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)

ОТЧЕТ

По лабораторной работе №2

По курсу: «Анализ алгоритмов»

Тема: «Алгоритм Копперсмита — Винограда»

Студент: Керимов А. Ш.

Группа: ИУ7-54Б

Преподаватели: Волкова Л. Л.,
Строганов Ю. В.

Москва, 2019

Оглавление

Введение

Алгоритм Копперсмита — Винограда — алгоритм умножения квадратных матриц, предложенный в 1987 году Д. Копперсмитом и Ш. Виноградом [1]. В исходной версии асимптотическая сложность алгоритма составляла $O(n^{2,3755})$, где n — размер стороны матрицы. Алгоритм Копперсмита — Винограда, с учетом серии улучшений и доработок в последующие годы, обладает лучшей асимптотикой среди известных алгоритмов умножения матриц [2].

На практике алгоритм Копперсмита — Винограда не используется, так как он имеет очень большую константу пропорциональности и начинает выигрывать в быстродействии у других известных алгоритмов только для матриц, размер которых превышает память современных компьютеров [3]. Поэтому пользуются алгоритмом Штрассена по причинам простоты реализации и меньшей константе в оценке трудоемкости.

Алгоритм Штрассена [4] предназначен для быстрого умножения матриц. Он был разработан Фолькером Штрассеном в 1969 году и является обобщением метода умножения Карацубы на матрицы.

В отличие от традиционного алгоритма умножения матриц, алгоритм Штрассена умножает матрицы за время $\Theta(n^{\log_2 7}) = O(n^{2.81})$, что даёт выигрыш на больших плотных матрицах начиная, примерно, от 64×64.

Несмотря на то, что алгоритм Штрассена является асимптотически не самым быстрым из существующих алгоритмов быстрого умножения матриц, он проще программируется и эффективнее при умножении матриц относительно малого размера.

Задачи работы

Реализовать алгоритмы умножения матриц, описанные ниже.

1. Классический алгоритм умножения.
2. Алгоритм Копперсмита — Винограда.
3. Улучшенный Алгоритм Копперсмита — Винограда.

1 Аналитическая часть

1.1 Описание алгоритмов

1.1.1 Стандартный алгоритм

Пусть даны две прямоугольные матрицы

$$A_{lm} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{l1} & a_{l2} & \dots & a_{lm} \end{pmatrix}, \quad B_{mn} = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \dots & b_{mn} \end{pmatrix}, \tag{1.1}$$

тогда матрица C

$$C_{ln} = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{l1} & c_{l2} & \dots & c_{ln} \end{pmatrix}, \tag{1.2}$$

где

$$c_{ij} = \sum_{r=1}^{m} a_{ir} b_{rj} \quad (i = \overline{1,l}; j = \overline{1,n}) \tag{1.3}$$

будет называться произведением матриц A и B. Стандартный алгоритм реализует данную формулу.

1.1.2 Алгоритм Копперсмита — Винограда

Если посмотреть на результат умножения двух матриц, то видно, что каждый элемент в нем представляет собой скалярное произведение соответствующих строки и столбца исходных матриц. Можно заметить также, что такое умножение допускает предварительную обработку, позволяющую часть работы выполнить заранее.

Рассмотрим два вектора $V = (v_1, v_2, v_3, v_4)$ и $W = (w_1, w_2, w_3, w_4)$. Их

скалярное произведение равно: $V \cdot W = v_1w_1 + v_2w_2 + v_3w_3 + v_4w_4,$, что эквивалентно

$$V \cdot W = (v_1+w_2)(v_2+w_1)+(v_3+w_4)(v_4+w_3)-v_1v_2-v_3v_4-w_1w_2-w_3w_4. \quad (1.4)$$

Несмотря на то, что второе выражение требует вычисления большего количества операций, чем стандартный алгоритм: вместо четырех умножений - шесть, а вместо трех сложений - десять, выражение в правой части последнего равенства допускает предварительную обработку: его части можно вычислить заранее и запомнить для каждой строки первой матрицы и для каждого столбца второй, то для каждого элемента будет необходимо выполнить лишь первые два умножения и последующие пять сложений, а также дополнительно два сложения. Из-за того, что операция сложения быстрее операции умножения, алгоритм должен работать быстрее стандартного [5].

1.2 Модель вычислений

Для последующего вычисления трудоемкости необходимо ввести модель вычислений:

1. +, -, /, %, ==, !=, <, >, <=, >=, [], ++, -- – имеют трудоемкость 1
2. трудоемкость оператора выбора `if условие then A else B` рассчитывается, как

$$f_{if} = f_{\text{условия}} + \begin{cases} f_A, & \text{если условие выполняется,} \\ f_B, & \text{иначе.} \end{cases} \quad (1.5)$$

3. трудоемкость цикла рассчитывается, как $f_{for} = f_{\text{инициализации}}+f_{\text{сравнения}}+N(f_{\text{тела}} + f_{\text{инициализации}} + f_{\text{сравнения}})$
4. трудоемкость вызова функции равна 0

Вывод

Были рассмотрены алгоритмы классического умножения матриц и алгоритм Винограда, основное отличие которых — наличие предварительной обработки, а также количество операций умножения.

2 Конструкторская часть

2.1 Разработка алгоритмов

2.1.1 Стандартный алгоритм

На рисунке 2.1 приведена схема стандартного алгоритма умножения матриц.

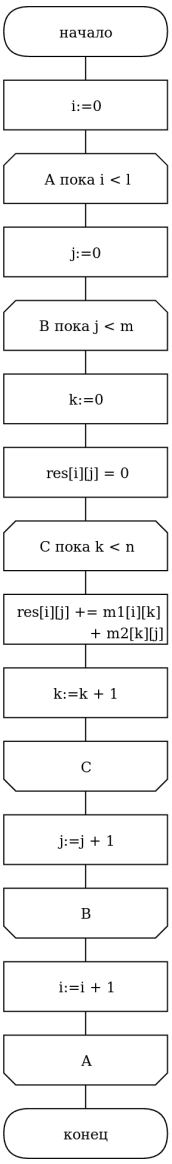

Рис. 2.1: Схема стандартного алгоритма умножения матриц

Видно, что для стандартного алгоритма не существует лучшего и худшего случаев, как таковых.

2.2 Алгоритм Копперсмита — Винограда

На рисунках 2.2 и 2.3 представлена схема алгоритма Копперсмита — Винограда.

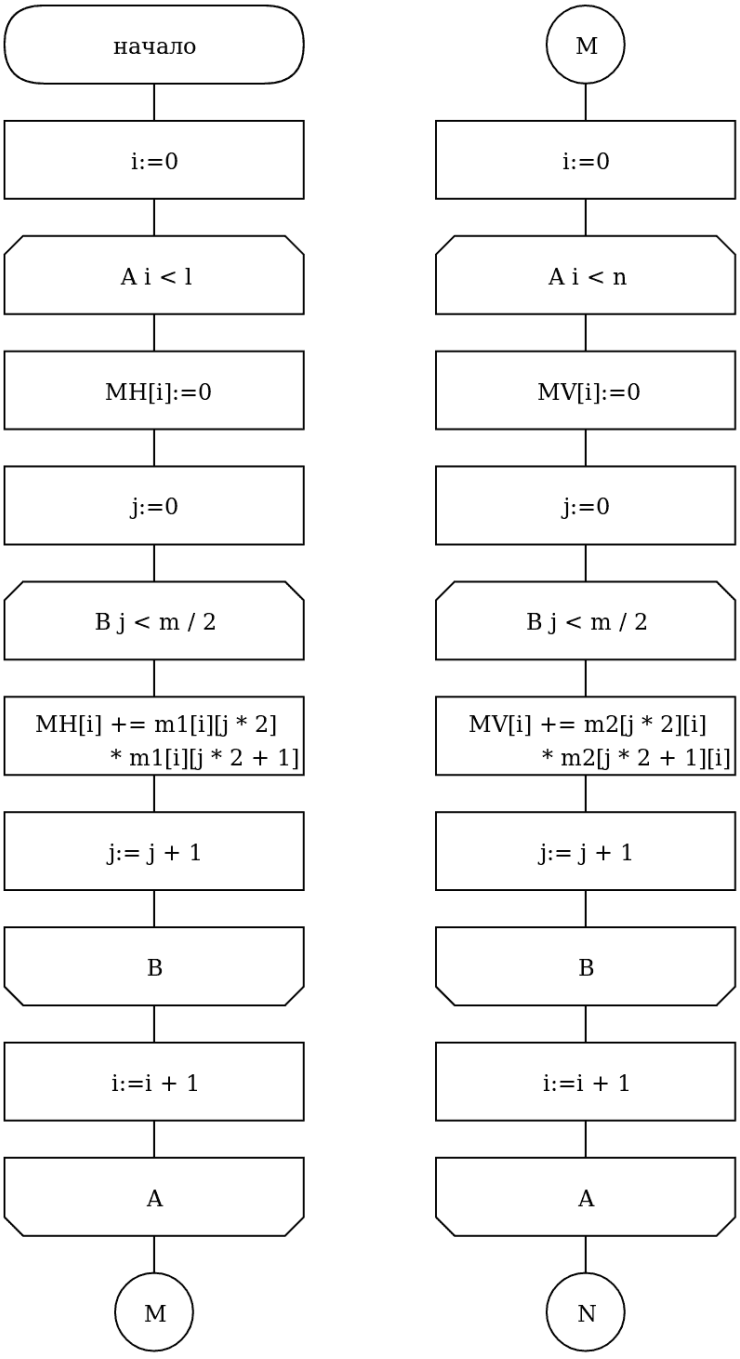

Рис. 2.2: Схема алгоритма Копперсмита — Винограда

Видно, что для алгоритма Виноградова худшим случаем являются матрицы нечетного размера, а лучшим четного, т.к. отпадает необходимость в

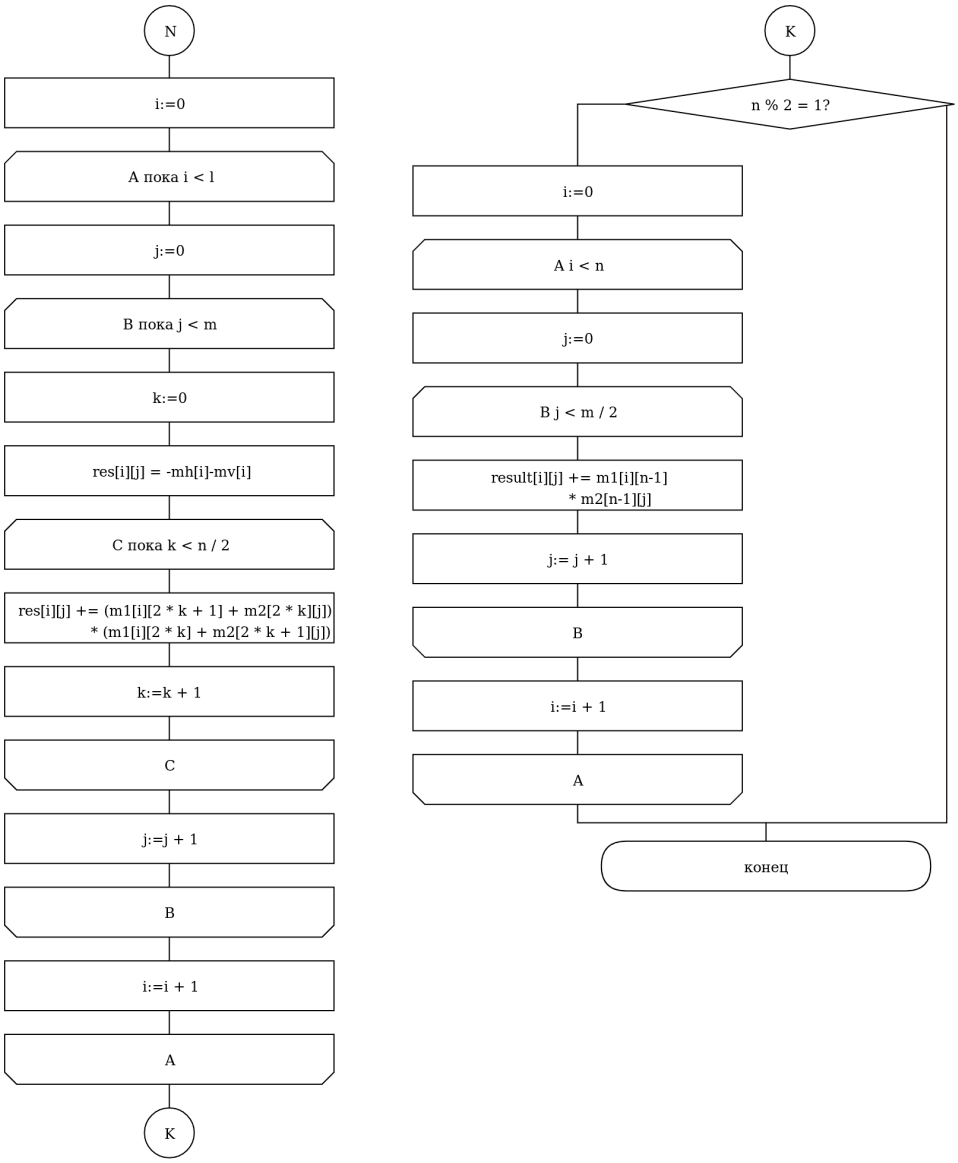

Рис. 2.3: Схема алгоритма Копперсмита — Винограда (продолжение)

последнем цикле.

В качестве оптимизаций можно:

- заранее считать в MH и MV отрицательные произведения
- заменить выражения вида `a = a + ...` на `a += ...`
- в циклах по k сделать шаг 2, избавившись тем самым от двух операций умножения на каждую итерацию

2.3 Трудоемкость алгоритмов

2.3.1 Стандартный алгоритм умножения матриц

Трудоёмкость стандартного алгоритма умножения матриц

$$f = 2 + l(2 + 2 + n(2 + 2 + m(2 + 9)) = 11lmn + 4ln + 4l + 2. \quad (2.1)$$

2.3.2 Алгоритм Копперсмита — Винограда

Трудоёмкость алгоритма Копперсмита — Винограда

1. $f_{row} = 2 + l(2 + 2 + 0.5n(2 + 10)) = 6ln + 4l + 2$
2. $f_{col} = 2 + n(2 + 2 + 0.5m(2 + 10)) = 6mn + 4n + 2$
3. $f_{matrix} = 2+l(2+2+m(2+2+7+0.5n(2+22)) = 12lmn+11lm+4l+2$
4. $f_{end} = \begin{cases} 2, & \text{чётная,} \\ 2 + 2 + l(2 + 2 + m(2 + 13)) = 15lm + 4l + 4, & \text{иначе.} \end{cases}$

Итого, для худшего случая (нечетный размер матрицы) $f = f_{row}+f_{col}+f_{matrix}+f_{end} = 12lmn + 6ln + 6mn + 26lm + 12l + 4n + 10 \approx 12lmn$

Для лучшего случая (четный размер матрицы: $f = f_{row}+f_{col}+f_{matrix}+f_{end} = 12lmn + 6ln + 6mn + 11lm + 8l + 4n + 8 \approx 12lmn$

2.3.3 Оптимизированный алгоритм Копперсмита — Винограда

Трудоёмкость оптимизированного алгоритма Копперсмита — Винограда

1. $f_{row} = 2 + l(2 + 2 + 0.5n(2 + 8)) = 5ln - l + 2$
2. $f_{col} = 2 + n(2 + 2 + 0.5m(2 + 8)) = 5mn - n + 2$

3. $f_{matrix} = 2+l(2+2+m(2+2+5+0.5n(2+15)) = 8.5lmn+9lm+4l+2$

4. $f_{end} = \begin{cases} 2, & \text{чётная} \\ 2+2+l(2+2+m(2+10)) = 12lm+4l+4, & \text{иначе.} \end{cases}$

Итого, для худшего случая (нечетный размер матрицы) $f = f_{row}+f_{col}+f_{matrix}+f_{end} = 8.5lmn + 5ln + 5mn + 21lm + 7l - n + 10 \approx 8lmn$

Для лучшего случая (четный размер матрицы): $f = f_{row}+f_{col}+f_{matrix}+f_{end} = 8.5lmn + 5ln + 5mn + 11lm + 3l - n + 8 \approx 8lmn$

Вывод

На основе теоретических данных, полученных из аналитического раздела, были построены схемы обоих алгоритмов умножения матриц. Оценены лучшие и худшие случаи их работы.

3 Технологическая часть

В данном разделе приведены средства реализации и листинг кода.

3.1 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран высокопроизводительный язык C++ [6], так как он предоставляет широкие возможности для эффективной реализации алгоритмов.

Для генерации псевдослучайных чисел использована функция 3.4.

Для замера времени работы алгоритма использованы точнейшие функции библиотеки `std::chrono`

3.2 Листинг кода

В листингах 3.1–3.3 приведены реализации алгоритмов умножения матриц. Функция замера времени работы алгоритма приведена в листинге 3.5.

```
matrix_t product(const matrix_t& m1, const matrix_t& m2) {
	const size_t l = m1.size();
	const size_t m = m2.size();

	if (!l || !m || m1[0].size() != m) {
		throw std::exception();
	}

	const size_t n = m2[0].size();

	auto result = create_matrix(l, n);
	for (size_t i = 0; i != l; ++i) {
		for (size_t j = 0; j != n; ++j) {
			for (size_t k = 0; k != m; ++k) {
				result[i][j] += m1[i][k] * m2[k][j];
			}
		}
	}

	return result;
}
```

Листинг 3.1: Стандартный алгоритм умножения матриц

```
namespace bad {
inline namespace detail {

row_t negative_row_products(const matrix_t& matrix, size_t rows, size_t cols) {
	auto result = row_t(rows, 0);
	for (size_t i = 0; i < rows; i++) {
		for (size_t j = 0; j < cols / 2; j++) {
			result[i] = result[i] + matrix[i][2*j] * matrix[i][2*j + 1];
		}
	}

	return result;
}

row_t negative_col_products(const matrix_t& matrix, size_t rows, size_t cols) {
	auto result = row_t(rows, 0);
	for (size_t i = 0; i < rows; i++) {
		for (size_t j = 0; j < cols / 2; j = j + 2) {
			result[i] = result[i] + matrix[2*j][i] * matrix[2*j + 1][i];
		}
	}

	return result;
}

} // namespace detail

matrix_t coppersmith_winograd_product(const matrix_t& m1, const matrix_t& m2) {
	const size_t l = m1.size();
	const size_t m = m2.size();

	if (!l || !m || m1[0].size() != m) {
		throw std::exception();
	}

	const size_t n = m2[0].size();

	const auto mh = negative_row_products(m1, l, m);
	const auto mv = negative_col_products(m2, n, m);

	auto result = create_matrix(l, n);
	for (size_t i = 0; i < l; i++) {
		for (size_t j = 0; j < m; j++) {
			result[i][j] = -(mh[i] + mv[j]);
			for (size_t k = 0; k < n / 2; k++) {
```

```
                        result[i][j] = result[i][j] + (m1[i][2*k + 1] + m2[2*k][j])
                            * (m1[i][2*k] + m2[2*k + 1][j]);
                }
            }
        }

        if (n % 2) {
            for (size_t i = 0; i < l; i++) {
                for (size_t j = 0; j < m; j++) {
                    result[i][j] = result[i][j] + m1[i][n - 1] * m2[n - 1][j];
                }
            }
        }

        return result;
}

} // namespace bad
```

Листинг 3.2: Алгоритм Копперсмита — Винограда

```
namespace good {
inline namespace detail {

row_t negative_row_products(const matrix_t& matrix, size_t rows, size_t cols) {
        auto result = row_t(rows, 0);
        for (size_t i = 0; i != rows; ++i) {
            for (size_t j = 0; j < cols - 1; j += 2) {
                result[i] -= matrix[i][j] * matrix[i][j + 1];
            }
        }

        return result;
}

row_t negative_col_products(const matrix_t& matrix, size_t rows, size_t cols) {
        auto result = row_t(rows, 0.);
        for (size_t j = 0; j < cols - 1; j += 2) {
            for (size_t i = 0; i != rows; ++i) {
                result[i] -= matrix[j][i] * matrix[j + 1][i];
            }
        }

        return result;
}

} // namespace detail

matrix_t coppersmith_winograd_product(const matrix_t& m1, const matrix_t& m2) {
```

```
        const size_t l = m1.size();
        const size_t m = m2.size();

        if (!l || !m || m1[0].size() != m) {
                throw std::exception();
        }

        const size_t n = m2[0].size();

        const auto mh = negative_row_products(m1, l, m);
        const auto mv = negative_col_products(m2, n, m);

        auto result = create_matrix(l, n);
        for (size_t i = 0; i != l; ++i) {
                for (size_t j = 0; j != m; ++j) {
                        result[i][j] = mh[i] + mv[j];
                        for (size_t k = 0; k < n - 1; k += 2) {
                                result[i][j] += (m1[i][k + 1] + m2[k][j]) * (m1[i][k] +
                                    m2[k + 1][j]);
                        }
                }
        }

        if (n & 1) {
                for (size_t i = 0; i != l; ++i) {
                        for (size_t j = 0; j != m; ++j) {
                                result[i][j] += m1[i][n - 1] * m2[n - 1][j];
                        }
                }
        }

        return result;
}

} // namespace good
```

Листинг 3.3: Оптимизированный алгоритм Копперсмита — Винограда

```
int dont_try_to_guess() {
        static thread_local std::mt19937 generator(std::random_device{}());
        static thread_local std::uniform_int_distribution<int> distribution(-1000, 1000);
        return distribution(generator);
}
```

Листинг 3.4: Продвинутый генератор псевдослучайных чисел

```
double count_time(product_func_type product_func, const matrix_t& m1, const matrix_t&
    m2) {
        constexpr size_t N = 1;
```

```
        const auto start = std::chrono::high_resolution_clock::now();
        for (int i = 0; i != N; ++i) {
                product_func(m1, m2);
        }
        const auto end = std::chrono::high_resolution_clock::now();

        return
            static_cast<double>(std::chrono::duration_cast<std::chrono::nanoseconds>(end
            - start).count()) / 1.0e6 / N;
}
```

Листинг 3.5: Функция замера времени работы алгоритмов

3.3 Тестирование фунций

В таблице 3.1 приведены тесты для функций, реализующих стандартный алгоритм умножения матриц, алгоритм Винограда и оптимизированный алгоритм Винограда. Тесты пройдены успешно.

Матрица 1	Матрица 2	Ожидаемый результат
$\begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} 6 & 12 & 18 \\ 6 & 12 & 18 \\ 6 & 12 & 18 \end{pmatrix}$
$\begin{pmatrix} 1 & 2 \\ 1 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 \\ 1 & 2 \end{pmatrix}$	$\begin{pmatrix} 3 & 6 \\ 3 & 6 \end{pmatrix}$
$\begin{pmatrix} 2 \end{pmatrix}$	$\begin{pmatrix} 2 \end{pmatrix}$	$\begin{pmatrix} 4 \end{pmatrix}$
$\begin{pmatrix} 1 & -2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} -1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} 0 & 4 & 6 \\ 4 & 12 & 18 \\ 4 & 12 & 18 \end{pmatrix}$
$\begin{pmatrix} 1 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 \end{pmatrix}$	Не могут быть перемножены

Таблица 3.1: Тестирование функций

Вывод

Правильный выбор инструментов разработки позволил эффективно реализовать алгоритмы, настроить модульное тестирование и выполнить исследовательский раздел лабораторной работы.

4 Исследовательская часть

4.1 Технические характеристики

- Операционная система: Ubuntu 19.10 64-bit.
- Память: 3,8 GiB.
- Процессор: Intel® Core™ i3-6006U CPU @ 2.00GHz

4.2 Время выполнения алгоритмов

Алгоритмы тестировались с помощью функции замера процессорного времени `std::chrono::high_resolution_clock::now()`

Результаты замеров приведены в таблицах 4.1 и 4.2. На рисунках 4.1 и 4.2 приведены графики зависимостей времени работы алгоритмов от размеров матриц. Здесь и далее: СА — стандартный алгоритм, АКВ — алгоритм Копперсмита — Винограда, УАКВ — улучшенный алгоритм Копперсмита — Винограда.

Размер матрицы	**Время, мс**		
	СА	**АКВ**	**УАКВ**
100	39.33352950	21.84159700	18.13774250
200	191.06879950	183.80495450	154.97888150
300	643.15356100	633.46189000	525.78953000
400	1648.43648550	1662.16656000	1374.16655050
500	3516.70210600	3389.16216450	2932.47787900
600	6132.82750350	6010.69887450	4984.55201700
700	9777.77753600	9541.70957550	7983.47457400
800	14795.68465900	14814.81216150	12807.17200350
900	21514.64538100	21362.82759250	17893.11289600
1000	30421.01086100	30058.90910500	25223.32126700

Таблица 4.1: Время работы алгоритмов при чётных размерах матриц

Размер матрицы	Время, мс		
	СА	**АКВ**	**УАКВ**
101	23.04713700	23.10028000	19.48659500
201	199.53655900	191.50602500	156.05946000
301	653.65077000	650.97895400	535.23752000
401	1625.69624600	1633.80264400	1331.79177500
501	3295.49082900	3363.90729600	2702.37849600
601	5886.34526600	6034.57134900	4863.69247000
701	9664.31852300	9781.15727700	8039.56718700
801	14855.55541700	15110.44329600	12477.65650800
901	21483.38911900	21818.22948000	18126.33664900
1001	29756.94948100	30313.03802300	25429.57026500

Таблица 4.2: Время работы алгоритмов при нечётных размерах матриц

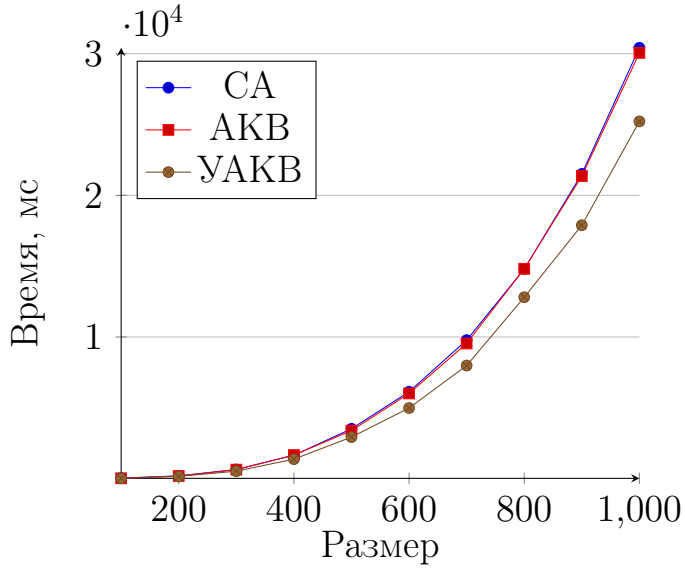

Рис. 4.1: Зависимость времени работы алгоритма от размера квадратной матрицы (чётного)

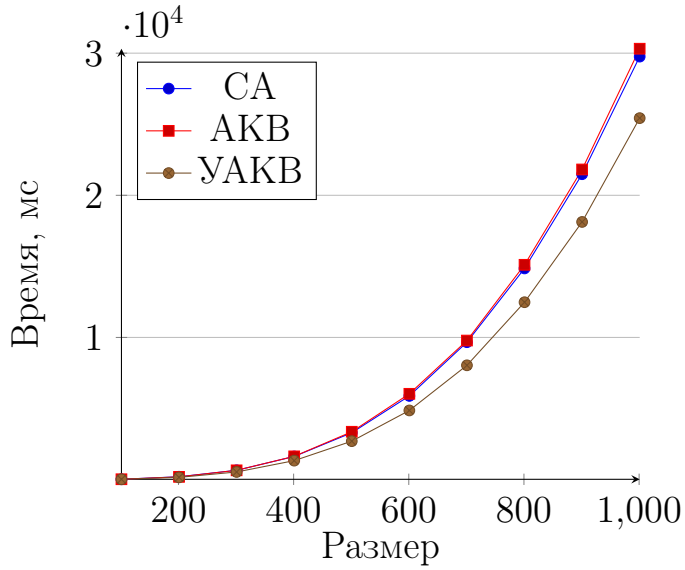

Рис. 4.2: Зависимость времени работы алгоритма от размера квадратной матрицы (нечётного)

Вывод

Время работы алгоритма Винограда незначительно меньше стандартного алгоритма умножения, однако оптимизированная реализации имеет заметный прирост в скорости работы, на матрицах размером 1000x1000 уже около 18%.

Заключение

В рамках лабораторной работы были рассмотрены и реализованы стандартный алгоритм умножения матриц и алгоритм Винограда. Была рассчитана их трудоемкость и произведены замеры времени работы реализованных алгоритмов. На основании этого произведено сравнение их эффективности. Оптимизированный алгоритм Винограда имеет заметный выигрыш в эффективности работы по сравнению с остальными алгоритмами: уже на матрицах размером 1000×1000 улучшенный алгоритм Копперсмита — Винограда работает примерно на 18% быстрее двух остальных рассмотренных алгоритмов.

Литература

[1] Coppersmith D., Winograd S. Matrix multiplication via arithmetic progressions // Journal of Symbolic Computation. 1990. no. 9. P. 251–280.

[2] Group-theoretic Algorithms for Matrix Multiplication / H. Cohn, R. Kleinberg, B. Szegedy et al. // Proceedings of the 46th Annual Symposium on Foundations of Computer Science. 2005. October. P. 379–388.

[3] Robinson S. Toward an Optimal Algorithm for Matrix Multiplication // SIAM News. 2005. November. Vol. 38, no. 9.

[4] Strassen V. Gaussian Elimination is not Optimal // Numerische Mathematik. 2005. Vol. 13, no. 9. P. 354–356.

[5] Погорелов Дмитрий Александрович Таразанов Артемий Михайлович Волкова Лилия Леонидовна. Оптимизация классического алгоритма Винограда для перемножения матриц // Журнал №1. 2019. Т. 49.

[6] Working Draft, Standard for ProgrammingLanguage C++. URL: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4713.pdf`. 2017.