МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ЦЕНТР ЗАОЧНОЇ, ДИСТАНЦІЙНОЇ ТА ВЕЧІРНЬОЇ ФОРМ НАВЧАННЯ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК
СЕКЦІЯ ІКТ

ВИПУСКНА РОБОТА

на тему:

«Порівняльний аналіз алгоритмів перемноження матриць в сучасних інформаційно-комунікаційних технологіях»

Завідувач випускаючої кафедри Довбиш А. С.
Керівник роботи Шаповалов С. П.
Студентка гр. Індн-71С Хамрай О. О.

Суми 2021 р.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

Центр заочної, дистанційної і вечірньої форм навчання

Кафедра комп'ютерних наук

Затверджую_____

Зав. кафедрою Довбиш А.С.

"_____" _____2021 р.

ЗАВДАННЯ

до випускної роботи

Студентки четвертого курсу, групи Індн-71С спеціальності "Комп'ютерних наук" дистанційної форми навчання Хамрай Олени Олексіївни

Тема: «Порівняльний аналіз алгоритмів перемноження матриць в сучасних інформаційно-комунікаційних технологіях»

Затверджена наказом по СумДУ

№_____ от _____2021 р.

Зміст пояснювальної записки: 1) аналітичний огляд методів перемноження матриць; 2) постановка завдання й формування завдань дослідження; 3) опис основних положень, математичних моделей і алгоритмів, що використовуються для рішення поставленого завдання; 5) розробка інформаційного й програмного забезпечення; 6) аналіз результатів моделювання.

Дата видачі завдання "_____"_____2021 р.

Керівник випускної роботи _____ Шаповалов С. П.

Завдання приняла до виконання _____ Хамрай О. О.

РЕФЕРАТ

Записка: 36 стор., 10 рис., 2 табл., 3 додатки, 12 джерел.

Об’єкт дослідження — проблема перемноження матриць.

Мета роботи — комп’ютерний порівняльний аналіз алгоритмів перемноження матриць.

Методи дослідження — математичне моделювання, комп’ютерна реалізація алгоритмів на ЕОМ.

Результати — розроблено інформаційне та програмне забезпечення комп’ютерного порівняльного аналізу алгоритмів перемноження матриць. Проведено огляд алгоритмів перемноження матриць, обрано основні алгоритми для порівняльного аналізу. Розроблено комп’ютерну реалізацію наївного (ітеративного) алгоритму перемноження матриць та алгоритму Штрассена за допомогою алгоритмічної мови програмування C++.

ПЕРЕМНОЖЕННЯ МАТРИЦЬ, НАЇВНИЙ АЛГОРИТМ, АЛГОРИТМ “РОЗДІЛЮЙ ТА ВОЛОДАРЮЙ”, АЛГОРИТМ ШТРАССЕНА, АЛГОРИТМ КОППЕРСМІТА-ВИНОГРАДА, ПОРІВНЯЛЬНИЙ КОМП’ЮТЕРНИЙ АНАЛІЗ, МОВА ПРОГРАМУВАННЯ C++.

ЗМІСТ

ВСТУП

Множення матриць широко використовується в інформаційно-комунікаційних технологіях. Підвищення ефективності обчислення результату може привести до значного збільшення продуктивності в області кіноіндустрії, астрономії, комп'ютерної графіки та багато чого іншого. Для досягнення високої ефективності було проведено велику кількість досліджень.

Значна кількість аналітиків за останні кілька років звернули увагу на те, як компанії збирають і передають величезні обсяги інформації. Виникають проблеми з передачею великої кількості даних. На подолання проблем витрачаються ресурси.

Дані можуть бути стиснуті, такі як текстовий файл або файл зображення. Стиснення має виконуватися таким чином, щоб не було втрати даних. Стиснення зображень допомагає вирішити проблему відправки зображень великого розміру. У дискретному косинусному перетворенні (DCT) для стиснення зображень JPEG ми маємо методи квантування і кодування. При цих перетвореннях застосовуються алгоритми множення матриць. Результати, отримані в ході експерименту при порівнянні з множенням матриць і множенням матриць Штрассенса, показують збільшення продуктивності DCT.

Оскільки множення матриць є такою центральною операцією в багатьох числових алгоритмах, багато праці було вкладено в те, щоб зробити алгоритми множення матриць ефективними. Множення матриць у обчислювальних задачах знайшло застосування у багатьох галузях, включаючи наукові обчислення та розпізнавання шаблонів, а також у не пов'язаних між собою задачах, таких як підрахунок шляхів через граф. Багато різних алгоритмів розроблено для множення матриць на різних типах апаратних засобів, включаючи паралельні та розподілені системи, де обчислювальна робота розподіляється на декілька процесорів (можливо, по мережі).

1 АНАЛІЗ ПРОБЛЕМИ. ПОСТАНОВКА ЗАДАЧІ ДОСЛІДЖЕННЯ

1.1Аналітичній огляд алгоритмів перемноження матриць

З самого початку комп'ютерної ери дослідники намагаються знайти оптимальний спосіб множення матриць - фундаментальна операція, яка є вузьким місцем для багатьох важливих алгоритмів. Множення матриць - одна з основних операцій, що використовується у багатьох різних сферах, таких як машинне навчання, інженерне та фізичне моделювання тощо. Це робить реалізацію ефективних та чисельно стабільних алгоритмів операції важливими. Швидше множення матриць дало б ефективніші алгоритми для багатьох стандартних задачі лінійної алгебри, такі як інвертування матриць, розв'язування систем лінійних рівнянь та пошук детермінант.

Матричне множення - один з небагатьох алгоритмів, які дозволяють ефективно залучити всі обчислювальні ресурси сучасних процесорів і графічних прискорювачів. Тому не дивно, що багато алгоритмів намагаються звести до матричного множення - додаткові витрати, пов'язані із підготовкою даних, як правило з лишком окупаються загальним прискоренням алгоритмів.

Стандартний метод множення n × n матриць вимагає множення O (n^3).У 1968 р. Виноград зробив відкриття, що змістило акцент з множення на додавання і яке полягало у тому, що використовуючи інший метод обчислення внутрішнього добутку, можна знайти добуток двох матриць $n \times n$ використовуючи схожу кількість операцій. Це було важливо, оскільки додавання було обчислювально менш вимогливим, ніж множення.

Того ж року Штрассен запропонував чіткий алгоритм, який міг помножити дві матриці $2^n \times 2^n$ менш ніж за $6{,}7^n$ операцій, що є меншим у порівнянні із методом Винограда або наївним алгоритмом, де кількість операцій дорівнює приблизно 8^n. Це дає $\omega \leq \log_2(7) < 2.81$.

У 1978 р. Пан знайшов алгоритми подальшого зменшення ω за допомогою техніки трилінійного агрегування. Ця методика використовує той факт, що обчислення сліду добутку трьох матриць $n \times n$ еквівалентно задачі множення

двох матриць $n \times n$ (у перерахунку на загальну кількість множень). Пан показує, що ми можемо помножити дві матриці 70 × 70 за 143640 операцій. Це дає $\omega \leq \log_{70} 143640 < 2.79512$, і далі ми можемо виконати множення матриці 46 × 46 за 41952 операції, що дає $\omega \leq 2.78017$.

У 1980 р. Біні та співавтори отримують $\omega \leq 2{,}7799$. У 1981 р. Шонхаге показав, що алгоритм, який може приблизно виконувати множинні незалежні матричні множення, може бути використаний для подальшого зменшення ω. В результаті використання запропонованого ним методу $\omega \leq 2{,}5479$.

Найшвидший з відомих алгоритмів, розроблений в 1987 році алгоритм Копперсміта-Винограда, працює за O ($n^{2.38}$) час. Більшість дослідників вважають, що оптимальний алгоритм працюватиме по суті за час O (n^2), проте до недавнього часу подальшого прогресу у його пошуку не було. [1-4]

Генрі Кон, Роберт Кляйнберг, Балаз Сегеді та Кріс Уманс повторно вивели алгоритм Копперсміта-Винограда, використовуючи теоретичну побудову групи. Вони також показали, що будь-яка з двох різних гіпотез означатиме, що оптимальним показником множення матриць є 2, як вже давно підозрювали. Проте їм не вдалося сформулювати конкретного рішення, яке призвело б до кращого часу виконання ніж Копперсміт-Виноград. З того часу Бласіак, Кон, Черч, Грохув, Наслунд, Савін та Уманс спростували декотрі з їхніх домислів за допомогою методу Slice Rank. [5]

Існує хронологія досліджень, проведених на предмет покращання асимптотичної оцінки алгоритму множення матриць (рис. 1.1).

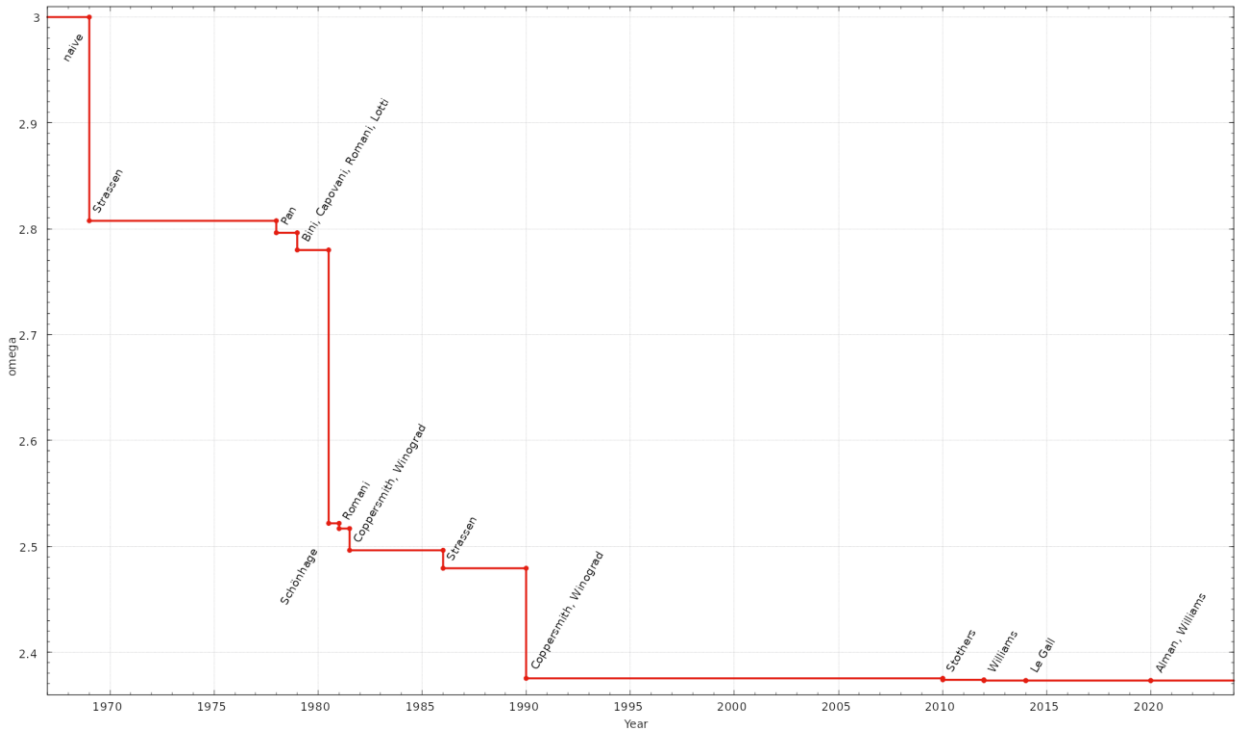

Рисунок 1.1 — Хронологічний порядок покращень оцінок показника ω для обчислювальної складності матричного множення $O(n^{\omega})$

Проблема граничної асимптотичної оцінки швидкості множення великих матриць та проблема розробки найстійкіших та найшвидших алгоритмів перемноження великих матриць, що можуть бути застововані на практиці, й досі залишається однією з актуальних задач.

1.2 Постановка задачі

Провести аналіз алгоритмів множення матриць та зробити висновки щодо їх застосування.

1. Провести аналіз та створити програмне забезпечення.
2. Виконати порівняльний аналіз швидкодії алгоритмів.
3. Провести тестування програмного продукту на прикладах.

2 МАТЕМАТИЧНА ПОСТАНОВКА ЗАДАЧІ ТА ВИБІР МЕТОДУ ЇЇ РІШЕННЯ

2.1 Алгоритми з асимптотичною складністю O(n^3)

Однією з основних операцій над матрицями є множення матриць. Матриця, отримана внаслідок операції множення, називається добутком матриць. Добуток двох матриць (AB) — це всі можливі комбінації скалярних добутків вектор-рядків однієї матриці-співмножника та вектор-стовбців другої.

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{l1} & a_{l2} & \cdots & a_{lm} \end{bmatrix}, \quad B = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix}.$$

$$C = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{l1} & c_{l2} & \cdots & c_{ln} \end{bmatrix}, \quad c_{ij} = \sum_{r=1}^{m} a_{ir} b_{rj} \quad (i = 1, 2, \ldots l;\ j = 1, 2, \ldots n). \tag{1.1}$$

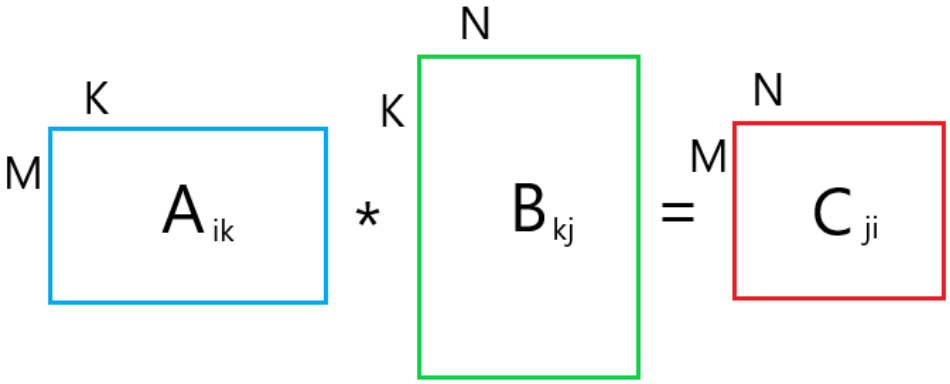

Рисунок 2.1 – Наївний алгоритм множення матриць

Тільки тоді, коли кількість стовпців першого співмножника дорівнює кількості рядків другого співмножника, можна виконати операцію множення двох матриць. У цьому випадку матриця вважається узгодженою. Зокрема, якщо обидва співмножники є квадратними матрицями одного порядку, множення завжди можна виконати.

Таким чином, з існування добутку AB не випливає існування добутку BA.

Найпростіший алгоритм перемноження матриць можна реалізувати за допомогою трьох ітераційних циклів, тому його називають ітеративним або наївним. В цьому алгоритмі ми ітеруємо крізь рядки першої матриці та множимо кожний рядок із кожним стовпцем другої матриці. Для кожного рядка та стовпця ми підраховуємо суму добутків відповідних елементів рядків та стовпців.

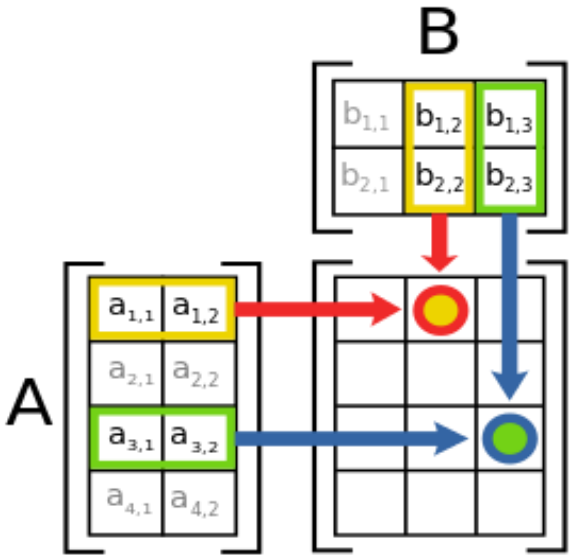

Рисунок 2.2 – Графічне зображення виконання наївного алгоритму

Алгоритм “Розділюй та володарюй” полягає у рекурсивному розбитті задачі на підзадачі. Знайдемо добуток матриць $C=AB$, де кожна з матриць має розмір $n \times n$. Припустимо що $n=2^k$. Розіб’ємо кожну з матриць на матриці розміром $n/2 \times n/2$. Тоді задача набуває вигляду

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

де $C_{ij} = \sum_{k=1}^{2} A_{ik} B_{kj}$ і A_{ij}, B_{ij} та C_{ij} - матричні блоки меншого розміру 2^{k-1}. Розіб’ємо кожну матрицю на матричні блоки меншого розміру для знаходження матриць C_{ij}. У такий спосіб, на найнижчому рівні рекурсії перемножатимуться матриці розміром 2×2. У разі, якщо розмір матриць що перемножаюся не є то-

чним ступенем 2, слід доповнити вхідні матриці нульовими елементами до необхідного розміру.

Асимптотична складність цього алгоритму складає $O(n^3)$, так само як і для наївного алгоритму, що робить їх найбільш часомісткими, проте існують ефективніші алгоритми для перемноження великих матриць. Проблема побудови найшвидшого та найстійкішого практичного алгоритму множення великих матриць, так само як і питання про граничні швидкості множення великих матриць залишається однією з невирішених задач у лінійній алгебрі.

2.2 Алгоритм Штрассена

Першим алгоритмом, який пришвидшував наївний алгоритм, був алгоритм, запропонований німецьким математиком Фолькером Штрассеном у 1969 році. Це алгоритм швидкого множення квадратних матриць зі значенням розміру рівному числу 2 в деякому степені зі складністю (1.1) $O(n^{\log_2 7})$.

В основі його лежить спосіб розбиття матриць на блоки 2x2, що вимагає лише 7 множень на відміну від наївного алгоритму, який вимагає 8 множень. Асимптотична складність цього алгоритму дорівнює $O(n^{\log_2 7}) \approx O(n^{2.807355})$. Алгоритм Штрассена має нижчу чисельну стійкість та є більш складним у порівнянні із стандартним способом, але є швидшим для матриць великого розміру. Його використання надає вагомого заощадження часу для обчислення великих матриць над точними областями. Порівняно зі стандартними алгоритмами, недоліками цього методу є висока складність програмування, низька чисельна стабільність та більше використання пам'яті. Базуючись на методі Штрассена було розроблено низку алгоритмів, що покращують швидкість, чисельну стійкість, а також інші характеристики цього методу. Однак завдяки своїй простоті алгоритм Штрассена все ще залишається одним із алгоритмів для перемноження великих матриць, що застосовується на практиці.

Звичайний алгоритм множення матриць працює за стандартною формулою (1.2) $C_{ij} = \sum_{i=1}^{n} (A_{in} + B_{nj})$, де A і B – відповідно множники, кількість стовпців матриці A дорівнює кількості стовпців матриці B, матриця C є добутком. Відповідно до цього алгоритму, для квадратної матриці розміром 2 елементи для знаходження добутку необхідно здійснити 8 множень і 4 додавання:

(1.3) $C_{11} = A_{11}B_{11} + A_{12}B_{21}$

(1.4) $C_{12} = A_{11}B_{12} + A_{12}B_{22}$

(1.5) $C_{21} = A_{21}B_{11} + A_{22}B_{21}$

(1.6) $C_{22} = A_{21}B_{12} + A_{22}B_{22}$

Ідея алгоритму Штрассена полягає у тому, щоб зосередити увагу на множенні, припускаючи, що ця операція займатиме найбільше часу. Загалом для обчислення кожного елементу квадратної матриці розміром 2 на 2 елементи за алгоритмом Штрассена необхідно 7 множень і 18 додавань. Внаслідок перетворень можна отримати наступні формули:

(1.7) $M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$

(1.8) $M_2 = (A_{21} + A_{22})B_{11}$

(1.9) $M_3 = A_{11}(B_{12} - B_{22})$

(1.10) $M_4 = A_{11}(B_{21} - B_{11})$

(1.11) $M_5 = (A_{11} + A_{12})B_{22}$

(1.12) $M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$

(1.13) $M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$

(1.14) $C_{11} = M_1 + M_4 - M_5 + M_7$

(1.15) $C_{12} = M_3 + M_5$

(1.16) $C_{21} = M_2 + M_4$

(1.17) $C_{22} = M_1 - M_2 + M_3 + M_6$

Така умова виконується тільки для квадратних матриць. Алгоритм Штрассена використовує парадигму «Розділяй і володарюй» і рекурсивний поділ матриці на кожному етапі рекурсії на 4 рівні блочні матриці. Кожна з блочних матриць у свою чергу містить 4 інші рівні блочні матриці, розміром 2 на 2 елементи. Відповідно до формул на кожному етапі, алгоритм викликає рекурсивно метод множення 7 разів замість 8, що й дає виграш у складності і відповідно швидкості множення. [6]

Існує деяка межа розміру матриці, за якої алгоритм Штрассена починає працювати повільніше за звичайний. Число розміру матриці, за якого необхідно використовувати інший алгоритм може варіюватися від 32 до 128 і залежить від характеристик процесора і мови програмування. Єдиною причиною цього є те, що відбувається більша кількість додавань ніж в стандартному алгоритмі. Тому алгоритм Штрассена використовує також і звичайне множення. Коли на етапі рекурсії блочна матриця досягає деякого числа розміру, множення завершується за стандартним алгоритмом. [7-8]

У практичному застосуванні це відчуватиметься на великих матрицях.

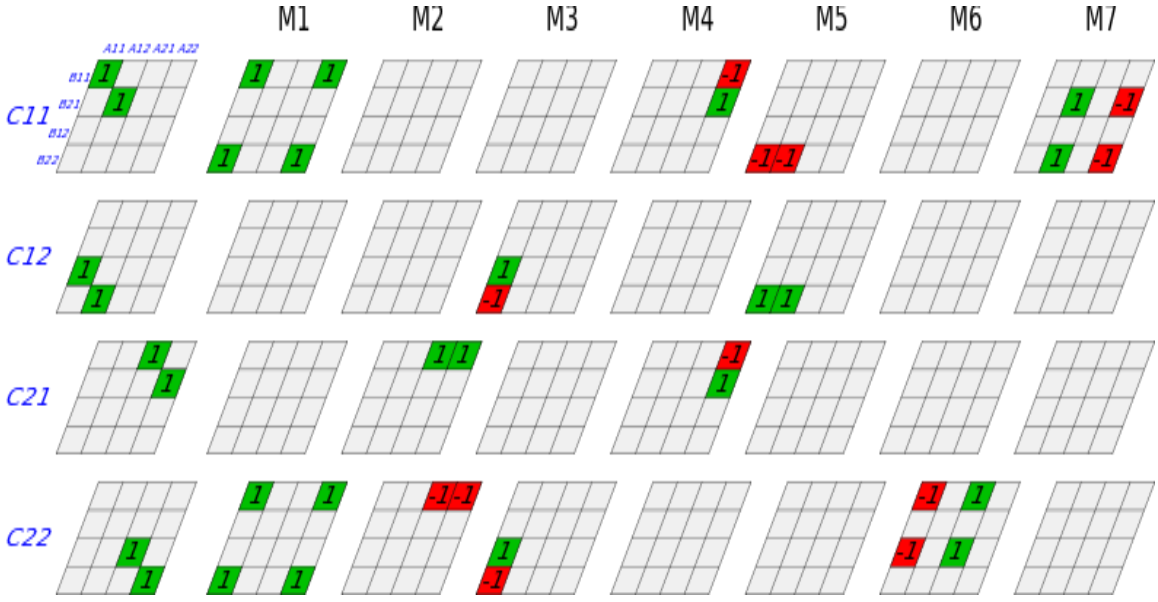

Рисунок 2.3 – Представлення множення матриць за алгоритмом Штрассена

Лівий стовпець відображає множення матриці 2x2. Наївний алгоритм множення матриць вимагає виконання одного множення для кожної «1» у ліво-

му стовпці. Усі інші стовпці є одним із 7 множень в алгоритмі, а сума стовпців дає повне множення ліворуч.

Алгоритм Штрассена використовує парадигму «Розділяй і володарюй» і рекурсивний поділ матриці на кожному етапі рекурсії на 4 рівні блочні матриці. Кожна з блочних матриць у свою чергу містить 4 інші рівні блочні матриці, розміром 2 на 2 елементи. Відповідно до формул на кожному етапі, алгоритм викликає рекурсивно метод множення 7 разів замість 8, що й дає виграш у складності і відповідно швидкості множення.

2.3 Алгоритм Копперсміта-Винограда

Алгоритм Копперсміта-Винограда засновано на ідеях, схожих із алгоритмом Штрассена та до 2010 року він був найбільш асимптотично швидким. Асимптотична складність цього алгоритму складає $O(n^{2.375477})$, що є кращим за алгоритм Штрассена.[12] Цей алгоритм часто використовується як основа інших алгоритмів для доведення теоретичних меж часу. Наразі алгоритм Копперсміта-Винограда та похідні він нього алгоритми є асимптотично найшвидшими. Однак те, що отримані поліпшення є незначними, свідчить про наявність «бар'єру Копперсміта-Винограда» при оцінці швидкодії алгоритмів. Також, на відміну від алгоритму Штрассена, алгоритм Копперсміта-Винограда на практиці не використовується, оскільки надає перевагу лише матрицям астрономічного розміру, що їх неможливо обробити сучасним обладнанням, що робить його галактичним алгоритмом.

3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Вибір мови програмування та розробка ПЗ

Для створення програмного забезпечення було обрано мову C++. На відміну від таких мов як Python (без додаткових бібліотек, таких як numpy), C++ має набагато вищу швидкість виконання. На стадії дизайну ПЗ було імплементовано наївний алгоритм на мові Python та на мові C++. Час виконання наївного алгоритму для множення тестової матриці 3000x3000 на мові Python зайняв близько 53 хвилин, а множення цієї самої матриці за допомогою наївного алгоритму, імплементованого на мові C++ зайняв 5 хвилин 22 секунди. Виходячи з цього результату розробка робочого варіанту ПЗ велася на мові C++.

У вигляді двох окремих програм було імплементовано наївний алгоритм перемноження матриць (Додаток Б) а також алгоритм Штрассена (Додаток В).

Обидві програмні реалізації алгоритмів зчитують тестові матриці-множники з файлу *.mnoz. Результуюча матриця виводиться на stdout. За необхідності запису у файл використовується стандартна функція терміналу — перенаправлення (“>”) stdout на stdin для запису результату у файл.

Наведемо псевдокод наївного алгоритму:

procedure NaiveMatrixMultiplication (A, B)

n = A.rows

let C be a new n × n matrix

for i = 1 to n **do**

for j = 1 to n **do**

$c_{ij} = 0$

for k = 1 to n **do**

$c_{ij} = c_{ij} + a_{ik} * b_{kj}$

end for

```
        end for
    end for
    return C
end procedure
```

Наведемо опис функцій та змінних програми:

rslt - композитний тип даних, що містить в собі 2 матриці

zchit - функція, що зчитує дві матриці з файлу

matA, matB, matC - матриці, які ініціалізовані як вектор вектору типу integer

filein - змінна що вказує ім’я файлу, де містяться вхідні данні

keepab — змінна типу rslt для зберігання матриць, зчитаних з файлу

eee, rrr, zzz, fff, aaa - лічильники типу integer

lnin - проміжна змінна типу string, що використовується для читання матриць з файлу

ijkalg - функція, в якій реалізовано алгоритм прямого перемножання матриць

matout - функція для виводу результуючої матриці

main - основна функція програми, в якій виконуються інші функції

Наведемо псевдокод алгоритму Штрассена [11, 12]:

```
StrassenMatrixMultiplication(MatrA, MatrB)
divide MatrA in MatrA11, MatrA12, MatrA21, MatrA22
divide MatrB in MatrB11, MatrB12, MatrB21, MatrB22
  SubMatrP1 is calculated as (MatrA11 + MatrA22) * (MatrB11 + MatrB22) by
StrassenMatrixMultiplication
  SubMatrP2 is calculated as (MatrA21 + MatrA22) * MatrB11 by
StrassenMatrixMultiplication
```

SubMatrP3 is **calculated** as MatrA11 * (MatrB12 + MatrB22) by StrassenMatrixMultiplication

SubMatrP4 is **calculated** as MatrA22 * (MatrB21 + MatrB11) by StrassenMatrixMultiplication

SubMatrP5 is **calculated** as (MatrA11 + MatrA12) * MatrB22 by StrassenMatrixMultiplication

SubMatrP6 is **calculated** as (MatrA21 + MatrA11) * (MatrB11 + MatrB12) by StrassenMatrixMultiplication

SubMatrP7 is **calculated** as (MatrA12 + MatrA22) * (MatrB21 + MatrB22) by StrassenMatrixMultiplication

MatrC11 is **calculated** as SubMatrP1 + SubMatrP4 - SubMatrP5 + SubMatrP7

MatrC12 is **calculated** as SubMatrP3 + SubMatrP5

MatrC21 is **calculated** as SubMatrP2 + SubMatrP4

MatrC22 is **calculated** as SubMatrP1 - SubMatrP2 + SubMatrP3 + SubMatrP6

combine MatrC11, MatrC12, MatrC21, MatrC22 in MatrC and **return** it as result of

function StrassenMatrixMultiplication

Наведемо опис функцій та змінних програми:

szofdiv - змінна типу int що використовується для порівняння алгоритмів

smmR - допоміжна функція з реалізацією алгоритма штрассена та опціональним перемиканням на загальний алгоритм множення матриць

smm = функція в якій реалізовано глобальний цикл алгоритму штрассена та розбиття основних матриць на підматриці

twomtr - функція в якій реалізовано додавання матриць

rizn - функція в якій реалізовано віднімання матриць

matout - функція в якій реалізовано вивід результуючої матриці

chit - функція зчитування матриць з файлу

main - основна функція програми, в якій виконуються інші функції

np2 - допоміжна функція, що рахує ступінь 2

mt, nmt , szofdiv, z, mmm, ttt, nnn, vvv - лічильники та допоміжні змінні типу integer

matA, matB, matC - матриці, які ініціалізовані як вектор вектору типу double

vnutr - допоміжний вектор типу double

matApidg, matBpidg, matCpidg, asub11, asub12, asub21, asub22, bsub11, bsub12, bsub21, bsub22, psub1, psub2, psub3, psub4, psub5, psub6, psub7, rsla, rslb, csub11, csub12, csub21, csub22 = допоміжні змінні, які використовуються безпосередньо в реалізації алгоритму штрассена. Ініціалізован як вектор вектору типу double

rdk - допоміжна змінна типу string для роботи з файлами

iryad - допоміжна змінна для роботи з файловими потоками

3.2 Порівняльний аналіз швидкодії алгоритмів

Відмінною рисою алгоритму Штрассена є те, що він вимагає лише 7 операцій множення замість 8, характерних для стандартного алгоритму. З іншого боку, алгоритм Штрассена вимагає додавання та віднімання блоків, що може додавати складності. Додавання матриць розміром N/2 вимагає лише $(N/2)^2$ операцій, в той час як множення є більш витратним — $2(N/2)^3$. Звичайне перемноження матриць потребує близько $2N^3$, де $(N=2^n)$ арифметичних операцій, тобто асимптотична складність його дорівнює $\Theta(N3)$.

Кількість додавань та множень, що вимагає алгоритм Штрассена, можна порахувати таким чином: $f(n) = 7f(n-1) + \ell 4^n$, де $f(n)$ — кількість операцій для матриці розміром $2^n x 2^n$, а ℓ — деяка стала, що залежить від кількості додавань,

виконуваних при кожному застосуванні алгоритму. Отже, $f(n) = (7 + o(1))^n$, таким чином асимптотична складність матриць розміром N=2n із застосуванням алгоритму Штрассена складає $O([7 + o(1)]^n) = O(N^{\log_2 7+o(1)}) \approx O(N^{2.8074})$. Наївний алгоритм вимагає 8 множень замість 7, що дає $O(8^{\log_2 n}) = O(N^{\log_2 8}) = O(N^3)$. [9, 10]

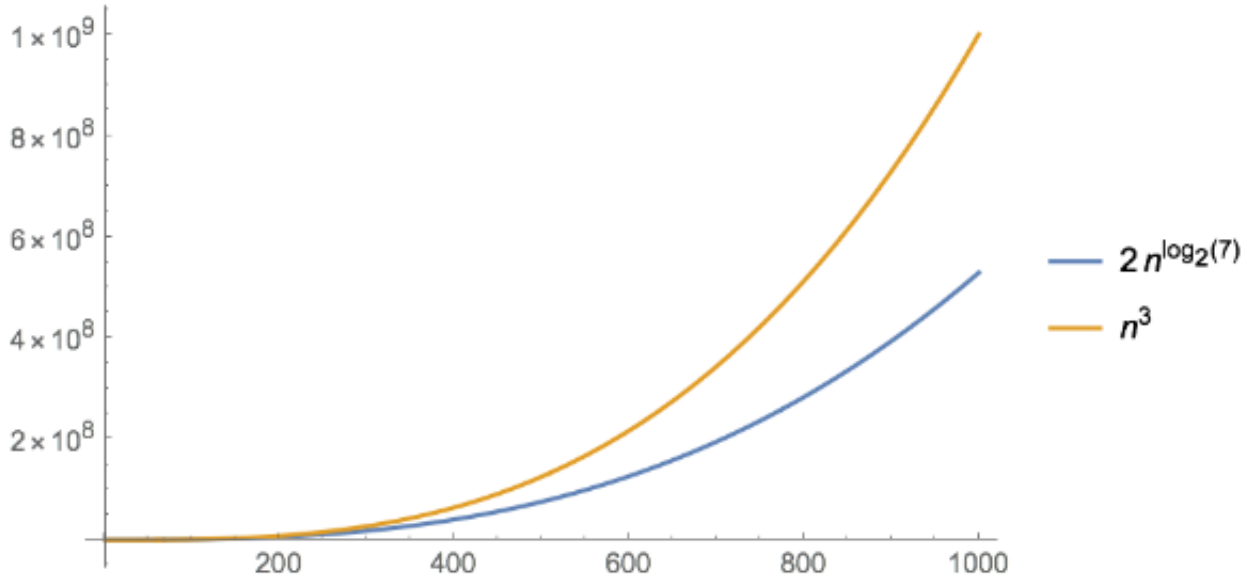

Рисунок 3.1 — Відношення асимптотичної складності наївного алгоритму до алгоритму Штрассена

Порівняння цих двох алгоритмів показує, що алгоритм Штрассена є асимптотично швидшим за наївний алгоритм. Існує деяка величина N-порогу, по досягненню якого матриці більших розмірів перемножуються швидше ніж із застосуванням наївного алгоритму. Однак для матриць малого розміру алгоритм Штрассена не є швидшим: витрати на додаткові додавання блоків матриць перевищують економію в кількості множень.

3.3 Тестування програмного продукту на прикладах

Для генерації тестових даних було розроблено скрипт на мові Python (Додаток А). В якості тестових вхідних даних для подальших обрахунків було згенеровано пари квадратних матриць розмірностей 40x40, 80x80, 400x400,

800x800, 1000x1000, 2000x2000, 4000x4000 та 8000x8000. Елементами матриць-множників є випадкові одно-, дво- та тризначні числа. Дві матриці-множники розділені новим рядком, “\n”, а елементи матриць розділені за допомогою горизонтальної табуляції, “\t”. Нижче наведено приклад тестових матриць-множників та результуючої матриці-добутку розміром 10x10 (Рисунок 3.2, Рисунок 3.3).

```
667     229     12      536     935     483     473     631     818     146
449     938     574     193     772     706     492     808     566     920
247     407     561     322     364     312     786     133     967     126
290     956     270     479     303     308     833     506     333     786
476     727     66      531     687     307     577     16      925     341
504     407     885     335     367     877     499     268     734     368
552     563     935     535     147     663     254     318     99      372
521     569     930     739     972     748     772     829     3       502
627     740     134     423     389     785     184     149     590     264
934     790     358     796     863     242     478     24      743     238

493     404     296     375     55      271     661     473     935     510
441     304     784     971     723     433     556     496     591     895
904     83      627     54      433     589     640     509     292     985
293     184     346     663     144     929     699     363     347     480
448     674     121     238     565     920     822     660     290     867
239     342     446     196     718     449     514     755     140     537
153     305     74      150     858     203     837     324     251     486
725     356     799     884     623     770     116     220     817     895
429     106     987     652     752     747     843     608     942     206
926     157     563     243     138     765     398     482     280     284
```

Рисунок 3.2 — Дві матриці-множники

```
2147995 1712611 2327236 2343266 2593932 3166618 3184166 2471304 2732936 2888836
3480860 1953626 3461862 2940904 3400403 3921088 3666882 3207136 3105872 4115124
1888590 1090729 2228353 1775960 2528622 2428410 2971892 2129530 2148409 2416529
2523341 1420789 2581524 2482033 2674199 2922371 2987689 2317701 2368440 3077017
1964143 1417794 2316461 2250369 2513116 2827134 3255067 2414522 2445185 2555939
2626474 1392914 2757326 1993084 2788325 3014183 3326430 2760973 2454325 3181058
2403082 1155605 2269651 1797958 2020223 2528983 2592880 2199801 1963946 2980756
3364537 2117286 2895422 2638573 3285767 4033278 3900615 3152840 2752458 4550468
1876164 1311030 2256936 2095940 2202665 2467347 2704306 2313145 2229821 2495203
2439843 1728558 2529944 2552432 2570185 3300753 3799855 2783694 2911210 3270713
```

Рисунок 3.3 — Результуюча матриця-добуток

Множення кожної пари матриць виконувалось за допомогою наївного алгоритму тричі для матриць кожної розмірності, так само як і за допомогою ал-

горитму Штрассена. Середнє арифметичне значення часу виконання у секундах заносилось у таблицю (Таблиця 3.1). Вимірювання часу виконання програми проводилось за допомогою команди time. Результуючі матриці виводились у файл для контролю якості вихідних даних. Також проводилось вимірювання часу при виводі на dev/null/. Різниця знаходилась в межах похибки, тому нею можна знехтувати. Тестування проводилося на сучасному комп'ютері (Таблиця 3.2).

Таблиця 3.1 Час виконання наївного алгоритму та алгоритму Штрассена для тестових матриць в секундах

N	**Наївний алгоритм**	**Алгоритм Штрассена**
40	0,003	0,005
80	0,007	0,023
400	0,542	1,006
800	4,627	7,150
930	7,038	7,049
1000	8,799	7,096
2000	75,550	49,504
4000	877,336	346,752
8000	7566,803	2402,639

Таблиця 3.2 — конфігурація тестового комп'ютера

Елемент	**Опис**
CPU	AMD Ryzen 7 3800X (16) @ 3.900GHz
RAM	64GB 3600C16
Накопичувач	Твердотільний накопичувач M.2 NVMe Samsung SSD 970 EVO Plus 500GB
OS	Fedora release 32 (Thirty Two) x86_64

Kernel	5.11.21-100.fc32.x86_64
Компілятор	g++ (GCC) 10.3.1 20210422 (Red Hat 10.3.1-1)

Як відомо, існує деяка N-межа, по досягненню якої матриці більшого розміру перемножуються повільніше за допомогою наївного алгоритму у порівнянні із алгоритмом Штрассена. Для знаходження цієї межі графічним способом було побудовано графік відношення часу виконання алгоритму Штрассена до часу виконання наївного алгоритму (рис. 3.4).

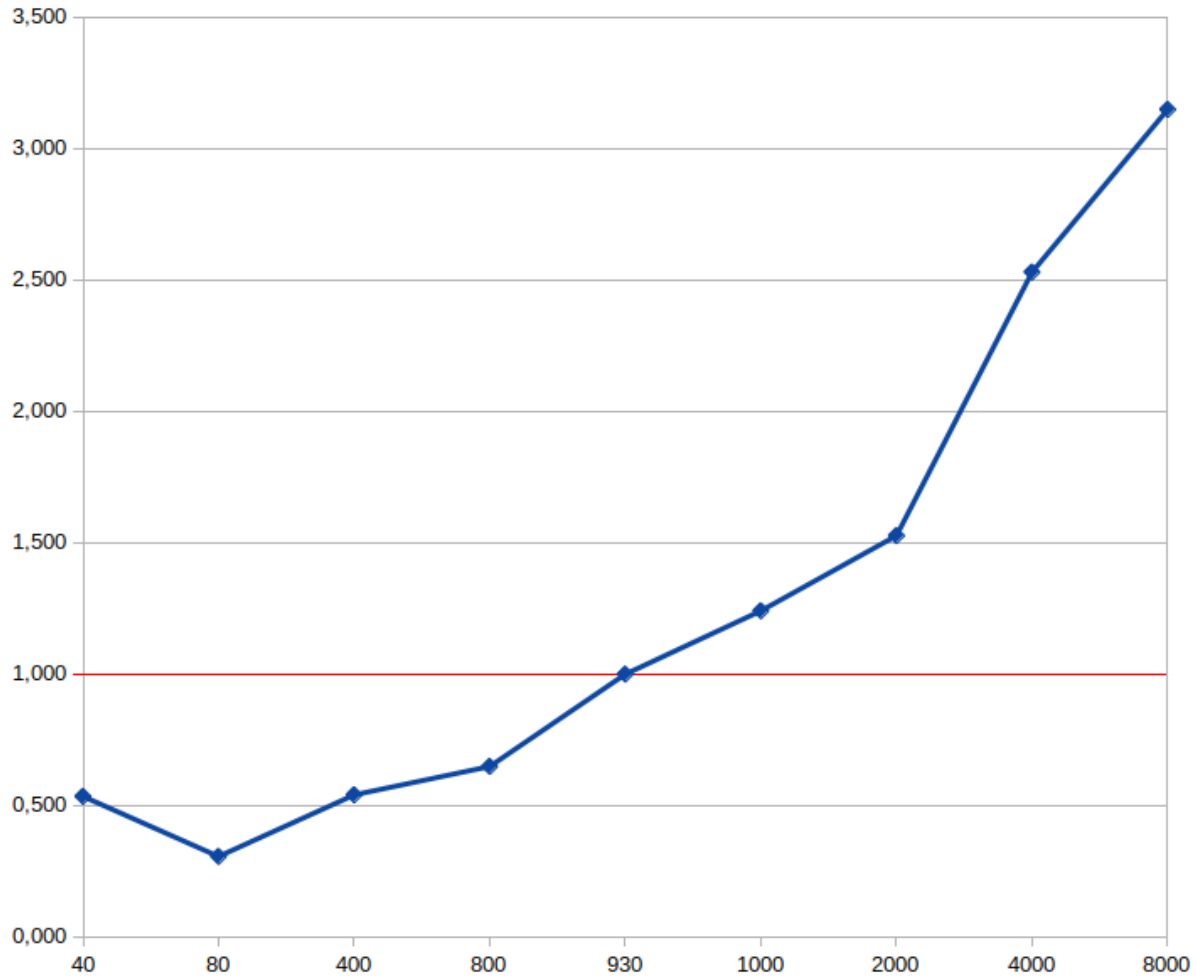

Рисунок 3.4 — Відношення часу виконання наївного алгоритму до алгоритму Штрассена для матриць кожної розмірності

Згідно графіку, розмірність перемножуваних матриць, після якої має сенс використовувати алгоритм Штрассена, лежить близько 930x930. Для перевірки було згенеровано додаткову пару тестових матриць цієї розмірності і виконане

перемноження із дотриманням тестової методології, визначеної для інших тестових матриць.

Середній час роботи наївного алгоритму склав 7,038 секунд, а алгоритму Штрассена — 7,049 секунд, демонструючи що близько саме цієї розмірності лежить межа, за якою використання алгоритму Штрассена замість наївного алгоритму набуває більшого сенсу для заощадження часу.

Як видно з графіку часу виконання наївного алгоритму та алгоритму Штрассена (рис. 3.5), матриці розмірностей від 40x40 до 800x800 були перемножені за допомогою наївного алгоритму за менший проміжок часу. Для матриць розмірностей 1000x1000 і вище алгоритм Штрассена виконувався швидше. Для матриць розміром 930x930 можна використовувати обидва алгоритми без суттєвої втрати швидкості обчислення.

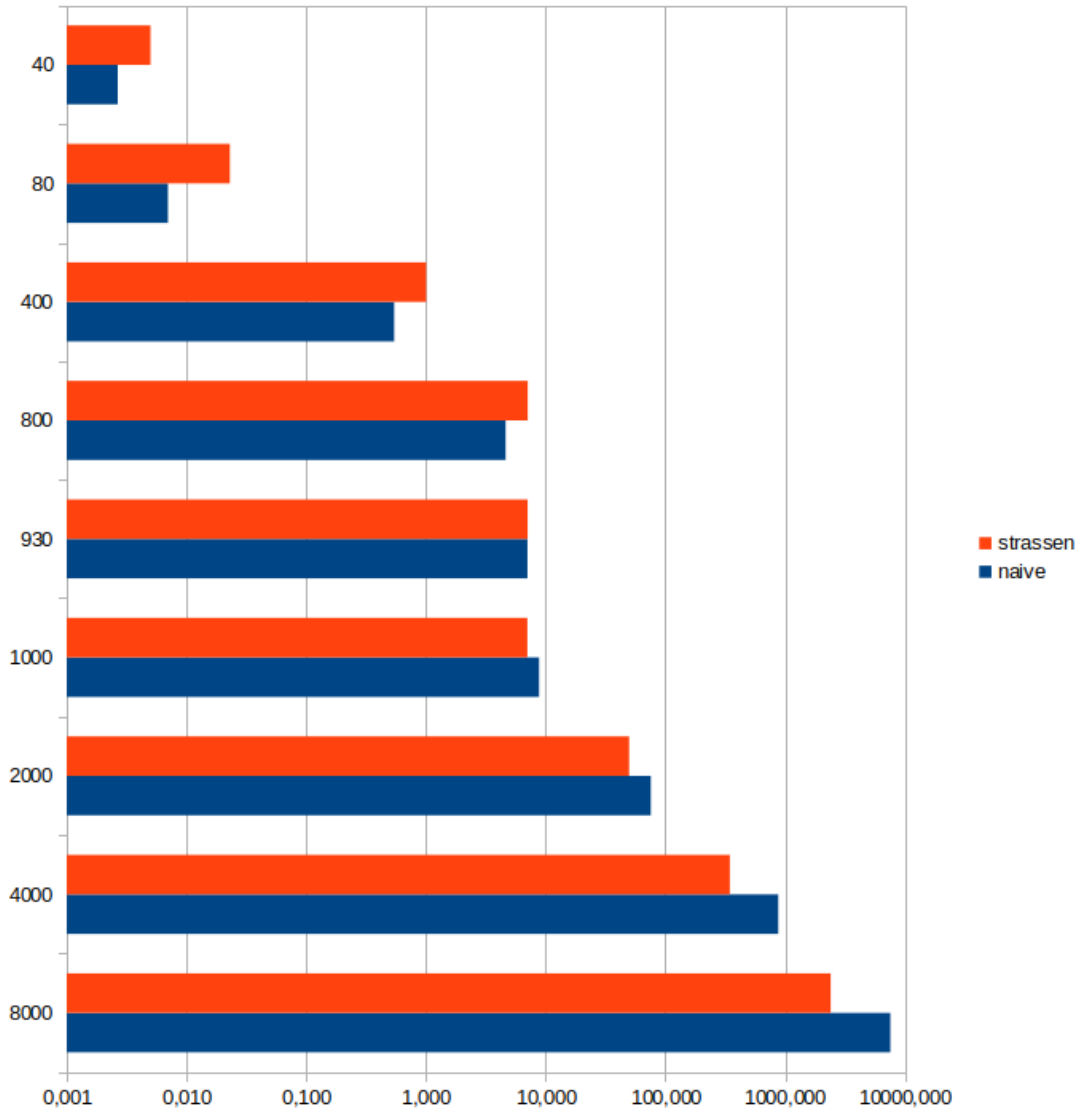

Рисунок 3.5 — Час виконання наївного алгоритму та алгоритму Штрассена за логарифмічною шкалою

Для матриць із розмірністю вище за 930x930 має сенс використовувати алгоритм Штрассена. Як видно на графіку (рис 3.6), обрахунок добутку матриць 8000x8000 за допомогою наївного алгоритму тривав в середньому 7566,803 секунд, а за допомогою алгоритму Штрассена - 2402,639, тобто виграш часу при використанні алгоритму Штрассена становив приблизно 86 хвилин.

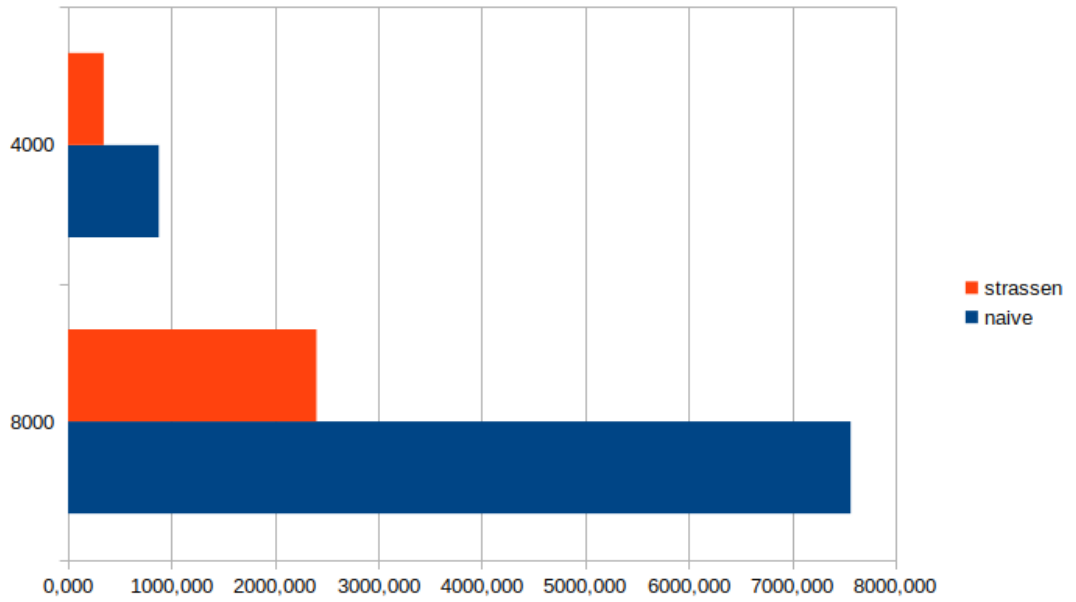

Рисунок 3.6 — Час виконання наївного алгоритму та алгоритму Штрассена для матриць 4000x4000 та 8000x8000

Заощадження часу при обрахунках великих матриць має велике практичне значення, проте також може існувати необхідність обчислювати велику кількість матриць порівняно невеликого розміру. Наприклад, обчислення матриць розміром 400x400 за допомогою наївного алгоритму було виконано приблизно вдвічі швидше за алгоритм Штрассена.

ВИСНОВКИ

Досліджено застосування алгоритмів перемноження матриць в інформаційно-телекомунікаційних технологіях, проблеми та виклики, пов’язані із швидкодією алгоритмів перемноження матриць. Проведено їхній порівняльний аналіз. Здійснено імплементацію наївного алгоритму а також алгоритму Штрассена на мові C++. Проведено тестування швидкості виконання обрахунків за допомогою обох алгоритмів, а також аналіз отриманих результатів. Знайдено межу розмірностей перемножуваних матриць, по досягненню якої слід надавати перевагу тому чи іншому алгоритму.

Алгоритм Штрассена складніший у реалізації ніж стандартний алгоритм, проте судячи їз результатів, отриманих з тестів, стандартний алгоритм працює швидше за алгоритм Штрассена на матрицях малого розміру, тобто менше за 930x930. Алгоритм Штрассена починає перемагати в часі на матрицях більшого розміру. Алгоритм Копперсміта-Винограда вважається не практичним, бо може бути використаним лише для матриць галактичних розмірів.

Тим не менш, в асимптотичних оцінках перемагає поки-що саме алгоритм Копперсміта-Винограда, який також ліг в основу багатьох інших досліджень у цій області.

СПИСОК ЛІТЕРАТУРИ

1. Wengrow J. A Common-Sense Guide to Data Structures and Algorithms. - Pragmatic Bookshelf, 2020. – 507 p.
2. Мелешко Є.В., Якименко М.С., Поліщук Л.І. Алгоритми та структури даних: Навчальний посібник для студентів технічних спеціальностей денної та заочної форми навчання. – Кропивницький: Видавець – Лисенко В.Ф., 2019. – 156 с.
3. Paul E. Black Dictionary of Algorithms and Data Structures [Електронний ресурс] – Режим доступу до ресурсу: https://xlinux.nist.gov/dads/
4. Стратієнко Н. К. Алгоритми і структури даних: практикум : навч. посібник / Н. К. Стратієнко, М. Д. Годлевський, І. О. Бородіна ; Нац. техн. ун-т «Харків. політехн. ін-т.» – Харків : НТУ «ХПІ», 2017. – 224 с.
5. Robinson S. Toward an Optimal Algorithm for Matrix Multiplication// SIAM News, V. 38, N 9, 2005 [Електронний ресурс] – Режим доступу до ресурсу: https://archive.siam.org/pdf/news/174.pdf
6. CME 323: Distributed Algorithms and Optimization, Spring 2016 [Електронний ресурс] – Режим доступу до ресурсу: https://stanford.edu/~rezab/classes/cme323/S16/notes/Lecture03/cme323_lec3.pdf
7. Ananth M., Vishwas S., Anala M. R., Cache Friendly Strategies to Optimize Matrix Multiplication// 2017 IEEE 7th International Advance Computing Conference (IACC), v. 1, 2017. - p. 23-27
8. Manning V. Optimizing matrix multiplication [Електронний ресурс] – Режим доступу до ресурсу: https://silo.tips/download/optimizing-matrix-multiplication-amitabha-banerjee
9. Kakaradov B. Ultra-Fast Matrix Multiplication: An Empirical Analysis of Highly Optimized Vector Algorithms - Computer Science, 2004 – c. 33 - 36

10. Stothers A. On the Complexity of Matrix Multiplication - University of Edinburgh, 2010
11. Богатирьов О. О., Красношлик Н. О. Алгоритмічні та кеш-орієнтовані способи підвищення ефективності матричних обчислень – Вісник Запорізького національного університету №1, 2009 – с. 27 - 35
12. Paterson M. CS341 Topics in Algorithms [Електронний ресурс] – Режим доступу до ресурсу: https://warwick.ac.uk/fac/sci/dcs/teaching/material/cs341/matrixmult09.pdf

ДОДАТОК А

Лістинг скрипта для генерації тестових матриць:

```
#!/usr/bin/env python

import random
random.seed(5678)

def genRandMat(y):
    mxv = 999
    mtrx = []
    for q in range(y):
        mtrx.append([random.randint(0, mxv) for lgt in range(y)])
    return mtrx

def svmtrx(mtrxA, mtrxB, fnm):
    zz = open(fnm, "w")
    for q, mtrx in enumerate([mtrxA, mtrxB]):
        if q != 0:
            zz.write("\n")
        for rdk in mtrx:
            zz.write("\t".join(map(str, rdk)) + "\n")

y = 8000
mtrxA = genRandMat(y)
mtrxB = genRandMat(y)
svmtrx(mtrxA, mtrxB, "8000.mnoz")
```

ДОДАТОК Б

Лістинг програмної реалізації наївного алгоритму

```
##include <sstream>
#include <string>
#include <fstream>
#include <iostream>
#include <vector>

using namespace std;

struct rslt {
    vector< vector<int> > matA;
    vector< vector<int> > matB;
};

rslt zchit(string filein) {
    vector< vector<int> > matA, matB;
    rslt keepab;
    string lnin;
    ifstream zfajlu;
    zfajlu.open (filein.c_str());

    int eee = 0;
    while (getline(zfajlu, lnin) && !lnin.empty()) {
        istringstream eeess(lnin);
        matA.resize(matA.size() + 1);
        int rrr, zzz = 0;
        while (eeess >> rrr) {
            matA[eee].push_back(rrr);
            zzz++;
        }
        eee++;
    }

    eee = 0;
    while (getline(zfajlu, lnin)) {
        istringstream eeess(lnin);
        matB.resize(matB.size() + 1);
        int rrr;
        int zzz = 0;
        while (eeess >> rrr) {
            matB[eee].push_back(rrr);
            zzz++;
        }
        eee++;
    }

    zfajlu.close();
    keepab.matA = matA;
    keepab.matB = matB;
    return keepab;
}

// обрахунок ітеративним алгоритмом:

vector< vector<int> > ijkalg(vector< vector<int> > matA,
                                    vector< vector<int> > matB) {
    int fff = matA.size();
```

```
    vector<int> tmp(fff, 0);
    vector< vector<int> > matC(fff, tmp);

    for (int eee = 0; eee < fff; eee++) {
        for (int zzz = 0; zzz < fff; zzz++) {
            for (int aaa = 0; aaa < fff; aaa++) {
                matC[eee][zzz] += matA[eee][aaa] * matB[aaa][zzz];
            }
        }
    }
    return matC;
}

// вивід матриці-добутку

void matout(vector< vector<int> > matX) {
    vector< vector<int> >::iterator irr;
    vector<int>::iterator vnutr;
    for (irr=matX.begin(); irr != matX.end(); irr++) {
        for (vnutr = irr->begin(); vnutr != irr->end(); vnutr++) {
            cout << *vnutr;
            if(vnutr+1 != irr->end()) {
                cout << "\t";
            }
        }
        cout << endl;
    }
}

int main (int cgmt, char* vgmt[]) {
    string filein;
    if (cgmt < 3) {
        filein = "8000.mnoz";
    } else {
        filein = vgmt[2];
    }
    rslt finmat = zchit (filein);
    vector< vector<int> > matC = ijkalg(finmat.matA, finmat.matB);
    matout(matC);
    return 0;
}
```

ДОДАТОК В

Лістинг програмної реалізації алгоритму Штрассена

```
#include <sstream>
#include <string>
#include <fstream>
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>


int szofdiv;

using namespace std;


void smm(vector< vector<int> > &matA,
             vector< vector<int> > &matB,
             vector< vector<int> > &matC, unsigned int mt);
unsigned int np2(int z);
void smmR(vector< vector<int> > &matA,
             vector< vector<int> > &matB,
             vector< vector<int> > &matC,
             int mt);
void twomtr(vector< vector<int> > &matA,
         vector< vector<int> > &matB,
         vector< vector<int> > &matC, int mt);
void rizn(vector< vector<int> > &matA,
             vector< vector<int> > &matB,
             vector< vector<int> > &matC, int mt);

void matout(vector< vector<int> > mtxx, int z);
void chit(string vhmat, vector< vector<int> > &matA, vector< vector<int> >
&matB);

// перемноження за ітеративним алгоритмом:

void ikjalg(vector< vector<int> > matA,
                                    vector< vector<int> > matB,
                                    vector< vector<int> > &matC, int z) {
    for (int mmm = 0; mmm < z; mmm++) {
        for (int ttt = 0; ttt < z; ttt++) {
            for (int nnn = 0; nnn < z; nnn++) {
                matC[mmm][nnn] += matA[mmm][ttt] * matB[ttt][nnn];
            }
        }
    }
}

void smmR(vector< vector<int> > &matA,
             vector< vector<int> > &matB,
             vector< vector<int> > &matC, int mt) {
    if (mt <= szofdiv) {
        ikjalg(matA, matB, matC, mt);
        return;
    }

    else {
        int nmt = mt/2;
```

```
        vector<int> vnutr (nmt);
        vector< vector<int> >
            asub11(nmt,vnutr), asub12(nmt,vnutr), asub21(nmt,vnutr),
asub22(nmt,vnutr),
            bsub11(nmt,vnutr), bsub12(nmt,vnutr), bsub21(nmt,vnutr),
bsub22(nmt,vnutr),
              csub11(nmt,vnutr), csub12(nmt,vnutr), csub21(nmt,vnutr),
csub22(nmt,vnutr),
            psub1(nmt,vnutr), psub2(nmt,vnutr), psub3(nmt,vnutr),
psub4(nmt,vnutr),
            psub5(nmt,vnutr), psub6(nmt,vnutr), psub7(nmt,vnutr),
            rsla(nmt,vnutr), rslb(nmt,vnutr);

        int mmm, nnn;

        // ділення матриць на 4 підматриці:

        for (mmm = 0; mmm < nmt; mmm++) {
            for (nnn = 0; nnn < nmt; nnn++) {
                asub11[mmm][nnn] = matA[mmm][nnn];
                asub12[mmm][nnn] = matA[mmm][nnn + nmt];
                asub21[mmm][nnn] = matA[mmm + nmt][nnn];
                asub22[mmm][nnn] = matA[mmm + nmt][nnn + nmt];

                bsub11[mmm][nnn] = matB[mmm][nnn];
                bsub12[mmm][nnn] = matB[mmm][nnn + nmt];
                bsub21[mmm][nnn] = matB[mmm + nmt][nnn];
                bsub22[mmm][nnn] = matB[mmm + nmt][nnn + nmt];
            }
        }

        // обрахунок з п1 по п7:

        twomtr(asub11, asub22, rsla, nmt);
        twomtr(bsub11, bsub22, rslb, nmt);
        smmR(rsla, rslb, psub1, nmt);

        twomtr(asub21, asub22, rsla, nmt);
        smmR(rsla, bsub11, psub2, nmt);

        rizn(bsub12, bsub22, rslb, nmt);
        smmR(asub11, rslb, psub3, nmt);

        rizn(bsub21, bsub11, rslb, nmt);
        smmR(asub22, rslb, psub4, nmt);

        twomtr(asub11, asub12, rsla, nmt);
        smmR(rsla, bsub22, psub5, nmt);

        rizn(asub21, asub11, rsla, nmt);
        twomtr(bsub11, bsub12, rslb, nmt);
        smmR(rsla, rslb, psub6, nmt);

        rizn(asub12, asub22, rsla, nmt);
        twomtr(bsub21, bsub22, rslb, nmt);
        smmR(rsla, rslb, psub7, nmt);


        // обрахунок c12 c21 c11 c22:

        twomtr(psub3, psub5, csub12, nmt);
        twomtr(psub2, psub4, csub21, nmt);

        twomtr(psub1, psub4, rsla, nmt);
```

```
        twomtr(rsla, psub7, rslb, nmt);
        rizn(rslb, psub5, csub11, nmt);

        twomtr(psub1, psub3, rsla, nmt);
        twomtr(rsla, psub6, rslb, nmt);
        rizn(rslb, psub2, csub22, nmt);

        // групування результатів

        for (mmm = 0; mmm < nmt ; mmm++) {
            for (nnn = 0 ; nnn < nmt ; nnn++) {
                matC[mmm][nnn] = csub11[mmm][nnn];
                matC[mmm][nnn + nmt] = csub12[mmm][nnn];
                matC[mmm + nmt][nnn] = csub21[mmm][nnn];
                matC[mmm + nmt][nnn + nmt] = csub22[mmm][nnn];
            }
        }
    }
}

unsigned int np2(int z) {
    return pow(2, int(ceil(log2(z))));
}

void smm(vector< vector<int> > &matA,
             vector< vector<int> > &matB,
             vector< vector<int> > &matC, unsigned int z) {
    unsigned int vvv = np2(z);
    vector<int> vnutr(vvv);
    vector< vector<int> > matApidg(vvv, vnutr), matBpidg(vvv, vnutr),
matCpidg(vvv, vnutr);

    for(unsigned int mmm=0; mmm<z; mmm++) {
        for (unsigned int nnn=0; nnn<z; nnn++) {
            matApidg[mmm][nnn] = matA[mmm][nnn];
            matBpidg[mmm][nnn] = matB[mmm][nnn];
        }
    }

    smmR(matApidg, matBpidg, matCpidg, vvv);
    for(unsigned int mmm=0; mmm<z; mmm++) {
        for (unsigned int nnn=0; nnn<z; nnn++) {
            matC[mmm][nnn] = matCpidg[mmm][nnn];
        }
    }
}

void twomtr(vector< vector<int> > &matA,
         vector< vector<int> > &matB,
         vector< vector<int> > &matC, int mt) {
    int mmm, nnn;

    for (mmm = 0; mmm < mt; mmm++) {
        for (nnn = 0; nnn < mt; nnn++) {
            matC[mmm][nnn] = matA[mmm][nnn] + matB[mmm][nnn];
        }
    }
}

void rizn(vector< vector<int> > &matA,
             vector< vector<int> > &matB,
             vector< vector<int> > &matC, int mt) {
    int mmm, nnn;
```

```
    for (mmm = 0; mmm < mt; mmm++) {
        for (nnn = 0; nnn < mt; nnn++) {
            matC[mmm][nnn] = matA[mmm][nnn] - matB[mmm][nnn];
        }
    }
}

int mtxsz(string vhmat) {
    string rdk;
    ifstream vidkr;
    vidkr.open (vhmat.c_str());
    getline(vidkr, rdk);
    return count(rdk.begin(), rdk.end(), '\t') + 1;
}

void matout(vector< vector<int> > mtxx, int z) {
    for (int mmm=0; mmm < z; mmm++) {
        for (int nnn=0; nnn < z; nnn++) {
            if (nnn != 0) {
                cout << "\t";
            }
            cout << mtxx[mmm][nnn];
        }
        cout << endl;
    }
}

void chit(string vhmat, vector< vector<int> > &matA, vector< vector<int> >
&matB) {
    string rdk;
    FILE* mtxxfile = freopen(vhmat.c_str(), "r", stdin);

    if (mtxxfile == 0) {
        cerr << "File can't be read " << vhmat << endl;
        return;
    }

    int mmm = 0, nnn, hhh;
    while (getline(cin, rdk) && !rdk.empty()) {
        istringstream iryad(rdk);
        nnn = 0;
        while (iryad >> hhh) {
            matA[mmm][nnn] = hhh;
            nnn++;
        }
        mmm++;
    }

    mmm = 0;
    while (getline(cin, rdk)) {
        istringstream iryad(rdk);
        nnn = 0;
        while (iryad >> hhh) {
            matB[mmm][nnn] = hhh;
            nnn++;
        }
        mmm++;
    }

    fclose (mtxxfile);
}

int main (int crgt, char* vrgt[]) {
    string vhmat;
```

```
    if (crgt < 3) {
        vhmat = "8000.mnoz";
    } else {
        vhmat = vrgt[2];
    }

    if (crgt < 5) {
        szofdiv = 16;
    } else {
        szofdiv = atoi(vrgt[4]);
    }

    int z = mtxsz(vhmat);
    vector<int> vnutr (z);
    vector< vector<int> > matA(z, vnutr), matB(z, vnutr), matC(z, vnutr);
    chit (vhmat, matA, matB);
    smm(matA, matB, matC, z);
    matout(matC, z);
    return 0;
}
```