МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное
образовательное учреждение высшего образования
«Самарский национальный исследовательский университет
имени академика С.П. Королева»
(Самарский университет)

Институт информатики, математики и электроники
Факультет информатики
Кафедра технической кибернетики

Отчет по лабораторной работе №2

Векторные алгоритмы матричных разложений

Вариант №2

Студенты группы 6407 Анурин А.С.
Гринина В.С.
Проверил Головашкин Д.Л.

САМАРА 2018

СОДЕРЖАНИЕ

Введение 4

1 Теоретические сведения 5

1.1 Алгоритмы KJI и IKJ LU-разложения матриц 5

1.2 Блочная gaxpy-версия алгоритма LU-разложения 5

2 Цель работы 7

3 Инструментарий 8

4 Параметры эксперимента 9

5 Теоретическое ожидание 10

6 Результаты 11

7 Анализ результатов 14

8 Сравнение с теоретическим ожиданием 16

Заключение 17

Список использованных источников 18

Приложение А Код программы 19

ЗАДАНИЕ

Вариант №2

Алгоритмы: KJI, IKJ, блочный gaxpy, библиотечный.

ВВЕДЕНИЕ

Задача LU-разложения матриц встречается на практике достаточно часто. Представив произвольную матрицу в виде произведения нижней унитреугольной и верхней треугольной позволяет использовать эти свойства в последующих вычислениях. Существует множество библиотечных реализаций различных алгоритмов. Наивный алгоритм имеет сложность порядка $O\left(n^3\right)$ операций. Более того, существует множество подходов к реализации этого алгоритма, в частности, KJI и IKJ. Также существует блочный алгоритм. Из-за особенностей работы с кэшем процессора, разные реализации одного и того же подхода, хоть и имеют одинаковое количество флопов, на практике могут различаться по скорости в несколько раз.

1 Теоретические сведения

1.1 Алгоритмы KJI и IKJ LU-разложения матриц

Пусть дана матрица A размерности $n \times n$, тогда результатом LU разложения будут две матрицы той же размерности: нижняя унитреугольная L и верхняя треугольная U, при этом $A = LU$.

В данной работе без ограничения общности будем рассматривать только матрицы A, в которых все диагональные миноры не вырождены, так как это гарантирует, что при поиске LU-разложения не придется искать матрицу перестановок.

Ниже приведен алгоритм KJI разложения:

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

Также представим алгоритм IKJ разложения:

```
for i = 2:n
  for j = 1:n-1
    L(i, j) = A(i, j) / A(j, j)
  end
  for k = 1:n-1
    A(i, k+1:n) = A(i, k+1:n) - L(i, k) * A(k, k+1:n)
  end
end
```

1.2 Блочная gaxpy-версия алгоритма LU-разложения

Пусть дана матрица A размерности $n \times n$. Размер

Рекурсивно будем делить их на четыре подматрицы.

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & A' \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & I \end{pmatrix},$$

Для нахождения разложения применим следующий алгоритм:

1. Найдем LU разложение матрицы $A_{11} = L_{11}U_{11}$.
2. Решим СЛАУ $A_{12} = L_{11}U_{12}$, где найдем U_{12}.

3. Решим СЛАУ $A_{21} = L_{21}U_{11}$, где найдем L_{21}.
4. Рассмотрим матрицу $A' = A_{22} - L_{21}U_{12}$ и вернемся к шагу 1.

Приведем псевдокод алгоритма.

```
l = 1
while l <= n
  r = min(n, l + r - 1)
  alpha = r - l + 1
  Сделаем alpha шагов алгоритма исключения Гаусса для A(l:n, l:n)
  Заменим A(l:r, r+1:n) в соответствии с решением
    A(l:r, l:r) * Z = A(l:r, r+1:n)
  A(r+1:n, r+1:n) = A(r+1:n, r+1:n) - A(r+1:n, l:r) * A(l:r, r+1:n)
  l = r + 1
end
```

2 Цель работы

Целью данной работы является сравнение времени работы, анализ и реализация следующих алгоритмов LU-разложения матриц:

1. KJI.
2. IKJ.
3. Блочный gaxpy.
4. Библиотечный (CBLAS).

Каждая реализация будет отдельно сравнивается с компиляторными оптимизациями и без них. По результатам эксперимента мы отсортируем алгоритмы по времени исполнения.

3 Инструментарий

Для вычислений использовалось следующее оборудование:

- Ноутбук с процессором 2.3GHz quad-core Intel Core i7 processor (Turbo Boost up to 3.5GHz) with 6MB shared L3 cache.
- Частота работы ядра: 2.3 ГГц.
- Доступная оперативная память: 16GB of 1600MHz DDR3L.
- L3 кэш: 6 МБ.
- Язык программирования C++, компилятор clang 9.1.0.

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

4 Параметры эксперимента

Тестирование проводилось на размерах матриц, экспоненциально меняющихся от 2^2 до 2^{11}. Нижней границей размеров матриц взята размерность 4, потому что при меньших значениях время исполнения приближается к точности измерения времени. В качестве верхней границы была взята размерность 2048 как наибольшая размерность, вычисление LU-разложения на которой не дает заснуть.

Был произведен экспериментальный анализ оптимального размера блока для блочного алгоритма. Размер матрицы был взят максимальный из рассматриваемых, то есть 2048. Размер блока менялся так же, как и размерности матриц.

5 Теоретическое ожидание

Ожидается, что самой быстрым алгоритмом окажется библиотечный, так как его оптимизировали дипломированные специалисты на протяжении десятилетий. Из всех алгоритмов, нами реализованных, алгоритм блочного LU-разложения должен оказаться на втором месте из-за более оптимальной работы с кэш-памятью процессора [1]. Неочевидно, какой алгоритм окажется на третьем месте: IKJ или KJI, однако можно предположить, что из-за характера обращений к памяти, совпадающего с типом хранения матрицы в памяти, алгоритм IKJ окажется быстрее, чем KJI.

Предполагается, что реализации, скомпилированные с оптимизациями, должны оказаться строго быстрее. Однако в случае библиотечного алгоритма разница может оказаться маленькой в связи с тем, что мы будем обращаться к уже скомпилированному библиотечному коду.

Оптимальный размер блока в блочном алгоритме представляется возможным оценить по следующей формуле [2]:

$$\text{blockSize} = \sqrt{\frac{\text{cacheSize}}{\text{typeSize} \cdot \text{matrixCount}}} \quad (1)$$

Где cacheSize — размер кэша (6 МБ);

typeSize — размер типа данных (8 байт);

matrixCount — количество матриц (3);

blockSize — размер блока.

Таким образом, ожидаемый размер блока — 512.

6 Результаты

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

Для того, чтобы сравнивать блочный алгоритм с остальными, сначала нужно выяснить оптимальный размер блока. Рисунок 1 показывает результаты сравнения разных блоков. Из-за нелинейного характера искомого отношения времени исполнения от размера блока, для большей наглядности будем использовать логарифмическую шкалу.

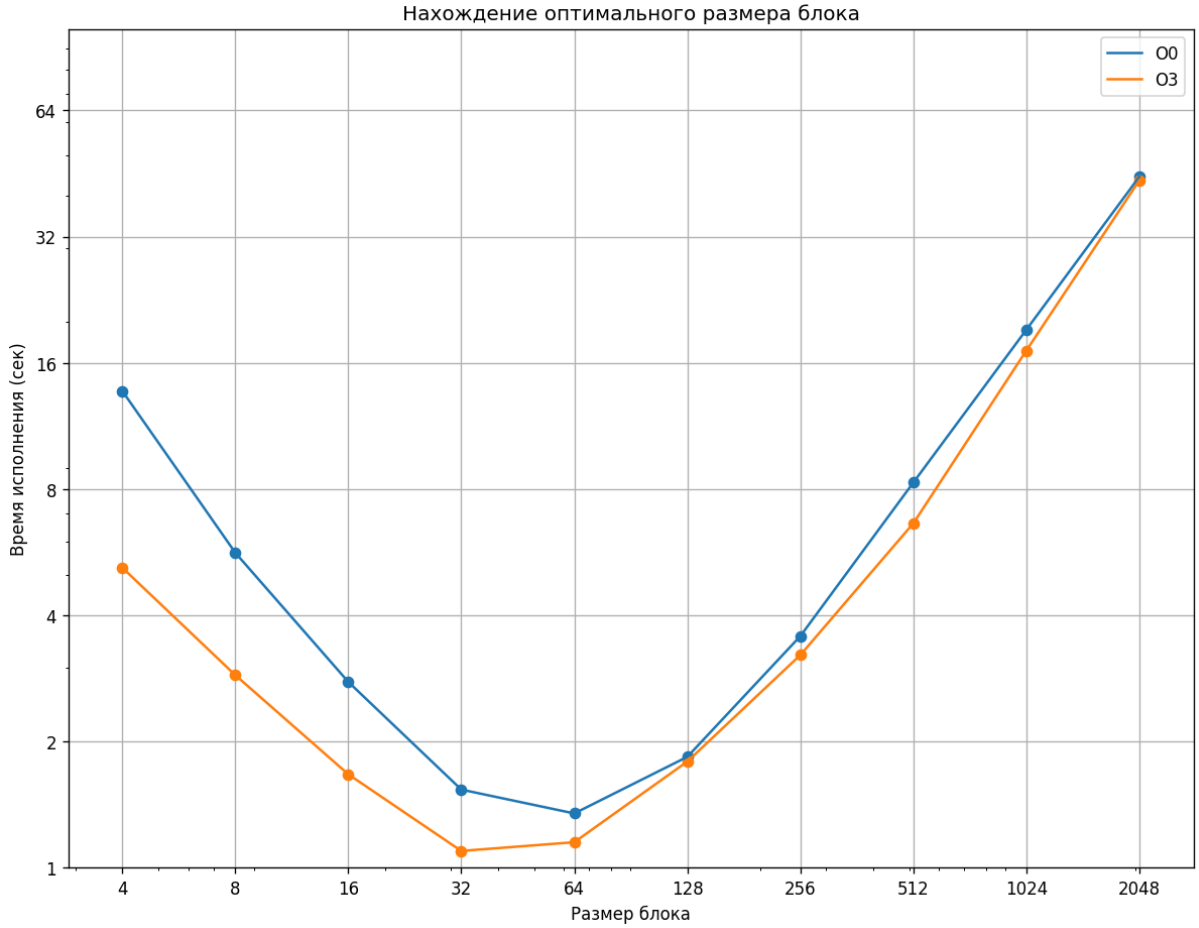

Рисунок 1 – Выбор оптимального размера блока.

Видно, что кривая времени исполнения в зависимости от размера блока имеет определенный u-характер. Оптимальное значение находится чуть левее середины, что и иллюстрируется данной кривой.

Приведем времена исполнения различных алгоритмов. Рисунок 2 показывает зависимость времени исполнения от размерности матрицы для реализаций алгоритмов без оптимизаций. Рисунок 3 показывает то же самое, но с компиляторными оптимизациями.

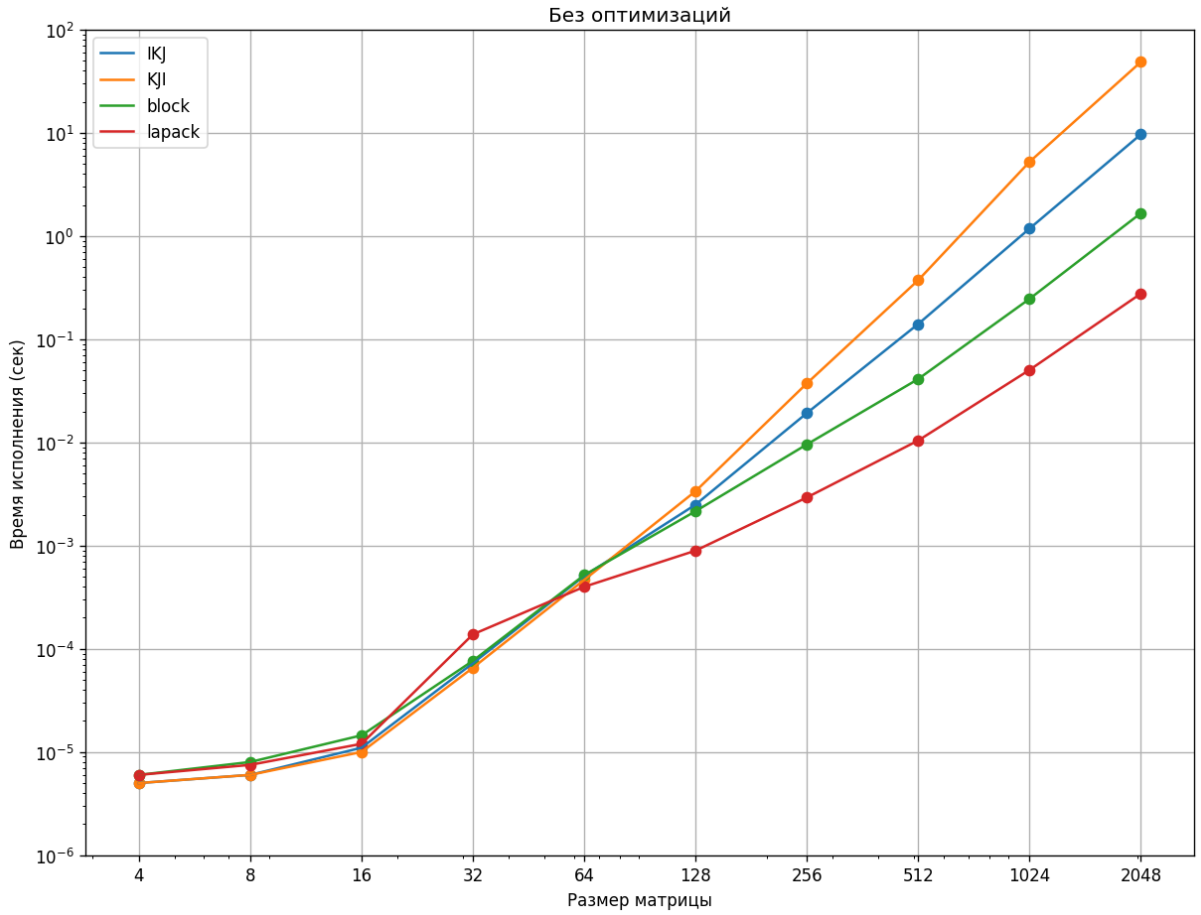

Рисунок 2 — Результаты без оптимизаций

Первое наблюдение, которое можно сделать, это то, что включение компиляторных оптимизаций приводит к ускорению работы алгоритмов. Таблица 1 приводит ускорения для рассмотренных алгоритмов.

Таблица 1 — Влияние компиляторных оптимизаций

Алгоритм	Ускорение
IJK	719,82%
KJI	105,51%
Блочный	162,9%
Библиотечный	135,21%

При маленьких размерах матриц ожидаемый порядок алгоритмов нарушается. Это связано с тем, что аккуратность работы с кэшем не имеет

значения в случаях, когда все матрицы умещаются в кэш. Так же некоторые алгоритмы, в частности библиотечный, проводят некоторую инициализацию, которая в случае небольших матриц может превышать время самих вычислений.

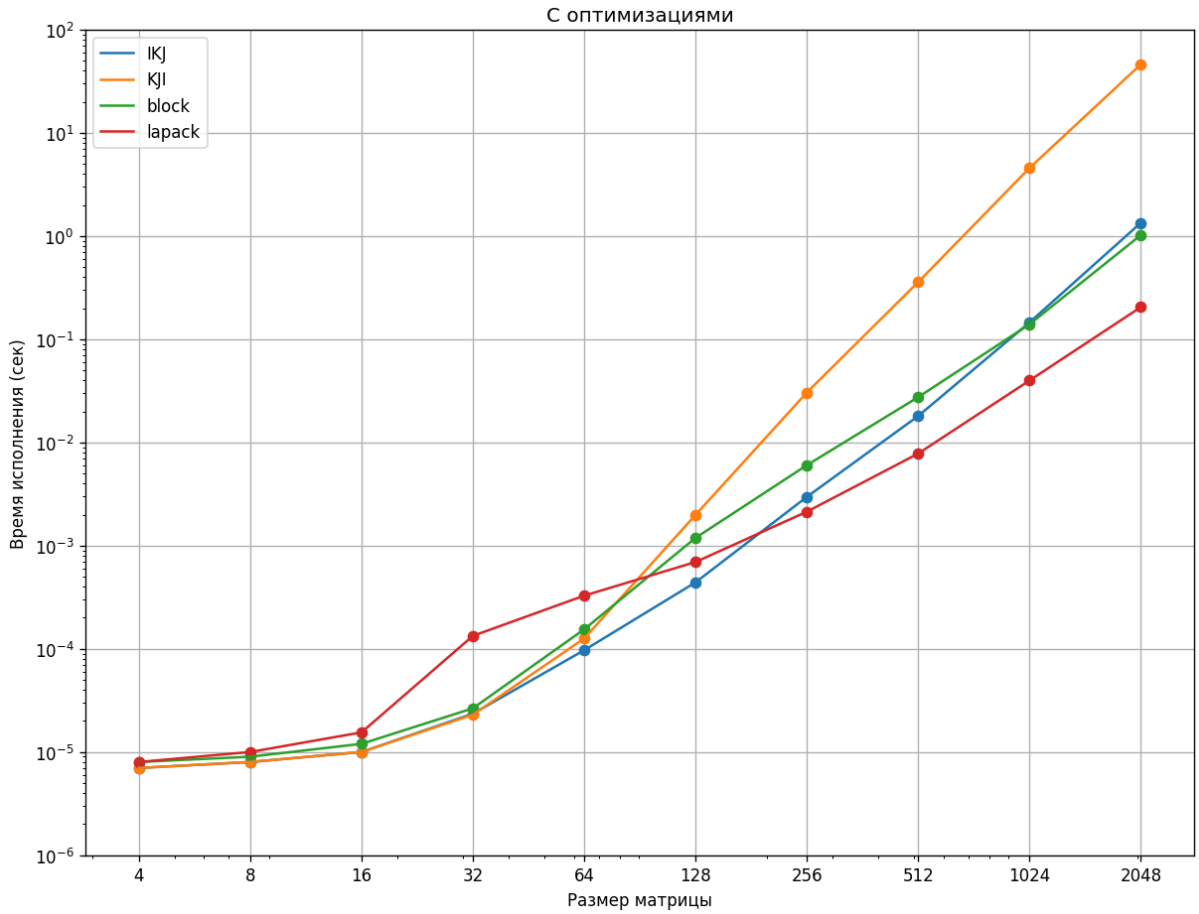

Рисунок 3 — Результаты с оптимизациями

Видно, что где-то между размерами матриц 16 и 32 библиотечный алгоритм начинает проводить некую тяжелую инициализацию, которая резко меняет его производительность.

7 Анализ результатов

Библиотечный алгоритм показал лучший результат. С использованием компиляторных оптимизаций алгоритм IKJ и блочный показали схожую производительность. Это показывает, что неасимптотические оптимизации, такие как оптимизация работы с кэшем, имеют право на существование.

Блочный алгоритм, несмотря на то, что использует много дополнительной памяти, оказался на втором месте. Скорее всего, так получилось из-за использования библиотечных алгоритмов для решения подзадач внутри блочного алгоритма. Алгоритм IKJ оказался быстрее своего товарища KJI в связи с тем, что IKJ делает последовательные запросы меняя индекс j, что соответствует тому, как матрицы хранятся в памяти («row-major»). Алгоритм KJI каждую последующую итерацию запрашивает элемент из другой строки, что сильно уменьшает эффективность процессорного кэша.

Можно заметить, что компиляторные оптимизации оказали очень большое влияние на производительность алгоритмов, особенно IJK. Он получил очень большой прирост производительности (>7 раз) благодаря векторным оптимизациям. Видно, что производительность алгоритма KJI почти не изменилась, ввиду того, что неправильная работа с кэшем процессора не спасается даже компиляторными оптимизации.

Производительность блочного алгоритма менялась в зависимости от выбранного размера блока. При маленьком размере блока количество запросов на выделение и освобождение памяти вырастает, так как каждый блок мы копируем в отдельный массив. Это приводит к общему замедлению алгоритма. При значениях размера блока, которые близки к практическому оптимальному значению, весь блок, над которым производятся вычисления в данный момент, целиком помещается в кэш-память. В такой ситуации уменьшается количество обменов «кэш — основная память», которые

приходится совершать процессору. При большом размере блока он перестает целиком помещаться в кэш-память, следовательно количество таких обменов вновь возрастает и производительность падает.

8 Сравнение с теоретическим ожиданием

В целом, результаты эксперимента достаточно полно совпали с теоретическим ожиданием.

Как и предполагалось, библиотечный алгоритм показал лучший результат. Согласно документации, авторы используемой реализации LAPACK пользуются итеративным алгоритмом Сивана Толедо. Видимо, эта реализация очень хорошо оптимизирована.

Блочный алгоритм действительно занял второе место, однако при наличии компиляторных оптимизаций отрыв весьма невелик. Алгоритмы KJI и IKJ вели себя так, как ожидалось.

Практические результаты выбора оптимального размера блока не очень хорошо совпали с теоретическими. Формула оценки оптимального размера блока несовершенна в том, что она не принимает в расчет множество дополнительных матриц, которые приходится выделять при выполнении таких шагов блочного алгоритма, как гауссово исключение, решение линейной системы и перемножение матриц.

ЗАКЛЮЧЕНИЕ

В ходе данной работы были продемонстрированы различия между несколькими реализациями нескольких алгоритмов нахождения LU-разложения матриц. Компиляторные оптимизации оказали строго положительное влияние на производительность алгоритмов.

Лучшее время показал библиотечный алгоритм, что неудивительно.

Блочный алгоритм показал свое, хоть и относительно небольшое, превосходство над наивными алгоритмами, что показывает практическую пользу от теоретических результатов.

Анализ различных размеров блоков в блочном алгоритме показал несоответствие оптимального результата теоретически предсказанному.

В целом, практические результаты эксперимента достаточно точно соответствуют теоретическим предсказаниям.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Вильямс В. Алгоритмы. Введение в разработку и анализ [Текст] / Вильямс В. – М. 2006. – 576 с.

2 Голуб Д. Матричные вычисления [Текст] / Голуб Д., Ван Ч. – М.: Мир, 1999. – 548 с.

ПРИЛОЖЕНИЕ А КОД ПРОГРАММЫ

```
#include <lapacke.h>
#include "../util.h"
void LU_KJI(const int n, double *A) {
  for (int k = 0; k < n; k++) {
    for (int i = k + 1; i < n; i++) {
      A[i * n + k] /= A[k * n + k];
    }
    for (int j = k + 1; j < n; j++) {
      for (int i = k + 1; i < n; i++) {
        A[i * n + j] -= A[i * n + k] * A[k * n + j];
      }
    }
  }
}
void LU_IKJ(const int n, double *A) {
  double *L = matrix_copy(n, A);
  for (int i = 1; i < n; i++) {
    for (int k = 0; k < i; k++) {
      L[i * n + k] = A[i * n + k] / A[k * n + k];
      for (int j = k; j < n; j++) {
        A[i * n + j] -= L[i * n + k] * A[k * n + j];
      }
    }
  }
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
      A[i * n + j] = L[i * n + j];
    }
  }
  free(L);
}
void LU_lapack(const int n, double *A) {
  int param_n = n;
  int *ipiv = (int *) malloc(n * sizeof(int));
  for (int i = 0; i < n; i++) {
    ipiv[i] = i + 1;
  }
  double *A_FORTRAN = C_to_F(n, n, A);
  int info;
  LAPACK_dgetrf(
      &param_n,
      &param_n,
      A_FORTRAN,
      &param_n,
      ipiv,
      &info
  );
  if (info != 0) {
    printf("ERROORORORORORO\n");
    exit(1);
  }
  double *A_RES_C = F_to_C(n, n, A_FORTRAN);
  free(A_FORTRAN);
  memcpy(A, A_RES_C, n * n * sizeof(double));
  free(ipiv);
}
```

```
int min(int a, int b) {
  return a < b ? a : b;
}
void gaxpy_gaussian_elimination(const int n, double *A, int num_steps) {
  for (int j = 0; j < num_steps; j++) {
    for (int k = 0; k < j; k++) {
      for (int i = k + 1; i < j; i++) {
        A[i * n + j] -= A[i * n + k] * A[k * n + j];
      }
    }
    for (int k = 0; k < j; k++) {
      for (int i = j; i < n; i++) {
        A[i * n + j] -= A[i * n + k] * A[k * n + j];
      }
    }
    for (int i = j + 1; i < n; i++) {
      A[i * n + j] /= A[j * n + j];
    }
  }
}
void LU_block(const int n, double *A, int alpha) {
  double *L = matrix_copy(n, A);
  double *U = matrix_copy(n, A);
  int l = 0;
  while (l < n) {
    int r = min(n, l + alpha);
    int block_size = r - l;
    // исключение гаусса
    {
      // копирование
      double *sub_A = subcopy(n, n, A, l, l, n - l, n - l);
      // счет
      gaxpy_gaussian_elimination(n - l, sub_A, block_size);
      // копирование назад
      copy_back(n, n, A, l, l, n - l, n - l, sub_A);
      free(sub_A);
    }
    // решение системы
    {
      if (r < n) {
        // копирование
        double *sub_A_C = subcopy(n, n, A, l, l, block_size, block_size);
        double *sub_B_C = subcopy(n, n, A, l, r, block_size, n - r);
        // перевели в COLUMN-MAJOR
        double *sub_A_FORTRAN = C_to_F(block_size, block_size, sub_A_C);
        double *sub_B_FORTRAN = C_to_F(block_size, n - r, sub_B_C);
        int param_N = block_size;
        int param_NRHS = n - r;
        int param_LDA = block_size;
        int *param_IPIV = (int *) malloc(block_size * sizeof(int));
        for (int i = 0; i < block_size; i++) {
          param_IPIV[i] = i + 1;
        }
        int param_LDB = block_size;
        int param_info;
        LAPACK_dgesv(
            &param_N,
            &param_NRHS,
```

```
            sub_A_FORTRAN,
            &param_LDA,
            param_IPIV,
            sub_B_FORTRAN,
            &param_LDB,
            &param_info
        );
        if (param_info != 0) {
          printf("ERROOROROROROROROR\n");
          exit(1);
        }
        // перевели обратно в ROW-MAJOR
        double *sub_Z = F_to_C(block_size, n - r, sub_B_FORTRAN);
        // скопировали обратно в A
        copy_back(n, n, A, l, r, block_size, n - r, sub_Z);
        free(sub_A_C);
        free(sub_B_C);
        free(sub_A_FORTRAN);
        free(sub_B_FORTRAN);
        free(sub_Z);
      }
    }
    // перемножение
    {
      // копирование
      double *sub_A = subcopy(n, n, A, r, l, n - r, block_size);
      double *sub_B = subcopy(n, n, A, l, r, block_size, n - r);
      double *sub_C = malloc_vector_0((n - r) * (n - r));
      // перемножили
      cblas_matrix_multiply(
          n - r, block_size, sub_A,
          block_size, n - r, sub_B,
          sub_C);
      // вычли из исходной
      for (int i = 0; i < n - r; i++) {
        for (int j = 0; j < n - r; j++) {
          A[(r + i) * n + (r + j)] -= sub_C[i * (n - r) + j];
        }
      }
      free(sub_A);
      free(sub_B);
      free(sub_C);
    }
    l = r;
  }
}
void test_block(int n, int block_size) {
  double *A = malloc_matrix_0(n);
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      A[i * n + j] = i == j ? 10 : 1;
    }
  }
  ll time_start;
  double *LU_block_result = matrix_copy(n, A);
  time_start = get_nanos();
  LU_block(n, LU_block_result, block_size);
  double LU_block_duration = (get_nanos() - time_start) / 1e9;
```

```
  print_matrix(n, LU_block_result);
  // CSV
  printf("LU_block,%d,%d,%f\n", n, block_size, LU_block_duration);
  free(LU_block_result);
}
void test_blocks() {
  printf("algo,n,block_size,time\n");
  for (int block_size = 4; block_size <= 16; block_size *= 2) {
    for (int k = 0; k < 10; k++) {
      test_block(2048, block_size);
    }
  }
}
void test_algos(int n, int block_size) {
  double *A = malloc_matrix_0(n);
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      A[i * n + j] = i == j ? 10 : 1;
    }
  }
  ll time_start;
  double *LU_KJI_result = matrix_copy(n, A);
  time_start = get_nanos();
  LU_KJI(n, LU_KJI_result);
  double LU_KJI_duration = (get_nanos() - time_start) / 1e9;
  double *LU_IKJ_result = matrix_copy(n, A);
  time_start = get_nanos();
  LU_IKJ(n, LU_IKJ_result);
  double LU_IKJ_duration = (get_nanos() - time_start) / 1e9;
  double *LU_block_result = matrix_copy(n, A);
  time_start = get_nanos();
  LU_block(n, LU_block_result, block_size);
  double LU_block_duration = (get_nanos() - time_start) / 1e9;
  double *LU_lapack_result = matrix_copy(n, A);
  time_start = get_nanos();
  LU_lapack(n, LU_lapack_result);
  double LU_lapack_duration = (get_nanos() - time_start) / 1e9;
  printf("KJI,%d,%f\n", n, LU_KJI_duration);
  printf("IKJ,%d,%f\n", n, LU_IKJ_duration);
  printf("block,%d,%f\n", n, LU_block_duration);
  printf("lapack,%d,%f\n", n, LU_lapack_duration);
  free(A);
  free(LU_KJI_result);
  free(LU_IKJ_result);
  free(LU_block_result);
  free(LU_lapack_result);
}
void test_all_algos() {
  printf("algo,n,time\n");
  for (int n = 4; n <= 2048; n *= 2) {
    int block_size = min(n, 64);
    for (int k = 0; k < 10; k++) {
      test_algos(n, block_size);
    }
  }
}
int main() test_all_algos();
```