Правительство Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего профессионального образования
**"Национальный исследовательский университет
Высшая школа экономики"**

Образовательная программа «Прикладная математика»
бакалавр

**ОТЧЕТ**
**по проектной работе**

# ПАРАЛЛЕЛЬНАЯ РЕАЛИЗАЦИЯ БЛОЧНОГО ПЕРЕМНОЖЕНИЯ МАТРИЦ

Выполнил студент гр. БПМ215:
*Морычев Григорий Михайлович*

Руководитель проекта:
*Загвоздина Ксения Олеговна*

Оценка:

Москва 2022

# Содержание

# Введение

Стандартный алгоритм перемножения матриц можно назвать неэффективным с точки зрения процесса обращения к памяти. В действительности, матрицы большого размера невозможно целиком разместить в кэш-память компьютера при её ограниченном объеме. В таком случае классический алгоритм перемножения обращается к отдельным участкам исходных матриц, сохраняет их в кэш-память, использует, и при следующей итерации — очищает кэш-память для заполнения её новым участком. Все эти манипуляции затрачивают большое количество времени, вследствие чего производительность алгоритма очень сильно падает. Алгоритм блочного перемножения матриц разбивает исходные матрицы на блоки меньшего размера, которые становится возможно целиком уместить в кэш-память компьютера, и далее — перемножает матрицы поблочно. Такая реализация не требует времени на дополнительные манипуляции с памятью и работает более эффективно.

# 1 Постановка задачи

Задачами проекта являются реализация блочного алгоритма перемножения согласованных матриц любых размеров и дальнейшее распараллеливание этого алгоритма, а также сравнение его работы с классическим алгоритмом перемножения матриц. Цель работы — выявить условия оправданного применения блочного алгоритма перемножения матриц.

# 2 Блочные матрицы

Блочная матрица — это способ представления обычной матрицы путем выделения в ней подматриц меньшего размера. Эти подматрицы являются элементами блочной матрицы и обязательно подчиняются следующим условиям:

- элементы одной строки имеют одинаковую высоту

- элементы одного столбца имеют одинаковую ширину

*Пример:*

$$A = \left(\begin{array}{cc|ccc} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \end{array}\right) = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

где:

$$A_{11} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \; A_{12} = \begin{bmatrix} a_{13} & a_{14} & a_{15} \\ a_{23} & a_{24} & a_{25} \end{bmatrix} \; A_{21} = \begin{bmatrix} a_{31} & a_{32} \end{bmatrix} \; A_{22} = \begin{bmatrix} a_{33} & a_{34} & a_{35} \end{bmatrix}$$

## 2.1 Умножение блочных матриц

Умножение блочных матриц никак не отличается от умножения обычных матриц. Результатом умножения также является блочная матрица.

$$A^* = \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1m} \\ A_{21} & A_{22} & \dots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \dots & A_{nm} \end{pmatrix} \quad B^* = \begin{pmatrix} B_{11} & B_{12} & \dots & B_{1k} \\ B_{21} & B_{22} & \dots & B_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m1} & A_{m2} & \dots & A_{mk} \end{pmatrix}$$

$$A^* \times B^* = C = \begin{pmatrix} C_{11} & C_{12} & \dots & C_{1k} \\ C_{21} & C_{22} & \dots & C_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ C_{n1} & C_{n2} & \dots & C_{nk} \end{pmatrix}$$

где

$$C_{ij} = \sum_{r=1}^{m} A_{ir} B_{rj} \tag{1}$$

# 3 Реализация алгоритма

Алгоритм реализован на языке Python и оформлен в виде пользовательского приложения. Код программы представлен ниже:

```
import numpy as np
A_height = int(input('Enter the height of the first matrix: '))
print('Enter the first matrix')
A_input = [list(map(float, input().split())) for _ in range(A_height)]

B_height = int(input('\nEnter the height of the second matrix: '))
print('Enter the second matrix')
B_input = [list(map(float, input().split())) for _ in range(B_height)]

matrix_A = np.array(A_input, np.float64)
matrix_B = np.array(B_input, np.float64)


def matrix_splitting(A, B):
    A_width = len(A[0])
    A_height = len(A)

    B_width = len(B[0])

    line_sep_index = int(A_width // 2)
    column_sep_index = int(A_height // 2)

    bline_sep_index = int(B_width // 2)

    A_11 = A[0:column_sep_index, 0:line_sep_index].tolist()
    A_12 = A[0:column_sep_index, line_sep_index::].tolist()
    A_21 = A[column_sep_index::, 0:line_sep_index].tolist()
    A_22 = A[column_sep_index::, line_sep_index::].tolist()

    B_11 = B[0:line_sep_index, 0:bline_sep_index].tolist()
    B_12 = B[0:line_sep_index, bline_sep_index::].tolist()
```

```
    B_21 = B[line_sep_index::, 0:bline_sep_index].tolist()
    B_22 = B[line_sep_index::, bline_sep_index::].tolist()

    return [A_11, A_12, A_21, A_22], [B_11, B_12, B_21, B_22]


A, B = matrix_splitting(matrix_A, matrix_B)


def multiplication(A, B, a_height, b_width, n=2):

    if n == 2:
        C_11 = (multiplication(A[0], B[0], len(A[0]), len(B[0][0]), n=1) +
            multiplication(A[1], B[2], len(A[1]), len(B[2][0]), n=1)).tolist()
        C_12 = (multiplication(A[0], B[1], len(A[0]), len(B[1][0]), n=1) +
            multiplication(A[1], B[3], len(A[1]), len(B[3][0]), n=1)).tolist()
        C_21 = (multiplication(A[2], B[0], len(A[2]), len(B[0][0]), n=1) +
            multiplication(A[3], B[2], len(A[3]), len(B[2][0]), n=1)).tolist()
        C_22 = (multiplication(A[2], B[1], len(A[2]), len(B[1][0]), n=1) +
            multiplication(A[3], B[3], len(A[3]), len(B[3][0]), n=1)).tolist()
        return [C_11, C_12, C_21, C_22]
    if n == 1:
        C = [[0 for row in range(b_width)] for col in range(a_height)]
        for i in range(a_height):
            for j in range(b_width):
                sum = 0
                for k in range(len(A[0])):
                    sum += A[i][k] * B[k][j]
                C[i][j] = sum
        return np.array(C)


C = multiplication(A, B, len(A), len(B[0]), n=2)

print()
for i in range(len(C[0])):
    print(*C[0][i], *C[1][i])

for i in range(len(C[2])):
    print(*C[2][i], *C[3][i])
```

От пользователя требуется ввести согласованные матрицы A и B, которые он хочет перемножить: $(A \times B)$. По правилам умножения матриц, горизонтальные размеры матрицы $A$ должны совпадать с вертикальными размерами матрицы $B$. Далее, алгоритм, при помощи основных его функций **matrix_splitting** и **multiplication** разбивает каждую из данных матриц на четыре блока и выполняет поблочное умножение. Результатом программы является матрица $C$, состоящая также из четырех блоков. В конце программы она выводится в удобном для человека виде.

## 3.1 Функция matrix_splitting

Данная функция выполняет разбиение матриц на блоки. Принцип её работы следующий: переменные line_sep_index и column_sep_index являются индексами середины строк и середины столбцов матрицы $A$ соответственно. Далее, с помощью срезов numpy по этим индексам происходит формирование четырех

блоков: $A_{11}$, $A_{12}$, $A_{21}$ и $A_{22}$.

Так как блоки в матрицах $A$ и $B$ тоже должны согласоваться между собой, горизонтальный срез матрицы $B$ определятся вертикальным срезом матрицы $A$ (см. рис. 1), иначе поблочное умножение будет невозможным. Вертикальный срез матрицы $B$ определяется соответствующей переменной bline_sep_index, являющейся индексом середины строки матрицы $B$. Само разбиение на блоки происходит аналогично матрице $A$, при помощи срезов numpy.

По итогу функция возвращает 2 массива длины 4, заполненные блоками матриц $A$ и $B$.

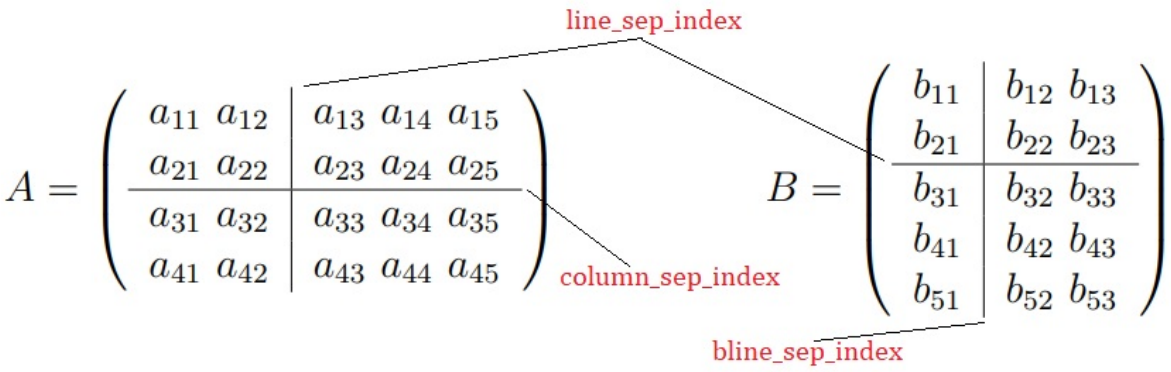


Рис. 1: Пример разбиения матриц функцией matrix_splitting

## 3.2 Функция multiplication

Данная функция выполняет рекуррентное умножение блочных матриц. На её вход поступают 2 массива блоков матриц $A$ и $B$, полученные в результате работы функции matrix_splitting, высота и ширина матриц $A$ и $B$ соответственно, а также обязательный параметр $n$, являющийся ключом, и определяющий глубину погружения.

- при $n = 2$, функция работает с входными массивами и реализует блочное умножение

- при $n = 1$, функция выполняет непосредственно умножение блоков

Так как количество блоков всегда заранее известно и равно 4 в обеих матрицах, то, соответственно, получившаяся матрица $C$ также будет состоять из четырех блоков: $C_{11}$, $C_{12}$, $C_{21}$ и $C_{22}$. Вычисление каждого из этих блоков производится по формуле (1) и сводится к перемножению обычных матриц с помощью той же функции multiplication, но с ключом $n = 1$.
По итогу функция возвращает массив длины 4, заполненный блоками матрицы $C$. Далее в коде этот массив представляется в виде искомой матрицы $C$.

# 4 Тесты

В ходе работы были произведены следующие тесты:

- Тесты 1-го рода: перемножение квадратных целочисленных матриц ($4 \times 4$; $10 \times 10$; $16 \times 16$)
- Тесты 2-го рода: перемножение прямоугольных целочисленных матриц ($4 \times 3$ и $3 \times 5$; $11 \times 5$ и $5 \times 2$; $7 \times 8$ и $8 \times 1$)
- Тесты 3-го рода: перемножение нецелочисленных матриц произвольных размеров

Результат работы каждого из тестов оказался верным, что позволило убедиться в корректности реализации алгоритма

# 5 Время работы и сравнение с классическим алгоритмом

За классический алгоритм будем принимать тройной вложенный цикл по всем элементам данных матриц. Код представлен ниже:

```
matrix_C = [[0 for row in range(b_width)] for col in range(a_height)]
for i in range(a_height):
    for j in range(b_width):
        sum = 0
        for k in range(len(matrix_A[0])):
            sum += matrix_A[i][k] * matrix_B[k][j]
        matrix_C[i][j] = sum
```

Измерять время работы будем с помощью библиотеки timeit. Для чистоты эксперимента, вывод матрицы $C$ опустим в обоих алгоритмах, также опустим использование функции tolist(), применяющейся в блочном алгоритме: она используется мной для упрощения работы с массивами и выводом данных, никак не влияет на результат работы алгоритма, но затрачивает заметное количество времени. С помощью инструмента numpy.random будем генерировать рандомные матрицы больших размеров. Для простоты все матрицы будут квадратные, одинакового размера.
Результаты замеров представлены в таблице 1.

Как можем видеть, на матрицах маленького размера, блочный алгоритм показывает себя несколько хуже чем классический. Это может быть обусловлено достаточным объемом кэш-памяти компьютера для хранения целой матрицы, а также значительной громоздкостью блочного алгоритма.

Однако, по мере увеличения размеров матриц заметен рост показателей: если на матрицах $6 \times 6$ блочный алгоритм тратит в 2,75 раз больше времени, то на матрицах $100 \times 100$ этот показатель уже равен 1,57. А преодолевая размер матриц $250 \times 250$, блочный алгоритм начинает превосходить классический, и как можем видеть дальше, на матрицах $500 \times 500$ и $1000 \times 1000$эта тенденция сохраняется.

| алгоритм | размер матриц | время, сек |
|---|---|---|
| Классический | $6 \times 6$ | 0,004 |
| | $100 \times 100$ | 0,21 |
| | $250 \times 250$ | 2,85 |
| | $500 \times 500$ | 25,04 |
| | $1000 \times 1000$ | 213,47 |
| Блочный | $6 \times 6$ | 0,011 |
| | $100 \times 100$ | 0,33 |
| | $250 \times 250$ | 2,71 |
| | $500 \times 500$ | 21,32 |
| | $1000 \times 1000$ | 159,11 |

Таблица 1: Результаты замеров

# 6 Заключение

В ходе работы мной был реализован и протестирован алгоритм блочного перемножения матриц на языке программирования Python. Реализация параллельного алгоритма не удалась. По проделанным наблюдениям можно утверждать, что данный алгоритм является более эффективным нежели классический при работе с большими матрицами, размер которых значительно превышает объем кэш-памяти компьютера. Соответственно, применимость данного алгоритма в таких ситуациях в целях оптимизации работы программы можно считать оправданной.

# Список литературы


[1] Кэш-независимые системы, В.С.Лапонин, ВМиК МГУ имени М.В. Ломоносова, кафедра вычислительных методов, лаборатория разностных методов, 2009г.

[2] Bindel, Fall , Matrix Computations, (CS 6210), Week 1, lec02, 2012

[3] Оптимизация вычислений в задаче матричного умножения, Козинов Е.А.б Мееров И.Б., Сиднев А.В., НГУ им. Н.И. Лобачевского, факультет вычислительной математики и кибернетики, 2013г.

[4] Параллельное программирование, И.А. Баранов, некоторые подходы к эффективной реализации блочных матричных алгоритмов на MIMD-компьютерах, институт кибернетики НАН Украины им. В.М. Глушкова

[5] Известия вузов, М.В. Юрушкин, С.Г. Семионов, перезамещение матриц к блочному виду с минимизацией использования дополнительной памяти, ЮФУ, 2017г.

[6] habr, перемножаем матрицы быстро или простая оптимизация программ, URL:https://habr.com/ru/post/21042/

[7] habr, умножение матриц: эффективная реализация шаг за шагом, URL:https://habr.com/ru/post/359272/