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 №1

Векторные алгоритмы умножения матриц

Вариант №1

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

САМАРА 2018

СОДЕРЖАНИЕ

ЗАДАНИЕ

Вариант №1

Алгоритмы: IJK, IKJ, блочный IKJ, Штрассена, библиотечный.

ВВЕДЕНИЕ

Задача перемножения матриц встречается на практике достаточно часто. Существует множество библиотечных реализаций различных алгоритмов. Наивный алгоритм имеет сложность порядка $O\left(n^3\right)$ операций. Однако из-за особенностей работы с кэшем процессора, разные реализации одного и того же алгоритма, хоть и имеют одинаковую асимптотику, на практике могут различаться по скорости в несколько раз. Алгоритм Штрассена позволяет достичь асимптотики $O\left(n^{\log_2 7}\right)$, жертвуя скрытой константой.

1 Теоретические сведения

1.1 Алгоритмы IJK и IKJ перемножения матриц

Пусть даны две матрицы A и B размерности $n \times n$, тогда результатом перемножения данных матриц будет матрица C размерности $n \times n$, причем

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j} \quad (1)$$

Ниже приведен алгоритм IJK перемножения матриц:

```
for i = 1:n
    for j = 1:n
        C(i, j) = C(i, j) + A(i, 1:n) * B(1:n, j)
    end
end
```

Алгоритм IKJ умножения матриц

```
for i = 1:n
    for k = 1:n
        C(i, 1:n) = C(i, 1:n) + A(i, k) * B(k, 1:n)
    end
end
```

1.2 Блочный алгоритм перемножения матриц

Блочный алгоритм перемножения матриц применяют с целью повышения эффективности использования кэш-памяти CPU. Исходные матрицы делятся на блоки, затем они умножаются, и результирующая матрица складывается из уже посчитанных блоков. Вычисление происходит по уже известной ранее формуле (1).

За счет выбора размерности оптимального блока, при умножении матриц блочным алгоритмом размер обрабатываемых данных на каждой итерации не превышает объема кэша.

Приведем код алгоритма блочного перемножения матриц:

```
for i_b = 1:n
    i_l = (i_b - 1) * blockSize + 1
    i_r = i_b * blockSize
    for j_b = 1:n
        j_l = (j_b - 1) * blockSize + 1
        j_r = j_b * blockSize
        for k_b = 1:n
            k_l = (k_b - 1) * blockSize + 1
```

```
            k_r = k_b * blockSize
            C(i_l:i_r,j_l:j_r) = C(i_l:i_r,j_l:j_r) +
              A(i_l:i_r,k_l:k_r) * B(k_l:k_r,j_l:j_r)
          end
        end
    end
```

1.3 Алгоритм Штрассена

Пусть даны две матрицы A и B размерности $n \times n$, где $n = 2^k, k \in \mathbb{Z}$

Рекурсивно будем делить их на четыре равные подматрицы.

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$C = AB = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Введем новые матрицы:

$$P_1 = \left(A_{11} + A_{22}\right)\left(B_{11} + B_{22}\right)$$
$$P_2 = \left(A_{21} + A_{12}\right)B_{11}$$
$$P_3 = A_{11}\left(B_{12} - B_{22}\right)$$
$$P_4 = A_{22}\left(B_{21} - B_{11}\right)$$
$$P_5 = \left(A_{11} + A_{12}\right)B_{22}$$
$$P_6 = \left(A_{21} - A_{12}\right)\left(B_{11} + B_{12}\right)$$
$$P_7 = \left(A_{12} - A_{22}\right)\left(B_{21} + B_{22}\right)$$

Выразим через них искомую матрицу C.

$$C_{11} = P_1 + P_4 - P_5 + P_7$$
$$C_{12} = P_3 + P_5$$
$$C_{21} = P_2 + P_4$$
$$C_{22} = P_1 - P_2 + P_3 + P_6$$

Магическим образом получилось, что для этого выражения нужно 7 умножений, а не 8. Приведем псевдокод алгоритма.

```
function c = Strassen(A, B, n)
  if n=1 then
    C=AB
  else
    p1 = strassen(A(1:n/2, 1:n/2) + A(n/2 +1:n, n/2 +1:n), B(1:n/2, 1:n/2)
+ B(n/2 +1:n, n/2 +1:n), n/2);
    p2 = …
    …
    p7 = strassen(A(1:n/2, n/2 +1:n) −  A(n/2 +1:n, n/2 +1:n), B(1:n/2, n/2
+1:n) + B(n/2 +1:n, n/2 +1:n), n/2);
    C(1:n/2, 1:n/2) = p1 + p4 − p5 + p7;
    …
    C(n/2 +1:n, n/2 +1:n) = p1 − p2 + p3 + p6;
end
```

2 Цель работы

Целью данной работы является сравнение времени работы, анализ и реализация следующих алгоритмов перемножения матриц:

1. IJK
2. IKJ
3. Штрассена
4. Библиотечный (CBLAS)
5. Блочный IKJ

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

Для упрощения реализации алгоритма Штрассена, тестирование проводилось на размерах матриц, являющихся степенями двойки. В качестве нижней границы была взята размерность 32, потому что при меньших значениях время исполнения приближается к точности измерения времени. В качестве верхней границы была взята размерность 2048 как наименьшая, при которой исполнение самой медленной реализации превышает одну минуту.

Был произведен экспериментальный анализ оптимального размера блока для блочного алгоритма. Размер матрицы был взят максимальный из рассматриваемых, то есть 2048. Размер блока менялся от 8 до 2048 по степеням двойки.

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 Штрассена должен оказаться на втором месте из-за своего преимущества по асимптотике [1]. Не очевидно, какой алгоритм окажется на третьем месте: блочный IKJ или IKJ. Они оба пытаются оптимизировать работу с кэшем, однако у блочного алгоритма есть определенный overhead, связанный с манипуляциями с блоками. Самым же медленным алгоритмом должен оказаться наивный IJK.

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]:

$$\text{blockSize} = \sqrt{\frac{\text{cacheSize}}{\text{typeSize} \cdot \text{matrixCount}}} \tag{2}$$

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 перемножения матриц, выяснение их преимуществ и недостатков.

Приведем времена исполнения различных алгоритмов. Рисунок 1 показывает зависимость времени исполнения от размерности матрицы для реализаций алгоритмов без оптимизаций. Рисунок 2 показывает то же самое, но с компиляторными оптимизациями. Из-за нелинейного характера асимптотик для большей наглядности будем использовать логарифмическую шкалу.

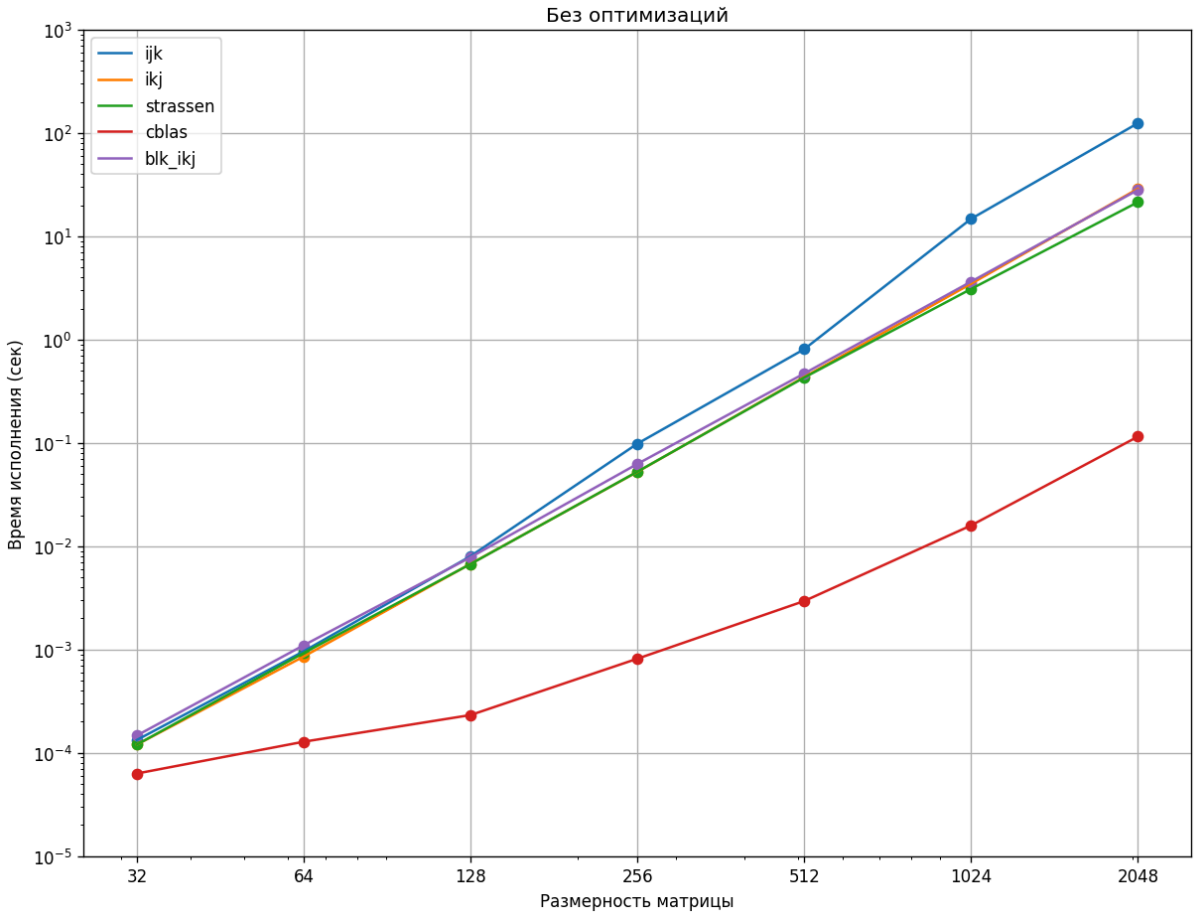

Рисунок 1 — Результаты без оптимизаций

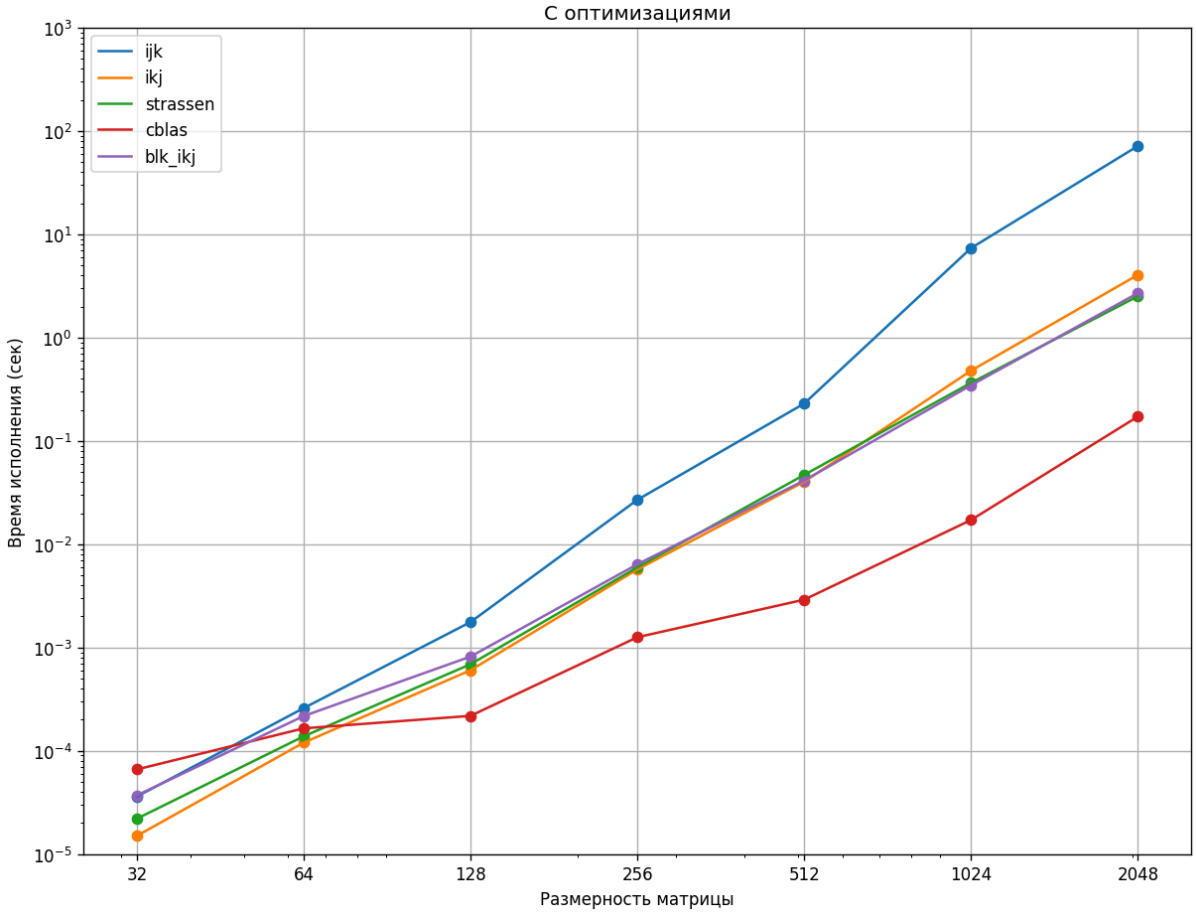

Рисунок 2 — Результаты с оптимизациями

Первый вывод, который можно сделать, это то, что оптимизация заметно улучшает производительность всех алгоритмов кроме библиотечного, как и ожидалось. Видно, что предсказания сбылись: библиотечный алгоритм работает быстрее всего, за ним следуют алгоритм Штрассена, блочный, IKJ и IJK.

В целом, включение компиляторных оптимизаций приводит к ускорению работы алгоритмов в 2 раза во всех случаях, кроме библиотечного.

При использовании логарифмической шкалы наклон линии соответствует показателю степени в асимптотике. Можно видеть, что для алгоритма Штрассена этот показатель немного меньше, чем для алгоритмов, родственных IJK.

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.

Проиллюстрируем выбор оптимального размера блока на рисунке 3. Как и ожидалось, оптимальным размером оказался 512, правда, ненамного.

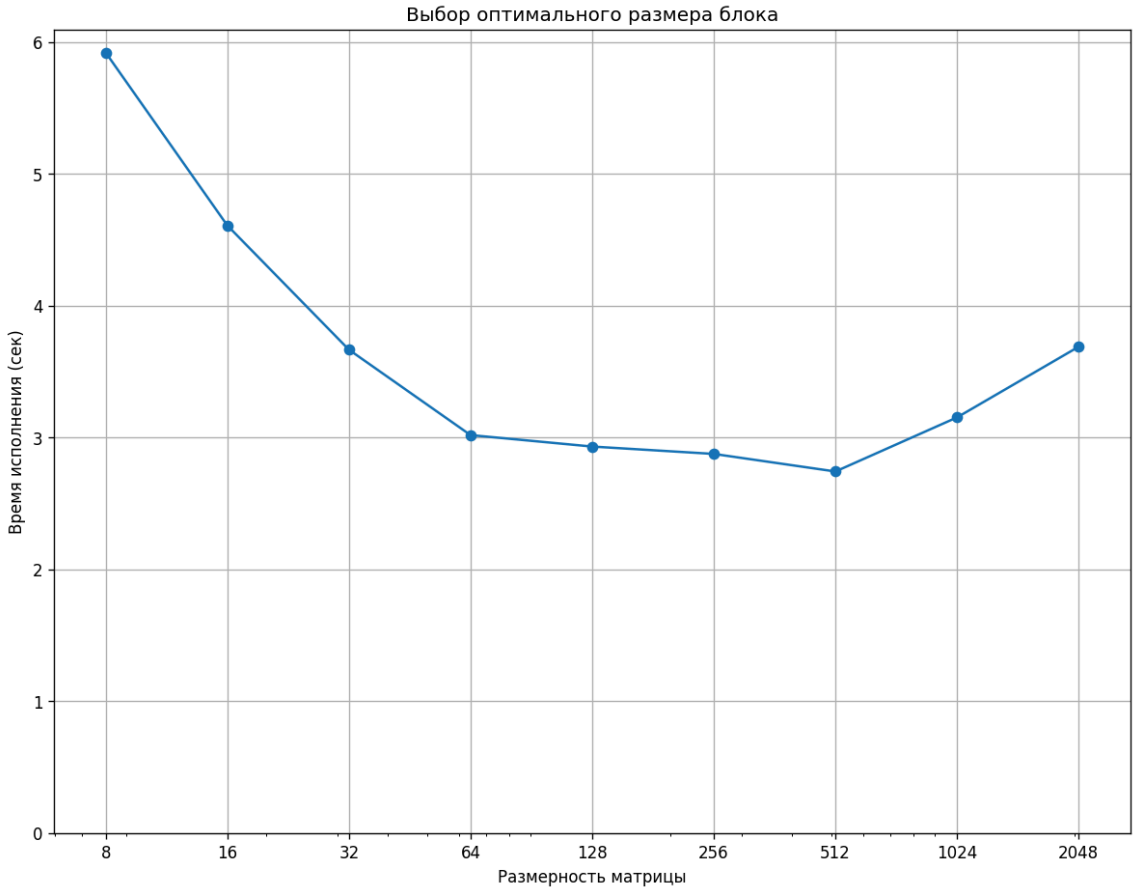

Рисунок 3 — Выбор оптимального размера блока

Видно, что кривая времени исполнения в зависимости от размера блока имеет определенный характер. Относительно маленькие значения приводят к увеличению количества побочных манипуляций с матрицами. Относительно большие значения стирают различия блочного алгоритма от обычного IKJ. Оптимальное значение находится где-то в середине, что и иллюстрируется данной кривой.

7 Анализ результатов

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

Алгоритм Штрассена, благодаря меньшему количеству производимых арифметических операций, заслуженно оказался на втором месте. Блочная модификация алгоритма IKJ показала себя (с оптимальным размером блока) лучше, чем алгоритмы IKJ и IJK. Алгоритм IJK был наименее быстрым.

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. Неочевидно, какими алгоритмами пользуются авторы библиотеки CBLAS, но получается у них достойно.

Удивительно, что практические результаты выбора оптимального размера блока очень хорошо совпали с теоретическими. Несмотря на то, что формула оценки оптимального размера блока несовершенна в том, что она не принимает в расчет неэксклюзивность исполнения программы на процессорном ядре, видно, что большую часть времени большая часть процессорного кэша используется по назначению.

Последние два места заняли самые простые алгоритмы: IKJ и IJK. Алгоритм IKJ был быстрее, чем IJK, из-за более оптимального взаимодействия с кэшем процессора.

ЗАКЛЮЧЕНИЕ

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

Лучшее время показал библиотечный алгоритм, что неудивительно.

Алгоритм Штрассена 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 соответствие 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 <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>
#include <sys/time.h>
#include <mach/clock.h>
#include <mach/mach.h>
#include <cblas.h>
#include "util.h"
typedef double T;
typedef long long ll;
void run_ijk(int n, const T *A, const T *B);
#define PRINT_MATRICES 0
#define BLOCK_SIZE 64
void algo_ijk(const int n, const T *A, const T *B, T *C) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      for (int k = 0; k < n; k++) {
        C[i * n + j] += A[i * n + k] * B[k * n + j];
      }
    }
  }
}
void algo_ikj(const int n, const T *A, const T *B, T *C) {
  for (int i = 0; i < n; i++) {
    for (int k = 0; k < n; k++) {
      for (int j = 0; j < n; j++) {
        C[i * n + j] += A[i * n + k] * B[k * n + j];
      }
    }
  }
}
void strassen_split_submatrix(int n, const T *A, T *A11, T *A12, T *A21, T
*A22) {
  int n_half = n >> 1;
  for (int i = 0; i < n_half; i++) {
    for (int j = 0; j < n_half; j++) {
      A11[i * n_half + j] = A[i * n + j];
      A12[i * n_half + j] = A[i * n + (j + n_half)];
      A21[i * n_half + j] = A[(i + n_half) * n + j];
      A22[i * n_half + j] = A[(i + n_half) * n + (j + n_half)];
    }
  }
}
void strassen_add(
    const int n,
    const T *A, const int A_n, const int A_i0, const int A_j0,
    const T *B, const int B_n, const int B_i0, const int B_j0,
    T *C, const int C_n, const int C_i0, const int C_j0,
    int sign
) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      C[(i + C_i0) * C_n + (j + C_j0)] = A[(i + A_i0) * A_n + (j + A_j0)] +
sign * B[(i + B_i0) * B_n + (j + B_j0)];
```

```
    }
  }
}
void strassen_recurse(const int n, const T *A, const T *B, T *C) {
  if (n <= 512) {
    algo_ikj(n, A, B, C);
    return;
  }
  int n_half = n >> 1;
  T *A11 = malloc0(n_half);
  T *A12 = malloc0(n_half);
  T *A21 = malloc0(n_half);
  T *A22 = malloc0(n_half);
  strassen_split_submatrix(n, A, A11, A12, A21, A22);
  T *B11 = malloc0(n_half);
  T *B12 = malloc0(n_half);
  T *B21 = malloc0(n_half);
  T *B22 = malloc0(n_half);
  strassen_split_submatrix(n, B, B11, B12, B21, B22);
  T *T1 = malloc0(n_half);
  T *T2 = malloc0(n_half);
  T *M1 = malloc0(n_half);
  T *M2 = malloc0(n_half);
  T *M3 = malloc0(n_half);
  T *M4 = malloc0(n_half);
  T *M5 = malloc0(n_half);
  T *M6 = malloc0(n_half);
  T *M7 = malloc0(n_half);
  // M1
  strassen_add(
      n_half,
      A11, n_half, 0, 0,
      A22, n_half, 0, 0,
      T1, n_half, 0, 0,
      1
  );
  strassen_add(
      n_half,
      B11, n_half, 0, 0,
      B22, n_half, 0, 0,
      T2, n_half, 0, 0,
      1
  );
  strassen_recurse(n_half, T1, T2, M1);
  // M2
  strassen_add(
      n_half,
      A21, n_half, 0, 0,
      A22, n_half, 0, 0,
      T1, n_half, 0, 0,
      1
  );
  strassen_recurse(n_half, T1, B11, M2);
  // M3
  strassen_add(
      n_half,
      B12, n_half, 0, 0,
      B22, n_half, 0, 0,
```

```
    T1, n_half, 0, 0,
    -1
);
strassen_recurse(n_half, A11, T1, M3);
// M4
strassen_add(
    n_half,
    B21, n_half, 0, 0,
    B11, n_half, 0, 0,
    T1, n_half, 0, 0,
    -1
);
strassen_recurse(n_half, A22, T1, M4);
// M5
strassen_add(
    n_half,
    A11, n_half, 0, 0,
    A12, n_half, 0, 0,
    T1, n_half, 0, 0,
    1
);
strassen_recurse(n_half, T1, B22, M5);
// M6
strassen_add(
    n_half,
    A21, n_half, 0, 0,
    A11, n_half, 0, 0,
    T1, n_half, 0, 0,
    -1
);
strassen_add(
    n_half,
    B11, n_half, 0, 0,
    B12, n_half, 0, 0,
    T2, n_half, 0, 0,
    1
);
strassen_recurse(n_half, T1, T2, M6);
// M7
strassen_add(
    n_half,
    A12, n_half, 0, 0,
    A22, n_half, 0, 0,
    T1, n_half, 0, 0,
    -1
);
strassen_add(
    n_half,
    B21, n_half, 0, 0,
    B22, n_half, 0, 0,
    T2, n_half, 0, 0,
    1
);
strassen_recurse(n_half, T1, T2, M7);
// C11 = M1 + M4 - M5 + M7
strassen_add(
    n_half,
    M1, n_half, 0, 0,
```

```
    M4, n_half, 0, 0,
    C, n, 0, 0,
    1
);
strassen_add(
    n_half,
    C, n, 0, 0,
    M5, n_half, 0, 0,
    C, n, 0, 0,
    -1
);
strassen_add(
    n_half,
    C, n, 0, 0,
    M7, n_half, 0, 0,
    C, n, 0, 0,
    1
);
// C12 = M3 + M5
strassen_add(
    n_half,
    M3, n_half, 0, 0,
    M5, n_half, 0, 0,
    C, n, 0, n_half,
    1
);
// C21 = M2 + M4
strassen_add(
    n_half,
    M2, n_half, 0, 0,
    M4, n_half, 0, 0,
    C, n, n_half, 0,
    1
);
// C22 = M1 - M2 + M3 + M6
strassen_add(
    n_half,
    M1, n_half, 0, 0,
    M2, n_half, 0, 0,
    C, n, n_half, n_half,
    -1
);
strassen_add(
    n_half,
    C, n, n_half, n_half,
    M3, n_half, 0, 0,
    C, n, n_half, n_half,
    1
);
strassen_add(
    n_half,
    C, n, n_half, n_half,
    M6, n_half, 0, 0,
    C, n, n_half, n_half,
    1
);
free(A11);
free(A12);
```

```
  free(A21);
  free(A22);
  free(B11);
  free(B12);
  free(B21);
  free(B22);
  free(T1);
  free(T2);
  free(M1);
  free(M2);
  free(M3);
  free(M4);
  free(M5);
  free(M6);
  free(M7);
}
void algo_strassen(const int n, const T *A, const T *B, T *C) {
  strassen_recurse(n, A, B, C);
}
void algo_cblas(const int n, const double *A, const double *B, double *C) {
  cblas_dgemm(
      CblasRowMajor, // OPENBLAS_CONST enum CBLAS_ORDER Order,
      CblasNoTrans, // OPENBLAS_CONST enum CBLAS_TRANSPOSE TransA
      CblasNoTrans, // OPENBLAS_CONST enum CBLAS_TRANSPOSE TransB
      n, // OPENBLAS_CONST blasint M
      n, // OPENBLAS_CONST blasint N
      n, // OPENBLAS_CONST blasint K,
      1.0, // OPENBLAS_CONST double alpha
      A, // OPENBLAS_CONST double *A
      n, // OPENBLAS_CONST blasint lda
      B, // OPENBLAS_CONST double *B
      n, // OPENBLAS_CONST blasint ldb
      0.0, // OPENBLAS_CONST double beta
      C, // double *C
      n  // OPENBLAS_CONST blasint ldc
  );
}
void algo_block(const int n, const int block_size, const T *A, const T *B,
T *C) {
//  int block_count = highest_bit((int) (sqrt(n))) / 2;
  int block_count = n / block_size;
  T *A_small = malloc0(block_size);
  T *B_small = malloc0(block_size);
  T *C_small = malloc0(block_size);
  for (int i = 0; i < block_count; i++) {
    for (int k = 0; k < block_count; k++) {
      for (int j = 0; j < block_count; j++) {
        for (int ii = 0; ii < block_size; ii++) {
          for (int jj = 0; jj < block_size; jj++) {
            A_small[ii * block_size + jj] = A[(i * block_size + ii) * n + k
* block_size + jj];
            B_small[ii * block_size + jj] = B[(k * block_size + ii) * n + j
* block_size + jj];
          }
        }
        algo_ikj(block_size, A_small, B_small, C_small);
        for (int ii = 0; ii < block_size; ii++) {
          for (int jj = 0; jj < block_size; jj++) {
```

```
            C[(i * block_size + ii) * n + j * block_size + jj] +=
C_small[ii * block_size + jj];
          }
        }
      }
    }
  }
  free(A_small);
  free(B_small);
  free(C_small);
}
void run_ijk(int n, const T *A, const T *B) {
  T *C = malloc0(n);
  ll start_nanos = get_nanos();
  algo_ijk(n, A, B, C);
  ll end_nanos = get_nanos();
  double secs = (end_nanos - start_nanos) / 1e9;
  printf("%010.6f,%020lld,%s,%010d\n", secs, matrix_hash(n, C), "ijk", 0);
  if (PRINT_MATRICES) {
    print_matrix(n, A);
    print_matrix(n, B);
    print_matrix(n, C);
  }
  free(C);
}
void run_ikj(int n, const T *A, const T *B) {
  T *C = malloc0(n);
  ll start_nanos = get_nanos();
  algo_ikj(n, A, B, C);
  ll end_nanos = get_nanos();
  double secs = (end_nanos - start_nanos) / 1e9;
  printf("%010.6f,%020lld,%s,%010d\n", secs, matrix_hash(n, C), "ikj", 0);
  if (PRINT_MATRICES) {
    print_matrix(n, A);
    print_matrix(n, B);
    print_matrix(n, C);
  }
  free(C);
}
void run_strassen(int n, const T *A, const T *B) {
  T *C = malloc0(n);
  ll start_nanos = get_nanos();
  algo_strassen(n, A, B, C);
  ll end_nanos = get_nanos();
  double secs = (end_nanos - start_nanos) / 1e9;
  printf("%010.6f,%020lld,%s,%010d\n", secs, matrix_hash(n, C), "strassen",
0);
  if (PRINT_MATRICES) {
    print_matrix(n, A);
    print_matrix(n, B);
    print_matrix(n, C);
  }
  free(C);
}
void run_cblas(int n, const T *A, const T *B) {
  T *C = malloc0(n);
  ll start_nanos = get_nanos();
  algo_cblas(n, A, B, C);
```

```
  ll end_nanos = get_nanos();
  double secs = (end_nanos - start_nanos) / 1e9;
  printf("%010.6f,%020lld,%s,%010d\n", secs, matrix_hash(n, C), "cblas",
0);
  if (PRINT_MATRICES) {
    print_matrix(n, A);
    print_matrix(n, B);
    print_matrix(n, C);
  }
  free(C);
}
void run_block(const int n, const int block_size, const T *A, const T *B) {
  T *C = malloc0(n);
  ll start_nanos = get_nanos();
  algo_block(n, block_size, A, B, C);
  ll end_nanos = get_nanos();
  double secs = (end_nanos - start_nanos) / 1e9;
  printf("%010.6f,%020lld,%s,%010d\n", secs, matrix_hash(n, C), "block",
block_size);
  if (PRINT_MATRICES) {
    print_matrix(n, A);
    print_matrix(n, B);
    print_matrix(n, C);
  }
  free(C);
}
int main(const int argc, const char *argv[]) {
  int n = (int) strtol(argv[1], NULL, 10);
  int block_size = (int) strtol(argv[2], NULL, 10);
  printf("n: %d\n", n);
  printf("block_size: %d\n", block_size);
  if ((n & (n - 1)) != 0) {
    printf("not a power of 2");
    exit(1);
  }
  T fill_value = 0;
  T *A = malloc0(n);
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      A[i * n + j] = fill_value++;
    }
  }
  T *B = malloc0(n);
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      B[i * n + j] = fill_value++;
    }
  }
  run_ijk(n, A, B);
  run_ikj(n, A, B);
  run_strassen(n, A, B);
  run_cblas(n, A, B);
  run_block(n, block_size, A, B);
  free(A);
  free(B);
  return 0;
}
```