



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»

Отчет по лабораторной работе № 5 по курсу «Анализ алгоритмов»

Тема Организация асинхронного взаимодействия потоков вычисления на примере
конвейерных вычислений

Студент Фам М. Х.

Группа ИУ7-52Б

Оценка (баллы)

Преподаватель Волкова Л. Л.

Содержание

Введение	3
1 Аналитическая часть	4
1.1 Описание конвейерной обработки данных	4
2 Конструкторская часть	5
2.1 Алгоритмы обработки матриц	5
3 Технологическая часть	12
3.1 Требования к программному обеспечению	12
3.2 Выбор языка программирования	12
3.3 Описание используемых типов данных	12
3.4 Реализация алгоритмов	13
3.5 Функциональные тесты	16
4 Исследовательская часть	17
4.1 Технические характеристики устройства	17
4.2 Демонстрация работы программы	17
4.3 Время выполнения алгоритмов	18
4.4 Вывод	19
Заключение	20
Список использованных источников	21

Введение

Задачу ускорения обработки данных можно решить с помощью введения конвейерной обработки. Вводится конвейерная лента и обрабатывающие устройства. Данные поступают на обрабатывающее устройство, которое после завершения обработки передает их дальше по ленте, и не ожидая завершения цикла, приступает к обработке следующих данных.

Целью данной лабораторной работы является изучение принципов конвейерной обработки данных.

Для достижения поставленной цели необходимо решить следующие задачи:

- исследовать основы конвейерной обработки данных;
- привести схемы алгоритмов, используемых для конвейерной и линейной обработок данных;
- реализовать перечисленные алгоритмы;
- провести сравнительный анализ времени работы этих алгоритмов;
- описать и обосновать полученные результаты в отчете о выполненной лабораторной работе.

1 Аналитическая часть

В этом разделе будет описан конвейерный принцип обработки данных.

1.1 Описание конвейерной обработки данных

Конвейер — способ организации вычислений, используемый в современных процессорах и контроллерах с целью повышения их производительности (увеличения числа инструкций, выполняемых в единицу времени — эксплуатация параллелизма на уровне инструкций), технология, используемая при разработке компьютеров и других цифровых электронных устройств [1].

Этот способ можно использовать в обработке данных, суть которой состоит в выделении отдельных этапов выполнения общей операции. Каждый этап, выполнив свою работу, передает результат следующему, одновременно принимая новую порцию данных.

В данной лабораторной работе необходимо реализовать следующую последовательность операций:

1. Формулировать матрицу B как сумму матрицы A и транспонированной матрицы A .
2. Формулировать матрицу C как сумму матрицы B и транспонированной матрицы B .
3. Заменить матрицу C на произведение C на A .

2 Конструкторская часть

В данном разделе будут приведены схемы конвейерной и линейной реализаций алгоритмов обработки данных.

2.1 Алгоритмы обработки матриц

На рис. 2.1 – 2.6 приведены схемы линейной и конвейерной реализаций алгоритмов обработки матрицы, схема трёх лент для конвейерной обработки матрицы, а также схемы реализаций этапов обработки матрицы.

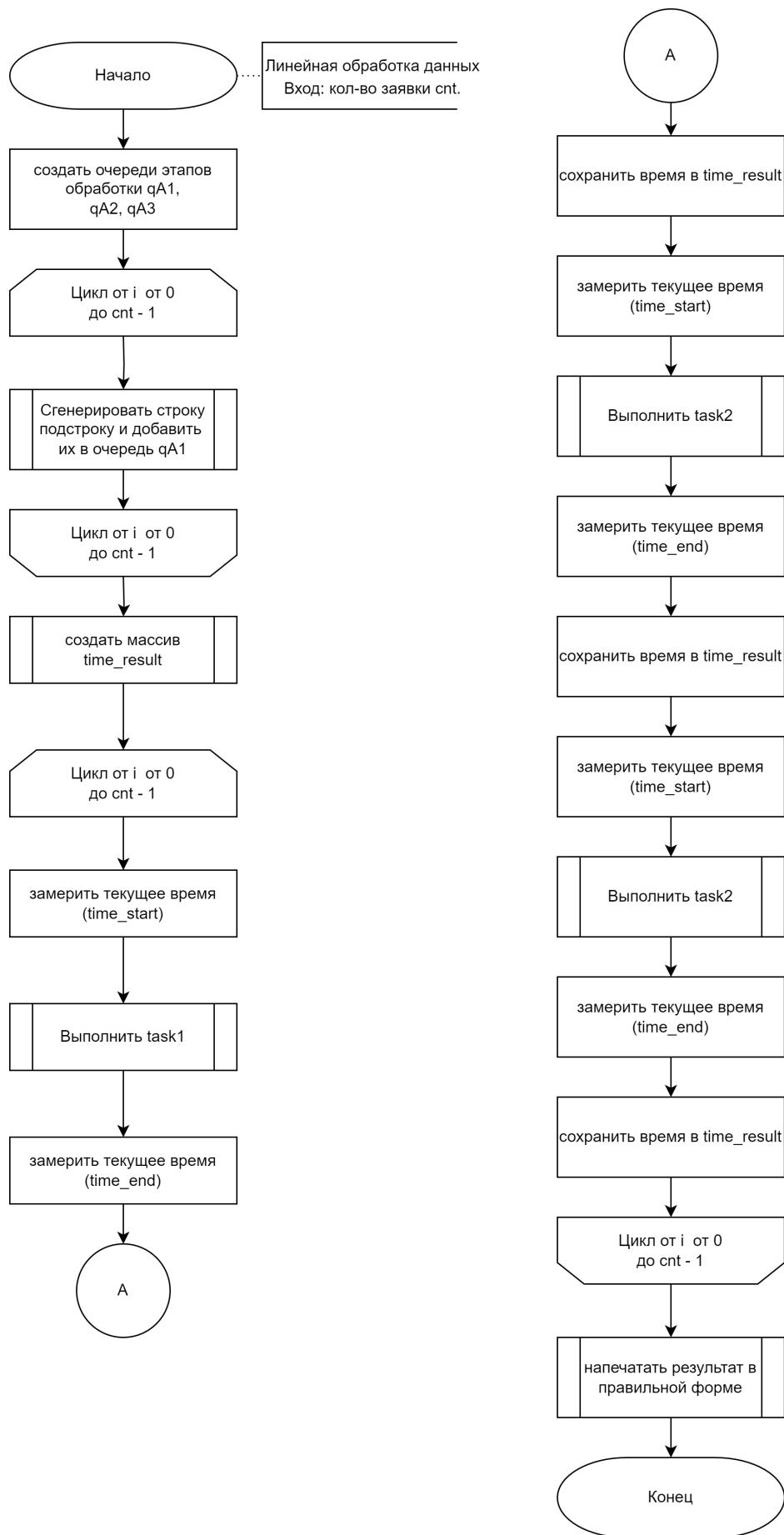


Рисунок 2.1 – Схема алгоритма линейной обработки матрицы

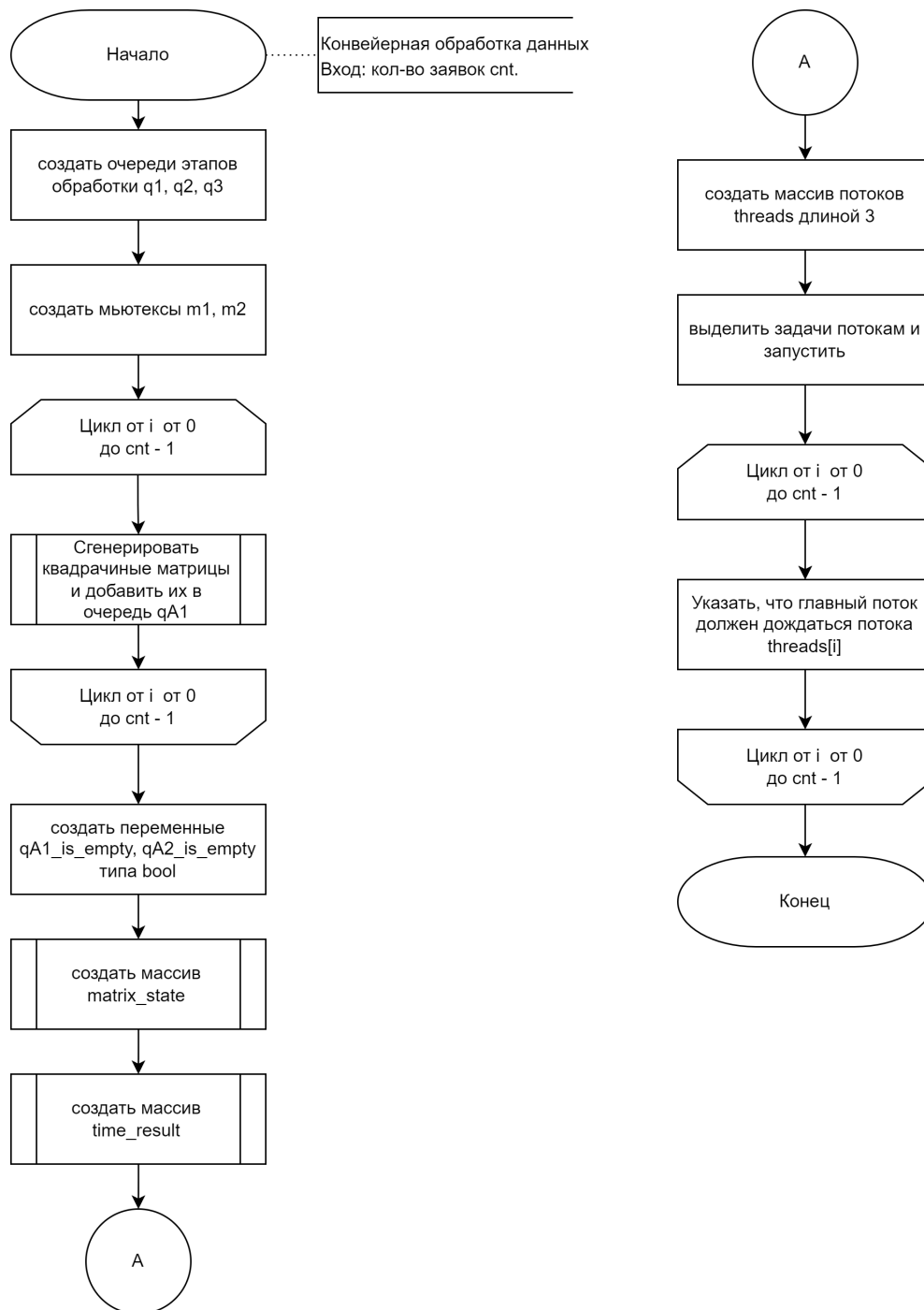


Рисунок 2.2 – Схема алгоритма конвейерной обработки матрицы

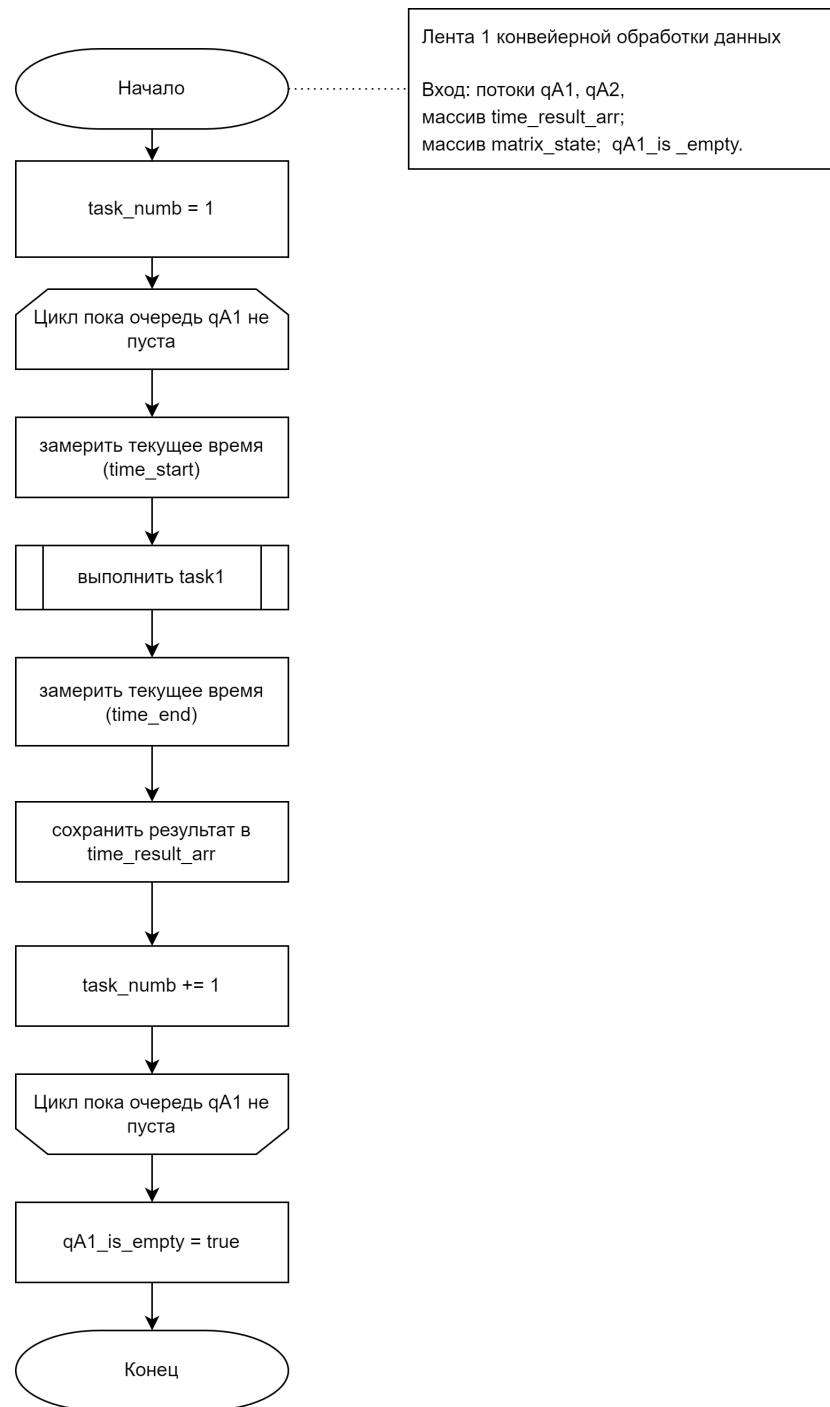


Рисунок 2.3 – Схема 1-ой ленты конвейерной обработки матрицы

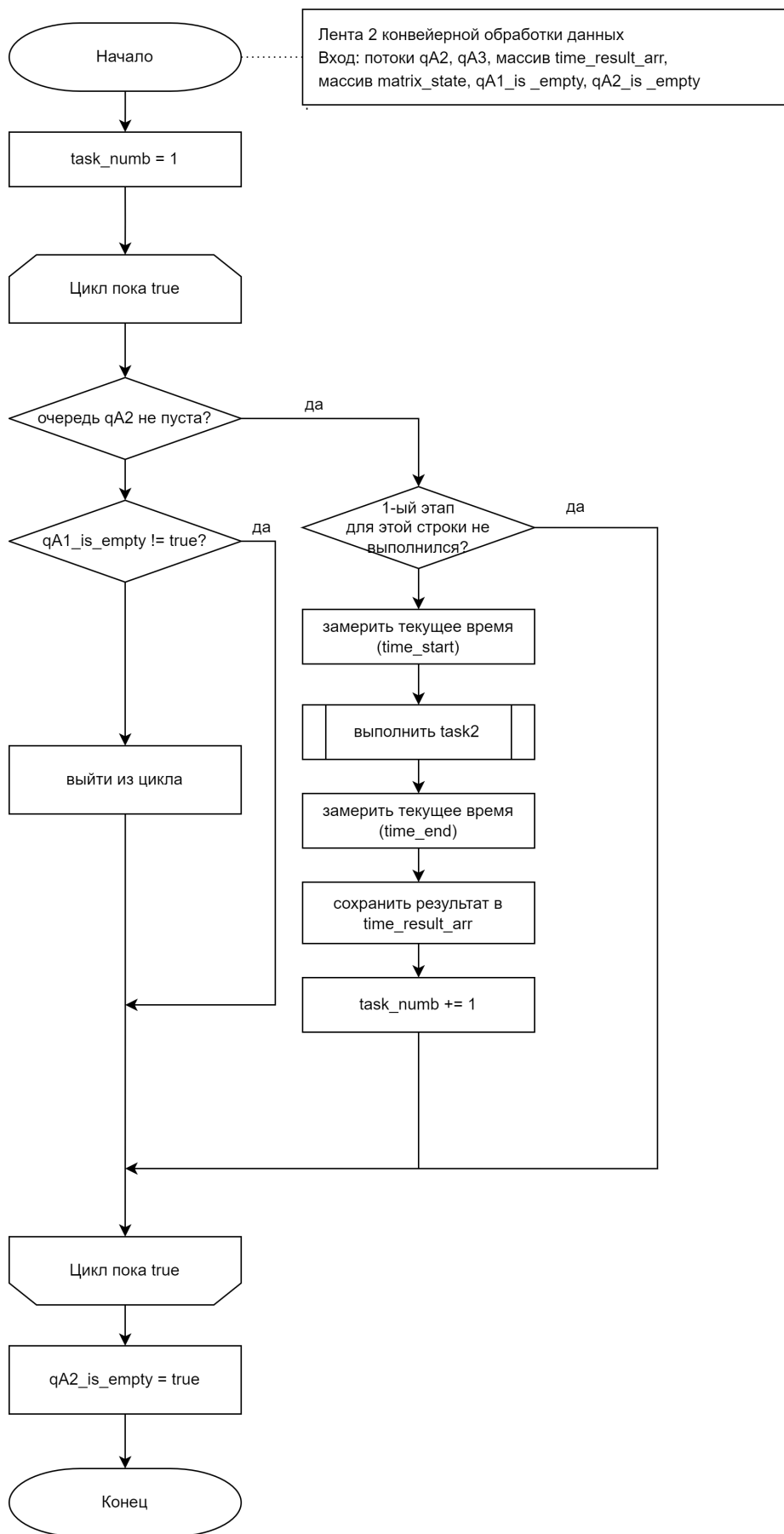


Рисунок 2.4 – Схема 2-ой ленты конвейерной обработки матрицы

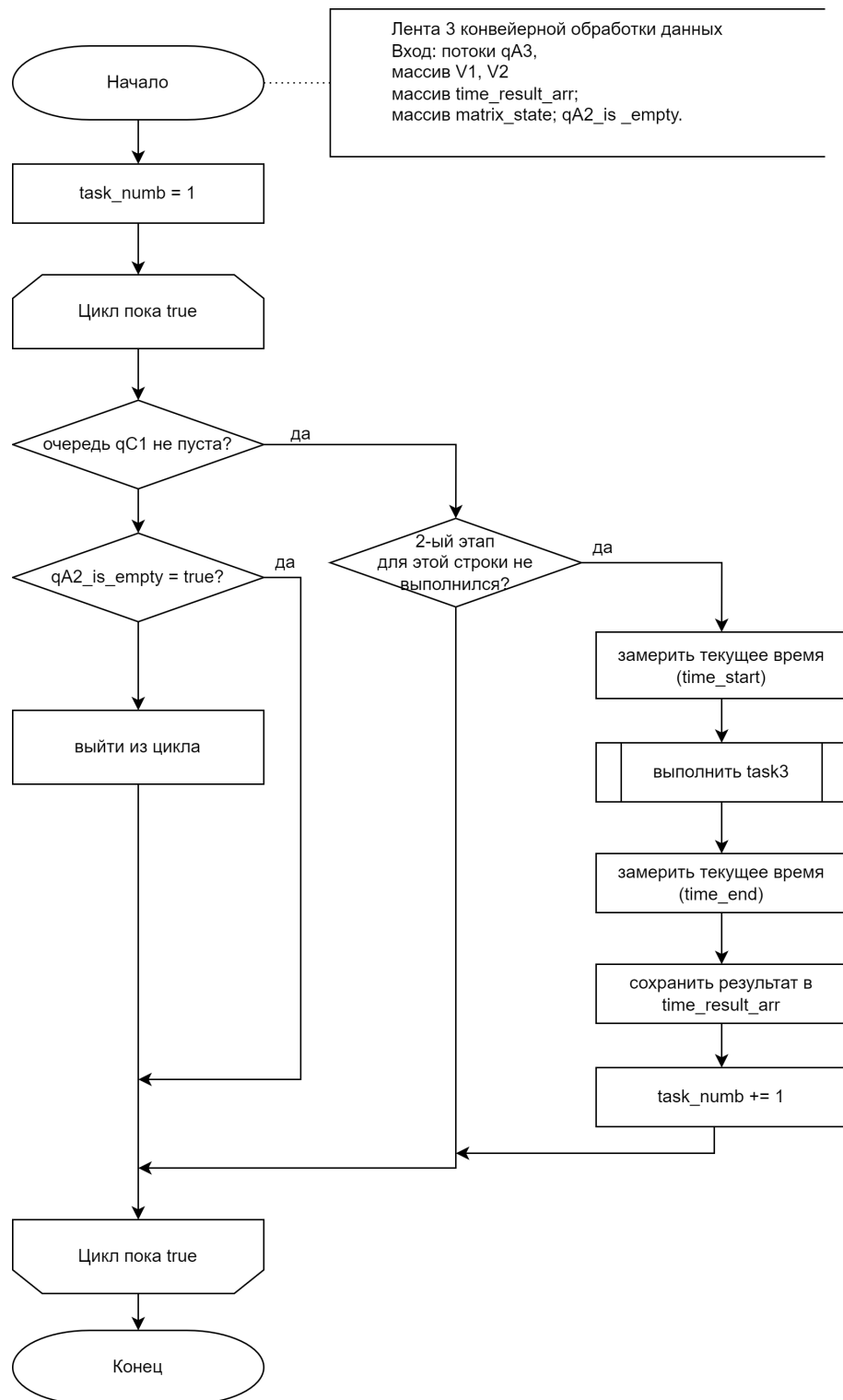


Рисунок 2.5 – Схема 3-ей ленты конвейерной обработки матрицы

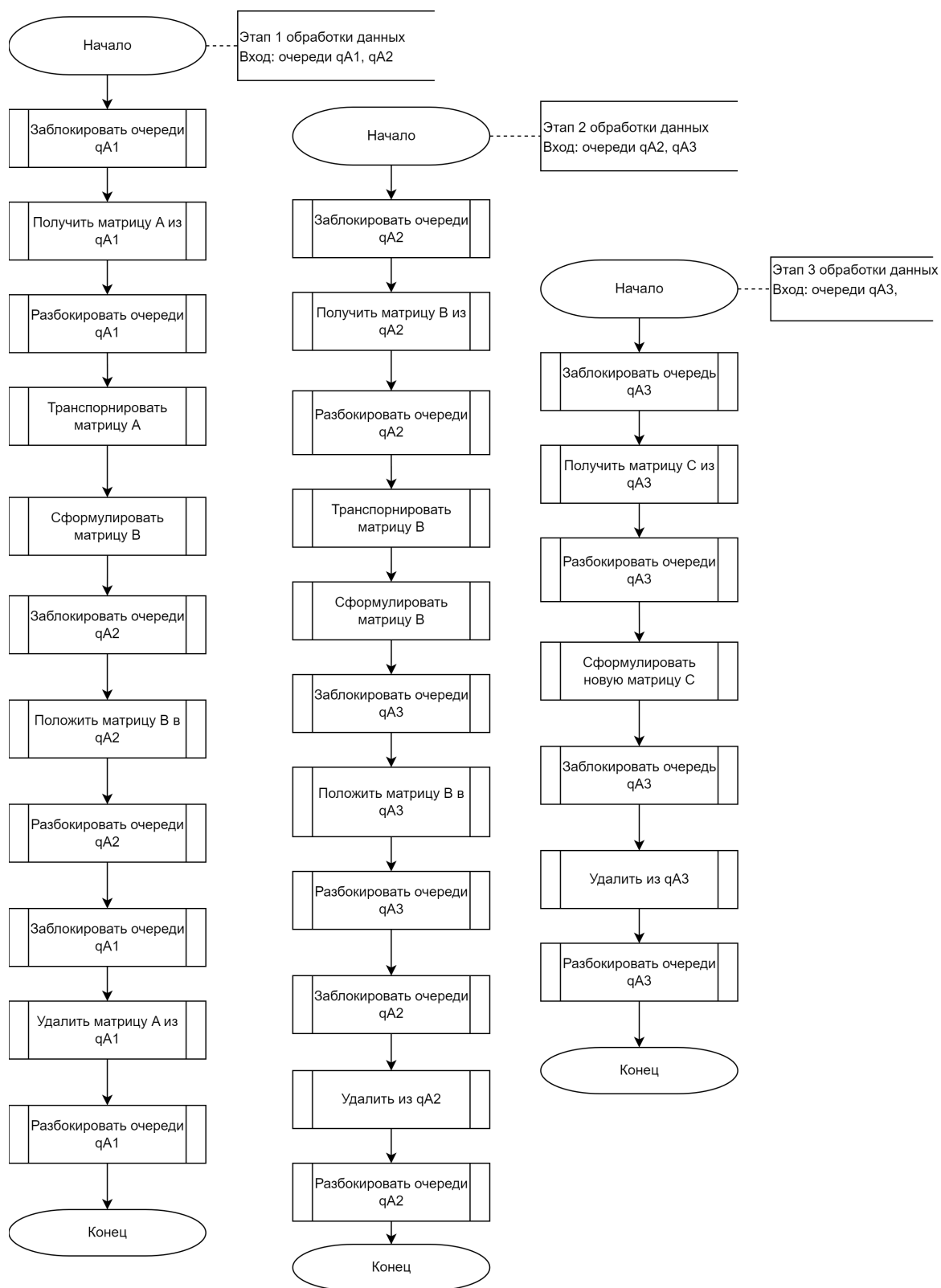


Рисунок 2.6 – Схема реализаций этапов обработки матрицы

3 Технологическая часть

В данном разделе будут приведены требования к программному обеспечению, средства реализации, листинги кода, а также функциональные тесты.

3.1 Требования к программному обеспечению

В качестве входных данных задается количество строк и столбцов матрицы *matr*, которое должно быть больше 0, а все элементы матрицы имеют тип *int*. Количество матриц больше 0. Выходные данные — табличка с номерами матриц, номерами этапов (лент) её обработки, временем начала обработки текущей матрицы на текущей ленте, временем окончания обработки текущей матрицы на текущей ленте.

3.2 Выбор языка программирования

В данной работе для реализации был выбран язык программирования *C++* [2], так как он предоставляет весь необходимый функционал для выполнения работы. Для замера времени работы использовалась функция *std::chrono::system_clock::now()* [3]. Визуализация графиков с помощью библиотеки *Matplotlib* [4].

3.3 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие структуры данных:

- матрица - двумерный вектор элементов типа *int*;
- размер матриц и их количество - числа типа *int*;

3.4 Реализация алгоритмов

В листингах 3.1–3.5 представлены функции для конвейерного и ленивого алгоритмов обработки матриц.

Листинг 3.1 – Алгоритм линейной обработки данных

```
1 void liner(int cnt, bool is_count)
2 {
3     queue<Matrix> qA1;
4     queue<Matrix> qA2;
5     queue<Matrix> qA3;
6     mutex m1;
7     mutex m2;
8
9     chrono::time_point<chrono::system_clock> time_start, time_end,
10        time_begin = chrono::system_clock::now();
11
12     vector<res_time_t> time_result_arr;
13     init_time_result_arr(time_result_arr, time_begin, cnt, 3);
14
15     for (int i = 0; i < cnt; i++)
16     {
17         Matrix m(100, 100);
18         m.randomMatrix();
19         qA1.push(m);
20     }
21
22     for (int i = 0; i < cnt; i++)
23     {
24         time_start = chrono::system_clock::now();
25         task1(ref(m1), ref(qA1), ref(qA2));
26         time_end = chrono::system_clock::now();
27
28         save_result(time_result_arr, time_start, time_end,
29                    time_result_arr[0].time_begin, i + 1, 1);
30
31         time_start = chrono::system_clock::now();
32         task2(ref(m1), ref(m2), ref(qA2), ref(qA3));
33         time_end = chrono::system_clock::now();
34
35         save_result(time_result_arr, time_start, time_end,
36                    time_result_arr[0].time_begin, i + 1, 2);
37
38         time_start = chrono::system_clock::now();
39         task3(ref(m2), ref(qA3));
40         time_end = chrono::system_clock::now();
```

```

39
40     save_result(time_result_arr, time_start, time_end,
41                 time_result_arr[0].time_begin, i + 1, 3);
42 }
43 if (is_count)
44 {
45     printf("|      %4d      |    %.6f  \n",
46           cnt, time_result_arr[cnt - 1].end);
47 }
48 else
49 {
50     print_res_time(time_result_arr, cnt * 3);
51 }
52 }

```

Листинг 3.2 – Алгоритм конвейерной обработки данных

```

1 void parallel(int cnt, bool is_count)
2 {
3     queue<Matrix> qA1;
4     queue<Matrix> qA2;
5     queue<Matrix> qA3;
6
7     bool qA1_is_empty = false;
8     bool qA2_is_empty = false;
9     mutex m1;
10    mutex m2;
11    mutex time_mutex;
12
13    for (int i = 0; i < cnt; i++)
14    {
15        Matrix m(100, 100);
16        m.randomMatrix();
17        qA1.push(m);
18    }
19
20    vector<strings_state_t> state(cnt);
21    for (int i = 0; i < cnt; i++)
22    {
23        strings_state_t tmp_state;
24        tmp_state.stage_1 = false;
25        tmp_state.stage_2 = false;
26        tmp_state.stage_3 = false;
27        state[i] = tmp_state;
28    }
29    chrono::time_point<chrono::system_clock> time_begin =
30        chrono::system_clock::now();
31    vector<res_time_t> time_result_arr;

```

```

31     init_time_result_arr(time_result_arr, time_begin, cnt, 3);
32     thread threads[3];
33     threads[0] = thread(parallel_stage_1, ref(time_mutex), ref(m1),
34                         ref(qA1), ref(qA2), ref(state), ref(time_result_arr),
35                         ref(qA1_is_empty));
36
37     threads[1] = thread(parallel_stage_2, ref(time_mutex), ref(m1),
38                         ref(m2), ref(qA2), ref(qA3), ref(state), ref(time_result_arr),
39                         ref(qA1_is_empty), ref(qA2_is_empty));
40
41     threads[2] = thread(parallel_stage_3, ref(time_mutex), ref(m2),
42                         ref(qA3), ref(state), ref(time_result_arr), ref(qA2_is_empty));
43
44     for (int i = 0; i < 3; i++)
45         threads[i].join();
46     if (is_count)
47         printf("|      %4d      |    %.6f   \n",
48               cnt, time_result_arr[cnt - 1].end);
49     else
50         print_res_time(time_result_arr, cnt * 3);
51 }

```

Листинг 3.3 – Алгоритм транспонирования матрицы

```

1     Matrix transposeMatrix()
2     {
3         int r = this->arr[0].size();
4         int c = this->arr.size();
5         Matrix res(r, c);
6         for (int i = 0; i < r; i++)
7             for (int j = 0; j < c; j++)
8                 res.arr[i][j] = this->arr[j][i];
9         return res;
10    }

```

Листинг 3.4 – Алгоритм сложения двух матриц

```

1     Matrix sumMatrix(Matrix& m)
2     {
3         int r = this->arr.size();
4         int c = this->arr[0].size();
5         Matrix res(r, c);
6         for (int i = 0; i < r; i++)
7             for (int j = 0; j < c; j++)
8                 res.arr[i][j] = this->arr[i][j] + m.arr[i][j];
9         return res;
10    }

```

Листинг 3.5 – Алгоритм умножения двух матриц

```

1     Matrix mulMatrix(Matrix& m)

```

```

2      {
3          int r = this->arr.size();
4          int c = this->arr[0].size();
5          Matrix res(r, c);
6          for (int i = 0; i < r; i++)
7              for (int j = 0; j < c; j++)
8                  for (int k = 0; k < r; k++)
9                      res.arr[i][j] += (arr[i][k] * m.arr[k][j]);
10         return res;
11     }

```

3.5 Функциональные тесты

В таблице 3.1 приведены функциональные тесты для конвейерного и линейного алгоритмов обработки матриц. Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты

Строк	Столбцов	Метод обр.	Алгоритм	Ожидаемый результат
0	10	10	Конвейерный	Сообщение об ошибке
k	10	10	Конвейерный	Сообщение об ошибке
10	10	k	Конвейерный	Сообщение об ошибке
100	100	20	Конвейерный	Вывод результ. таблички
100	100	20	Линейный	Вывод результ. таблички

4 Исследовательская часть

В данном разделе будет проведен сравнительный анализ алгоритмов по времени выполнения в зависимости от количества матриц и их размеров.

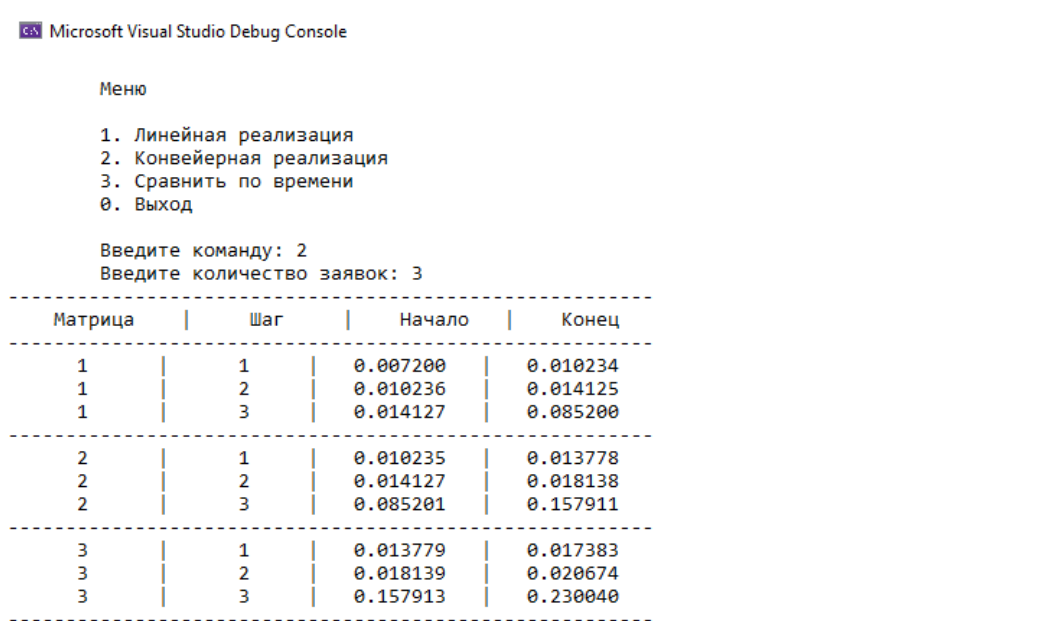
4.1 Технические характеристики устройства

Тестирование проводилось на устройстве со следующими техническими характеристиками:

- операционная система Window 10 Home Single Language;
- память 8 Гб;
- процессор 11th Gen Intel(R) Core(TM) i7-1165G7 2.80 ГГц, 4 ядра.

4.2 Демонстрация работы программы

На рис. 4.1 представлена демонстрация работы программы.



Microsoft Visual Studio Debug Console

Меню

1. Линейная реализация
2. Конвейерная реализация
3. Сравнить по времени
0. Выход

Введите команду: 2
Введите количество заявок: 3

Матрица	Шаг	Начало	Конец
1	1	0.007200	0.010234
1	2	0.010236	0.014125
1	3	0.014127	0.085200
2	1	0.010235	0.013778
2	2	0.014127	0.018138
2	3	0.085201	0.157911
3	1	0.013779	0.017383
3	2	0.018139	0.020674
3	3	0.157913	0.230040

Рисунок 4.1 – Демонстрация работы программы при конвейерной обработки данных

4.3 Время выполнения алгоритмов

Результаты замеров времени работы алгоритмов обработки матриц для конвейерной и ленточной реализаций представлены на рисунках 4.2 – 4.3. Замеры времени проводились в секундах и усреднялись для каждого набора одинаковых экспериментов.

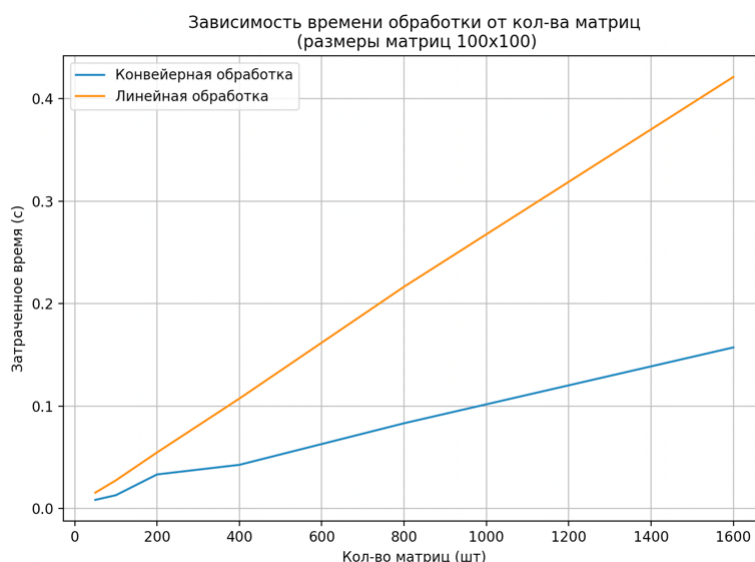


Рисунок 4.2 – Зависимость времени работы алгоритмов от кол-ва матриц (размеры матриц 100x100)

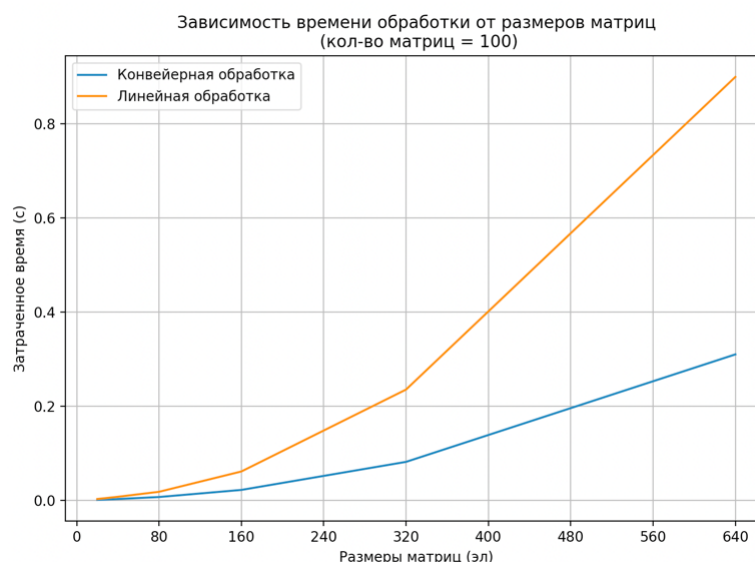


Рисунок 4.3 – Зависимость времени работы алгоритмов от размера матриц (кол-во матриц = 100)

4.4 Вывод

В этом разделе были указаны технические характеристики машины, на которой происходило сравнение времени работы алгоритмов обработки матриц для конвейерной и ленточной реализаций.

В результате замеров времени было установлено, что конвейерная реализация обработки лучше линейной при большом кол-ве матриц (в 2.5 раза при 400 матрицах, в 2.6 раза при 800 и в 2.7 при 1600). Так же конвейерная обработка показала себя лучше при увеличении размеров обрабатываемых матриц (в 2.8 раза при размере матриц 160x160, в 2.9 раза при размере 320x320 и в 2.9 раза при матрицах 640x640). Значит при большом кол-ве обрабатываемых матриц, а так же при матрицах большого размера стоит использовать конвейерную реализацию обработки, а не линейную.

Заключение

Было экспериментально подтверждено различие во временной эффективности конвейрной и линейной реализаций обработок матриц. В результате исследований можно сделать вывод о том, что при большом кол-ве обрабатываемых матриц, а так же при матрицах большого размера стоит использовать конвейрную реализацию обработки, а не линейную (при 1600 матриц конвейрная быстрее в 2.7 раза, а при матрицах 640x640 быстрее в 2.9 раза).

Цель лабораторной работы была достигнута, в ходе выполнения данной лабораторной работы были решены следующие задачи:

- изучены основы конвейрной обработки данных;
- приведены схемы алгоритмов, используемых для конвейрной и линейной обработок данных;
- реализованы перечисленные алгоритмы;
- проведен сравнительный анализ времени работы этих алгоритмов;
- описаны и обоснованы полученные результаты в отчете о выполненной лабораторной работе.

Список использованных источников

1. Конвейерная организация [Электронный ресурс]. Режим доступа: http://www.citforum.mstu.edu.ru/hardware/svk/glava_5.shtml
2. C++ reference [Электронный ресурс]. Режим доступа: <https://en.cppreference.com/w/>.
3. std::chrono::system_clock::now - cppreference.com [Электронный ресурс]. Режим доступа: https://en.cppreference.com/w/cpp/chrono/system_clock/now.
4. Matplotlib documentation [Электронный ресурс]. Режим доступа: <https://matplotlib.org/stable/index.html>