Сравнение алгоритмов матричного умножения

Студент Калашников С.Д.

Группа ИУ7-53Б

Преподаватель Волкова Л.Л. Строганов Ю.В.

Москва, 2022

СОДЕРЖАНИЕ

Введение

Матрицы — крайне мощный инструмент, используемый в каждой точной науке: физике, математике, прораммировании и т.д. Они позволяют выполнять операции, требующие большого количества входных параметров и сложные в вычислении, с относительной легкостью. В связи с этим встает вопрос об оптимизации основных алгоритмов, связанных с матрицами: сложение, умножение, транспонирование и т.п. В данной работе будет рассмотрена оптимизация алгоритмов матричного умножения.

Целью данной лабораторной работы является анализ трудоемкости реализаций алгоритмов матричного умножения.

Для достижения поставленной цели требуется решить ряд задач:

1) изучить алгоритмы умножения матриц: классический, Винограда, оптимизированный Винограда;
2) провести сравнительный анализ трудоёмкости алгоритмов на основе теоретических расчетов;
3) реализовать каждый из трех алгоритмов;
4) провести замеры процессорного времени для каждой из реализаций алгоритмов;
5) выполнить анализ полученных результатов;
6) по итогам работы составить отчет.

1. Аналитическая часть

1.1 Матрица

Пусть есть два конечных множества.

- Номера строк: $M = 1, 2, ..., m$.
- Номера столбцов: $N = 1, 2, ..., n$.

Где m и n — натуральные числа. Тогда матрицей A размера m на n называется структура вида:

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}; \tag{1.1}$$

где элемент матрицы a_{ij} находится на пересечении i-й строки и j-го столбца. При этом количество элементов матрицы равно $m * n$.

Можно выделить следующие операции над матрицами:

1) сложение матриц одинакового размера;
2) вычитание матриц одинакового размера;
3) умножение матриц в случае, если количество столбцов первой матрицы равно количеству строк второй матрицы. В итоговой матрице количество строк будет, как у первой матрицы, а столбцов – как у второй.

Замечание: операция умножения матриц не коммутативна – если *A* и *B* – квадратные матрицы, а *C* – результат их перемножения, то произведение *AB* и *BA* дадут разный результат *C*.

1.2 Описание алгоритмов

В этом разделе будут рассмотрены следующие алгоритмы матричного умножения: классический, Винограда, оптимизированный Винограда.

1.2.1 Классический алгоритм умножения матриц

Пусть даны две матрицы

$$A_{lm} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{l1} & a_{l2} & \dots & a_{lm} \end{pmatrix}, \quad B_{mn} = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \dots & b_{mn} \end{pmatrix}, \tag{1.2}$$

тогда матрица C

$$C_{ln} = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{l1} & c_{l2} & \dots & c_{ln} \end{pmatrix}, \tag{1.3}$$

где

$$c_{ij} = \sum_{r=1}^{m} a_{ir} b_{rj} \quad (i = \overline{1,l}; j = \overline{1,n}) \tag{1.4}$$

будет называться произведением матриц A и B.

Стандартный алгоритм реализует данную формулу.

1.2.2 Алгоритм Винограда

Алгоритм Винограда – алгоритм умножения квадратных матриц. Начальная версия имела асимптотическую сложность алгоритма примерно $O(n^{2,3755})$, где n – размер стороны матрицы, но после доработки он стал обладать лучшей асимптотикой среди всех алгоритмов умножения матриц.

Рассмотрим два вектора $U = (u_1, u_2, u_3, u_4)$ и $W = (w_1, w_2, w_3, w_4)$. Их скалярное произведение равно: $U \cdot W = u_1w_1 + u_2w_2 + u_3w_3 + u_4w_4$, что эквивалентно (1.5):

$$V \cdot W = (u_1 + w_2)(u_2 + w_1) + (u_3 + w_4)(u_4 + w_3) - u_1u_2 - u_3u_4 - w_1w_2 - w_3w_4. \tag{1.5}$$

За счёт предварительной обработки данных можно получить прирост производительности: несмотря на то, что полученное выражение требует большего количества операций, чем стандартное умножение матриц, выражение в правой части равенства можно вычислить заранее и запомнить для каждой строки первой матрицы и каждого столбца второй матрицы. Это позволит выполнить лишь два умножения и пять сложений, при учёте, что потом будет сложено только с двумя предварительно посчитанными суммами соседних элементов текущих строк и столбцов. Операция сложения выполняется быстрее, поэтому на практике алгоритм должен работать быстрее обычного алгоритма перемножения матриц.

Стоит упомянуть, что при нечётном значении размера матрицы нужно дополнительно добавить произведения крайних элементов соответствующих строк и столбцов.

1.2.3 Оптимизация алгоритма Винограда

При программной реализации рассмотренного выше алгоритма Винограда можно провести оптимизации.

1. Операции сложения и вычитания с присваиванием следует реализовывать при помощи соответствующего оператора $+=$ или $-=$ (при наличии данных операторов в выбранном языке программирования).

2. Операцию умножения на 2 программно эффективнее реализовывать как побитовый сдвиг влево на 1.

3. Некоторое слагаемые можно предвычислять заранее.

2. Конструкторская часть

2.1 Трудоемкость алгоритмов

Введем модель трудоемкости для оценки алгоритмов.

1. Операции из списка (2.1) имеют трудоёмкость равную 1:

$$+, -, /, *, \%, =, + =, - =, * =, / =, \% =, ==$$

$$! =, <, >, <=, >=, [], ++, -- \tag{2.1}$$

2. Трудоёмкость оператора выбора if условие then A else B рассчитывается, как (2.2):

$$f_{if} = f_{\text{условия}} + \begin{cases} f_A, & \text{если условие выполняется,} \\ f_B, & \text{иначе.} \end{cases} \tag{2.2}$$

3. Трудоёмкость цикла рассчитывается, как (2.3):

$$f_{for} = f_{\text{инициализации}} + f_{\text{сравнения}} + N(f_{\text{тела}} + f_{\text{инкремент}} + f_{\text{сравнения}}) \tag{2.3}$$

4. Трудоёмкость вызова функции равна 0.

Далее будет приведены оценки трудоемкости алгоритмов.

2.1.1 Стандартный алгоритм умножения матриц

Для стандартного алгоритма умножения матриц трудоемкость будет слагаться из трех пунктов.

- Внешнего цикла по $i \in [1..M]$, трудоёмкость которого: $f = 2 + M \cdot (2 + f_{body_j})$.
- Цикла по $j \in [1..N]$, трудоёмкость которого: $f = 2 + N \cdot (2 + 2 + f_{body_k})$.
- Цикла по $k \in [1..K]$, трудоёмкость которого: $f = 2 + 13K$.

Поскольку трудоемкость стандартного алгоритма равна трудоемкости внешнего цикла, то:

$$f_{standard} = 2 + M \cdot (4 + N \cdot (4 + 13K)) = 2 + 4M + 4MN + 13MNK \quad (2.4)$$

$$\approx 13MNK \quad (2.5)$$

2.1.2 Алгоритм Винограда

Чтобы вычислить трудоемкость алгоритма Винограда, нужно учесть следующие пункты.

- Создания и инициализации массивов a_{tmp} и b_{tmp}, трудоёмкость которых (2.6):

$$f_{init} = M + N; \quad (2.6)$$

- Заполнения массива a_{tmp}, трудоёмкость которого (2.7):

$$f_{a_{tmp}} = 2 + K(4 + \frac{M}{2} \cdot 12); \quad (2.7)$$

- Заполнения массива b_{tmp}, трудоёмкость которого (2.8):

$$f_{b_{tmp}} = 2 + K(4 + \frac{N}{2} \cdot 12); \quad (2.8)$$

- Цикла заполнения для чётных размеров, трудоёмкость которого (2.3):

$$f_{cycle} = 2 + M \cdot (4 + N \cdot (11 + \frac{K}{2} \cdot 23)); \quad (2.9)$$

- Цикла, который дополнительно нужен для подсчёта значений при нечётном размере матрицы, трудоемкость которого (2.10):

$$f_{last} = \begin{cases} 2, & \text{чётная,} \\ 4 + M \cdot (4 + 14N), & \text{иначе.} \end{cases} \tag{2.10}$$

Тогда для худшего случая (нечётный общий размер матриц) имеем (2.11):

$$f_{worst} = f_{a_{tmp}} + f_{b_{tmp}} + f_{cycle} + f_{last} \approx 11.5 \cdot MNK \tag{2.11}$$

Для лучшего случая (чётный общий размер матриц) имеем (2.12):

$$f_{best} = f_{a_{tmp}} + f_{a_{tmp}} + f_{cycle} + f_{last} \approx 11.5 \cdot MNK \tag{2.12}$$

2.1.3 Оптимизированный алгоритм Винограда

Оптимизация заключается в следующих пунктах.

- Использовании побитового сдвига вместо умножения на 2.
- Замены операции сложения и вычитания на операции $+=$ и $-=$ соответственно.
- Педвычисление первой суммы в циклах заполнения вспомогательных массивов.

Тогда трудоемкость оптимизированного алгоритма Винограда состоит из:

1. создания и инициализации массивов a_{tmp} и b_{tmp} (2.6);
2. Заполнения массива a_{tmp}, трудоёмкость которого (2.13):

$$f_{a_{tmp}} = 2 + K(4 + 7 + (\frac{M}{2} - 1) \cdot 12); \tag{2.13}$$

3. Заполнения массива b_{tmp}, трудоёмкость которого (2.14):

$$f_{b_{tmp}} = 2 + K(4 + 7 + (\frac{N}{2} - 1) \cdot 12); \tag{2.14}$$

4. Цикла заполнения для чётных размеров, трудоёмкость которого (2.3):

5. Условие, которое нужно для дополнительных вычислений при нечётном размере матрицы, трудоемкость которого (2.10):

Тогда для худшего случая (нечётный общий размер матриц) имеем (2.15):

$$f_{worst} = f_{MH} + f_{MV} + f_{cycle} + f_{last} \approx 11.5 \cdot MNK \tag{2.15}$$

Для лучшего случая (чётный общий размер матриц) имеем (2.16):

$$f_{best} = f_{MH} + f_{MV} + f_{cycle} + f_{last} \approx 11.5 \cdot MNK \tag{2.16}$$

2.2 Описание алгоритмов

В данном разделе будут рассмотрены схемы алгоритмов плавной сортировки (рис. 2.1), сортировки слиянием (рис. 2.2, 2.3, 2.4).

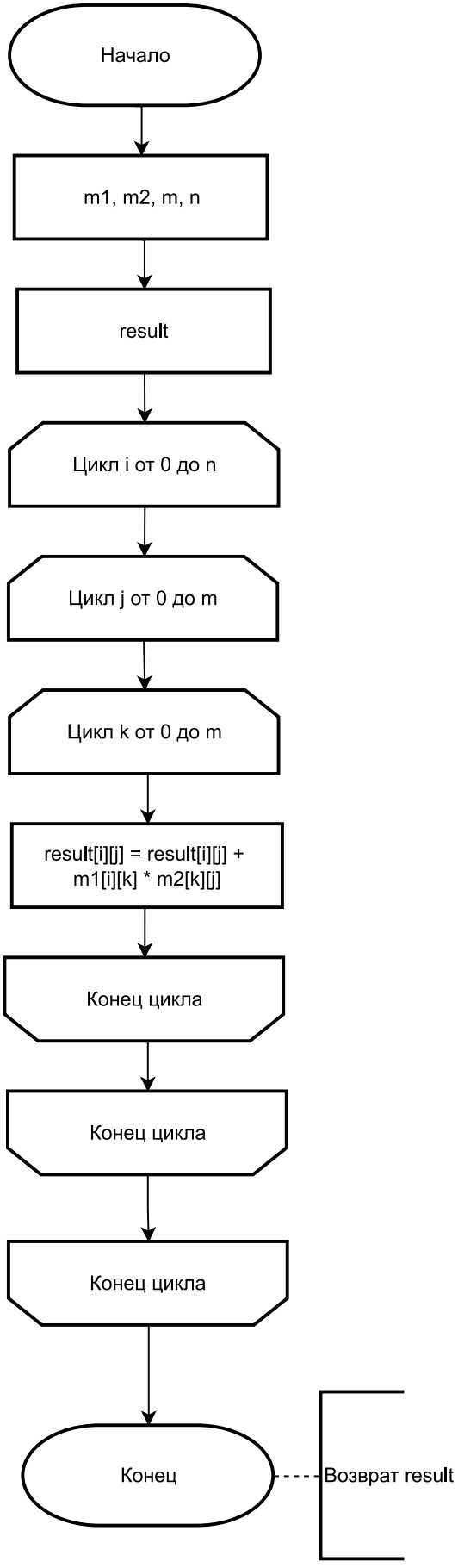

Рис. 2.1 – Схема алгоритма плавной сортировки

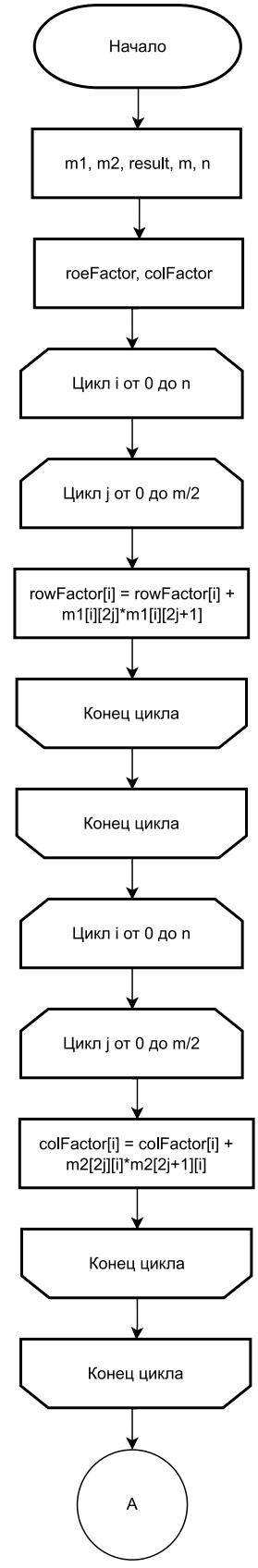

Рис. 2.2 – Схема функции findPosMaxElem, часть 1

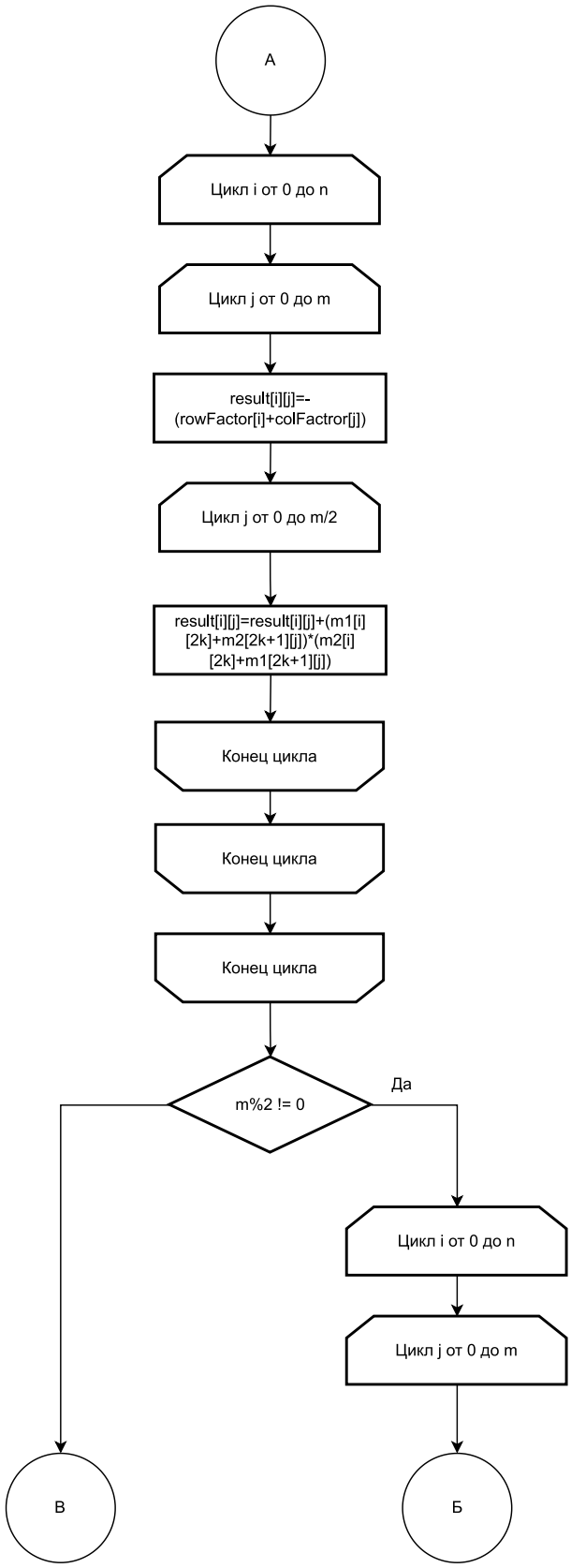

Рис. 2.3 – Схема функции findPosMaxElem, часть 2

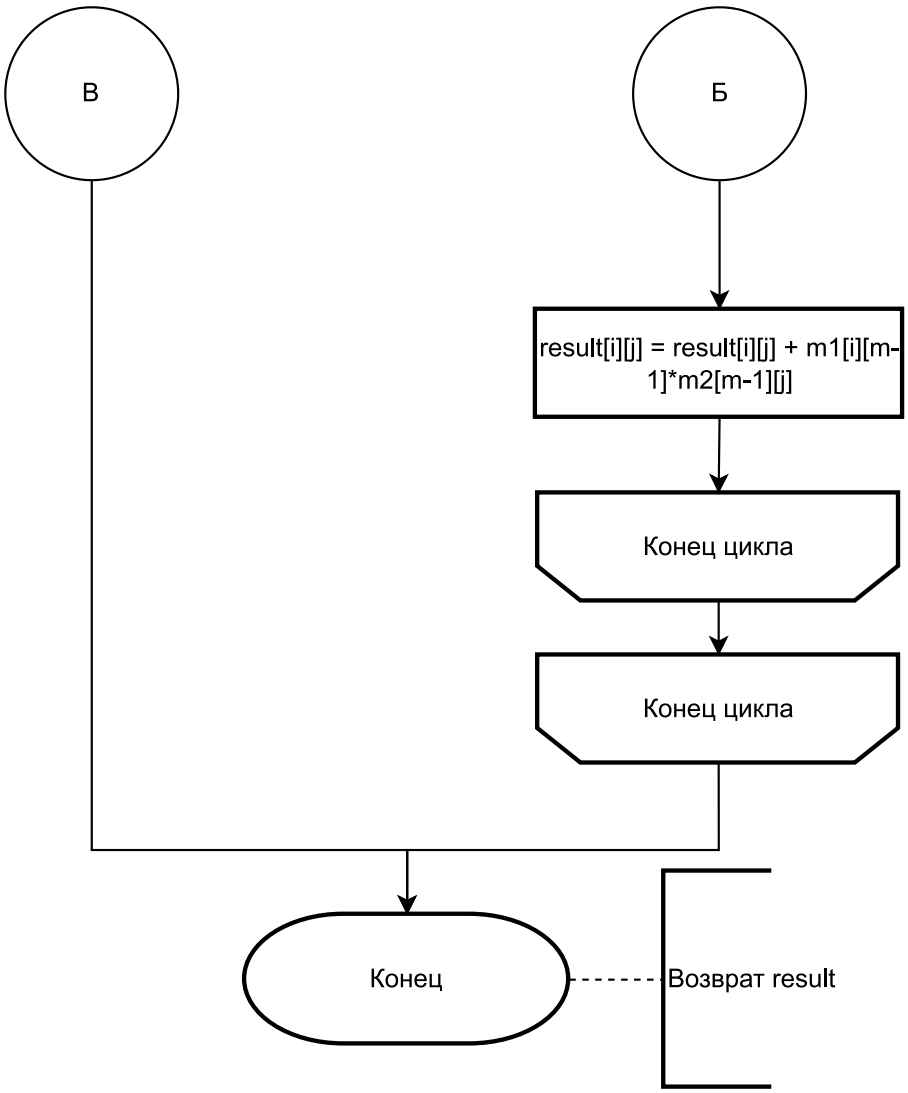

Рис. 2.4 – Схема функции makeHeapPool

Вывод

В данном разделе были проанализированны три алгоритма сортировки. Рассчитаны трудоёмкости алгоритмов в лучшем случае (л.с.) и в худшем случае (х.с.).

Сортировка слиянием: л.с. – $O(nlog(n))$, х.с. – $O(nlog(n))$.

Плавная сортировка: л.с. – $O(n)$, х.с. – $O(nlog(n))$.

Блочная сортировка: л.с. – $O(n)$, х.с. – $O(nlog(n))$.

3. Технологическая часть

В данном разделе будут рассмотрены средства реализации, а также представлены листинги сортировок.

3.1 Средства реализации

В данной работе для реализации был выбран язык программирования $c++$. В текущей лабораторной работе требуется замерить процессорное время для выполняемой программы, а также построить графики. Для визуализации результатов использовался язык $Python$.

Время работы было замерено с помощью функции *GetProcessTimes(...)* [1] из библиотеки $Windows.h$.

3.2 Сведения о модулях программы

Программа состоит из следующих модулей:

- $AA2.cpp$ - файл, содержащий весь служебный код;
- $MatrixMull.cpp$ - файл, содержащий код всех алгоритмов матричного умножения;
- $TimeCompare.cpp$ - файл, производящий замеры времени;
- $GetCPUTime.cpp$ - файл, определяющий функцию замера времени;
- $Gen.cpp$ - файл, генерирующий входную матрицу;

- *MatrixMull.hpp* - заголовочный файл модуля *MatrixMull.cpp*;
- *TimeCompare.hpp* - заголовочный файл модуля *TimeCompare.cpp*;
- *GetCPUTime.hpp* - заголовочный файл модуля *GetCPUTime.cpp*;
- *Gen.hpp* - заголовочный файл модуля *Gen.cpp*.

3.3 Реализация алгоритмов

В листингах 3.1, 3.2, 3.3, представлены реализации алгоритмов матричного умножения (стандартный, винограда, оптимизированный винограда).

Листинг 3.1 — Стандартный алгоритм умножения матриц

```
data_t matrix_multiplication(const data_t m1, const data_t m2)
{
  data_t result;
  result.n = m1.n;
  result.m = m2.m;
  result.matrix = create_matrix(m1.n, m2.m);
  if (!result.matrix)
  return result;
  for (int i = 0; i < m1.n; i++)
  for (int j = 0; j < m2.m; j++)
  {
    result.matrix[i][j] = 0;
    for (int k = 0; k < m1.m; k++)
    result.matrix[i][j] += m1.matrix[i][k] * m2.matrix[k][j];
  }
  return result;
}
```

Листинг 3.2 — Алгоритм Винограда умножения матриц

```
data_t matrix_multiplication_Vinograd(const data_t m1, const data_t m2)
{
  data_t result;
  result.n = m1.n;
  result.m = m2.m;
  result.matrix = create_matrix(m1.n, m2.m);
  if (!result.matrix)
  return result;
  int n = m1.n;
  int m = n;

  int *rowFactor = (int *)calloc(n, sizeof(int)), *colFactor = (int*)
     calloc(n, sizeof(int));

  for (int i = 0; i < n; ++i)
  {
    for (int j = 0; j < m / 2; ++j)
    rowFactor[i] = rowFactor[i] + m1.matrix[i][2 * j] * m1.matrix[i][2
       * j + 1];
  }

  for (int i = 0; i < n; ++i)
  {
    for (int j = 0; j < m / 2; ++j)
    colFactor[i] = colFactor[i] + m2.matrix[2 * j][i] * m2.matrix[2 * j
        + 1][i];
  };

  for (int i = 0; i < n; ++i)
  for (int j = 0; j < n; ++j)
  {
    result.matrix[i][j] = -(rowFactor[i] + colFactor[j]);
    for (int k = 0; k < m / 2; ++k)
    {
      result.matrix[i][j] = result.matrix[i][j] + (m1.matrix[i][2 * k]
         + m2.matrix[2 * k + 1][j]) * (m1.matrix[i][2 * k + 1] + m2.
         matrix[2 * k][j]);
    }
```

```
34    }
35
36    if (m % 2 != 0)
37    {
38      for (int i = 0; i < n; ++i)
39      for (int j = 0; j < n; ++j)
40      {
41        result.matrix[i][j] = result.matrix[i][j] + m1.matrix[i][m - 1] *
              m2.matrix[m - 1][j];
42      }
43    }
44    free(rowFactor);
45    free(colFactor);
46    return result;
47  }
```

Листинг 3.3 — Оптимизированный алгоритм Винограда умножения матриц

```
data_t matrix_multiplication_VinogradOptimase(const data_t m1, const
    data_t m2)
{
  data_t result;
  result.n = m1.n;
  result.m = m2.m;
  result.matrix = create_matrix(m1.n, m2.m);
  if (!result.matrix)
  return result;
  int n = m1.n;
  int m = n;
  int d = n>>1;

  int *rowFactor = (int *)calloc(n, sizeof(int)), *colFactor = (int*)
     calloc(n, sizeof(int));

  for (int i = 0; i < n; ++i)
  {
    rowFactor[i] = m1.matrix[i][0] * m1.matrix[i][1];
    for (int j = 1; j < d; ++j)
    rowFactor[i] +=  m1.matrix[i][j<<1] * m1.matrix[i][(j<<1) + 1];
  }

  for (int i = 0; i < n; ++i)
  {
    colFactor[i] = m2.matrix[0][i] * m2.matrix[1][i];
    for (int j = 1; j < d; ++j)
    colFactor[i] += m2.matrix[j<<1][i] * m2.matrix[(j<<1) + 1][i];
  };

  for (int i = 0; i < n; ++i)
  for (int j = 0; j < n; ++j)
  {
    result.matrix[i][j] = -(rowFactor[i] + colFactor[j]);
    for (int k = 0; k < d; ++k)
    {
      result.matrix[i][j] += (m1.matrix[i][k << 1] + m2.matrix[(k<<1) +
          1][j]) * (m1.matrix[i][(k<<1) + 1] + m2.matrix[ k<<1][j]);
```

```
36        }
37    }
38
39    if (m % 2 != 0)
40    {
41        for (int i = 0; i < n; ++i)
42        for (int j = 0; j < n; ++j)
43        {
44            result.matrix[i][j] += m1.matrix[i][m - 1] * m2.matrix[m - 1][j];
45        }
46    }
47    free(rowFactor);
48    free(colFactor);
49    return result;
50 }
```

3.4 Функциональные тесты

В таблице 3.1 приведены тесты для функций, реализующих алгоритмы матричного умножения. Тесты для всех реализаций алгоритмов пройдены успешно.

Таблица 3.1 — Функциональные тесты

Матрица 1	Матрица 2	Ожидаемый результат
$\begin{pmatrix} 1 \end{pmatrix}$	$\begin{pmatrix} \end{pmatrix}$	Сообщение об ошибке
$\begin{pmatrix} 1 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$	Сообщение об ошибке
$\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$
$\begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{pmatrix}$
$\begin{pmatrix} 1 \end{pmatrix}$	$\begin{pmatrix} 2 \end{pmatrix}$	$\begin{pmatrix} 2 \end{pmatrix}$

Вывод

В данном разделе были представлены реализации следующих алгоритмов: стандартного матричного умножения, Винограда, Винограда с оптимизацией.

4. Исследовательская часть

В данном разделе будут приведены примеры работы программа, а также проведен сравнительный анализ реализаций алгоритмов при различных ситуациях на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры времени представлены далее:

- операционная система Windows 11 Pro Версия 22H2 (22621.674) [2];
- память 16 ГБ;
- процессор 11th Gen Intel(R) Core(TM) i5-11400 2.59 ГГц [3].

При тестировании компьютер был включен в сеть электропитания. Во время замеров процессорного времени устройство было нагружено только встроенными приложениями окружения, а также системой тестирования.

4.2 Демонстрация работы программы

На рисунке 4.1 представлен результат работы программы. На экран выводятся результаты заемров времени для разных размеров матриц и разных видов алгоритмов матричного умножения в мс.

```
                    Time in ms
Size          Easy          Vinograde   VinogradeOptimise
 10              0       0.015625              0
 20        0.03125        0.03125       0.015625
 30         0.0625        0.03125        0.03125
 40       0.046875         0.0625         0.0625
 50        0.34375        0.28125       0.296875
 60       0.671875         0.4375         0.4375
 70        1.04688        0.65625        0.78125
 80        1.57812        1.15625        1.14062
 90        2.34375         1.6875          1.625
100        3.07812        2.23438        2.23438
200             25        18.0781        18.1406
300        85.7031        63.2656        61.4531
400        206.312         150.75        150.516
500        411.062        301.531        302.391
```

Рис. 4.1 – Пример работы программы

4.3 Время выполнения реализаций алгоритмов

Как было сказано выше, используется функция замера процессорного времени GetProcessTimes(...) из библиотеки Windows.h. Функция возвращает пользовательское процессорное временя типа float.

Использовать функцию приходится дважды, затем из конечного времени нужно вычесть начальное, чтобы получить результат.

Входные данные: размер матрицы от 10 до 500, целые числа от 0 до 200.

Результаты замеров времени работы реализаций алгоритмов матричного умножения на различных входных данных (в мс) приведены в таблице 4.1.

Таблица 4.1 — Процессорное время работы реализаций алгоритмов

Размер	Классический	Винограда	Оптимизированный
10	0	0.015625	0
20	0.03125	0.03125	0.015625
30	0.0625	0.03125	0.03125
40	0.046875	0.0625	0.0625
50	0.34375	0.28125	0.296875
60	0.671875	0.4375	0.4375
70	1.04688	0.65625	0.78125
80	1.57812	1.15625	1.14062
90	2.34375	1.6875	1.625
100	3.07812	2.23438	2.23438
200	25	18.0781	18.1406
300	85.7031	63.2656	61.4531
400	206.312	150.75	150.516
500	411.062	301.531	302.391

Также на рисунке 4.2 приведены графические результаты замеров времени работы алгоритмов в зависимости от размера входной матрицы.

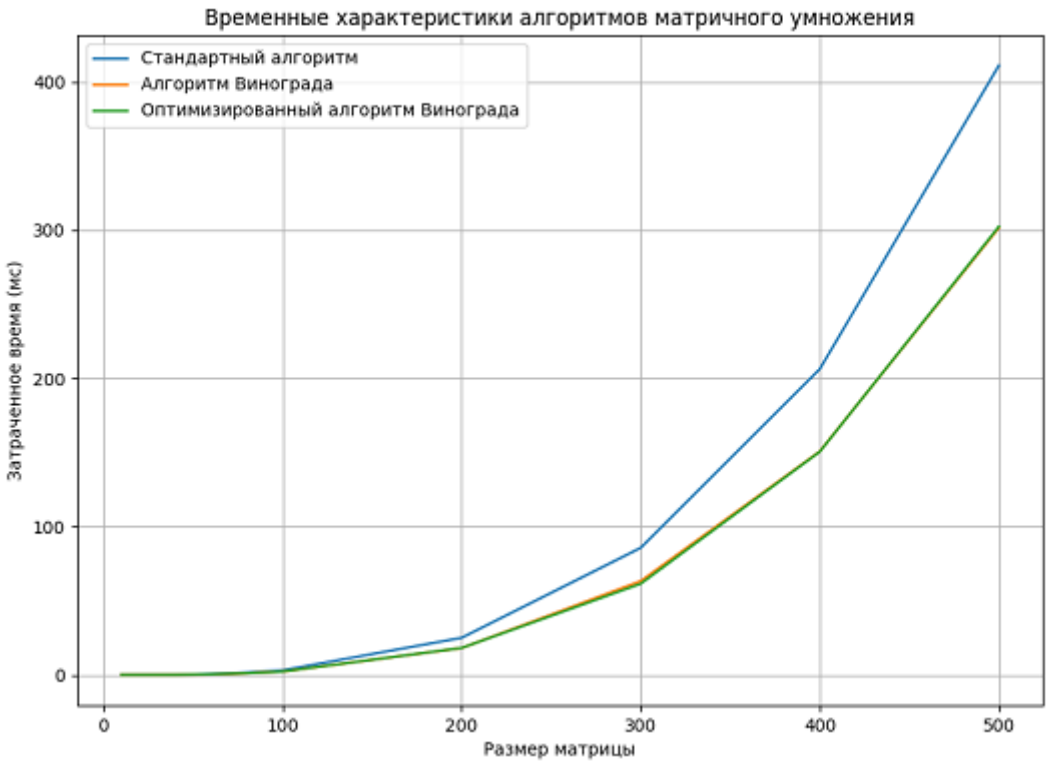

Рис. 4.2 – Процессорное время вычислений

Вывод

Теоретические результаты замеров и полученные практически результаты совпадают.

Список использованных источников

[1] GetProcessTimes function [Эл. ресурс]. Режим доступа: https://clck.ru/32NCYi (дата обращения: 13.10.2022).

[2] Windows 11, version 22H2 [Эл. ресурс]. Режим доступа: https://clck.ru/32NCXx (дата обращения: 14.10.2022).

[3] Процессор Intel® Core™ i7 [Эл. ресурс]. Режим доступа: https://clck.ru/yeQa8 (дата обращения: 14.10.2022).