МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»

Институт информатики, математики и электроники
Факультет информатики
Кафедра технической кибернетики

Векторные алгоритмы матричных разложений

Отчет по лабораторной работе № 2
Вариант 1

Студенты группы 6407 Берлин Д.И.
Советников В.Р.
Проверил Головашкин Д.Л.

САМАРА 2018

СОДЕРЖАНИЕ

ЗАДАНИЕ

В данной работе необходимо реализовать LU-разложение, используя алгоритмы KIJ, JKI, а также I - блочный алгоритм.

Теоретические сведения

Алгоритмы KIJ и JKI LU-разложения матриц

Пусть дана матрица A размерности $n \times n$, тогда результатом разложения данной матрицы будут: нижнетреугольная матрица L и верхнетреугольная матрица U размерности $n \times n$, причем, матрица L будет иметь на главной диагонали единицы и

$$a_{ij} = \sum_{k=1}^{min\{i,j\}} l_{ik} \cdot u_{kj}.$$

LU-разложение возможно только если все диагональные миноры не вырождены, этого можно добиться при помощи диагонального преобладания.

Ниже приведен алгоритм KIJ разложения с замещением:

```
for k=1:n-1
        A(k+1:n,k) = A(k+1:n,k)/A(k,k)
        A(k+1:n, k+1:n) = A(k+1:n, k+1:n)  + A(k+1:n, k)*A(k, k+1:n)
end
```

Алгоритм JKI разложения матриц с замещением:

```
for j=1:n
        for k=1:j-1
                A(k+1:j,j) = A(k+1:j,j) - A(k+1:j,k)* A(k,j)
        end
        for k=1:j-1
                A(j:n,j) = A(j:n,j) - A(j:n,k)A(k,j)
        end
        A(j+1:n,j) = A(j+1:n,j)/A(j,j)
end
```

I - блочный алгоритм LU-разложения матриц

Блочный алгоритм разложения матриц применяют с целью повышения эффективности использования кэш-памяти CPU. Исходная матрица A делится на одинаковые блоки размерностью m, затем матрицу разделяют следующим образом:

$\begin{pmatrix} A_{11} & A_{21} \\ A_{12} & A_{22} \end{pmatrix}$, где A_{11} имеет размерность $R^{m\times m}$, A_{12} имеет размерность $R^{n-m\times m}$, A_{21} имеет размерность $R^{m\times n-m}$, A_{22} имеет размерность $R^{n-m\times n-m}$.

Дальше выполняется рекурсивная конструкция, до тех пор, пока не будет разложена вся матрица A:

1) Произвести LU-разложение матрицы $A_{11} = L_{11}U_{11}$.
2) Решить СЛАУ вида: $L_{21}U_{11} = A_{21}$ и найти L_{21}.
3) Решить СЛАУ вида: $L_{11}U_{12} = A_{12}$ и найти U_{12}.
4) $A_{22} = A_{22} - L_{21}U_{12}$ и повторить все действия для матрицы A_{22}.

За счет выбора размерности оптимального блока, при разложении матриц блочным алгоритмом размер обрабатываемых данных на каждой итерации не превышает объема кэш-памяти.

Цель эксперимента

В данной работе планируется изучение и применение следующих алгоритмов:

1. алгоритм KIJ;
2. алгоритм JKI;
3. блочный алгоритм на основе векторного алгоритма KIJ;

Также необходимо провести сравнение времени работы и определение эффективности каждого алгоритма в зависимости от размерности матриц, как с использованием оптимизации, так и без неё.

Инструментарий

Для вычислений использовалось следующее оборудование:

- ноутбук с процессором AMD A8-3520M;
- частота работы ядра 1.60 GHz;
- Установленная память (ОЗУ) 8 ГБ;
- операционная 64-х разрядная система Windows 7 home basic;
- L2 кэш 4 МБ;
- для построения графиков использовался язык программирования Octave;
- язык программирования C++, компилятор Visual C++;
- Библиотека BLAS.

Данное оборудование удовлетворяет всем необходимым аппаратным и программным требованиям, т.к. оно позволяет выполнять высокопроизводительные вычисления с достаточно высокой степенью точности.

Параметры эксперимента

В качестве нижней границы матрицы была взята размерность матриц 64x64, максимальная граница размерности матриц 1024x1024. Шаг взят 64, как достаточно большое значение, на котором можно увидеть различие между двумя следующими друг за другом значениями времени. Максимальная граница выбрана таким образом, поскольку при размерности свыше 1024 время выполнения будет серьёзно увеличиваться. А выбор минимальной границы обусловлен тем, что при размерах матрицы меньше, чем 64x64 все алгоритмы по времени ведут себя примерно одинаково.

Язык программирования C++ выбран исходя из личных предпочтений, а так же ввиду строчного хранения матриц.

Оптимальный размер блока для блочного алгоритма LU-разложения был выбран экспериментально.

Теоретическое ожидание

При запуске программы ожидается самое быстрое выполнение алгоритма из стандартной библиотеки BLAS. На втором месте должен быть блочный алгоритм с оптимальным размером блока, так как он использует алгоритм KIJ. На третьем месте ожидается KIJ алгоритм, ввиду выбора языка программирования. И на последнем алгоритм JKI.

Теоретически ожидается ≈ кубическая зависимость времени от размерности матриц.

Ожидается, что время работы алгоритмов, в которых используется механизм оптимизации, окажется значительно меньше, чем время алгоритмов, которые выполняются без использования векторизации.

Оптимальный размер блока для блочного алгоритма найдём по формуле:

$$block = \sqrt{\frac{cache}{double\_size \cdot 3}} \approx 413,$$

где $cache$ – размер КЭШ памяти (Байт);

$double\_size$ – размер типа double. Он равен 8 байт;

$block$ – размерность блока. Его нужно выразить;

3 – количество матриц, каждая из которых хранится в памяти.

Результаты

Целью данной работы является сравнение работы алгоритмов разложения матриц, выяснение их преимуществ и недостатков.

При проведении эксперимента по определению оптимального блока для блочного алгоритма для матрицы размерностью 1024x1024, не удалось однозначно установить оптимальный размер блока.

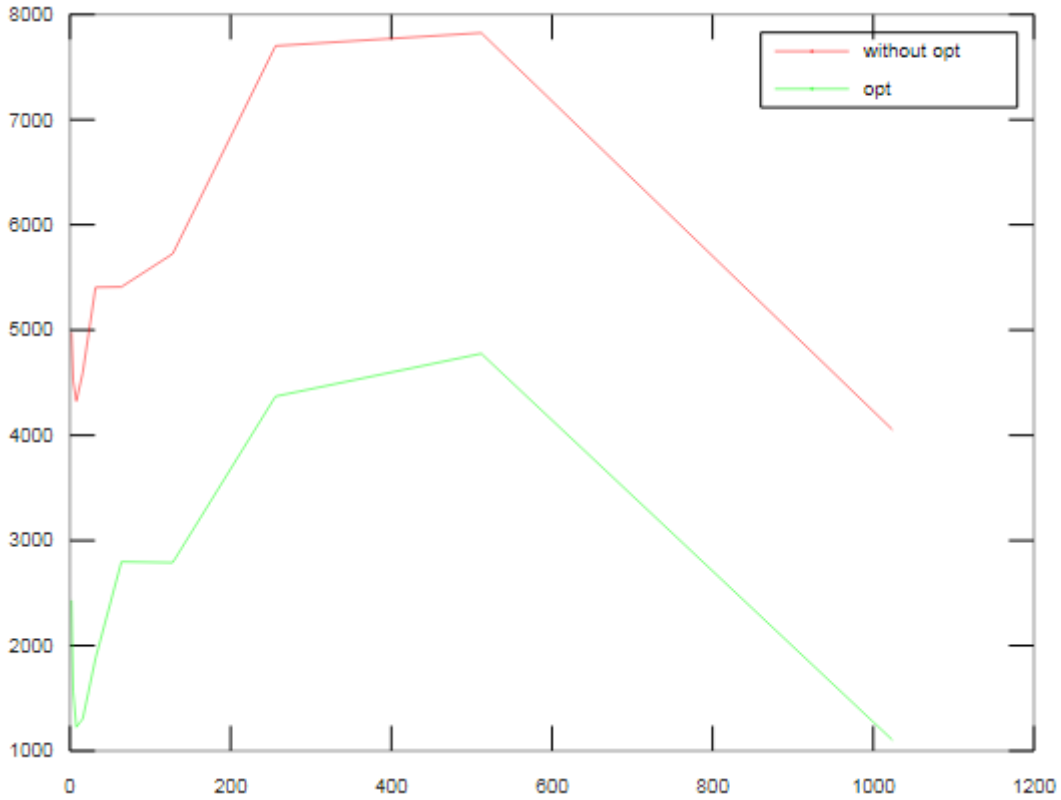

Рисунок 1 – Определение оптимального блока для блочного алгоритма

Ниже в таблице 1 отображены значения времени разложения матрицы размерностями 1024 на 1024 с помощью блочного алгоритма при различных размерах блока.

Таблица 1 - Значения времени для блочного алгоритма при различном размере блока (Время измеряется в мс)

Размер блока	2	4	8	16	32	64	128	256	512	1024
Время	4978	4521	4325	4529	5406	5410	5725	7700	7825	4023
Время (С опт.)	2432	1568	1224	1305	1881	2797	2790	4370	4776	1103

При разложении матриц различной размерности, с использованием различных алгоритмов, был получен результат, который можно увидеть на рисунках 2 и 3.

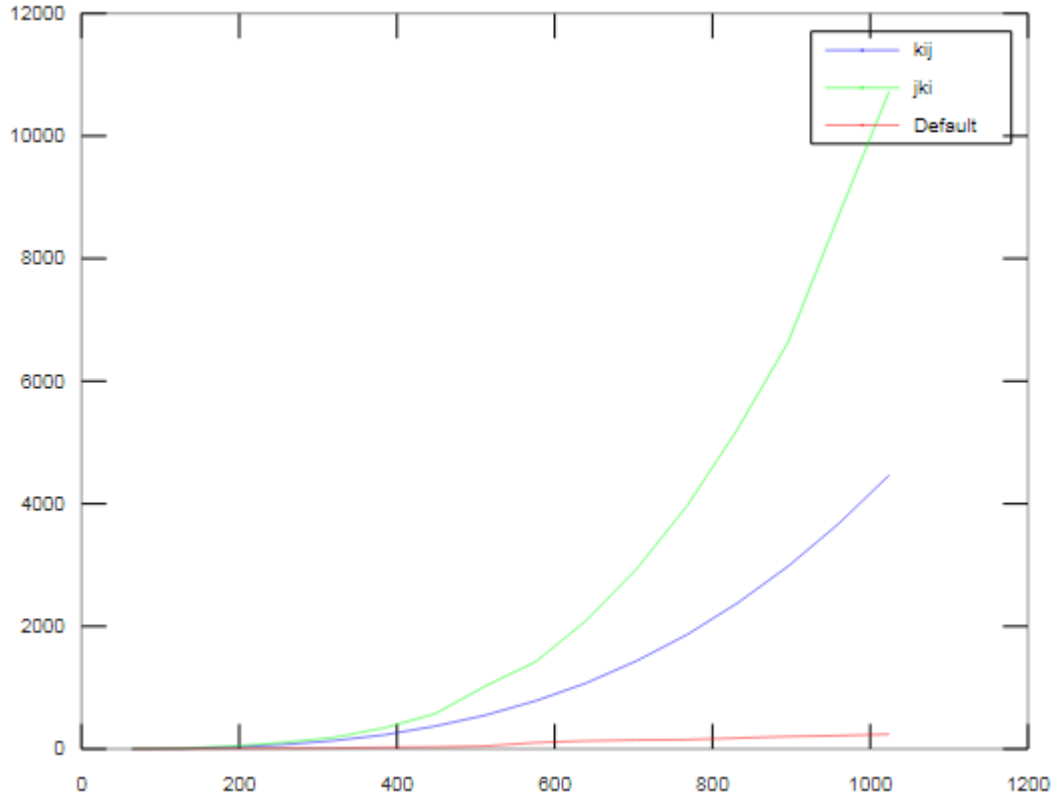

Рисунок 2 – График зависимости времени выполнения от размерности матриц без использования оптимизации

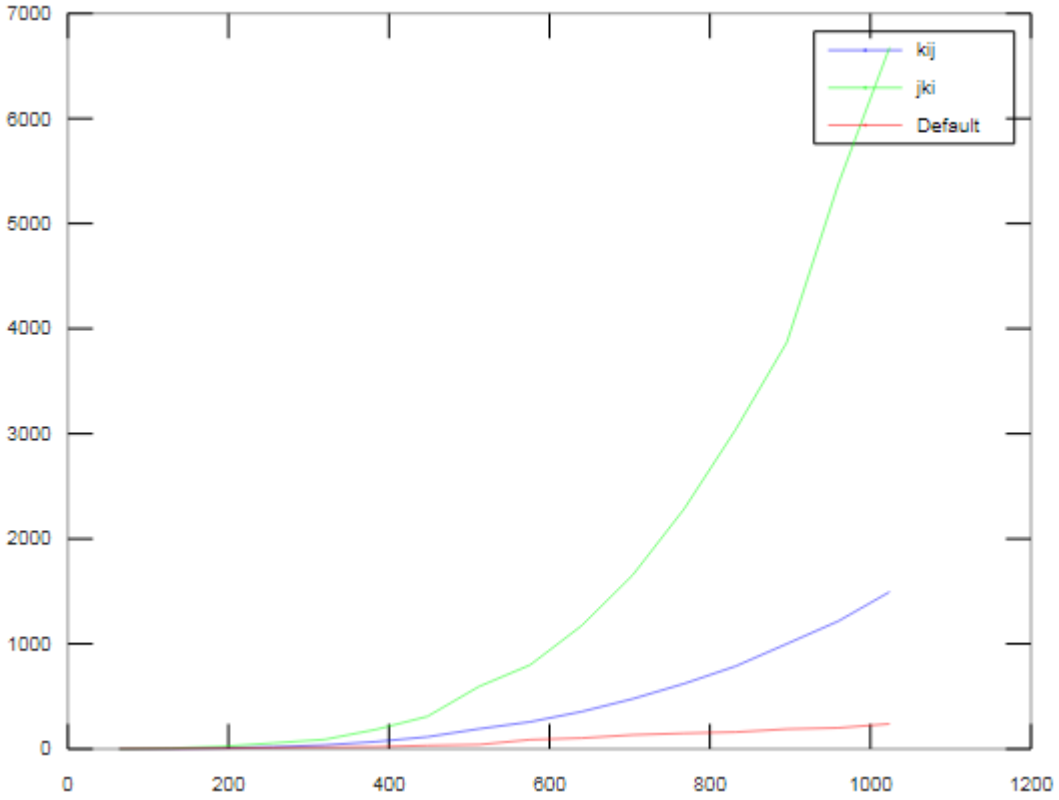

Рисунок 3 – График зависимости времени выполнения от размерности матриц с использования оптимизации

На основании рисунков 2,3 можно заметить: до n = 256 примерно все алгоритмы работают одинаково. Как и предполагалось лидирующими по быстродействию алгоритмами являются алгоритм из стандартной библиотеки BLAS. Самым медленным, как и ожидалось, оказался стандартный алгоритм JKI. Алгоритм KIJ и его блочная реализация работали примерно одинаковое количество времени. Из рисунка 3 видно, как после включения оптимизации быстродействие всех алгоритмов заметно улучшилось.

Ниже в таблицах 2 представлены значения времени в зависимости от размерностей входных матриц для каждого алгоритма.

Таблица 2 – Результаты работы программы без использования оптимизации (Время измеряется в мс)

Размерность матрицы	Время работы Алгоритма		
	KIJ	JKI	BLAS
64	1	2	2
128	8	11	3
192	28	42	7
256	67	104	10
320	133	185	14
384	228	343	23
448	369	571	30
512	549	1024	45
576	784	1422	100
640	1075	2093	130
704	1436	2941	143
768	1866	3968	150
832	2381	5222	178
896	2981	6634	203
960	3671	8696	115

1024	4465	10728	239

Будем использовать стандартный механизм оптимизации встроенный в среду разработки Visual Studio C++. Найдём также все значения времени для всех алгоритмов с использованием этого механизма. Ниже в таблице 3 представлены значения времени в зависимости от размера входных матриц для каждого алгоритма с использованием оптимизации.

Таблица 3 - Результаты работы программы с использованием оптимизации (Время измеряется в мс)

Размерность матрицы	Время работы Алгоритма		
	KIJ	JKI	BLAS
64	0	0	1
128	2	5	3
192	8	23	7
256	19	56	11
320	37	90	15
384	68	185	20
448	114	309	33
512	191	592	40
576	258	798	87
640	354	1172	103
704	477	1655	133
768	622	2286	149
832	788	3039	160
896	1001	3871	188
960	1215	5376	200

1024	1495	6676	239

Анализ результатов

Для начала проанализируем результаты работы блочного алгоритма, нам не удалось отыскать оптимальные размеры блока, вероятно, это связано с особенностями практической реализации этого алгоритма, а так же с занятостью КЭШ памяти на конкретный момент работы программы. Не смотря на это, график ведет себя в соответствии с теоретическими ожиданиями, на отрезке от [0,16], тогда значение оптимального размера блока, для данной реализации, можно считать равным 8.

Проанализируем результаты, показанные на рисунках 2, 3, стандартный BLAS алгоритм, как и ожидалось, оказалось намного быстрее остальных. Алгоритм KIJ, как и его блочный вариант разделяют второе место по скорости, поскольку способ доступа к памяти, совпадает со способом хранения в языке C++, алгоритм JKI оказался самым медленным, поскольку доступ к памяти не является оптимальным.

ЗАКЛЮЧЕНИЕ

В данной работе была продемонстрирована работа алгоритмов LU-разложения матриц на различных размерностях с использованием оптимизации и без нее. Были проведено сравнение временных показателей разложения матрицы следующих алгоритмов: KIJ, JKI, блочный алгоритм на основе KIJ, алгоритм из стандартной библиотеки BLAS. Анализируя временные показатели, мы пришли к выводу, что наилучший результат, помимо библиотеки BLAS, показывает алгоритм KIJ, поскольку способ доступа к памяти совпадает со способом хранения. Ввиду особенностей инструментария нам не удалось продемонстрировать преимущества блочного алгоритма, однако удалось показать зависимость скорость работы от размера блока, а так же получить результат показывающий, что более быстрые в теории алгоритмы, могут уступать более медленным в конкретных ситуациях, ввиду факторов, которые в ходе работы программы можно считать случайными. Все результаты продемонстрированы в виде рисунков и таблиц.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Голуб Дж. Матричные вычисления / Дж. Голуб, Ч.Ван. Лоун; пер. с англ. Ю.М. Нечепуренко, А.Ю. Романова, А.В. Собянина, Е.Е. Тыртышникова; по ред. В.В. Воеводина. – М.: Мир, 1999. – 548 с
2. Вильямс Алгоритмы. Введение в разработку и анализ — М.: 2006. 189–195. — 576 с.

Приложение А
Код программы на языке С++

//Универсальная функция для алгоритмов KIJ,JKI

```
#include <iostream>
#include <ctime>
#include <fstream>

void algoLU() {
	double n;
	std::ofstream out;
	out.open("resultopt.txt");
	for (n = 64; n < 1025; n = n + 64) {
		double** a = new double*[n];
		for (int i = 0; i < n; i++) {
			a[i] = new double[n];
		}
		for (int i = 0; i < n; i++) {
			for (int j = 0; j < n; j++) {
				a[i][j] = 2;
				if (i == j)
					a[i][j] += 100;
			}
		}
		int startTime = clock(); ///kij
		for (int k = 0; k < n - 1; k++) {
			for (int i = k + 1; i < n; i++) {
				a[i][k] = a[i][k]/a[k][k];
			}
			for (int i = k + 1; i < n; i++) {
				for (int j = k + 1; j < n; j++) {
					a[i][j] = a[i][j] - a[i][k] * a[k][j];
				}
			}
		}
		int endTime = clock();///
		for (int i = 0; i < n; i++) {
			for (int j = 0; j < n; j++) {
				a[i][j] = 2;
				if (i == j)
					a[i][j] += 100;
			}
		}
		int startTime1 = clock(); ///jki
		for (int j = 0; j < n; j++){
				for (int k = 0; k < j; k++) {
					for (int i = k + 1; i < j; i++) {
						a[i][j] = a[i][j] - a[i][k] * a[k][j];
					}
				}
```

```
                        for (int k = 0; k < j; k++) {
                                for (int i = j; i < n; i++) {
                                        a[i][j] = a[i][j] - a[i][k] * a[k][j];
                                }
                        }

                for (int i = j + 1; i < n; i++) {
                        a[i][j] = a[i][j] / a[j][j];
                }
        }
        int endTime1 = clock();/////
        std::cout << "N: " << n << " kij: " << endTime - startTime << " jki: " << endTime1 -
startTime1 << "\n";
        out << n << ", " << endTime - startTime << ", " << endTime1 - startTime1 << ";" <<
"\n";
    }
    out.close();
}
```

//Блочный алгоритм

```
void blockLU(double** a, double** l, double** u, int n, int alpha) {
    int betta = 0;
    while (betta < n - 1) {
        for (int k = betta; k < betta + alpha - 1; k++) {
            for (int i = k + 1; i < betta + alpha; i++) {
                a[i][k] = a[i][k] / a[k][k];
            }
            for (int i = k + 1; i < betta + alpha; i++) {
                for (int j = k + 1; j < betta + alpha; j++) {
                    a[i][j] = a[i][j] - a[i][k] * a[k][j];
                }
            }
        }
        for (int i = betta; i < alpha + betta; i++) {
            for (int j = betta; j < betta + alpha; j++) {
                if (i > j) {
                    l[i][j] = a[i][j];
                    l[i][i] = 1;
                }
                else {
                    u[i][j] = a[i][j];
                }
            }
        }
        l[betta][betta] = 1;
        if (betta == n - alpha) {
            break;
        }
        for (int k = betta + alpha; k < n; k++) {
            for (int i = betta; i < betta + alpha - 1; i++) {
                l[k][i] = a[k][i] / u[i][i];
                for (int j = i + 1; j < betta + alpha; j++) {
                    a[k][j] = a[k][j] - l[k][i] * u[i][j];
                }
            }
            l[k][betta + alpha - 1] = a[k][betta + alpha - 1] / u[betta + alpha - 1][betta +
alpha - 1];
        }
        for (int k = betta + alpha; k < n; k++) {
            for (int i = betta; i < betta + alpha - 1; i++) {
                u[i][k] = a[i][k];
                for (int j = i + 1; j < betta + alpha; j++) {
                    a[j][k] = a[j][k] - l[j][i] * u[i][k];
                }
            }
            u[betta + alpha - 1][k] = a[betta + alpha - 1][k];
        }
```

```
            betta = betta + alpha;
            for (int i = betta; i < n; i++) {
                  for (int j = betta; j < n; j++) {
                        for (int k = betta - alpha; k < betta; k++) {
                              a[i][j] = a[i][j] - l[i][k] * u[k][j];
                        }
                  }
            }
      }
}
```