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»

НАПРАВЛЕНИЕ ПОДГОТОВКИ «09.03.04 Программная инженерия»

ОТЧЕТ

ПО ЛАБОРАТОРНОЙ РАБОТЕ №2

Название: Алгоритм Копперсмита-Винограда

Дисциплина: Анализ алгоритмов

Студент	ИУ7-56Б		Ковель А.Д.
	Группа	Подпись, дата	И. О. Фамилия
Преподаватель			Волкова Л.Л.
		Подпись, дата	И. О. Фамилия

Москва, 2022 г.

Оглавление

Введение

Разработка и совершенствование матричных алгоритмов является важнейшей алгоритмической задачей. Непосредственное применение классического матричного умножения требует времени порядка $O(n^3)$. Однако существуют алгоритмы умножения матриц, работающие быстрее очевидного. В линейной алгебре алгоритм Копперсмита – Винограда[1], названный в честь Д. Копперсмита и Ш. Винограда , был асимптотически самый быстрый из известных алгоритмов умножения матриц с 1990 по 2010 год. В данной работе внимание акцентируется на алгоритме Копперсмита – Винограда и его улучшениях.

Алгоритм не используется на практике, потому что он дает преимущество только для матриц настолько больших размеров, что они не могут быть обработаны современным вычислительным оборудованием. Если матрица не велика, эти алгоритмы не приводят к большой разнице во времени вычислений.

Цель лабораторной работы – теоретическое изучение алгоритма Копперсмита – Винограда, разработка его улучшений и сравнение с классическим алгоритмом на основе полученных в ходе лабораторной работы экспериментальных данных. Для этого необходимо проанализировать изучаемый алгоритм, выделить основные особенности и недостатки для дальнейшей оптимизации.

1 Аналитический раздел

1.1 Применение математического подхода

Даны матрицы, $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times p}$, произведение матриц, $B \in \mathbb{R}^{m \times p}$ – такая матрица, $C = A \times B$, каждый элемент которой вычисляется согласно формуле 1.1:

$$c_{i,j} = \sum_{n}^{k=1} a_{i,k} \cdot b_{k,j}, \text{ где } i = \overline{1,m}, j = \overline{1,p} \tag{1.1}$$

Стандартный алгоритм умножения матриц реализует формулу(1.1).

Операция умножения двух матриц выполнима только в том случае, если число столбцов в первом сомножителе равно числу строк во втором. В частности, умножение всегда выполнимо, если оба сомножителя — квадратные матрицы одного и того же порядка.

1.2 Алгоритм Копперсмита – Винограда

Для начала стоит обратить внимание на альтернативный подсчет выражения $a_1 \cdot b_1 + a_2 \cdot b_2$ осуществляется согласно 1.2:

$$\begin{aligned} &\lambda_1 = a_1 \cdot a_1 \\ &\lambda_2 = b_1 \cdot b_1 \\ &\lambda_3 = (a_1 + b_2) \cdot (a_2 + b_1) \\ &\text{результат: } \lambda_3 - \lambda_1 - \lambda_2 \end{aligned} \tag{1.2}$$

Классическое умножение матриц, по своей сути, является нахождением некоторого числа скалярных произведений каждого столбца первого множителя с каждой строкой второго. Процедура может быть усовершенствована: если один вектор V встречается множество раз, то операция нахождения векторного произведения для него может быть выполнена единожды. Идея препроцессирования в случае перемножения квадратных матриц, $n \times n$ приводит к определению алгоритма Копперсмита – Винограда.

Для вектора, $x = (x_1 \cdots x_n)$, можно записать(1.3):

$$W(x) = x_1 \cdot x_2 + x_3 \cdot x_4 + \cdots + x_{n-1} \cdot x_n \tag{1.3}$$

Тогда умножения матрицы выполняется согласно следующему алгоритму:

1. Для каждой строки R_i матрицы M вычислить $W(R_i)$ и для каждого столбца C_i матрицы M вычислить $W(C_i)$;

2. Для каждой пары (i, j), где r соответствует R_i и C соответствует C_i, вычислить 1.4:

$$\begin{gathered} r \cdot c = (r_1 + c_2) + (r_2 + c_1) \cdot (r_3 + c_4) \cdot \\ (r_4 + c_3) + \cdots + (r_{n-1} + c_n) + (r_n + c_{n-1}) - W(r) - W(c). \end{gathered} \tag{1.4}$$

Если оценивать подход Копперсмита – Винограда опуская идею препроцессирования, то можно заметить, что арифметических операций в ней больше, чем в формуле классического скалярного произведения Однако, сохранение результатов $W(C_i)$ и $W(R_i)$ позволяет выполнять меньше операций, чем при нахождении матричного произведения математически. Разница при таком подходе, очевидно, будет заметна на матрицах настолько больших размеров, что они не могут быть обработаны ЭВМ.

1.3 Вывод

Была выявлена основная особенность подхода Копперсмита – Винограда – идея предварительной обработки. Разница во времени выполнения при такой оптимизации будет экспериментально вычислена в исследовательском разделе.

2 Конструкторский раздел

2.1 Трудоемкость алгоритмов

Для получения функции трудоемкости алгоритма необходимо ввести модель оценки трудоемкости. Трудоемкость "элементарных"операций оценивается следующим образом:

1. Трудоемкость 1 имеют операции:

$$+, -, =, <, >, <=, >=, ==, + =, - =,$$
$$++, --, [], \&\&, ||, >>, <<$$

2. Трудоемкость 2 имеют операции:

$$*, /, \backslash, \%$$

3. Трудоемкость конструкции ветвления определяется согласно формуле 2.1

$$f_{if} = f_{condition} + \begin{cases} min\left(f_{true}, f_{false}\right) & \text{в лучшем случае,} \\ max\left(f_{true}, f_{false}\right) & \text{в худшем случае.} \end{cases} \quad (2.1)$$

4. Трудоемкость цикла расчитывается по формуле 2.2

$$f_{loop} = f_{init} + f_{cmp} + N\left(f_{body} + f_{inc} + f_{cmp}\right), \quad (2.2)$$

где
f_{init} – трудоемкость инициализации,
f_{body} – трудоемкость тела цикла,
f_{iter} – трудоемкость инкремента,
f_{cmp} – трудоемкость сравнения,
N – количество повторов.

5. Трудоемкость вызова функции равна 0.

2.2 Оптимизация алгоритма Копперсмита – Винограда

Алгоритм Копперсмита – Винограда можно оптимизировать следующим образом:

- Счетчики цикла можно объявить единожды и обнулять их по требованию. В этом случае стоимость инициализации при расчете трудоемкости цикла сократится;
- Увеличение числа на определенное число можно заменить на операцию $+=$, поскольку она имеет меньший вес чем в сумме сложение и присвоение;
- Для алгоритма худшим случаем являются матрицы с нечётным общим размером, а лучшим - с четным. Соответственно, алгоритм можно ускорить в соответствии с четностью размера.

2.3 Трудоемкость алгоритмов

2.3.1 Классический алгоритм

Пусть на вход алгоритму поступают матрицы M_{left} и M_{right} с размерностью $n \times m$ и $m \times q$. Тогда трудоемкость классического алгоритма определяется по формуле 2.3

$$f_{alg} = f_{loop_i} = 2 + n\left(2 + f_{loop_j}\right) = 2 + n\left(2 + 2 + q\left(2 + f_{loop_k}\right)\right) = \\ = 2 + n\left(2 + 2 + q\left(2 + +2 + 14 \cdot m\right)\right) \approx 14mnq = 14MNK \quad (2.3)$$

2.3.2 Алгоритм Копперсмита — Винограда

Трудоёмкость алгоритма Копперсмита — Винограда состоит из:

- создания и инициализации массивов MH и MV, трудоёмкость которого (2.4):

$$f_{init} = M + N; \tag{2.4}$$

- заполнения массива MH, трудоёмкость которого (2.5):

$$f_{MH} = 2 + K(2 + \frac{M}{2} \cdot 11); \tag{2.5}$$

- заполнения массива MV, трудоёмкость которого (2.6):

$$f_{MV} = 2 + K(2 + \frac{N}{2} \cdot 11); \tag{2.6}$$

- цикла заполнения для чётных размеров, трудоёмкость которого (2.7):

$$f_{cycle} = 2 + M \cdot (4 + N \cdot (11 + \frac{K}{2} \cdot 23)); \tag{2.7}$$

- цикла, для дополнения умножения суммой последних нечётных строки и столбца, если общий размер нечётный, трудоемкость которого (2.8):

$$f_{last} = \begin{cases} 2, & \text{чётная,} \\ 4 + M \cdot (4 + 14N), & \text{иначе.} \end{cases} \tag{2.8}$$

Итого, для худшего случая (нечётный общий размер матриц) имеем (2.9):

$$f = f_{MH} + f_{MV} + f_{cycle} + f_{last} \approx 11.5 \cdot MNK \tag{2.9}$$

Для лучшего случая (чётный общий размер матриц) имеем (2.10):

$$f = f_{MH} + f_{MV} + f_{cycle} + f_{last} \approx 11.5 \cdot MNK \tag{2.10}$$

2.3.3 Оптимизированный алгоритм Копперсмита — Винограда

Оптимизированный алгоритм Винограда представляет собой обычный алгоритм Винограда, за исключением следующих оптимизаций:

- вычисление происходит заранее;
- используется битовый сдвиг, вместо деления на 2;
- последний цикл для нечётных элементов включён в основной цикл, используя дополнительные операции в случае нечётности N.

Трудоёмкость улучшенного алгоритма Копперсмита — Винограда состоит из:

- создания и инициализации массивов MH и MV, трудоёмкость которого (2.11):

$$f_{init} = M + N; \tag{2.11}$$

- заполнения массива MH, трудоёмкость которого (2.12):

$$f_{MH} = 2 + K(2 + \frac{M}{2} \cdot 8); \tag{2.12}$$

- заполнения массива MV, трудоёмкость которого (2.13):

$$f_{MV} = 2 + K(2 + \frac{M}{2} \cdot 8); \tag{2.13}$$

- цикла заполнения для чётных размеров, трудоёмкость которого (2.14):

$$f_{cycle} = 2 + M \cdot (4 + N \cdot (11 + \frac{K}{2} \cdot 18)); \tag{2.14}$$

- условие, для дополнения умножения суммой последних нечётных строки и столбца, если общий размер нечётный, трудоемкость которого (2.15):

$$f_{last} = \begin{cases} 1, & \text{чётная,} \\ 4 + M \cdot (4 + 10N), & \text{иначе.} \end{cases} \tag{2.15}$$

Итого, для худшего случая (нечётный общий размер матриц) имеем (2.16):

$$f = f_{MH} + f_{MV} + f_{cycle} + f_{last} \approx 9MNK \tag{2.16}$$

Для лучшего случая (чётный общий размер матриц) имеем (2.17):

$$f = f_{MH} + f_{MV} + f_{cycle} + f_{last} \approx 9MNK \tag{2.17}$$

2.4 Схемы алгоритмов

На рисунке 2.1 приведена схема классического алгоритма умножения матриц. На рисунке 2.2 приведена схема алгоритма Копперсмита – Винограда. Рисунок 2.3 демонстрируют схему оптимизированного алгоритма Копперсмита – Винограда.

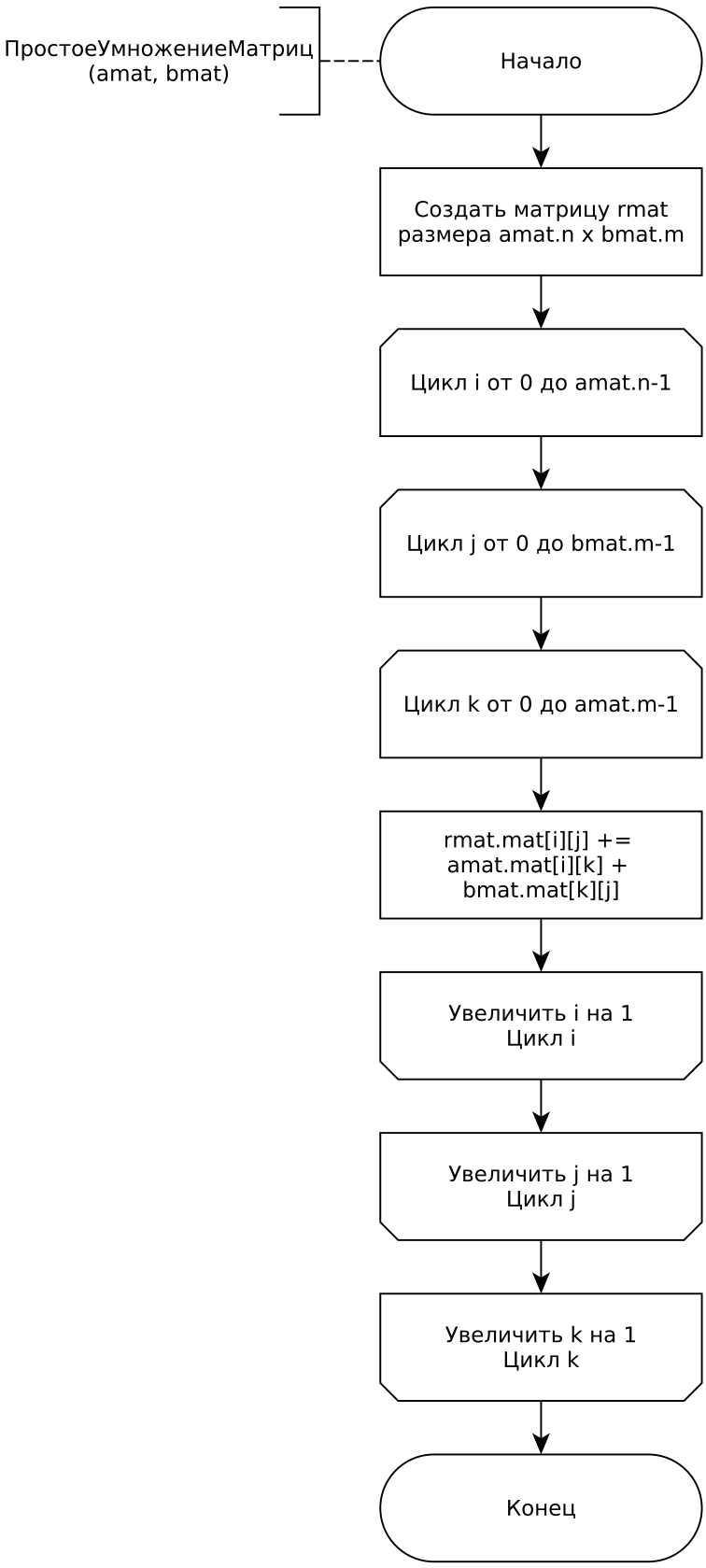

Рисунок 2.1 – Схема классического алгоритма умножения матриц

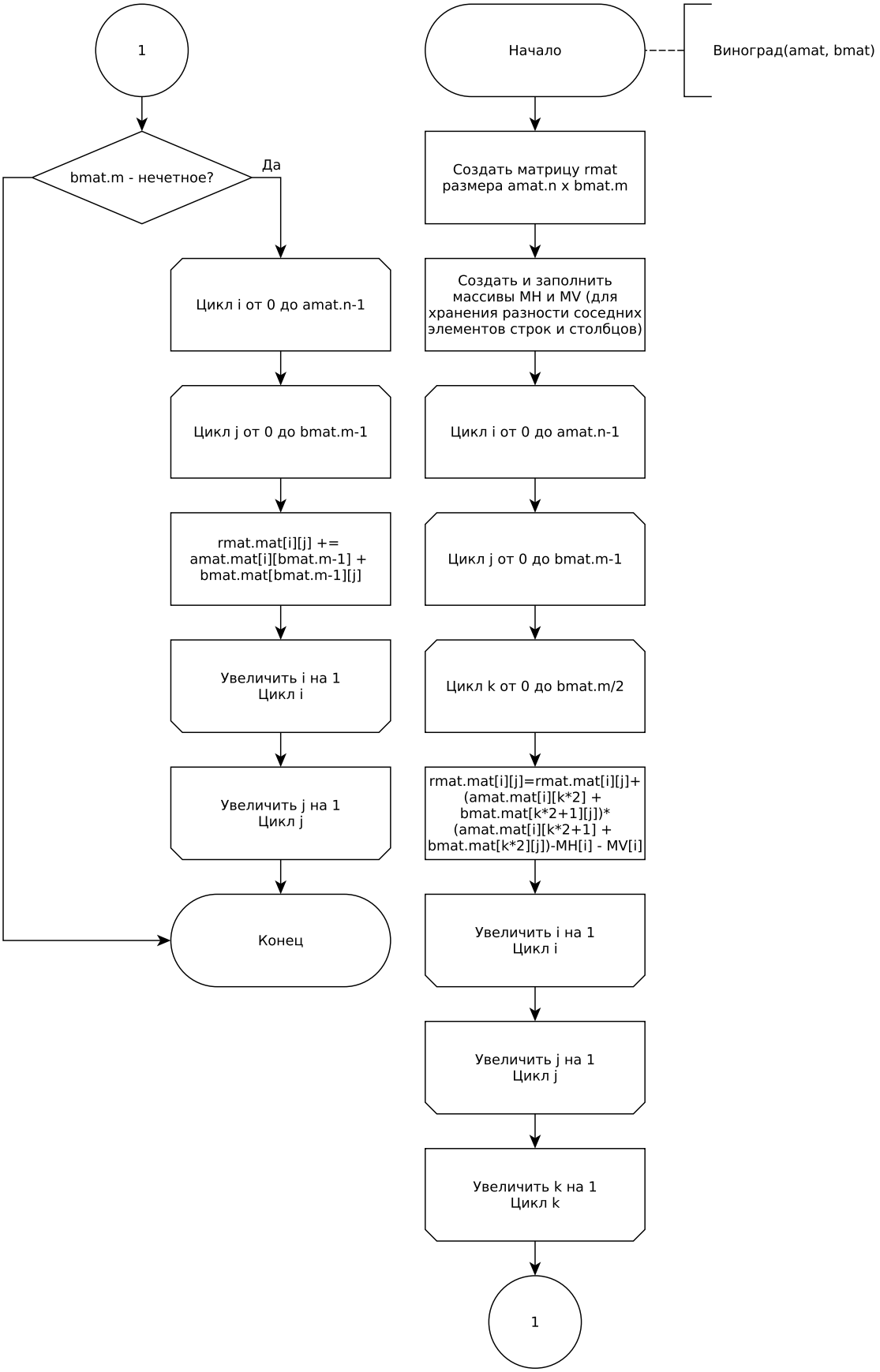

Рисунок 2.2 – Схема алгоритма умножения матриц Копперсмита – Винограда

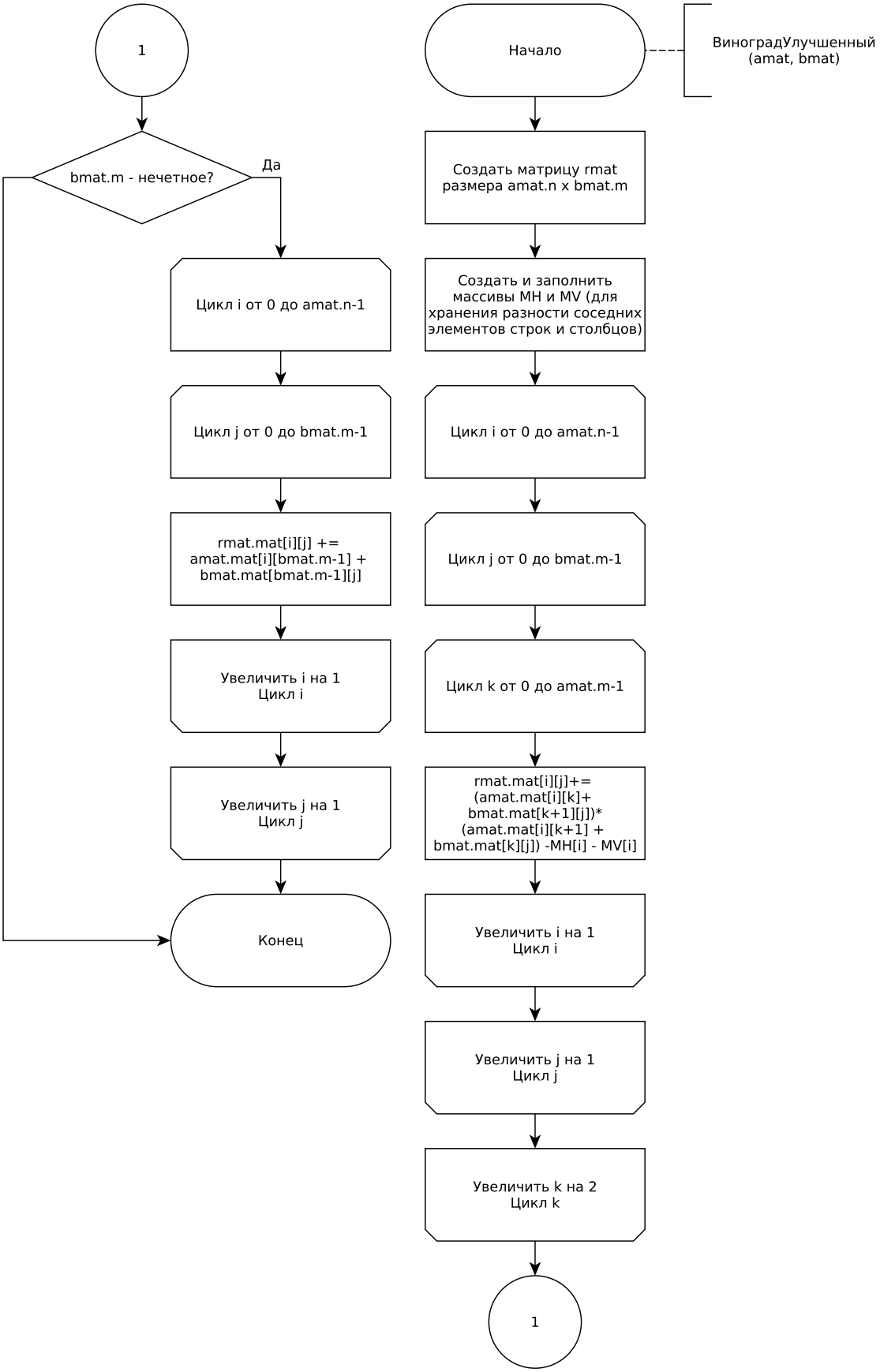

Рисунок 2.3 – Схема оптимизированного алгоритма умножения матриц Копперсмита – Винограда

2.4.1 Вывод

Алгоритмы были проанализированы с точки зрения временных затрат. Было выявлено, что алгоритм Копперсмита – Винограда работает на $1.5MNK$ быстрее, чем классический матричный. Оптимизация алгоритма же дает выигрыш на $2.5MNK$.

Были построены схемы алгоритмов. Теоретически были исследованы способы оптимизации алгоритма Копперсмита – Винограда. Было получено достаточно теоретических сведений для разработки ПО, решающего поставленную задачу.

3 Технологический раздел

3.1 Требования к ПО

Программное обеспечение должно удовлетворять следующим требованиям:

- Программа получает на вход с клавиатуры две матрицы размеров в пределах 10000 $\times$ 10000 либо получает два числа – размерность матрицы в пределах 10000;
- Программа выдает матрицу - произведение двух полученных матриц.

3.2 Средства реализации

Для реализации ПО был выбран язык программирования Golang[2]. Данный язык предоставляет следующие возможности:

- Средства объектно-ориентированного программирования ограничиваются интерфейсами, что позволяет создавать абстракции, которые нужны для представления матрицы;
- Поддерживает библиотеки C и C++, что позволит измерить процессорное время, также имеет свою обширную библиотеку.

В качестве среды разработки была выбрана среда VS Code[3], запуск происходил через команду go run main.go.

3.3 Листинги кода

Листинг 3.1 демонстрирует классический алгоритм умножения.

Листинг 3.1 – Классический алгоритм умножения

```
func SimpleMult(amat, bmat MInt) MInt {
  rmat := formResMat(amat.n, bmat.m)
  for i := 0; i < rmat.n; i++ {
    for j := 0; j < rmat.m; j++ {
      for k := 0; k < amat.m; k++ {
        rmat.mat[i][j] += amat.mat[i][k] * bmat.mat[k][j]
      }
    }
  }
  return rmat
}
```

Листинг 3.2 – умножение матриц алгоритмом Винограда.

Листинг 3.2 – Алгоритм умнложения Виноградом

```
func WinogradMult(amat, bmat MInt) MInt {
  rmat := formResMat(amat.n, bmat.m)
  rowcf := precomputeWinogradRows(amat)
  colcf := precomputeWinogradCols(bmat)

  for i := 0; i < rmat.n; i++ {
    for j := 0; j < rmat.m; j++ {
      for k := 0; k < rmat.m/2; k++ {
        rmat.mat[i][j] = rmat.mat[i][j] +
        (amat.mat[i][k*2]+bmat.mat[k*2+1][j])*
        (amat.mat[i][k*2+1]+bmat.mat[k*2][j]) -
        rowcf.mat[i][k] - colcf.mat[k][j]
      }
    }
  }
  if rmat.m%2 != 0 {
    for i := 0; i < rmat.n; i++ {
      for j := 0; j < rmat.m; j++ {
        rmat.mat[i][j] += amat.mat[i][amat.m-1] *
          bmat.mat[bmat.n-1][j]
      }
    }
  }
  return rmat
}
```

Листинг 3.3 – умножение оптимизированным согласно 2.2 алгоритмом винограда.

Листинг 3.3 – Оптимизированный алгоритм умнложения Виноградом

```
func WinogradMultImp(amat, bmat MInt) MInt {

  rmat := formResMat(amat.n, bmat.m)

  rowcf := precomputeWinogradRowsImp(amat)
  colcf := precomputeWinogradColsImp(bmat)

  for i := 0; i < rmat.n; i++ {
    for j := 0; j < rmat.m; j++ {
      for k := 0; k < rmat.m-1; k += 2 {

        l := k/2 + k%2
        rmat.mat[i][j] +=
        (amat.mat[i][k]+bmat.mat[k+1][j])
        *(amat.mat[i][k+1]+bmat.mat[k][j]) -
        rowcf.mat[i][l] - colcf.mat[l][j]

      }
    }
  }

  if rmat.m%2 != 0 {
    k := amat.m - 1
    for i := 0; i < rmat.n; i++ {
      for j := 0; j < rmat.m; j++ {

        rmat.mat[i][j] += amat.mat[i][k] * bmat.mat[k][j]

      }
    }
  }

  return rmat
}
```

Функции оптимизации, реализующие формулу 1.3 представлены на листинге 3.4

Листинг 3.4 – Функции препроцессирования для алгоримов Винограда

```
func precomputeWinogradRowsImp(mat MInt) MInt {
  s := mat.m / 2
  cf := formResMat(mat.n, s)

  for i := 0; i < mat.n; i++ {
    for j := 0; j < mat.m-1; j += 2 {
      cf.mat[i][j/s+j%2] = mat.mat[i][j] * mat.mat[i][j+1]
    }
  }

  return cf
}

func precomputeWinogradColsImp(mat MInt) MInt {
  s := mat.n / 2
  cf := formResMat(s, mat.m)

  for i := 0; i < mat.n-1; i += 2 {
    for j := 0; j < mat.m; j++ {
      cf.mat[i/s+i%2][j] = mat.mat[i][j] * mat.mat[i+1][j]
    }
  }

  return cf
}
```

3.4 Тестирование ПО

В таблице 3.1 представлены следующие переменные:

- AG — Матрица полученная классическим алгоритмом перемножения матриц;
- CW — Матрица полученная алгоритмом Винограда;
- CW OPT — Матрица полученная оптимизированным алгоритмом Винограда .

Таблица 3.1 – Тестовые случаи

№	Входная матрица №1	Входная матрица №2	Результат		
			AG	CW	CW OPT
1	$\begin{matrix} -3 & 5 & -1 & 7 \\ -8 & 2 & -2 & 1 \\ 0 & -3 & -4 & 0 \\ -6 & 0 & 5 & 1 \end{matrix}$	$\begin{matrix} -3 & 5 & -1 & 7 \\ -8 & 2 & -2 & 1 \\ 0 & -3 & -4 & 0 \\ -6 & 0 & 5 & 1 \end{matrix}$	$\begin{matrix} -73 & -2 & 32 & -9 \\ 2 & -30 & 17 & -53 \\ 24 & 6 & 22 & -3 \\ 12 & -45 & -9 & -41 \end{matrix}$	$\begin{matrix} -73 & -2 & 32 & -9 \\ 2 & -30 & 17 & -53 \\ 24 & 6 & 22 & -3 \\ 12 & -45 & -9 & -41 \end{matrix}$	$\begin{matrix} -73 & -2 & 32 & -9 \\ 2 & -30 & 17 & -53 \\ 24 & 6 & 22 & -3 \\ 12 & -45 & -9 & -41 \end{matrix}$
2	$\begin{matrix} 0 & 0 & -2 \\ 2 & 3 & 2 \\ -4 & 0 & -1 \end{matrix}$	$\begin{matrix} 0 & 0 & -2 \\ 2 & 3 & 2 \\ -4 & 0 & -1 \end{matrix}$	$\begin{matrix} 8 & 0 & 2 \\ -2 & 9 & 0 \\ 4 & 0 & 9 \end{matrix}$	$\begin{matrix} 8 & 0 & 2 \\ -2 & 9 & 0 \\ 4 & 0 & 9 \end{matrix}$	$\begin{matrix} 8 & 0 & 2 \\ -2 & 9 & 0 \\ 4 & 0 & 9 \end{matrix}$
3	$\begin{matrix} 5 & 7 \\ 7 & 0 \\ 3 & 1 \end{matrix}$	$\begin{matrix} 5 & 7 & 7 & 0 \\ 3 & 1 & -7 & 5 \end{matrix}$	$\begin{matrix} 46 & 42 & -14 & 35 \\ 35 & 49 & 49 & 0 \\ 18 & 22 & 14 & 5 \end{matrix}$	$\begin{matrix} 46 & 42 & -14 & 35 \\ 35 & 49 & 49 & 0 \\ 18 & 22 & 14 & 5 \end{matrix}$	$\begin{matrix} 46 & 42 & -14 & 35 \\ 35 & 49 & 49 & 0 \\ 18 & 22 & 14 & 5 \end{matrix}$
4	$\begin{matrix} -4 & 9 \\ -4 & -1 \\ -1 & 5 \\ 5 & 3 \end{matrix}$	$\begin{matrix} -4 & 9 & -4 \\ -1 & -1 & 5 \end{matrix}$	$\begin{matrix} 7 & -45 & 61 \\ 17 & -35 & 11 \\ -1 & -14 & 29 \\ -23 & 42 & -5 \end{matrix}$	$\begin{matrix} 7 & -45 & 61 \\ 17 & -35 & 11 \\ -1 & -14 & 29 \\ -23 & 42 & -5 \end{matrix}$	$\begin{matrix} 7 & -45 & 61 \\ 17 & -35 & 11 \\ -1 & -14 & 29 \\ -23 & 42 & -5 \end{matrix}$

3.5 Вывод

Было написано и протестировано программное обеспечение для решения поставленной задачи.

4 Исследовательская часть

4.1 Технические характеристики

Тестирование выполнялось на устройстве со следующими техническими характеристиками:

- Операционная система Pop!_OS 22.04 LTS [4] Linux [5];
- Оперативная память 16 GiB;
- Процессор AMD® Ryzen 7 2700 eight-core processor × 16 [6].

Во время тестирования устройство было подключено к блоку питания и не нагружено никакими приложениями, кроме встроенных приложений окружения, окружением и системой тестирования.

4.2 Таблица времени выполнения алгоритмов

Результаты профилирования алгоритмов приведены в таблице 4.1. Результаты тестирования приведены в таблице 3.1. Здесь CW – алгоритм Копперсмита – Винограда, ALG – классический и CW OPT – оптимизированный Копперсмита – Винограда.

4.3 Графики функций

Таблица 4.1 – Время выполнения алгоритмов

Таблица 4.1.1 – Четная размерность матриц

n	Время, ns		
	ALG	CW	CW OPT
10	4.49e-06	3.91e-06	4.01e-06
20	2.74e-05	2.24e-05	2.30e-05
30	8.84e-05	6.72e-05	7.01e-05
40	2.09e-04	1.49e-04	1.58e-04
50	3.96e-04	2.98e-04	3.14e-04
60	6.58e-04	4.95e-04	5.15e-04
70	1.07e-03	7.95e-04	8.43e-04
80	1.60e-03	1.20e-03	1.22e-03
90	2.26e-03	1.68e-03	1.77e-03
100	3.14e-03	2.25e-03	2.38e-03
150	1.06e-02	7.88e-03	8.33e-03
200	2.84e-02	2.20e-02	2.34e-02
250	6.73e-02	5.34e-02	5.74e-02
300	1.14e-01	8.97e-02	9.60e-02
350	1.80e-01	1.39e-01	1.50e-01
400	2.67e-01	2.05e-01	2.21e-01
450	4.17e-01	3.23e-01	3.48e-01
500	5.71e-01	4.42e-01	4.75e-01

Таблица 4.1.2 – Нечетная размерность матриц

n	Время, ns		
	ALG	CW OPT	CW
11	5.17e-06	4.56e-06	5.05e-06
21	3.25e-05	2.58e-05	2.59e-05
31	9.63e-05	7.23e-05	7.69e-05
41	2.14e-04	1.64e-04	1.73e-04
51	4.10e-04	3.15e-04	3.38e-04
61	7.13e-04	5.37e-04	5.70e-04
71	1.11e-03	8.36e-04	8.79e-04
81	1.67e-03	1.28e-03	1.35e-03
91	2.33e-03	1.73e-03	1.85e-03
101	3.23e-03	2.36e-03	2.49e-03
151	1.08e-02	8.03e-03	8.50e-03
201	2.74e-02	2.13e-02	2.29e-02
251	6.80e-02	5.27e-02	5.66e-02
301	1.16e-01	9.13e-02	9.82e-02
351	1.82e-01	1.40e-01	1.52e-01
401	2.58e-01	1.99e-01	2.14e-01
451	4.21e-01	3.26e-01	3.50e-01
501	5.74e-01	4.43e-01	4.76e-01

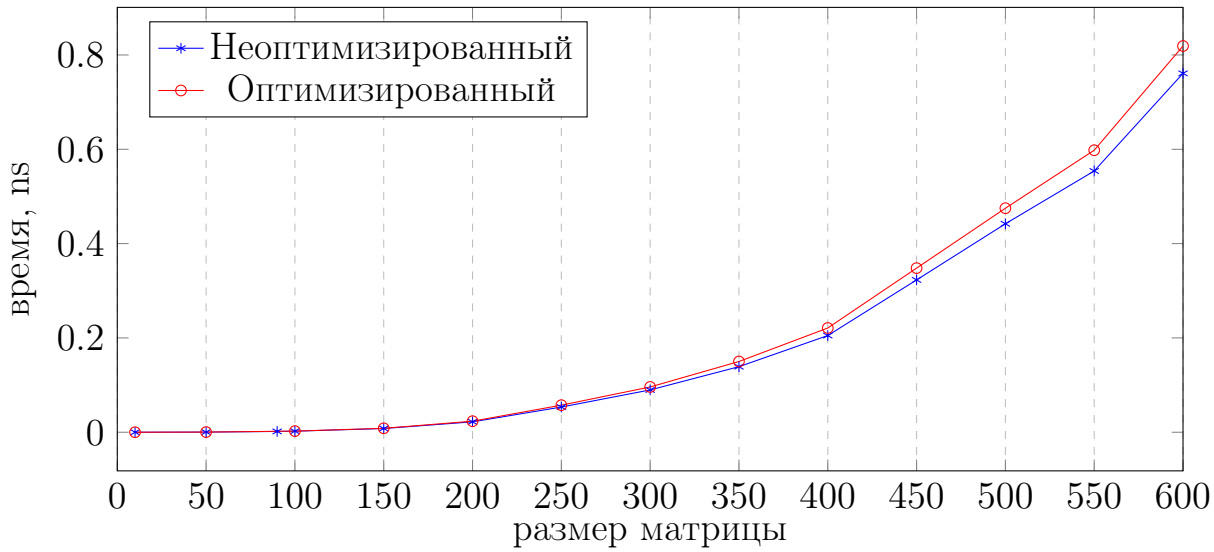

Рисунок 4.1.1 – Четная размерность матрицы

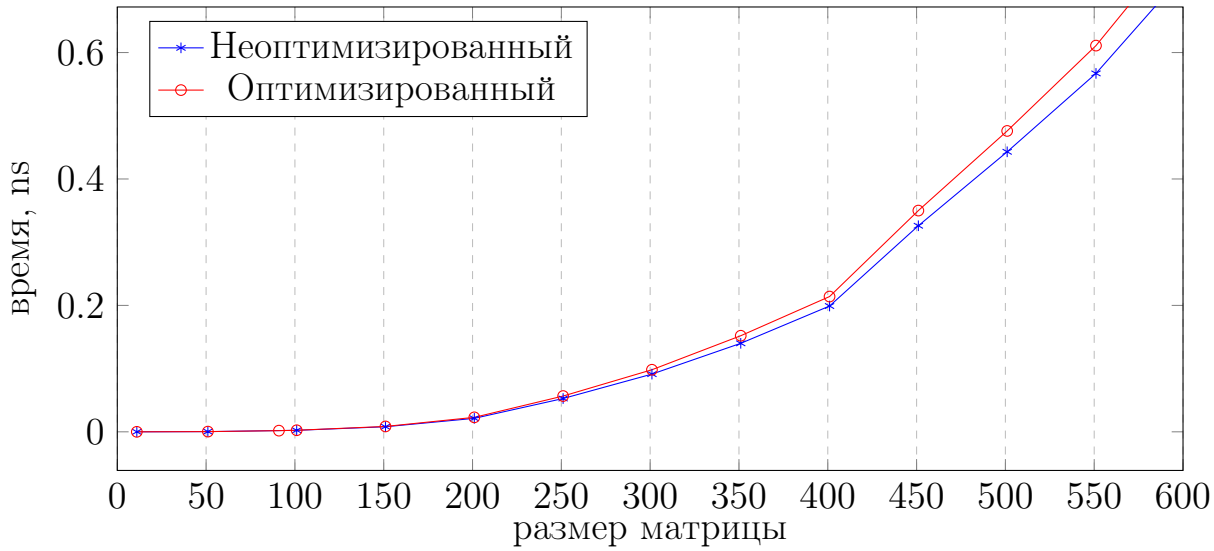

Рисунок 4.1.2 – Нечетная размерность матрицы

Рисунок 4.1 – Время выполнения алгоритма Копперсмита – Винограда

Вывод

В результате проведенного эксперимента был получен следующий вывод: алгоритм Винограда работает дольше за счет большого количества операций. Оптимизированный алгоритм работает быстрее, так как там меньше операций по сравнению с обычным алгоритмом Винограда и стандартным алгоритмом.

Заключение

В ходе выполнения лабораторной работы были решены следующие задачи:

- были изучены и реализованы 3 алгоритма перемножения матриц: обычный, Копперсмита-Винограда, модифицированный Копперсмита-Винограда;
- был произведен анализ трудоёмкости алгоритмов на основе теоретических расчетов и выбранной модели вычислений;
- был сделан сравнительный анализ алгоритмов на основе экспериментальных данных;
- подготовлен отчет о лабораторной работе.

Оптимизированный алгоритм Винограда быстрее обычного на 5 процентов при размерах матрицы 500 на 500.

Поставленная цель достигнута.

Литература

[1] Coppersmith Don, Winograd Shmuel. Matrix Multiplication via Arithmetic Progressions // Journal of Symbolic Computation. 1990.

[2] Golang Документация[Электронный ресурс]. Режим доступа: `https://go.dev/doc/` (дата обращения: 24.09.2022).

[3] Vscode Документация[Электронный ресурс]. Режим доступа: `https://code.visualstudio.com/docs` (дата обращения: 24.09.2022).

[4] Pop OS 22.04 LTS [Электронный ресурс]. Режим доступа: `https://pop.system76.com` (дата обращения: 04.09.2022).

[5] Linux – Документация [Электронный ресурс]. Режим доступа: `https://docs.kernel.org` (дата обращения: 24.09.2022).

[6] Процессор AMD® Ryzen 7 2700 eight-core processor × 16 [Электронный ресурс]. Режим доступа: `https://www.amd.com/en/products/cpu/amd-ryzen-7-2700` (дата обращения: 04.09.2022).