**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ**
Факультет інформаційних технологій

УДК 004.9:512.643

«ПОГОДЖЕНО»	**«ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ»**
Декан факультету інформаційних технологій	Завідувач кафедри комп'ютерних наук
Глазунова О.Г., д.п.н., професор	Голуб Б.Л., к.т.н., доцент
______________	______________
______ __________ **2021 р.**	______ __________ **2021 р.**

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Алгоритми матричного множення на GPU для цілочисельних і поліноміальних матриць»

Спеціальність 121 «Інженерія програмного забезпечення»
(код і назва)

Освітня програма Програмне забезпечення інформаційних систем
(назва)

Орієнтація освітньої програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Гарант освітньої програми
кандидат технічних наук, доцент ______________ Б.Л. Голуб
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Керівник магістерської кваліфікаційної роботи
професор, доктор физ.-мат. наук ______________ Г.І. Малашонок
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Виконав ______________ Д.Р. Петренко
(підпис) (ПІБ студента)

КИЇВ-2021

НАЦІОНАЛЬНИЙ УНІВЕРСТИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет (ННІ) інформаційних технологій

ЗАТВЕРДЖУЮ
Завідувач кафедри
к.т.н., доцент Б.Л Голуб
(науковий ступінь, вчене звання) (підпис) (ПІБ)
«___» ______________ 2021 року

З А В Д А Н Н Я

ДО ВИКОНАННЯ МАГІСТЕРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ СТУДЕНТУ

Петренко Денису Руслановичу
(прізвище, ім'я, по батькові)

Спеціальність 121 «Інженерія програмного забезпечення»
(код і назва)

Освітня програма Програмне забезпечення інформаційних систем
(назва)

Орієнтація освітньої програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Тема магістерської кваліфікаційної роботи «Алгоритми матричного множення на GPU для цілочисельних і поліноміальних матриць»

затверджена наказом ректора НУБіП України від «_29_» ___10___ 2020 р. № 1636 «С»

Термін подання завершеної роботи на кафедру ______________
(рік, місяць, число)

Вихідні дані до магістерської кваліфікаційної роботи:

Матеріали переддипломної практики

Перелік питань, що підлягають дослідженню:

1. Можливості ефективного обчислення матричного множення на GPU для цілочисельних матриць
2. Можливості ефективного обчислення матричного множення на GPU для поліноміальних матриць
3. Розробка ефективних алгоритмів для обчислення матричного множення на GPU для цілочисельних і поліноміальних матриць

Дата видачі завдання «__29__» ______10______ 2020 р.

Керівник магістерської кваліфікаційної роботи

професор, доктор физ.-мат. наук ______________ Г.І. Малашонок
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Завдання прийняв до виконання ______________ Д.Р. Петренко
(підпис) (ПІБ студента)

ЗМІСТ

Анотація

Робота присвячена дослідженню та розробці практичних алгоритмів матричного множення для числових та поліноміальних матриць з використанням графічних процесорів. Ці алгоритми враховують архітектурні особливості графічних процесорів, які містять тисячі арифметичних ядер. Досліджуються алгоритми для числових матриць двох видів: над цілими числами та над числами з плаваючою точкою. Виконується порівняльний аналіз швидкості виконання операцій на ЦП та ГП, операцій над числами з різними типами даних. Ми використовуємо програмну модель CUDA (Computer Unified Device Architecture) та мова програмування С++.

Розроблені алгоритми досліджувалися в експериментах на графічному процесорі GeForce GTX 1660. Максимальний розмір матриці експериментів досягав 4000. Прискорення обчислень, яке досягалося під час використання графічної карти проти центральним процесором становило 267 разів для матриць 640х640 над цілими числами; 94 разів для матриць 640х640 над 64-х розрядними числами з плаваючою точкою, 215 разів для матриць 640х640 над 32-х розрядними числами з плаваючою точкою.

Ключові слова: CUDA; GPGPU; паралельні і послідовні алгоритми; мова С++; матричне множення.

Вступ

Актуальність. Матричне множення є одним з фундаментальних операцій на сучасних обчисленнях. Воно виконується рутинним чином мільярди разів щодня по всьому світу в обчисленнях для лінійної алгебри, полілінійної алгебри та поліноміальної алгебри, звичайних диференціальних рівнянь, похідної, інтегральних рівнянь, комбінаторики, статистики, біоінформатики, фізики та інших галузей науки, техніки і обробки сигналів і образів. І хоча пошуки оптимального алгоритму множення матриць, здатного виконати обчислення за квадратичного часу, відновились і переживають активну фазу [1] [2] [3], більшість розроблених субкубічних алгоритмів розроблених в останні два десятиліття показують свою ефективність лише на матрицях астрономічних розмірів. Здебільше, це обумовлено введенню складних математичних моделей і проблемами їх алгоритмізації.

Стрімкий розвиток в науці привів і до появи нової технології програмно-апаратної архітектури CUDA [4], в якій стала можлива одночасна обробка масиву даних з використанням множини активних обчислювальних одиниць (ядер). Ідея скорочення часу необхідного для виконання задачі за рахунок її розподілення між наявними ядрами показала свою ефективність порівняно з традиційною одно поточною системою. Проте, потрібно зазначити, що такі паралельні алгоритми часто відрізняються підвищеною складністю порівняно з послідовними аналогами.

Мета дослідження – розвиток ідеї підвищення ефективності обчислювання використовуючи сучасні технології Nvidia CUDA

Завдання дослідження – здійснити аналіз предметної області, сучасних рішень щодо вирішення поставленої проблеми, архітектурних особливостей графічних процесорів у порівнянні з ЦП; розроблення алгоритмів для множення цілочисельних і поліноміальних матриць і на даних отриманих в ході розбору, тестування і аналізу програми зробити висновок, щодо якості побудованого рішення.

Об'єкт дослідження – практичні паралельні алгоритми матричного множення.

Предмет дослідження – Теоретичні, методичні засади та практичні аспекти вирішення задачі лінійної алгебри з використанням графічного процесора.

Джерела дослідження. Електронні версії друкованої літератури, програмна документація, електронні ресурси (в тому числі спеціалізовані форуми, вихідні коди програм та бібліотеки), відео та навчальні посібники.

Наукова новизна одержаних результатів. Запропонована архітектура і результати роботи можуть використовуватись для подальшого дослідження. Освітлені оптимізаційні рішення та можливі проблеми при роботі з такими типами даних, що були виявлені в ході виконання детального аналізу (з порівнянням традиційних та розроблених алгоритмів).

Апробація результатів дослідження

1. Петренко Д. Р. Особливості реалізації арифметики довільної точності на графічних прискорювачах // Збірник матеріалів XII Міжнародної науково-практичної конференції молодих вчених «Інформаційні технології: економіка, техніка, освіта». – Київ. – 2021. – ст. 111-112 (тези доступні за посиланням: https://drive.google.com/file/d/1LhyVBHBvpMKiV3gIJFsF3EKv_n6Mf Y2L/view).

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ МАТРИЧНОГО АЛГОРИТМІВ МНОЖЕННЯ

1.1 Історія розвитку та застосування алгоритмів матричного множення

Теорія матриць розпочала активний розвиток у середині XIX століття. У цей час вже були формульовані правила складання та множення матриць. Результати теорії матриць розроблені та відображені в роботах вчених: ірландський математик і фізик Вільям Гамільтон (1805-1865), англійський математик Артур Келі (1821-1895), німецькі математики Карл Вейєрштрас (1815-1897) та Фернінд, французький математик Марі Енмон Каміль Жордан (1838-1922) і ввів термін «матриця» Джеймс Сільвестр (1814-1897) у 1850 р [5].

Класичний алгоритм множення матриць розміру n × n використовує n^3 множень елементів і $n^3 - n^2$ додавань. В сучасних реалізаціях матричного множення ці операції виконуються набагато швидше, ніж зазвичай, і різні важливі обчислення для природничих наук, техніки та електроінженерії були суттєво прискорені лише за рахунок зведення їх до матричного множення.

До 1969 р. більшість фахівців з прикладної та обчислювальної математики по всьому світу були впевнені, що цей алгоритм є оптимальним, і прискорення до менш ніж $2n^{\omega}$ скалярних арифметичних операцій, де $\omega \leq 7 \ \approx 2.808$ представлене Ф. Штрассеном у роботі [6], сприймалося з сумнівами, незважаючи на те, що в роботі згадувалося про отриману раніше економію приблизно 50% скалярних множень для класичного алгоритму (див. [7] та [8]).

Проте значення 7 не піддавалося атакам протягом майже десяти років, до 1978 р., коли цей рекорд був побитий у роботі [9] за допомогою розвинених методів трилінійного агрегування, введених у 1972 р. (див. [10] та [11]). Ця техніка була новаторською в галузі тензорних розкладів і зовсім не пов’язаною з ідеями Штрассена. Поєднання цих алгоритмів з деякими іншими методами дозволило ще зменшити показник у 1978-1981 рр. і потім знову в 1986 р., після чого алгоритм був покращено в роботі Копперсміта – Винограда, який має асимптотичну складність 2.372.

Через три десятиліття показник все ще не подолав бар’єр 2.37, але найбільше занепокоєння викликало наявність рекурсій: для практично можливого множення матриць порядку $n < 1\,000\,000\,000$ значення показника 2.7734, отримане в роботі [12] в 1922 г. Показник практично здійснюваного матричного множення був отриманий виключно за допомогою трилінійного агрегування, тоді як менші значення показника були відомі лише для нездійсненного на практиці матричного множення: відповідні алгоритми вимагають численних рекурсивних кроків, і на кожному з них розмір завдання зростає квадратично. Такі алгоритми перевершують класичний алгоритм лише з вхідних даних великих розмірів, значно перевищують будь-який потенційний інтерес користувача.

1.2 Матриці та їх властивості

Розмір матриці визначається кількістю рядків і стовпців у матриці. Якщо матриця має m рядків і n стовпців, то вона називається матрицею $m \times n$ [13] [14] [15]. Матриці можуть бути різного розміру: прямокутні, квадратні, також є матриці-рядки та матриці-стовпці, які називають векторами. Матриця розміру m на n виглядає так:

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} = (a_{ij}) \in R^{m\times n} \quad (1.1)$$

де a_{ij} відповідає i-му рядку та j-му стовпцю матриці A.

Над матрицями можна виконувати певні дії, які, за аналогією з числами, називаються додавання, віднімання та множення. Так само існує дія, яка визначається лише для матриць – це транспонування матриць та перебування зворотної матриці до цієї.

Основні операції над матрицями:

Сумою $A+B$ матриць $A_{m\times n}=(a_{ij})$ та $B_{m\times n}=(b_{ij})$ називається матриця $C_{m\times n}=(c_{ij})$, де $c_{ij}=a_{ij}+b_{ij}$ для всіх $i=\underline{1,m}$ і $j=\underline{1,n}$.

Різницею $A-B$ матриць $A_{m\times n}=(a_{ij})$ та $B_{m\times n}=(b_{ij})$ називається матриця $C_{m\times n}=(c_{ij})$, де $c_{ij}=a_{ij}+b_{ij}$ для всіх $i=\underline{1,m}$ і $j=\underline{1,n}$.

Добутком матриці $A_{m\times n}=(a_{ij})$ на матрицю $B_{m\times n}=(b_{ij})$ називається матриця $C_{m\times n}=(c_{ij})$, для якої кожен елемент c_{ij} дорівнює сумі творів відповідних елементів і-го рядка матриці A на елементи j-го стовпця матриці B:

$$c_{ij}=\sum_{p=1}^{n} a_{ip}b_{pj}, i=\underline{1,m}, j=\underline{1,n} \qquad (1.2)$$

Додавання та множення матриць характеризуються такими властивостями:

1. $A+B=B+A$; (комутативність додавання)
2. $A+(B+C)=(A+B)+C$; (асоціативність додавання)
3. $(\alpha+\beta)\cdot A=\alpha A+\beta A$; (дистрибутивність множення на матрицю щодо складання чисел)
4. $\alpha\cdot(A+B)=\alpha A+\alpha B$; Дистрибутивність множення на число щодо складання матриць)
5. $A\cdot(B\cdot C)=(A\cdot B)\cdot C$;
6. $(\alpha\beta)A=\alpha(\beta A)$;
7. $A\cdot(B+C)=A\cdot B+A\cdot C$; $(B+C)\cdot A=B\cdot A+C\cdot A$.
8. $A\cdot E=A$; $E\cdot A=A$, де E – одинична матриця відповідного порядку.
9. $A\cdot O=O, O\cdot A=O$, де O – нульова матриця відповідного розміру.
10. $(A^T)^T=A$
11. $(A+B)^T=A^T+B^T$
12. $(A\cdot B)^T=B^T\cdot A^T$
13. $(\alpha A)^T=\alpha A^T$

1.4 Алгоритми множення матриць

1.4.1 Наївне множення матриць

Матричне множення відноситься до добутку двох матриць однакової розмірності [16]. Припустимо, що A є матрицею $m \times p$, а B – матрицею $p \times n$:

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1p} \\ a_{21} & a_{22} & \cdots & a_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mp} \end{pmatrix}, B = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{p1} & b_{p2} & \cdots & b_{pn} \end{pmatrix} \quad (1.1)$$

Тоді, в результаті множення матриці A на B ми отримаємо матрицю C розмірністю $m \times n$.

$$C = A \times B = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mn} \end{pmatrix} \quad (1.2)$$

Таку, що:

$$c_{ij} = a_{i1} \cdot b_{1j} + \cdots + a_{ip} \cdot b_{pj} = \sum_{k=1}^{p} a_{ik}\, b_{kj}, \quad (1.5)$$

де $i = \underline{1, m}$ і $j = \underline{1, n}$ [17]. Тобто: c_{ij} це множення й додавання i-го рядка A та j-го стовпця B (рис 1.1).

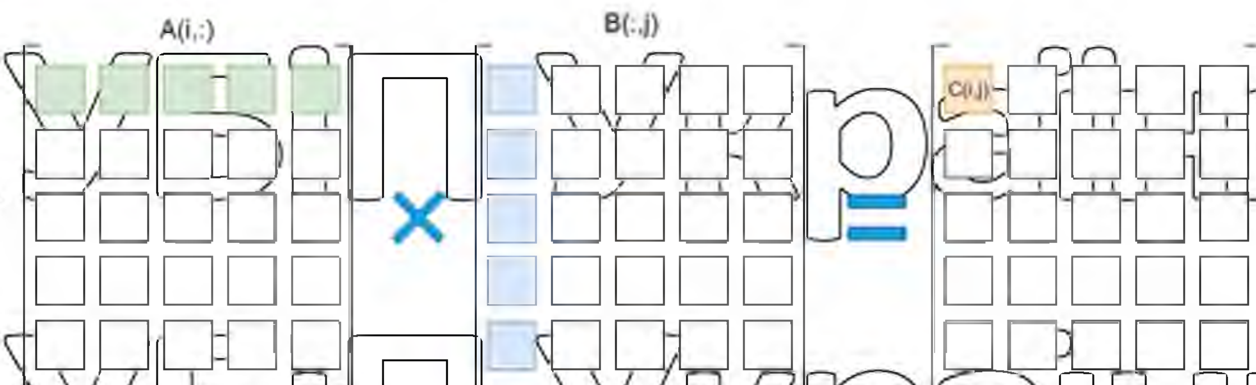


Рис. 1.2 Наївний метод множення матриць

У псевдокоді цей алгоритм виглядає так:

Алгоритм 1: Ітеративний алгоритм

```
Вхід: матриці A та B
Нехай C буде матрицею n × n
    для i з 1 по n:
        для j з 1 по p:
            для k з 1 по m
                C[i][j] ← C[i][j] + A[i][k]*B[k][j]
Повернути C
```

Поведінка кешу

Хоча три цикли в ітеративному алгоритмі множення матриць можна переставляти у довільному порядку без впливу на правильність обчислення чи асимптотичну тривалість виконання, на практиці порядок обходу матриць може мати значний вплив на практиці через моделі доступу до пам'яті та використанням алгоритмом кешу [18].

Більш оптимальним варіантом «наївного» алгоритму для перемноження матриць A та B є плиткова версія, де матриця ділиться на квадратні плитки розміру T на T(рис. 5)

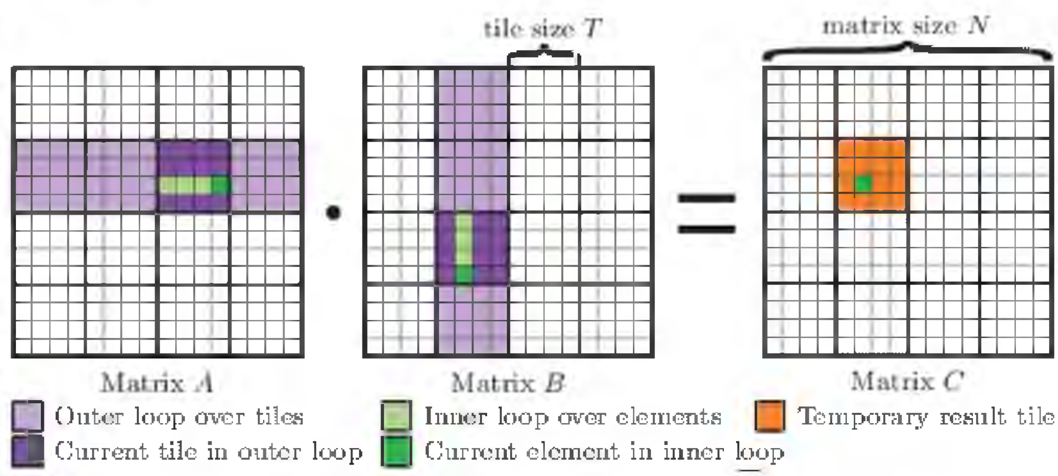


Рис. 1.3 Алгоритм блочного ітеративного множення

Псевдокод плиткової або блочної версії ітеративного алгоритму поданий нижче.

Алгоритм 2: Блочна версія ітеративного алгоритму

Вхід: матриці A та B

```
нехай C буде матрицею відповідного розміру
обрати розмір плитки T = Θ(√M)
для I з 1 по n кроками по T:
    для J з 1 по p кроками по T:
        для K з 1 по m кроками по T:
        //перемножити A_{I:I+T, K:K+T} та B_{K:K+T, J:J+T} до C_{I:I+T, J:J+T}:
            для i з I по min(I + T, n):
                для j з J по min(J + T, p):
                    sum = 0
                    для k з K по min(K + T, m):
                        sum ← sum + A_ik × B_kje
                    C_ij ← C_ij + sum
повернути C
```

Поблочне множення матриць

Наприклад, у найпростішому випадку А є матрицею $m \times n$, а В – матрицею $n \times m$. Розглянемо А як матрицю стовпців з m блоків, де кожен блок є вектором-рядком, і розглянемо В як матрицю рядків з m блоків, де кожен блок є вектором-стовпцем. Зауважимо, що для успішної реалізації множення блочної матриці кількість стовпців у А має дорівнювати кількості рядків у В [17]. Зазвичай застосовувана методологія множення матриці блоків полягає у фіксації розміру блоку, як показано на малюнку 1.2.

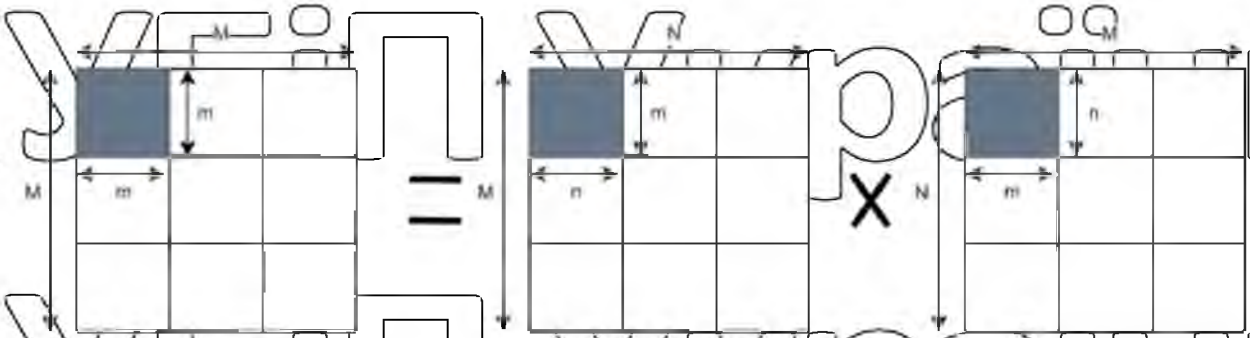


Рис. 1.4 Блочне множення матриць

Поблочне множення матриць використовується для підвищення продуктивності обчислювальної системи за рахунок розпаралелювання задачі серед декількох пристроїв чи потоків. Однак це значно збільшує складність комунікації до $\Theta(n^3)$, оскільки для обробки матриці на блоки потрібно багато часу. Тому необхідно оптимізувати обчислювальні та комунікаційні витрати з точки зору проектування та реалізації [19]

1.3.2 Дихотомічний блочно-рекурсивний алгоритм

Рекурсія – це базова структура для побудови потоку даних із багаторазовим виконанням тіла процедури. Розгортання рекурсії є складною методологією для оптимізації рекурсивних процедур [20]. Алгоритм «Розділяй і володарюй» – це різновид методів розгортання рекурсії. Він працює шляхом рекурсивного поділу основної проблеми на дві або більше підпроблеми, поки ці підпроблеми не стануть достатньо малими, щоб їх було легко розв'язати. Це може допомогти вирішити складні проблеми з меншим ступенем складності, але також затримує виконання програми. Як показано на малюнку 1.3, типовий алгоритм «розділяй і володарюй» [21] поділено на 3 кроки:

- Розділіть. Розділити основну проблему на кілька підпроблем.
- Перемагати. Вирішіть ці підпроблеми рекурсивно.
- Комбінуйте. Об'єднайте ці рішення, щоб отримати кінцевий результат.

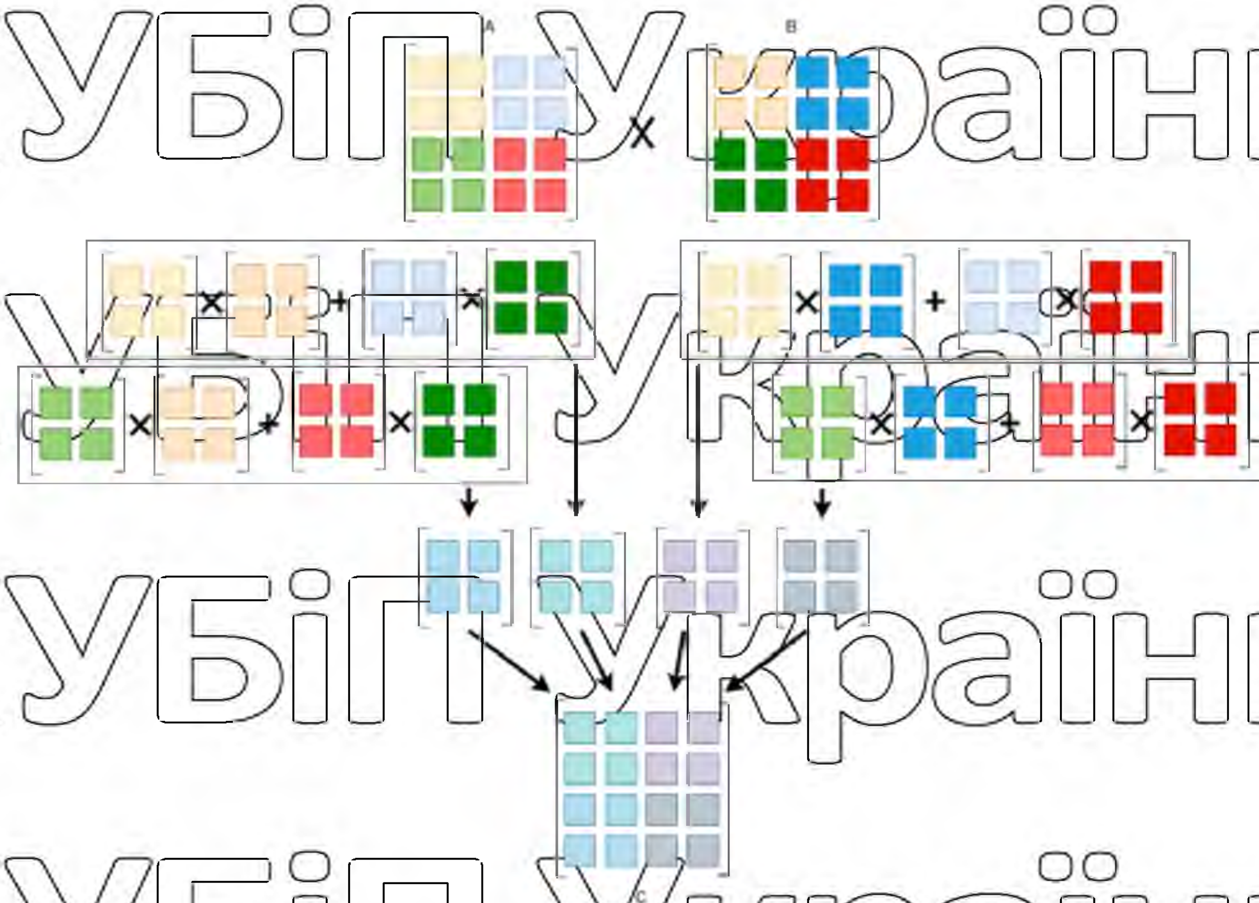


Рис. 1.1 Блочно-рекурсивний алгоритм

Припустимо, що n дорівнює ступеню 2 в кожній із матриць $n \times n$ для A і B. Це спрощене припущення дозволяє нам розбити велику матрицю $n \times n$ на менші блоки або квадранти розміру $n/2 \times n/2$ (попередньо переконавшись, що розмірність отриманих блоків $n/2$ є цілим числом). Цей процес називається розділенням блоків. Перевага даного методу [21] полягає в тому, що після того, як матриці розбиті на блоки і помножені, блоки поводяться так, ніби вони є атомарними елементами. Тоді добуток A і B можна виразити через його блоки. Таким чином ми можемо продовжувати розбивати матриці на квадранти через рекурсію чи ітеративним методом, поки вони не стануть достатньо малими, щоб їх можна було помножити наївним способом.

Нехай ми розбиваємо кожну з A, B і C на чотири матриці $n/2 \times n/2$

$$A = (A_{11}\ A_{12}\ A_{21}\ A_{22}\ ), \qquad B = (B_{11}\ B_{12}\ B_{21}\ B_{22}\ ),$$
$$C = (C_{11}\ C_{12}\ C_{21}\ C_{22}\ ), \tag{1.5}$$

Перепишимо рівняння $C = A \cdot B$ так, щоб:

$$(C_{11}\ C_{12}\ C_{21}\ C_{22}) = (A_{11}\ A_{12}\ A_{21}\ A_{22}) \cdot (B_{11}\ B_{12}\ B_{21}\ B_{22}) \tag{1.6}$$

Тоді, вирішивши рівняння 1.2 ми отримаємо:

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \tag{1.7}$$

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \tag{1.8}$$

$$C_{13} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \tag{1.9}$$

$$C_{14} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \tag{1.10}$$

Алгоритм 3: Блочно-рекурсивний алгоритм

Функція matMul(A, B, n)

```
Якщо n = 1  повернути A × B
Інакше
    Розбити на A11, B11, . . ., A22, B22 обчисливши m = n/2
        X1 ← matMul(A11, B11, n/2)
        X2 ← matMul(A12, B21, n/2)
        X3 ← matMul(A11, B12, n/2)
        X4 ← matMul(A12, B22, n/2)
        X5 ← matMul(A21, B11, n/2)
        X6 ← matMul(A22, B21, n/2)
        X7 ← matMul(A21, B12, n/2)
        X8 ← matMul(A22, B22, n/2)
        C 11 ← X1 + X2
```

```
13:  C 12 ← X3 + X4
14:  C 21 ← X5 + X6
15:  C 22 ← X7 + X8
16:  Повернути C
```

Операції в рядку 3 займають постійний час. Вартість об´єднання (рядки 12–15) дорівнює $\Theta(n^2)$ (додавання двох матриць n/2 × n/2 займає час $4 \cdot n/2 = \Theta(n^2)$). Є 8 рекурсивних викликів (рядки 4–11). Отже, нехай $T(n)$ – загальна кількість математичних операцій, виконаних matMul(A, B, n), тоді

$$T(n) = 8T\frac{n}{2} + \theta(n^2) \qquad (1.11)$$

Вирішення цієї майстер-теореми дає нам:

$$T(n) = \theta(n^8) = \theta(n^3) \qquad (1.12)$$

Отже, це не є покращення «наївного» алгоритму, наведеного раніше (який використовує n^3 операції), хоча переваги та застосування даного алгоритму будуть висвітлено пізніше у роботі.

1.3.3 Алгоритм Штрассена

У 1969 році Штрассен значно покращив процес множення матриці, опублікувавши свою роботу [6] про алгоритм складності $\theta(n^{2.81})$. У разі множення двох матриць 2×2 потрібно лише 7 множень і 18 додавань замість 8 множень і 4 додавань у наївному методі [22]. Обчислення в алгоритмі Штрассена (SA) виглядає наступним чином.

Нехай ми маємо дві квадратні матриці A і B розмірністю 2×2:

$$A = (a_{11}\ a_{12}\ a_{21}\ a_{22}\ ), \qquad B = (b_{11}\ b_{12}\ b_{21}\ b_{22}\ ) \tag{1.5}$$

Після виконання алгоритму буде отримання матриці $C = A \times B$

• Перший крок - виконати додавання/віднімання:

$$\begin{aligned} U_1 &= a_{11} + a_{22}; & V_1 &= b_{11} + b_{22}; \\ U_2 &= a_{21} + a_{22}; & V_2 &= b_{12} - b_{22}; \\ U_3 &= a_{11} + a_{12}; & V_3 &= a_{21} - b_{11}; \\ U_4 &= a_{21} - a_{11}; & V_4 &= b_{11} + b_{12}; \\ U_5 &= a_{12} - a_{22}; & V_5 &= b_{21} + b_{22}; \end{aligned} \tag{1.6}$$

• Другим кроком стане обчислення P1, P2,…, P7

$$\begin{aligned} P_1 &= U_1 \cdot V_1 \\ P_2 &= U_2 \cdot b_{11} \\ P_3 &= a_{11} \cdot V_2 \\ P_4 &= a_{22} \cdot V_3 \\ P_5 &= U_3 \cdot b_{22} \\ P_6 &= U_4 \cdot V_4 \\ P_7 &= U_5 \cdot V_5 \end{aligned} \tag{1.7}$$

• Останнім кроком є обчислення:

$$c_{11} = P_1 + P_4 - P_5 + P_7 \tag{1.8}$$

$$c_{12} = P_2 + P_4 \tag{1.9}$$

$$c_{21} = P_3 + P_5 \tag{1.10}$$

$$c_{22} = P_1 + P_3 - P_2 + P_6 \tag{1.11}$$

Чотири рівняння безпосередньо вище ведуть до вихідної матриці С:

$$C = (c_{11}\ c_{12}\ c_{21}\ c_{22}\ ) \tag{1.12}$$

Наведені вище формули можна використовувати для рекурсивного обчислення A × B наступним чином:

Алгоритм 4: Алгоритм Штрассена

```
Strassen(A, B)
1:     Якщо n = 1 повернути A × B
2:     Інакше
3:         Обчислити A11, B11, . . ., A22, B22 m = n/2
4:             P1 ← Strassen(A11, B12 − B22)
5:             P2 ← Strassen(A11 + A12, B22)
6:             P3 ← Strassen(A21 + A22, B11)
7:             P4 ← Strassen(A22, B21 − B11)
8:             P5 ← Strassen(A11 + A22, B11 + B22)
9:             P6 ← Strassen(A12 − A22, B21 + B22)
10:            P7 ← Strassen(A11 − A21, B11 + B12)
11:            C11 ← P5 + P4 − P2 + P6
12:            C12 ← P1 + P2
13:            C21 ← P3 + P4
14:            C22 ← P1 + P5 − P3 − P7
15:    Output C
```

Операції в рядку 3 займають постійний час. Вартість об'єднання 4 блоків (рядки 11–14) дорівнює $\Theta(n^2)$. Є 7 рекурсивних викликів (рядки 4–10). Отже, нехай $T(n)$ – загальна кількість математичних операцій, виконаних Strassen (A, B), тоді

$$T(n) = 7T\frac{n}{2} + \theta(n^2) \qquad (1.13)$$

Вирішення цієї майстер-теореми дає нам:

$$T(n) = \theta(n^{7}) = \theta(n^{2.8}) \quad (1.14)$$

1.3.4 Алгоритм Винограда-Штрассена

Віноград запропонував модифікацію [23], яка вимагає на 3 додавання менше, ніж алгоритм Штрассена. Цей алгоритм побудований так само, як і в SA. Нехай ми маємо дві квадратні матриці A і B розмірністю 2×2:

$$A = (a_{11}\ a_{12}\ a_{21}\ a_{22}), \qquad B = (b_{11}\ b_{12}\ b_{21}\ b_{22}) \quad (1.5)$$

- Перший крок - виконати додавання/віднімання:

$$\begin{aligned} U_1 &= a_{11} - a_{21}; & V_1 &= b_{22} - b_{12}; \\ U_2 &= a_{21} + a_{22}; & V_2 &= V_1 + b_{11}; \\ U_3 &= U_1 - a_{22}; & V_3 &= V_2 - b_{21}; \\ U_4 &= U_3 + a_{12}; & V_4 &= b_{12} - b_{11}; \end{aligned} \quad (1.6)$$

- Другим кроком стане обчислення P1, P2,…, P7

$$\begin{aligned} P_1 &= a_{11} \cdot b_{11} \\ P_2 &= a_{12} \cdot b_{21} \\ P_3 &= a_{22} \cdot V_3 \\ P_4 &= U_1 \cdot V_1 \\ P_5 &= U_2 \cdot V_4 \\ P_6 &= U_4 \cdot b_{22} \\ P_7 &= U_3 \cdot V_2 \end{aligned} \quad (1.7)$$

- Останнім кроком є обчислення:

$$c_{11} = P_1 + P_2 \quad (1.8)$$

$$c_{12} = ((P_1 - P_7) - P_5) + P_6 \quad (1.9)$$

$$c_{21} = (P_1 - P_7) - P_3 + P_4 \quad (1.10)$$

$$c_{22} = (P_1 - P_7 + P_5) + P_4 \tag{1.11}$$

Чотири рівняння безпосередньо вище ведуть до вихідної матриці C:

$$C = (c_{11}\ c_{12}\ c_{21}\ c_{22}\ ) \tag{1.12}$$

Важливо пам'ятати про повторне використання деяких виразів у WV. (a11 − a21) використовується як в U1, так і в U3, а (b22 − b12) використовується як у V1, так і в V2. (a11 − a21 − a22) використовується як в U3, так і в U4. (b11 − b12 + b22) використовується як у V2, так і у V3. (P1 − P7) у c12 також використовується в c21. (P1 − P7 + P5) в c12 також використовується в c22. Тому, дотримуючись правила неповторюваних операцій з однаковими параметрами, легко з'ясувати, що потрібно лише 8 і 7 додавань/віднімань, щоб отримати P_i і c_{ij} відповідно. Загальна арифметична вартість реалізації WV становить 7 множень і 15 додавань\віднімань.

Д'Альберто в своїй роботі [24] запропонував способи покращення точності обчислення для алгоритму Вінограда-Штрассена.

Алгоритм 5: Д'Альберто Вінограда-Штрассена

```
Winograd(A, B, C, n)
    Якщо n < τ повернути A × B
    Інакше
        T1 = A11 – A21;
        T2 = B22 – B12
        winograd(T1, T2, C21, n/2) //P4
        T1 = A21 + A22;
        T2 = B12 – B11;
        winograd(T1, T2, C22, n/2) //P5
        T1 = T1 – A11;
        T2 = B22 – T2;
        winograd(T1, T2, C11, n/2) //P1
        T1 = A12 – T1;
```

```
winograd(T1, B22, C12, n/2) //P6
C12 = C12 +C22;
winograd(A11, B11, T1, n/2) //P2
C12 = C11 + C12 + T1;
C11 = C11 + C21 + T1;
T2 = T2 – B21;
winograd(A22, T2, C21, n/2)
C21 = C11 – C21;
C22 = C11 + C22;
winograd(A12, B21, C11, n/2) //P3
C11 = C11 + T1;
```

Обрахунок матриць непарного розміру

До алгоритмів, що ґрунтуються на принципах Штрассена необхідно застосувати певну техніку – так як ранг таких алгоритмів здебільше дорівнює 2, ми повинні спочатку зробити матрицю парною і тільки після цього потім виконати обчислення. Спочатку Штрассен запропонував доповнити вхідні матриці додатковими рядками і стовпцями нулів, щоб розміри всіх матриць, які зустрічаються під час рекурсивних викликів, були парними. Після обчислення добутку зайві рядки та стовпці видаляються, щоб отримати бажаний результат. Ми називаємо цей підхід статичним заповненням, оскільки заповнення відбувається перед будь-якими рекурсивними викликами алгоритму Штрассена.

Крім того, щоразу, коли алгоритм Штрассена викликається рекурсивно, до кожного входу з непарним розміром рядка можна додати додатковий рядок нулів, а для кожного входу з непарним розміром стовпця можна додати додатковий стовпець нулів. Цей підхід до заповнення називається динамічним заповненням, оскільки заповнення відбувається протягом усього виконання алгоритму Штрассена. Використовується версія динамічного заповнення [25].

$$A = \left(\begin{array}{c|c} A_{11} & a_{12} \\ \hline a_{21} & a_{22} \end{array}\right), \quad B = \left(\begin{array}{c|c} B_{11} & b_{12} \\ \hline b_{21} & b_{22} \end{array}\right) \tag{1.13}$$

Інший підхід, який називається динамічним відшаровуванням [26] , має справу з непарними розмірами шляхом видалення зайвого рядка та/або стовпця за потреби та додавання їхнього внеску до остаточного результату в наступному етапі роботи з виправленням. Нехай A – матриця $m \times k$, а B – матриця $k \times n$. Припускаючи, що m, k і n всі непарні, A і B розбиваються на блочні матриці

$$\left(\begin{array}{c|c} C_{11} & c_{12} \\ \hline c_{21} & c_{22} \end{array}\right) = \left(\begin{array}{c|c} A_{11}B_{11} + a_{12}b_{21} & A_{11}b_{12} + a_{12}b_{22} \\ \hline a_{21}B_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{array}\right) \tag{1.14}$$

РОЗДІЛ 2. ГРАФІЧНІ ПРОЦЕСОРИ NVIDIA ТА ПРОГРАМУВАННЯ НА НИХ

Зі збільшенням потоку інформації збільшуються і вимоги в її обробці. На певному етапі затрати становляться настільки помітними, що ми неминуче приходимо до пошуків нових, більш ефективні підходів, адже наявні послідовні алгоритми вже не здатні виконати задачу за поставлений період часу [27]. Так, використання паралельних алгоритмів разом з багатопроцесорними системами стало не забаганкою, а необхідністю в нинішній час.

Отримати паралельний алгоритм розв'язання задачі можна шляхом розпаралелювання наявного послідовного алгоритму або шляхом розробки нового паралельного алгоритму.

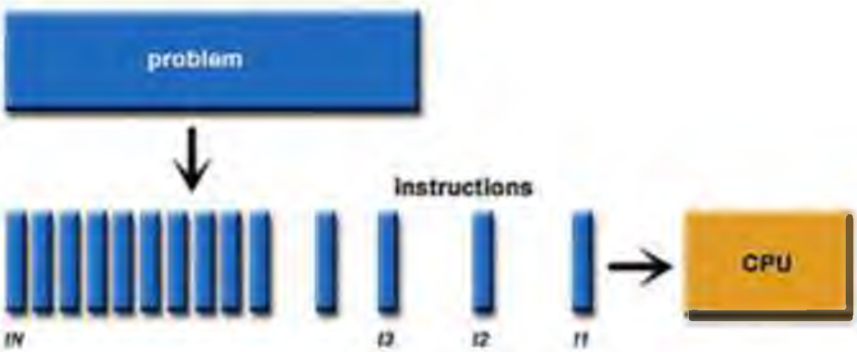


Рис. 2.1 Схематичне зображення послідовного виконання задачі

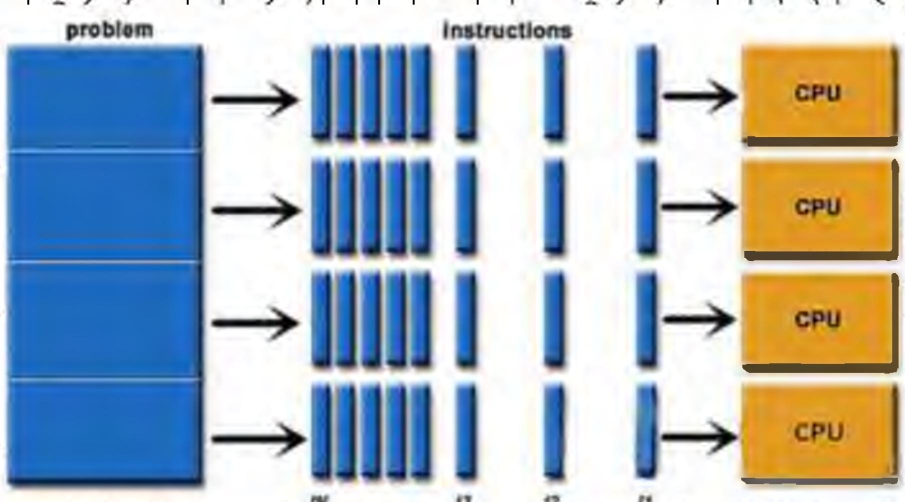


Рис. 2.2 Схематичне зображення паралельного виконання задачі

Навіть короткий перелік типів сучасних паралельних обчислювальних систем дає зрозуміти, що для орієнтування в цьому різноманітті потрібна чітка система класифікації.

У 1972 році Флінн [28] представив класифікацію комп'ютерних систем, яка зараз широко використовується комп'ютерною спільнотою. Основним визначальним архітектурним параметром він вибрав взаємодію потоку команд та потоку даних (операндів та результатів). Він розділив машини на 4 категорії на основі того, як машина співвідносить свої інструкції з даними, що обробляються. Це категорії:

1. SISD (Single Instruction stream/Single Data stream) – одиночний потік команд і даних.

2. SIMD (Single Instruction stream/Multiple Data stream) – одиночний потік команд і декілька потоків даних.

3. MISD(Multiple Instruction stream/Single Data stream) – декілька потоків команд, одиночний потік даних.

4. MIMD(Multiple Instruction stream/Multiple Data stream) – декілька потоків команд і даних.

2.1 Модель програмування CUDA

CUDA – це архітектура паралельних обчислень від NVIDIA, що дозволяє суттєво збільшити обчислювальну продуктивність завдяки використанню GPU. На сьогоднішній день продажі CUDA процесорів досягли мільйонів, а розробники програмного забезпечення, вчені та дослідники широко використовують CUDA у різних галузях, включаючи обробку відео та зображень, обчислювальну біологію та хімію, моделювання динаміки рідин, відновлення зображень, отриманих шляхом комп'ютерної томографії, сейсмічний аналіз, трасування променів та багато іншого [29] [30] [31].

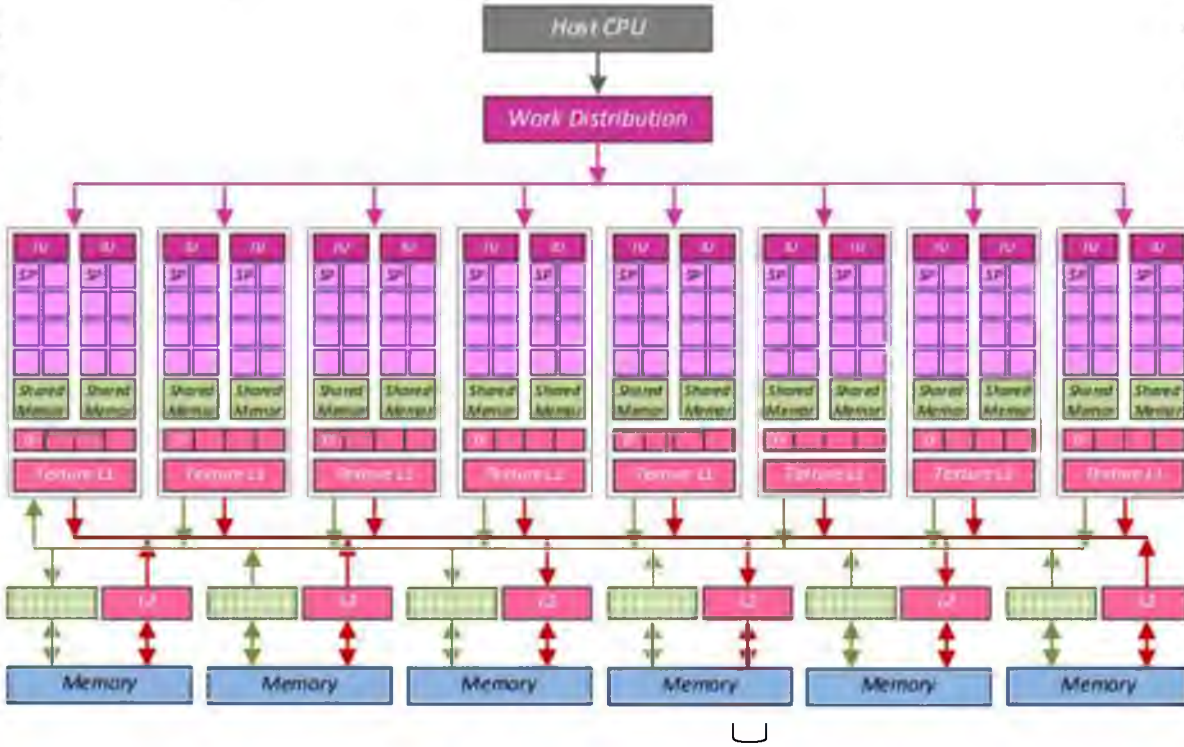


Рис. 2.1 Модель виконавчих пристроїв CUDA

Окремі потоки групуються в блоки потоків (thread block) однакового розміру, причому кожен блок потоків виконується на окремому мультипроцесорі. Кількість потоків у блоці обмежена (максимальні значення для конкретних пристроїв можуть бути знайдені [32] або отримані під час виконання за допомогою функцій CUDA API). Потоки всередині блоку потоків можуть ефективно взаємодіяти між собою за допомогою загальної пам'яті і синхронізації. Крім того, потоки можуть взаємодіяти через глобальну пам'ять чи за допомогою атомарних операцій.

На апаратному рівні потоки блоку групуються в так звані варпи (warps) [32] по 32 елементи (на всіх поточних пристроях), всередині яких всі потоки паралельно виконують однакові інструкції (за принципом SIMT – Single Instruction, Multiple Threads). При цьому важливо відзначити, що код ядер можна писати скалярно (тобто ядро містить код, який виконується кожним потоком, без урахування апаратних особливостей виконання). Виконання всіх потоків у варпі починається однаково, далі траєкторії виконання можуть розходитися – у разі розгалужень варп послідовно виконує всі можливі шляхи (при цьому працюють тільки потоки, що досягають цих шляхів, а решта тимчасово «засинають»). З цієї причини операції розгалуження можуть негативно позначатися на продуктивності – різні шляхи не можуть виконуватися паралельно (у той же час потоки одного варпа, що виконують один шлях, працюють паралельно).

У свою чергу, блоки потоків об'єднуються в сітку блоків потоків [32] (англ. grid of thread blocks). Слід зазначити, що взаємодія потоків з різних блоків під час роботи ядра утруднена: відсутні явні інструкції синхронізації на рівні блоків, взаємодія можлива через глобальну пам'ять та з використанням атомарних функцій (іншим варіантом є розбиття ядра на кілька ядер без внутрішньої взаємодії між потоками різних блоків). блоки потоків об'єднуються в грати блоків потоків (англ. grid of thread blocks).

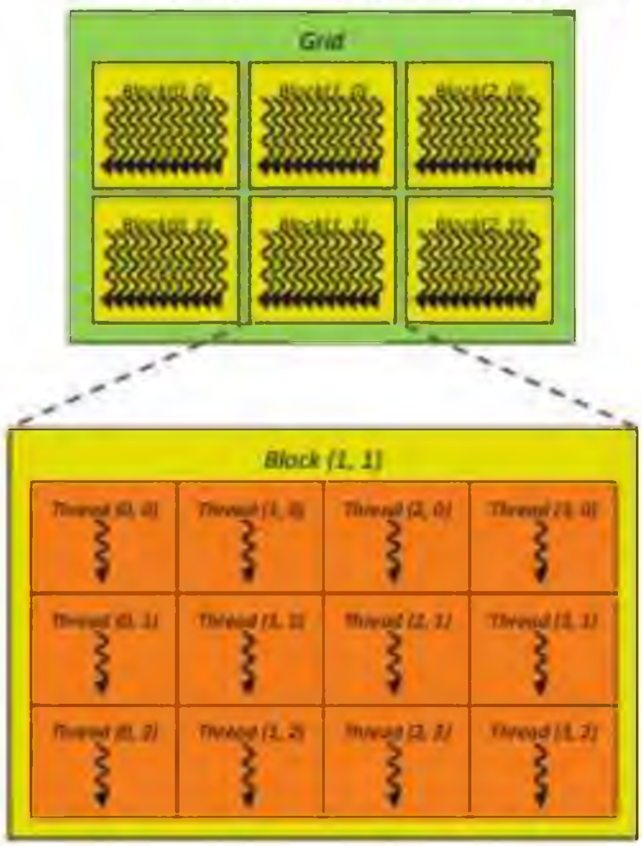


Рис. 2.2 Ієрархія потоків CUDA

Кожен потік усередині блоку потоків має свої координати (одно-, дво- чи тривимірні), які доступні через вбудовану змінну **threadIdx**. У свою чергу, координати блоку потоків (одно-, дво- або тривимірні) всередині решітки визначається вбудованою змінною **blockIdx**. Приклад ієрархії потоків наведено на рис 2.4. Дані вбудовані змінні є структурами поля .x, .y, .z.

Застосування даних понять може бути легко показано на прикладі додатка, що виконує складання двох матриць. Розіб'ємо обчислення суми матриць на обчислення суми підматриць (кожне з яких здійснюватиметься окремим блоком потоків). Кожен окремий потік всередині блоку потоків виробляє обчислення одного елемента з координатами, що залежать від координат блоку потоків всередині грат і координат потоку всередині блоку. Передбачається, що матриці мають N рядків і N стовпців і зберігаються у вигляді одновимірних масивів рядків. Далі наводиться лістинг із вихідним кодом функції-ядра мовою CUDA C++.

```
1   template <typename T> __global__ void matAdd(T* A, T* B, T* C,
2       int lda, int ldb, int ldc,
3       int XA, int YA)
4   {
5       int row = blockIdx.y * blockDim.y + threadIdx.y;
6       int col = blockIdx.x * blockDim.x + threadIdx.x;
7
8       if (row < XA && col < YA)
9       {
10          C[row * ldc + col] = A[row * lda + col] + B[row * ldb + col];
11      }
12  }
```

При цьому використовуються двовимірні індекси блоків і потоків. Передбачається, що буде створено всього (N, N) або більше потоків. Вирази у рядках 5 і 6 служать для обчислення індексу потоків серед усіх потоків всіх блоків за першою та другою координатами. Умова в рядку 8 необхідна для запобігання виходу індексів за межі масивів у випадку, коли потоків хоча б по одній з розмірностей буде строго більше N (що неминуче, коли N не кратне число потоків у блоці, тому що всі блоки мають однаковий розмір).

Ієрархія пам'яті.

Технологія CUDA надає доступ до кількох рівнів пам'яті:

- Кожен потік має свою локальну пам'ять (local memory).

• Всі потоки всередині блоку мають доступ до швидкої пам'яті, що розділяється (shared memory), час життя якої збігається з часом життя блоку. Пам'ять блоку, що розділяється, розбита на сторінки, при цьому доступ до даних на різних сторінках здійснюється паралельно.

• Всі потоки у всіх блоках мають доступ до глобальної пам'яті пристрою (global memory або device memory), яка зберігає свій стан протягом роботи програми.

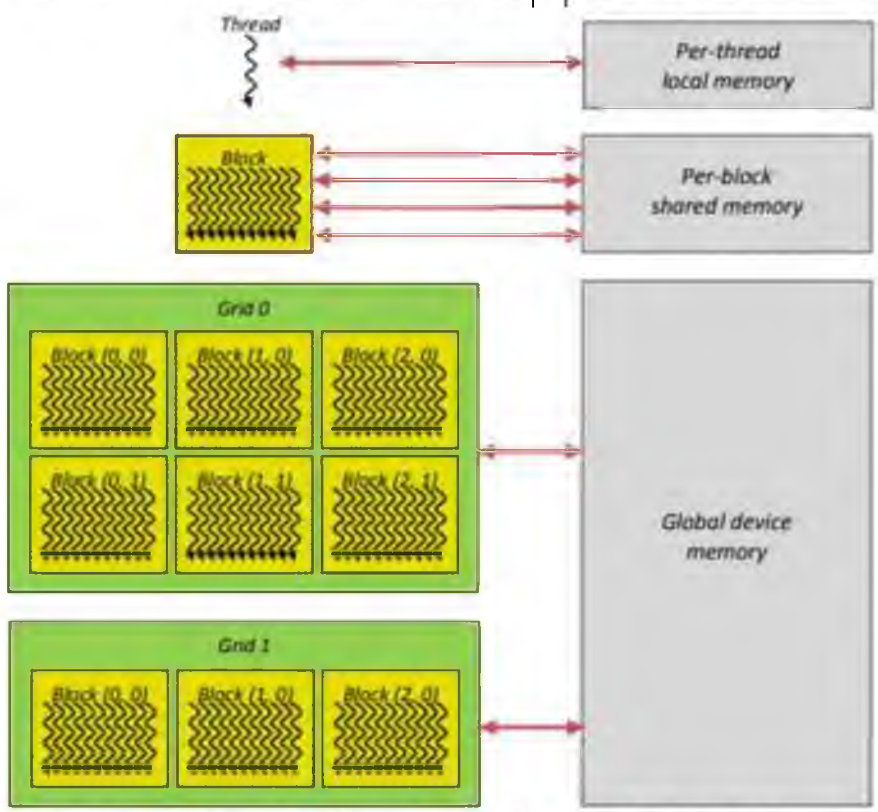


Рис. 2.3 Ієрархія пам'яті CUDA

Всім потокам також доступні два види загальної пам'яті для читання: константна (constant) і текстурна (texture). Як і в глобальній пам'яті пристрою, дані зберігаються протягом роботи програми. На відміну від інших типів пам'яті, текстурна пам'ять забезпечує різні режими адресації і підтримує фільтрацію для певних форматів даних. Фільтрація реалізована на апаратному рівні та може ефективно використовуватись у різних завданнях. Простір глобальної, константної та текстурної пам'яті оптимізовано для різних сценаріїв використання, які описані в [32] [33].

Вбудовані змінні. У додатку мовою CUDA C++ (у функціях, що виконуються на ГПУ) доступні такі вбудовані змінні:

• gridDim

Змінна типу dim3 містить поточну розмірність решітки;

• blockIdx

Змінна типу uint3 містить індекс блоку всередині решітки;

• blockDim

Змінна типу dim3 містить розмірність блоку потоків;

• threadIdx

Змінна типу uint3 містить індекс потоку всередині блоку;

• warpSize

Змінна типу int містить розмір варпа в кількості потоків.

Зазначені вбудовані змінні призначені тільки для читання і не можуть бути модифіковані програмою, що викликає.

Конфігурування виконання ядер.

Будь-який виклик функції зі специфікатором __global__ (ядра) повинен визначати конфігурацію виконання цього виклику. Конфігурація виконання визначає розмірність решітки і блоків, які будуть використовуватися для виконання функції на ГПУ. Конфігурація визначається за допомогою виразу спеціального виду <<>> між ім'ям функції та списком її аргументів, де:

• grid Змінна типу dim3, яка визначає розмірність та розмір сітки, так що grid.x × grid.y × grid.z дорівнює кількості блоків потоків, які будуть запущені (у ранніх версіях CUDA потрібно, щоб grid.z завжди дорівнювало 1) .

• block Змінна типу dim3, яка визначає розмірність і розмір кожного блоку потоків, block.x × block.y × block.z дорівнює кількості потоків на блок.

Таким чином, обов'язковими частинами конфігурації виконання є лише перші дві: кількість блоків та розмір блоку. Як приклад можна розглянути функцію зі специфікатором __global__ (ядро), оголошену так:

```
01 | __global__ void mean2 (float a, float b );
```

Нехай потрібно, щоб ця функція була виконана на двовимірній сітці блоків потоків розміру 5×5, при цьому кожен блок потоків мав розмір 4×4. Таким чином, загалом буде запущено $20 \times 20 = 400$ потоків (див. рис 2). Тоді для виклику ядра необхідно виконати наступний код:

```
dim3 grid ( 5, 5 );
dim3 block ( 4, 4 );
mean2<<< grid, block >>> ( a, b );
```

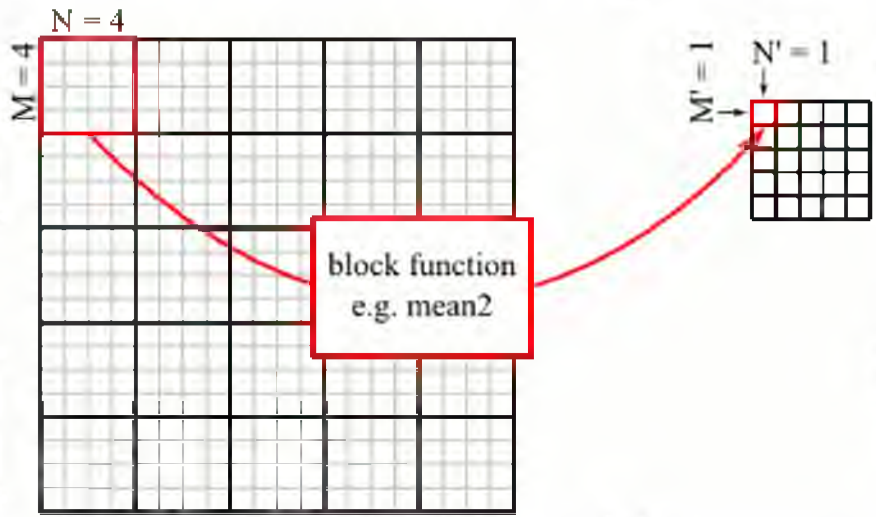


Рис. 2.4 Поділ даних на паралельні блоки

Аргументи зміни виконання обчислюються перед аргументами функції і передаються через загальну пам'ять на ГПУ.

Виклик функції може призвести до помилки програми, якщо аргументи grid і block не задовольняють обмежень, що накладаються на ГПУ, або size більше за максимальну кількість загальної пам'яті, доступної на ГПУ, за винятком обсягу спільної пам'яті, необхідної для статичного виділення, аргументів функції і конфігурації виконання.

Для бар'єрної синхронізації всіх потоків усередині одного блоку потоків використовується функція __syncthreads(). Функція для явної синхронізації потоків різних блоків у ході виконання ядра відсутня, цей ефект може бути досягнутий за допомогою атомарних операцій (проте, таке рішення, ймовірно, призведе до істотного зниження продуктивності).

2.2 Векторні обчислення

Вектори нагадують масиви тим, що містять кілька елементів одного типу. Але є дві важливі відмінності. По-перше, вектор даного типу може містити лише певну кількість елементів. По-друге, коли оперують вектором, усі елементи діють одночасно. За рахунок об'єднання і вирівнювання даних при використанні векторних типів, обробка даних масивів проходить швидше.

Приклад ядер програми складання масивів даних реалізованих на CUDA та різницю в часі виконання між векторними типами і звичайними показано в додатку А.

Порівняння векторних операцій CUDA з CPU.

За останні роки процесори досягли деяких фізичних обмежень і обмежень за потужністю. Оскільки вимоги до обчислень продовжують зростати, розробники ЦП вирішили цю проблему за допомогою трьох рішень:

• Додавання додаткових ядер. Таким чином, операційні системи можуть розподіляти запущені програми між різними ядрами. Крім того, програми можуть створювати кілька потоків, щоб максимізувати використання ядра.

• Додавання векторних операцій до кожного ядра. Це рішення дозволяє ЦП виконувати ті самі інструкції для вектора даних.

• Позачергове виконання кількох інструкцій. Сучасні процесори можуть виконувати до чотирьох інструкцій одночасно, якщо вони незалежні.

Використання векторних регістрів почалось в 1997 року з набору інструкцій MMX, який мав 80-розрядні регістри. Після цього були випущені набори інструкцій SSE (кілька їх версій, від SSE1 до SEE4.2), з 128-бітними регістрами. У 2011 році Intel випустила архітектуру Sandy Bridge з набором інструкцій AVX (256-розрядні регістри). У 2016 році був випущений перший процесор AVX-512 з 512-бітними регістрами (до 16х 32-бітних векторів з плаваючою чисельністю).

Для кращого розуміння реалізації векторних операцій на ЦП пояснимо їх архітектуру та як вони працюють на прикладі технологій SSE та AVX.

SSE і AVX мають по 16 регістрів. На SSE вони позначаються як XMM0-XMM15, а в AVX як YMM0-YMM15. Регістри XMM мають довжину 128 біт, тоді як YMM мають 256 біт.

SSE дає три визначення типів: __m128 , __m128d і __m128i (float, double і int відповідно).

AVX дає три визначення типів: __m256 , __m256d і __m256i. (float, double і int відповідно)

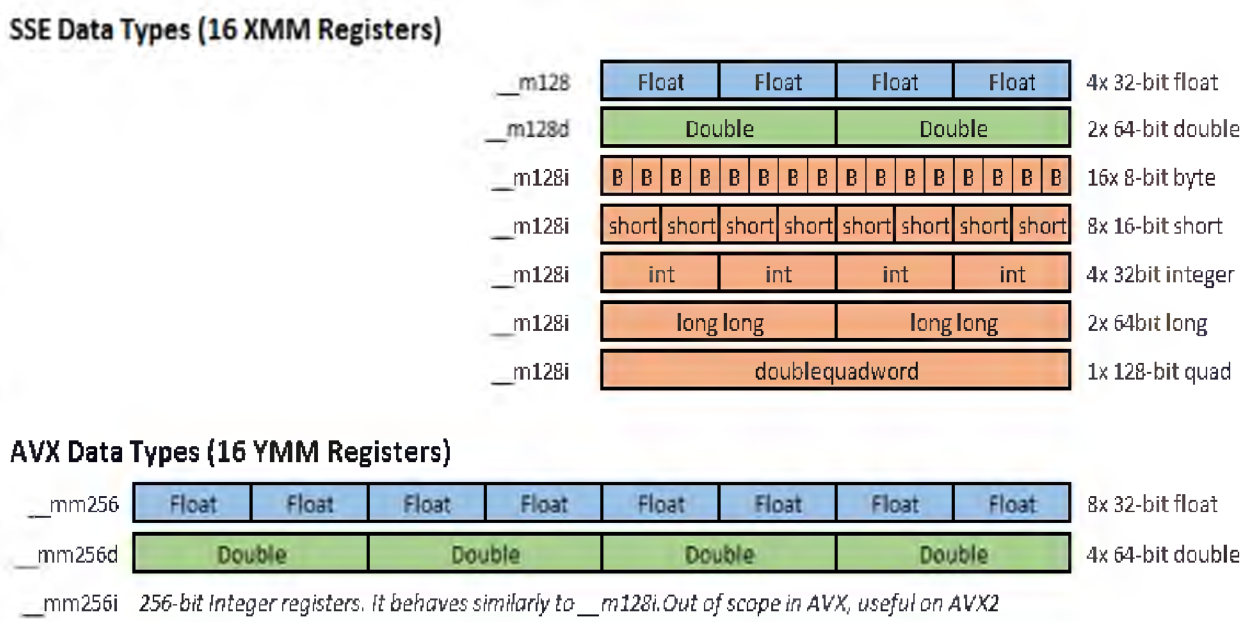


Рис. 2.1 Типи даних SSE і AVX

Так як типи даних з плаваючою комою (__m128, __m128d, __m256 і __m256d) мають лише один тип структури даних GCC надає доступ до компонентів даних у вигляді масиву. Тобто, такий синтаксис є вірним для компілятора:

```
__m256 vec = _mm256_set1_ps(6.665f); //Призначаємо вектору значення float
vec[0] = 2.22f;                      // Маємо доступ до першого значення
float f = (3.4f + vec[0]) * vec[7]; // GCC: «OK»
```

Виконання операцій на прикладі AVX

Коли виконується інструкція AVX, цей процес виглядає наступним чином:

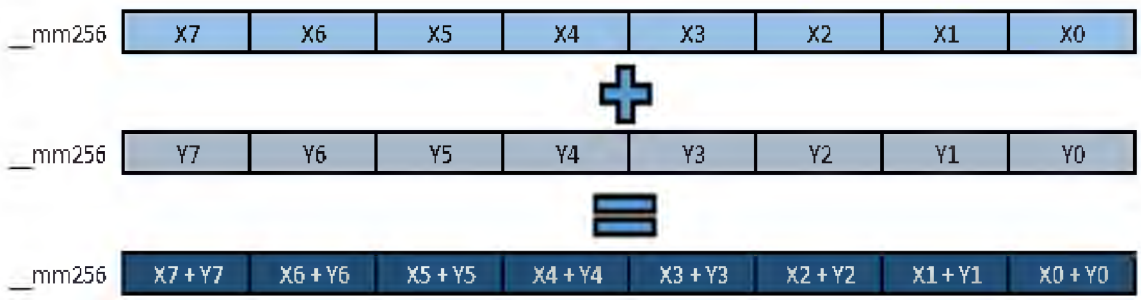


Рис. 2.2 AVX Операція складання

Усі операції виконуються одночасно. З точки зору продуктивності, вартість виконання одного додавання для float подібна до виконання VAdd на 8 float в AVX.

На відміну від CUDA інструкції AVX і SSE є інструкціями типу SIMD. Такі операції виконуються одночасно коли один 256-бітний регістр додається до іншого в тракті ЦП. Для отримання схожого результату нам необхідно 8 блоків FP32 або 4 FP64 виконуючий операцію разом над вектором даних. Такий тип операцій належить до типу SIMT (від англ. Single instruction, multiple threads)

Програмного код, який використовує векторні типи даних показано в Додатку А

РОЗДІЛ 3. АНАЛІЗ АЛГОРИТМІВ МАТРИЧНОГО МНОЖЕННЯ

В цьому розділі пояснюється аналіз, проведений для оцінки ефективності різних алгоритмів паралельного множення матриць на основі CUDA. Для здійснення повноцінного аналізу, було також розроблені алгоритми матричного множення на ГПУ, що використовують схожі або ті самі принципи. Для зниження похибки обчислення, кожний експеримент проводили щонайменше 10 разів і записували середні результати. Після отримання результуючої матриці, проводилась перевірка з базовим алгоритмом і обчислювалась похибка. Час виконання обчислення вимірюється у мілісекундах.

Інформація про систему на якій проводилися дослідження.

- CPU: AMD Ryzen 5 3500 3.60-4.1 GHz
- GPU: GeForce GTX 1660
- Оперативна пам'ять: 12 GB
- Операційна система: Windows 10 (64-bit)
- Інструмент розробки: Microsoft Visual Studio 2019

Рис. 3.1 Схема графічного ядра TU116

НУБіП України

НУБіП України

НУБіП України

НУБіП України

НУБіП України

НУБіП України

НУБіП України

Відеокарта NVIDIA GeForce GTX 1660 створена на основі 12 nm FinFET техпроцесу та заснована на графічному процесорі TU116-300 (TU116). Картка підтримує Directx 12 API. NVIDIA розмістила 6144 мегабайт оперативної пам'яті GDDR5, яка підключена із використанням 192-bit інтерфейсу.

Графічний процесор працює на частоті 1530 МГц, яку можна підвищити до 1785 МГц. Кількість ядер CUDA становить 1408, з швидкодією 4000 Мбіт/с та пропускною здатністю 96.0 Гбіт/

Таблиця 3.2

Характеристики графічного процесора NVIDIA GeForce GTX 1660 [16]

Найменування	**GeForce GTX 1660**
Ядро	TU116-300
Мікроархітектура	NVIDIA Turing
Техпроцес, нм	12
Базова\Динамічна частота GPU, МГц	1530–1785
Кількість SM	22
Кількість CUDA-ядер	1408
Кількість текстурних блоків	88
Кількість растрових блоків	48
Частота роботи пам'яті (DDR), МГц	2001 (8004)
Шина пам'яті	192-bit GDDR5
Об'єм пам'яті	6144
ПСП, ГБ/с	192,1
Shaders Model	6.4
Fill Rate, Mpix/s	85680
Fill Rate, Mtex/s	157100
DirectX	12 (12_1)
Інтерфейс	PCI-E 3.0

3.1 Поліноміальні і цілочисельні типи даних

3.1.1 BigInteger і дослідження його роботи.

Вбудовані примітивні числові типи не завжди можуть підходити для певної програми. Так, якщо нам необхідно розрахувати факторіал числа 100 в результаті обчислення ми отримаємо число розмірністю більше ніж 150 цифр. BigInteger використовується, коли необхідно зберігати та використовувати в програмі дуже великі числа, які виходять за межі допустимих значень для типів long і double.

Можна виділити такі основні ключові моменти створення бібліотеки довгої арифметики на GPU:

- швидкість доступу до пам'яті;
- паралельна реалізація алгоритмів простих арифметичних операцій;
- синхронізація між блоками паралельних потоків.

Структура класу

Архітектура класу BigInteger показана на рис. 1

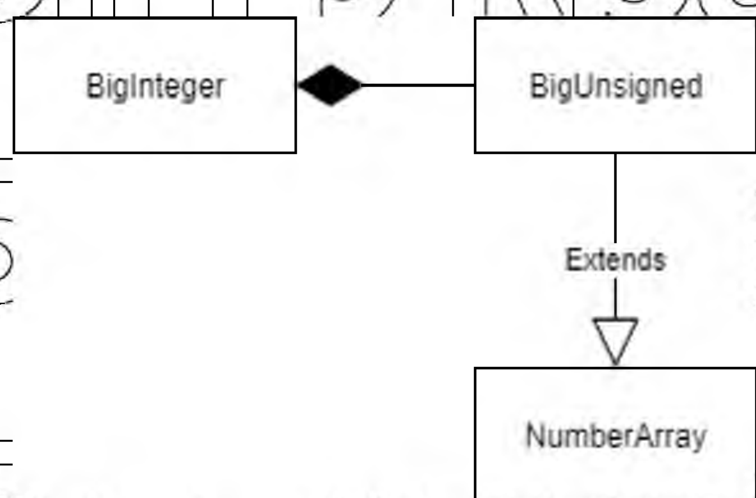


Рис 3.1 Залежності класу BigInteger

Конструктори класу створюють об'єкт класу BigInteger з рядка символів (знака числа та цифр) чи масиву байтів.

Нижче наведено основні конструктори.

BigInteger(const char* str)– об'єкт зберігатиме велике ціле число, задане рядком цифр, перед якими може стояти знак мінус;

BigInteger(const Blk* b, unsigned int blen)– конструктор, який копіює дані з заданого масиву блоків довжиною blen;

Таблиця 3.2

Основні методи класу

Методи	Опис
flipSign	Змінює знак на протилежний
getBlock getCapacity getLength getMagnitude getSign isZero	Група методів для отримання інформації (відповідно до внутрішньої структури)
add multiply negate subtract divideWithRemainder	Виконують основні арифметичні операції. Відповідно: додавання, множення, заперечення, поділ з залишком
toChars toInt toLong toShort	Група методів що повертають значення у відповідному представлені.

Після побудови прототипу ключового елементу класу бібліотеки довгої арифметики, було зроблено декілька тестів роботи класу на прикладі виконання матричного множення для цілочисельного типу даних і реалізованого.

Експериментальні дослідження. Основну увагу приділено дослідженню швидкості виконання на різних пристроях, за різних розмірів матриці, для визначення часової складності та прискорення щодо інших алгоритмів.

Таблиця 3.3

Час виконання класичного матричного множення CPU та GPU для типу Int32.

Розмір матриці	**AMD Ryzen 5 3500, мс**	**GeForce GTX 1660, мс**	**Пришвидшення, %**
16	0,0018	0,0403	4,37%
48	0,0370	0,0433	85,54%
64	0,0841	0,0422	199,62%
80	0,1617	0,0448	361,39%
144	0,9532	0,0594	1603,58%
176	1,7338	0,0721	2406,29%
208	2,8258	0,0848	3333,52%
272	8,3323	0,1413	5896,41%
304	11,4187	0,1770	6451,32%
336	15,7645	0,2092	7533,88%
400	26,3932	0,3268	8075,07%
432	33,6974	0,5590	6027,93%
464	41,6131	0,4752	8757,56%

Таблиця 3.4

Час виконання класичного матричного множення CPU та GPU для побудованого класу BigInteger

Розмір матриці	**AMD Ryzen 5 3500, мс**	**GeForce GTX 1660, мс**	**Пришвидшення, %**
10	0,02425	0,05456	44,45%
74	9,5068	1,28246	741,29%
138	59,663	4,91061	1214,98%
202	207,115	15,9062	1302,10%
266	433,924	45,4622	954,47%
330	822,25	90,0055	913,56%
394	1421,45	138,139	1029,00%

458	2280,51	251,012	908,53%

Як бачимо, GTX 1660 за рахунок більшої кількості CUDA ядер, пропускної здатності пам'яті та розміру пам'яті виграє за швидкістю виконання (рис. 3.3-3.4). Причому чим більший розмір матриці, тим помітнішою є різниця. BigInteger як тип даних, що реалізований програмно, дещо поступає типам даним підтриманим апаратно.

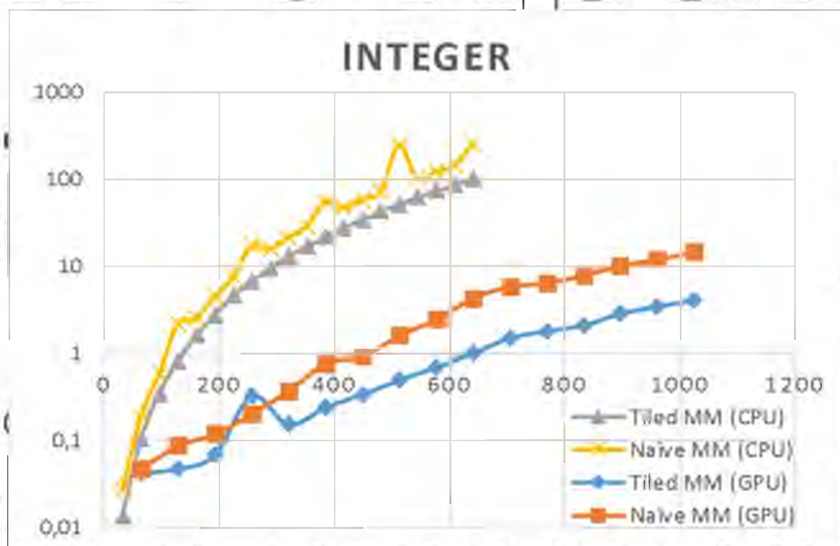


Рис. 3.1 Послідовна і паралельна реалізація алгоритму матричного множення для цілочисельного типу

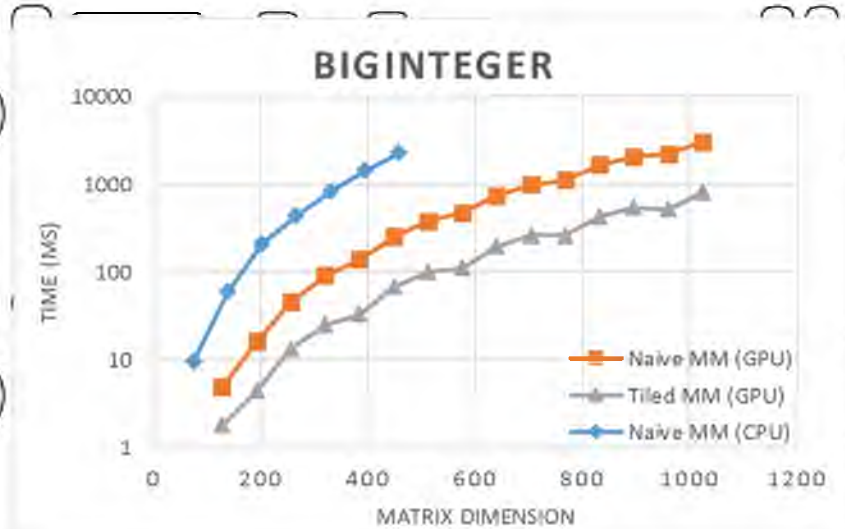


Рис. 3.2 Послідовна і паралельна реалізація алгоритму матричного множення для Big Integer

Через велику різницю у часі виконання (декілька порядків), графіки зображені у логарифмічному масштабі.

3.1.2 Polynomial і дослідження його роботи.

Загальна інформація про поліноми та їх використання

Застосування елементів теорії багаточленів можна зустріти у багатьох розділах вищої математики. Теорія багаточленів застосовується у курсах лінійної алгебри, математичного аналізу, теорії ймовірностей, методах наближених обчислень та інших розділах теоретичної та прикладної математики.

Поняття многочлена прийшло до нас з курсу математичного аналізу, в якому функція f(x) дійсної змінної x називається многочленом [34], якщо вона може бути представлена у вигляді:

$$f(x) \ = \ a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$$

де a_0, a_0, …,a_0 – дійсні числа (деякі з них і навіть усі можуть дорівнювати нулю).

Структура класу

Замість використовування ООП стилю програмування і створення повноцінного класу Polynomial, був використаний інший підхід – а саме ми створили окремі функції для роботи з структурою, яка містить лише дані в бінарному вигляді.

Було забезпечено підтримку таких основних типів даних як double, int, float.

Основні функції для роботи з Polynomial:

- Polynomial add(Polynomial other): повертає суму двох поліномов
- Polynomial subtract(Polynomial other): повертає різницю двох поліномов
- Polynomial multiply(Polynomial other): повертає добуток двох поліномов
- Polynomial divide(Polynomial other): повертає частку двох поліномов

Експериментальні дослідження.

У таблиці 3.6 наведено експериментальні результати множення матриці з типом даних Polynomial на GPU. Найбільші розмірні вектори, які підтримує

CUDA, є типами даних 4-dim (float4, double4, int4), тому поліноми встановлюються як R[x]/(x4 – 1) лише з R заміненим на float, double або int. Для Polynomial було реалізовано два алгоритми: наївний та блочний з використанням спільної пам'яті.

Таблиця 3.1

Часова характеристика алгоритмів матричного множення на CPU та GPU для кубічних поліномів з різними типами коефіцієнтів.

Типи даних	Integer		Float		Double	
Розмір матриці	Naïve MM, мс	Tiled MM, мс	Naïve MM, мс	Tiled MM, мс	Naïve MM, мс	Tiled MM
100	0,07	0,05	0,07	0,07	0,60	0,18
200	0,15	0,13	0,30	0,18	2,47	0,61
300	0,42	0,35	0,54	0,45	8,72	1,83
400	0,81	0,71	1,10	1,15	18,37	4,21
500	1,70	1,45	2,20	1,91	33,10	8,33
600	2,90	2,41	3,97	3,73	55,64	14,29
700	4,65	3,71	5,97	5,25	85,65	21,70
800	6,12	5,40	8,49	7,25	129,99	32,41
900	9,88	7,83	12,43	9,88	187,83	46,30
1000	12,07	10,26	12,75	10,95	258,06	63,23
1100	18,07	14,06	18,09	14,58	340,46	83,34
1200	20,66	17,57	21,31	18,25	437,37	107,44
1300	30,62	23,06	29,43	23,79	553,15	137,88
1400	35,21	28,02	34,05	29,15	687,23	172,20
1500	48,67	35,63	46,54	36,00	848,10	210,18
1600	49,15	42,36	49,33	46,32	1023,70	254,06
1700	69,84	53,09	71,29	52,78	1249,81	307,02
1800	74,39	60,78	78,15	60,76	1474,22	364,42
1900	100,78	72,74	94,15	73,74	1730,18	426,82
2000	111,49	82,26	101,72	85,40	2011,27	496,70

За рахунок використання спільної пам'яті в блочному алгоритмі для квадратної матриці розміром 2000 елементів ми отримуємо пришвидшення в 111/82 ≈ 35% для float і int. Для double пришвидшення досягає 2011/496 ≈ 405%. Також можна помітити, що час виконання задачі на double на порядок більше у порівнянні з іншими типами даних. Такі результати пов'язані з відсутністю у CUDA вбудованої підтримки такого типу даних (компілятор генерує підпрограму для обчислення таких даних використовуючи наявні FP блоки).

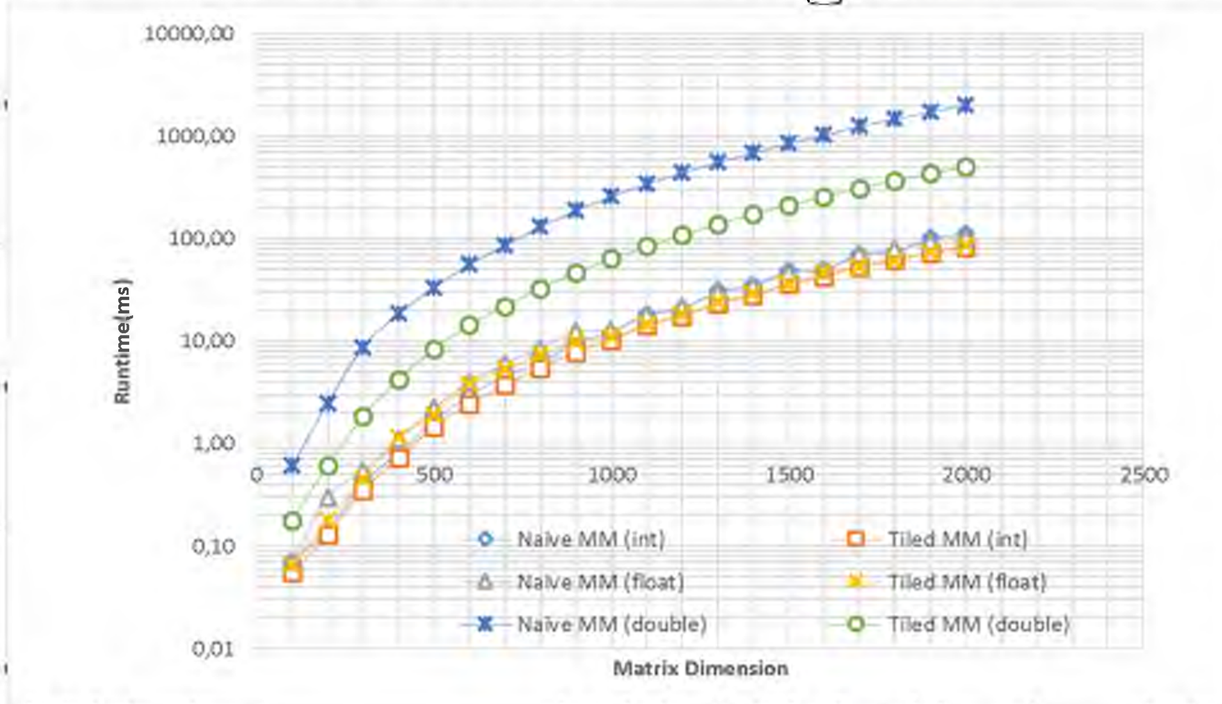


Рис. 3.1 Порівняння часу виконання паралельного алгоритму матричного множення для кубічних поліномів з різними типами коефіцієнтів.

Порівняння різних підходів в проектуванні типів даних на CUDA

Як функціональне програмування, так і об'єктно-орієнтоване програмування використовують різні методи зберігання та маніпулювання даними. У функціональному програмуванні дані не можуть зберігатися в об'єктах і їх можна трансформувати лише шляхом створення функцій. В об'єктно-орієнтованому програмуванні дані зберігаються в об'єктах.

BigInteger, що був побудований на принципах об'єктно-орієнтованого програмування, наслідує і всі риси присутні такому стилю. Наявність конструкторів і деструкторів, запитів на додаткову пам'ять, створення і контроль додаткових змінних - приводить до збільшення накладних витрат на само обчислення. Це особливо помітно, якщо ми порівнюємо час виконання матричного множення для int і BigInteger (див. табл. 3.6), де різниця досягає декількох порядків для одного алгоритму і однакових умовах роботи.

3.2 Оптимізація алгоритмів

В цьому розділі ми проаналізуємо різні оптимізаційні рішення, які були реалізовані на базі класичного алгоритму для матричного множення. Зіставимо методи і техніки оптимізації алгоритмів для різних обчислювальних пристроїв: графічного процесору і центрального процесору.

Базовим алгоритмом для порівняння ефективності була обрана наївна реалізація матричного множення (N). Так як оптимізація, як правило не впливає на асимптотичну складність, методи і техніки освітлені далі нижче цілковито придатні для використання і в інших алгоритмах.

Опишемо основні оптимізаційні стратегії, які були застосовані до базового алгоритму:

1) Tiling (T)

Основна ідея – змінити порядок обходу матриць для фасілітації використання кеш пам'яті за рахунок збільшення локальності даних(рис 3.6).

Рис. 3.1

2) Transposing (Tr)

Мета такої оптимізації – зменшити кількість кеш-промахів при зчитуванні строк матриці B за рахунок її транспонування (рис 3.7).

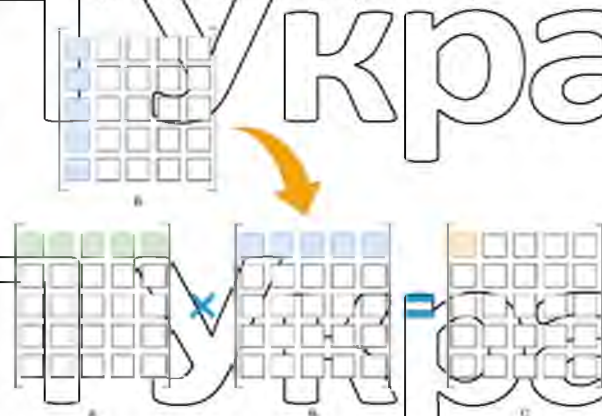


Рис. 3.2

3) Using shared memory (SM)

Перенесеення проміжних результатів в буфер з більшою пропускною здатністю і меншою латентністю для приховування затримки на зчитування пам’яті (рис. 3.8)

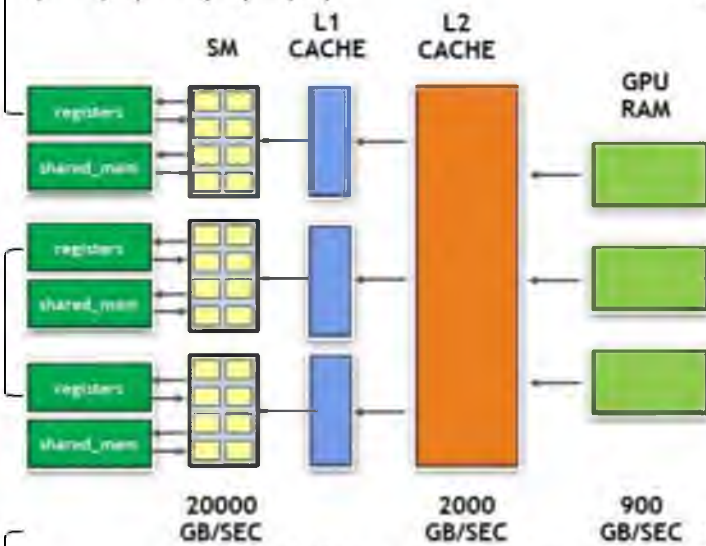


Рис. 3.3

4) Memory coalescing (MC)

Центральний процесор в сучасному комп’ютерному обладнанні найбільш ефективно виконують читання та запис в пам’ять, коли дані вирівняні певним чином. Наприклад, у 32-бітній архітектурі дані можуть бути вирівняні, якщо дані зберігаються в чотирьох послідовних байтах, а перший байт лежить на 4-байтовій межі. При такому розташуванні даних, замість зчитування одного елементу в одиницю часу, становиться можливо зчитування групи даних. За схожим принципом працює і графічний процесор (рис. 3.9)

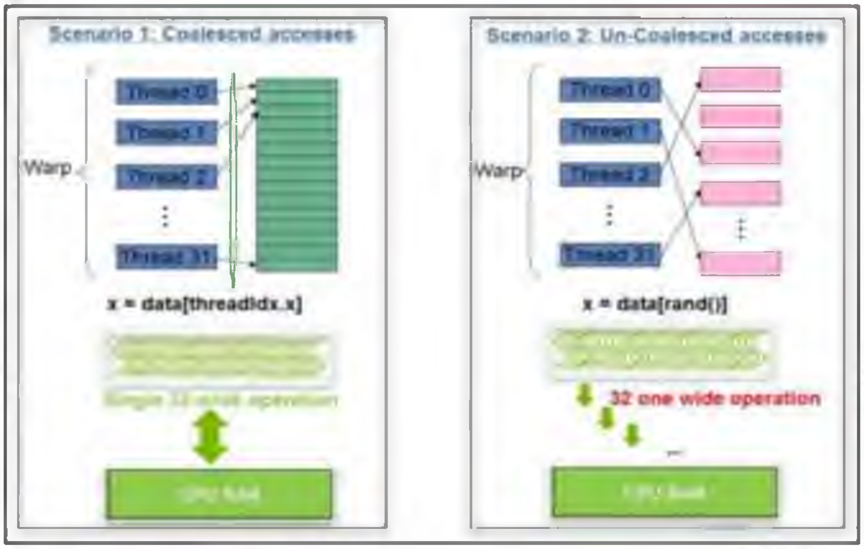


Рис. 3.4

5) Loop Unrolling (LU)

Метою розмотування циклу є збільшення швидкодії програми шляхом зменшення (або виключення) інструкцій контролю за циклом, таких як арифметика вказівників і перевірка на завершення циклу на кожній ітерації, зменшення витрат на галуження, а також «приховування латентності, особливо, затримку в читанні даних з пам’яті».

Приклад:

```
int i;
for ( i = 1; i < n; i++)
{
   a[i] = (i % b[i]);
}
```

Перетворюється в такий код:

```
int i;
for (i = 1; i < n - 3; i += 4)
{
   a[i] = (i % b[i]);
   a[i + 1] = ((i + 1) % b[i + 1]);
   a[i + 2] = ((i + 2) % b[i + 2]);
   a[i + 3] = ((i + 3) % b[i + 3]);
}
```

6) Loop Reordering (LR)

Зміна порядку обходу матриці для збільшення локальності даних. Приклад:

```
for (int i = 0; i < M; ++i) {
   for (int j = 0; j < M; ++j) {
      for (int k = 0; k < M; ++k) {
         C[j*M + i] += A[k*M + i] * B[j*M + k];
      }
   }
}
```

Перетворюється в такий код:

```
for (int j = 0; j < M; ++j) {
   for (int k = 0; k < M; ++k) {
      for (int i = 0; i < M; ++i) {
         C[j*M + i] += A[k*M + i] * B[j*M + k];
      }
   }
}
```

7) SIMD instruction (SI)

Проявляється у вигляді інтеграції до процесору спеціальних наборів інструкцій чи розширень команд (MMX, SSE, AVX тощо) для прискорення обробки певних видів обчислень (див. розділ 2.2)

8) Eliminating repeating calculations (ERC)

Винесення інваріантних циклів полягає у винесенні за межі циклів операцій, операнди яких не змінюються в процесі виконання циклу. Такі операції можуть бути виконані один раз до початку циклу, а отримані результати потім можуть використовуватися в тілі циклу.

Техніки перечисленні вище націлені на краще використання апаратних можливостей обчислювального приладу. В більшій степені вони ґрунтуються на архітектурі цільового пристрою. Враховуючи швидкодію сучасних центральних процесорів через вплив багатьох факторів (програмних та апаратних), утилізація всіх ресурсів обчислювального приладу зазвичай ускладнена.

3.2.2 Аналіз алгоритмів для однопоточної системи

Ми обчислюємо GFLOPS як основний показник для порівняння продуктивності різних реалізацій:

$$GFLOPS = \frac{2 \cdot m \cdot n \cdot k}{time\ (ms)} \cdot 10^{-6} \qquad (3.1)$$

Для цілочисельних типів даних формула остається та сама.

Експеримент для даних типу float

У таблиці 3.7 наведено експериментальні результати множення матриці з чисел з плаваючою комою на ЦП. Не зважаючи на однакову асимптотичну складність O(n), за рахунок використання тих чи інших стратегій оптимізації ми отримали пришвидшення у порівнянні з базовим «наївним» алгоритмом в десятки разів.

Час обчислення матричного множення (мс.) на центральному процесорі для даних типу float.

Таблиця 3.2

Розмір матриці	**N + Tr + ERC**	**N + T**	**N**	**N + ERC**	**N + ERC + LR**	**N + ERC + LR + T + SI**
16	0,002	0,003	0,002	0,002	0,001	0,001
32	0,013	0,023	0,016	0,013	0,004	0,004
48	0,052	0,066	0,062	0,050	0,013	0,009
64	0,134	0,172	0,153	0,135	0,027	0,020
80	0,265	0,364	0,320	0,244	0,050	0,035
96	0,450	0,617	0,591	0,438	0,085	0,047
112	0,789	1,034	0,986	0,771	0,133	0,118
128	1,247	2,116	2,120	2,097	0,192	0,153
144	1,809	2,094	2,058	1,779	0,272	0,204
160	2,513	2,807	2,874	2,509	0,370	0,284
176	3,419	3,775	3,833	3,397	0,488	0,294
192	4,457	5,004	4,991	4,501	0,626	0,503
208	5,752	6,396	6,383	5,687	0,817	0,491
224	7,287	7,998	8,001	7,277	0,994	0,616
240	9,077	9,794	9,876	9,006	1,242	0,963
256	11,240	16,903	16,338	16,203	1,501	1,516
272	13,315	14,244	14,512	13,256	1,748	1,450
288	15,949	16,721	17,416	16,098	2,094	1,793
304	18,859	19,678	20,339	18,851	2,458	2,017
320	22,239	23,648	23,777	22,133	2,939	1,741
336	25,869	27,058	27,441	25,660	3,342	2,777
352	29,664	31,083	31,900	29,882	3,932	3,238
368	34,149	35,430	36,965	34,218	4,475	3,625
384	38,938	56,770	54,762	54,997	5,345	3,952
400	43,919	45,675	47,083	44,661	5,731	4,956
416	49,770	50,996	53,237	50,738	6,361	4,441
432	56,147	57,291	60,602	56,964	7,080	5,971
448	62,484	65,964	66,822	63,556	7,876	6,172
464	69,761	71,427	73,989	70,643	8,804	6,423
480	77,459	79,589	82,887	78,887	9,678	7,295

496	85,297	88,053	90,726	87,069	10,656	7,644
512	95,572	236,939	412,968	369,279	11,915	7,789

Для наочності, ми зробили невеличку вибірку даних і перерахували час виконання за формулою 2.3 в стандартну одиницю (FLOPS) вимірювання швидкодії обчислювальних приладів для чисел з одинарною точністю.

Швидкодія в GFLOPS різних оптимізацій у порівнянні з базовим алгоритмом.

Таблиця 3.3

Розмір матриці	*N + Tr + ERC*	*N + T*	*N*	*N + ERC*	*N + ERC + LR*	*N + ERC + LR + T + SI*
128	3,36	1,98	1,98	2,00	21,85	27,39
256	2,99	1,99	2,05	2,07	22,35	22,14
512	2,81	0,94	0,65	0,73	22,53	34,46

В результаті оптимізації алгоритму ми отримали пришвидшення обчислення на ЦП в 10-30 разів відносно базового алгоритму (рис. 3.6). В більшій мірі, такий результат став можливим завдяки кращому використанню близької пам'яті (збільшення локальності даних) і усуненні частини холостих циклів процесора.

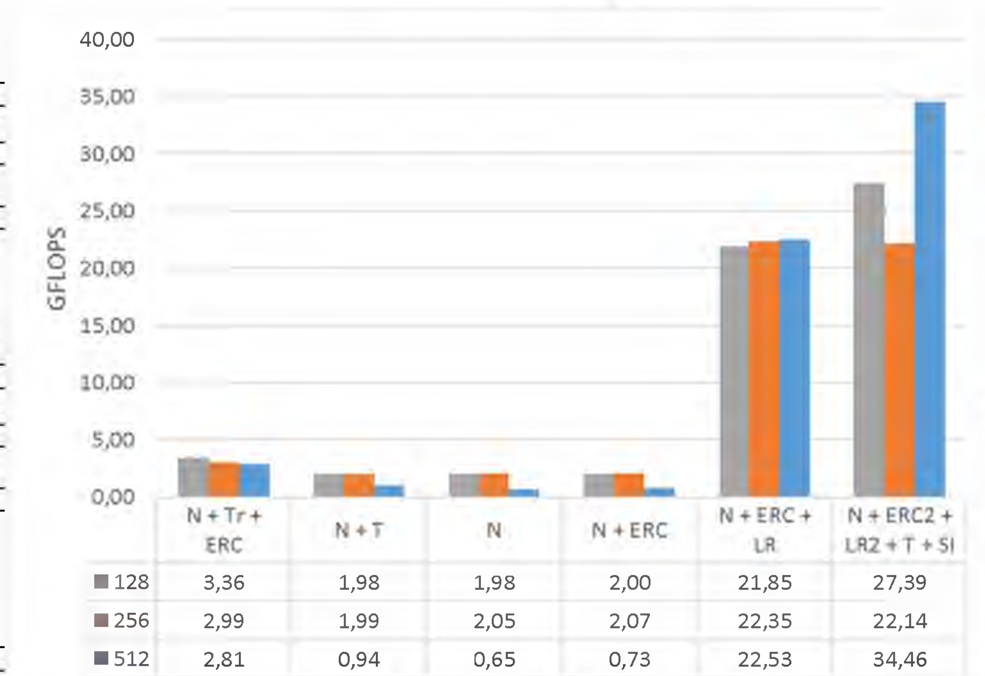


Рис. 3.1 Прискорення за рахунок використання оптимізаційних стратегій для AMD Ryzen 5 3500 (float)

НУБіП України

НУБіП України

НУБіП України

НУБіП України

НУБіП України

НУБіП України

НУБіП України

Аналогічне дослідження було проведено і для цілочисельного типу даних. Одним з переваг таких типів даних є те, що, за будь-яких умов, додавання числа 1 до числа 1 завжди буде дорівнювати 2 в електронній обчислювальній машині. Такі числа не накопичують похибки під час здійснення розрахунків і їх не потрібно додатково перевіряти. Нижче подані після оптимізації класичного алгоритму.

Час обчислення матричного множення (мс.) на центральному процесорі для даних типу int .

Таблиця 3.4

Розмір матриці	N + Tr + ERC	N + T	N	N + ERC	N + ERC + LR	N + ERC + LR + T + SI
16	0,002	0,005	0,004	0,002	0,001	0,001
32	0,014	0,026	0,025	0,012	0,004	0,003
48	0,046	0,085	0,081	0,037	0,014	0,007
64	0,106	0,192	0,180	0,087	0,034	0,022
80	0,202	0,362	0,361	0,185	0,064	0,035
96	0,367	0,594	0,569	0,299	0,112	0,090
112	0,606	0,934	0,911	0,453	0,176	0,114
128	0,835	2,126	2,151	2,281	0,262	0,132
144	1,182	2,052	1,908	0,942	0,373	0,192
160	1,755	2,871	2,641	1,576	0,512	0,282
176	2,145	3,865	3,433	1,746	0,684	0,357
192	2,974	5,077	4,371	3,875	0,885	0,515
208	3,881	6,399	5,763	3,165	1,136	0,642
224	4,700	7,686	7,139	4,280	1,411	0,723
240	5,859	9,224	8,543	4,591	1,732	0,904
256	7,114	17,585	16,759	17,265	2,139	1,332
272	8,048	13,659	12,703	8,119	2,523	1,442
288	9,643	16,146	15,518	12,429	3,047	1,610
304	11,029	19,331	17,487	11,330	3,490	2,075
320	12,929	23,311	21,034	17,222	4,114	2,100
336	14,996	25,291	23,715	15,143	4,761	2,740
352	17,199	28,818	28,396	23,197	5,640	3,291
368	19,625	32,719	31,388	20,264	6,233	3,692
384	22,338	56,957	55,125	57,831	7,288	4,905
400	25,042	42,700	40,668	26,422	8,083	4,869
416	28,595	48,725	49,527	39,128	9,074	5,438
432	31,419	54,579	51,243	33,896	10,211	5,980
448	35,147	63,584	57,968	47,259	11,349	7,077
464	38,998	67,059	65,741	41,979	12,507	6,811
480	43,165	75,578	72,052	59,483	14,007	7,791
496	47,379	83,749	79,603	54,199	15,406	9,052

512	53,072	172,55 8	253,00 2	255,569	16,745	8,768

Для цих даних, також було зроблено невеличку вибірку даних, яку було перераховано в стандартну одиницю вимірювання швидкодії обчислювальних приладів. Треба зазначити, що для int не існує власна стандартна одиниця вимірювання, як наприклад FLOPS для чисел з одинарною точністю. Тому, враховуючи, що довжина цілого числа в пам'яті для даної архітектури однакова, ми прийняли цю конвенцію в даній роботі для цілих чисел.

Швидкодія в GFLOPS різних оптимізацій у порівнянні з базовим алгоритмом.

Таблиця 3.5

Розмір матриці	***N + Tr + ERC***	***N + T***	***N***	***N + ERC***	***N + ERC + LR***	***N + ERC + LR + T + SI***
128	5,03	1,97	1,95	1,84	16,02	31,69
256	4,72	1,91	2,00	1,94	15,69	25,20
512	5,06	1,56	1,06	1,05	16,03	30,62

В результаті ми отримали схожі результати, що і з float, різниця в час для деяких оптимізацій пов'язана з апаратною підтримкою таких даних. Виконання операцій над числами з плаваючою точністю зазвичай потребує більше інструкцій порівняно з цілочисельними типами. Так, класичний алгоритм працює дещо повільніше з float ніж int.

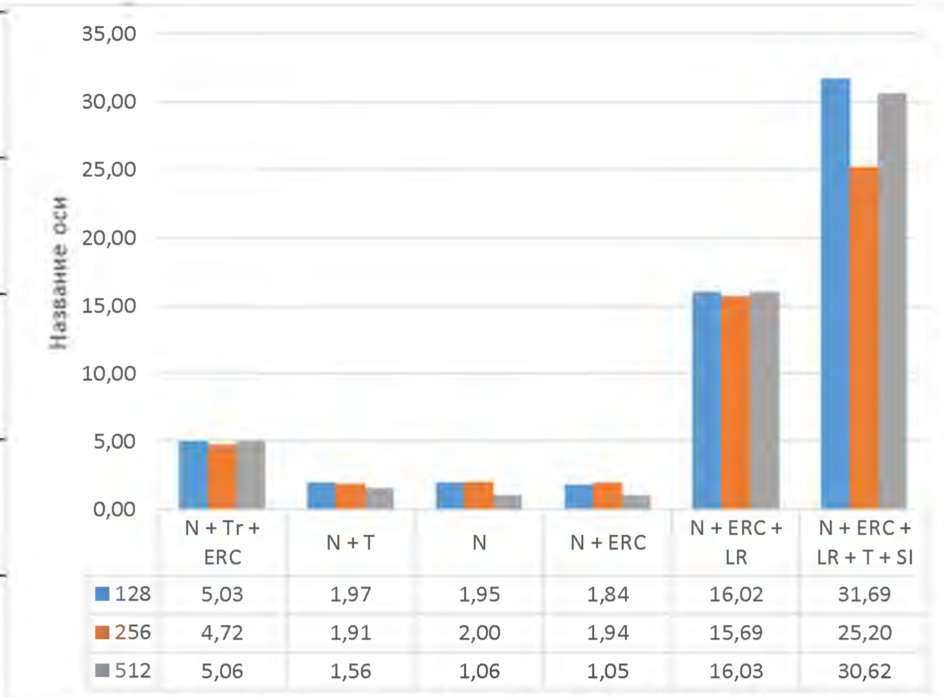


Рис. 3.1 Прискорення за рахунок використання оптимізаційних стратегій для AMD Ryzen 5 3500 (int)

3.2.3 Аналіз алгоритмів для многопоточної системи

GPU складається з глобальної пам'яті, кешу L1 і L2, спільної пам'яті та потокових мультипроцесорів (SM). Спільна пам'ять відіграє вирішальну роль у продуктивності графічного процесора. Таким чином, якщо ми розділимо A і B на блоки і зберігаємо результат множення цих блоків у спільній пам'яті, затримка на доступ до пам'яті зменшиться.

Час обчислення матричного множення (мс.) на графічному процесорі для даних типу int.

Таблиця 3.1

Розмір матриці	N	N + ERC	N + ERC + LR	N + ERC + LR + T + SI
64	0,047	0,042	0,042	0,041
128	0,087	0,054	0,046	0,038
192	0,117	0,076	0,070	0,056
256	0,201	0,110	0,111	0,067
320	0,368	0,198	0,165	0,097
384	0,756	0,297	0,257	0,134
448	0,911	0,422	0,434	0,239

512	1,600	0,605	0,519	0,249
576	2,428	0,872	0,721	0,328
640	4,271	1,237	1,392	0,439
704	5,804	1,935	1,347	0,556
768	6,409	1,773	1,589	0,581
832	7,791	2,320	1,875	0,696
896	10,046	2,802	2,214	0,850
960	11,989	3,769	3,075	1,031
1024	14,414	4,034	3,190	1,248
1088	16,403	4,966	4,074	2,156
1152	19,882	5,775	4,297	1,990
1216	22,809	6,721	5,478	2,229
1280	27,786	7,691	6,001	2,592
1344	30,692	8,969	7,063	3,043
1408	35,916	10,369	8,299	3,347
1472	41,296	12,158	9,025	3,967
1536	46,790	12,846	10,159	4,342
1600	52,659	15,209	11,402	5,185
1664	58,865	17,194	12,807	5,368
1728	66,426	18,309	14,511	5,982
1792	73,977	21,089	15,897	6,580
1856	81,638	23,281	18,273	7,268
1920	90,798	25,002	19,582	9,091
1984	99,913	27,919	22,179	10,110
2048	109,869	31,062	24,578	9,745

Для порівняння ми перерахували в GFLOPS значення часу виконання алгоритмів для довільно обраних розмірів матриць і представили їх в таблиці 3.12. Треба помітити, що виконання обчислень на графічному процесорі навіть для неоптимізованого рішення пришвидшує обчислення в сотні разів у порівнянні з центральним процесором. Такий ефект досягається за рахунок розпаралелювання послідовного алгоритму матричного множення серед тисячі CUDA ядер.

Швидкодія в GFLOPS різних оптимізацій у порівнянні з базовим алгоритмом.

Таблиця 3.2

Розмір матриці	*N*	*N + ERC*	*N + ERC + LR*	*N + ERC + LR + T + SI*
448	197,3	426,2	414,6	751,2
1152	153,8	529,5	711,6	1536,7
2048	156,4	553,1	699,0	1762,9

Так як архітектура графічного процесору дещо відрізняється від ЦП, відрізняється і спосіб оптимізації таких алгоритмів. В результаті ми змогли прискорити стандартний алгоритм в 10 разів (рис 3.8). Для матриць малого розміру ефект пришвидшення менш помітний. Це пояснюється недозавантаженням пристрою (кількості даних не вистачає для повного завантаження всіх паралельних ядер процесора).

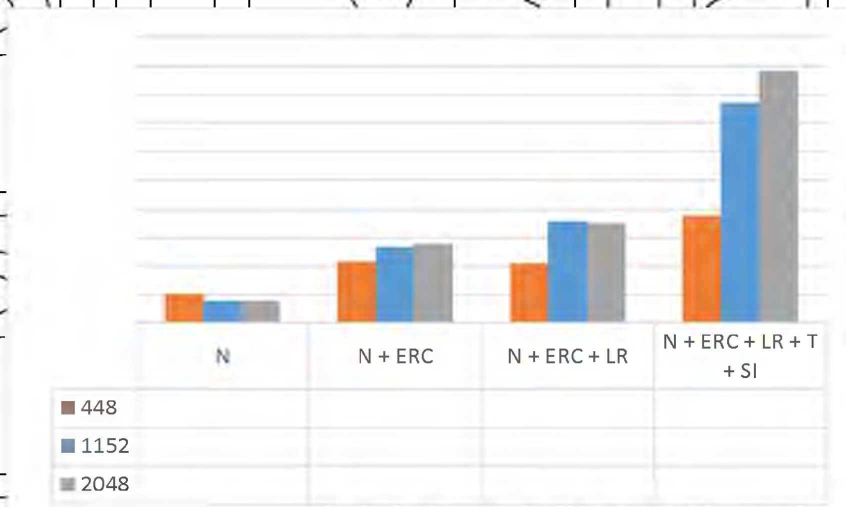


Рис. 3.1 Прискорення за рахунок використання оптимізаційних стратегій для GeForce GTX 1660 (int)

Ми провели схоже дослідження і для чисел з одинарною точністю. Треба зазначити, що для графічного процесора GeForce GTX 1660 кількість обчислювальних одиниць (блоків) FP32 для одного мультипроцесора дорівнює 64. Така сама і кількість блоків INT32 на 1 мультипроцесорі. Це значить, що за однакових умовах запуску, час виконання обчислень для таких типів даних повинен бути приблизно однаковим, що ми і бачимо в табл. 3.13.

Час обчислення матричного множення (мс.) на графічному процесорі для даних типу float.

Таблиця 3.3

Розмір матриці	N	N + ERC	N + ERC + LR	N + ERC + LR + T + SI
64	0,048	0,042	0,046	0,040
128	0,058	0,047	0,046	0,041
192	0,098	0,068	0,062	0,047
256	0,163	0,321	0,132	0,068

320	0,288	0,152	0,135	0,091
384	0,475	0,237	0,210	0,115
448	0,711	0,333	0,288	0,145
512	1,039	0,489	0,460	0,195
576	1,823	0,683	0,711	0,261
640	3,198	0,989	0,789	0,341
704	4,605	1,481	1,008	0,434
768	5,988	1,789	1,484	0,561
832	7,795	2,075	2,066	1,025
896	9,626	2,822	2,277	0,849
960	11,716	3,400	2,484	1,033
1024	13,863	3,990	3,202	1,251
1088	17,031	4,863	3,664	1,486
1152	19,925	5,679	4,419	2,089
1216	23,195	7,093	5,068	2,225
1280	27,194	7,734	5,905	2,588
1344	31,700	9,279	7,152	2,941
1408	36,347	10,824	8,222	3,333
1472	41,261	12,063	9,032	3,848
1536	46,604	13,004	10,404	4,594
1600	52,381	14,943	11,811	5,152
1664	58,809	17,413	12,742	5,842
1728	66,714	18,239	15,070	5,974
1792	73,430	21,033	16,382	6,654
1856	82,229	23,423	18,041	7,383
1920	90,896	25,916	19,735	8,633
1984	99,881	29,248	21,679	8,905
2048	110,760	30,535	24,556	10,523

В таблиці 3.14 ми бачимо, що для даних з меншим розміром ми отримуємо пришвидшення 1241\253 ~ 5 разів порівняно з неоптимізованим алгоритмом. Для даних з більшим розміром пришвидшення досягає 10 разів.

Швидкодія в GFLOPS різних оптимізацій у порівнянні з базовим алгоритмом.

Таблиця 3.4

Розмір матриці	**N**	**N + ERC**	**N + ERC + LR**	**N + ERC + LR + T + SI**
448	253,0	539,4	624,0	1241,3

1152	153,5	538,4	691,9	1464,0
2048	155,1	562,6	699,6	1632,7

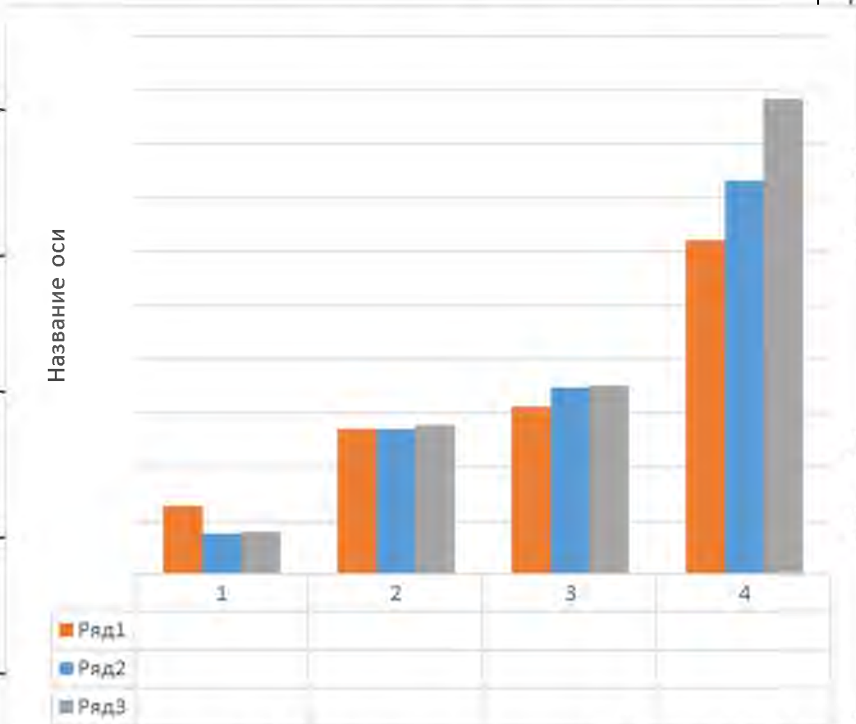


Рис. 3.1 Прискорення за рахунок використання оптимізаційних стратегій для GeForce GTX 1660 (float)

3.3 Загальний аналіз алгоритмів

В цій секції ми проаналізуємо роботу алгоритмів описаних в розділі 1. Алгоритми були побудовані з врахуванням архітектурних особливостей обчислювального приладу. Так, для графічних процесорів, реалізації алгоритмів матричного множення адаптовані під мультипоточність приладу та архітектурні особливості такої системи.

3.3.1 Аналіз складності алгоритмів для CPU

У програмуванні, обчислювальну складність алгоритмів зазвичай оцінюють за кількістю дій, які виконує алгоритм та за обсягом задіяної пам'яті. Хоча в секції 1.3 вже було теоретично обчислена складність для кожного з алгоритмів, ми проведемо емпіричний аналіз і зробим висновки щодо стабільності і швидкодії алгоритмів для різних розмірів даних на центральному процесорі AMD Ryzen 5 3500 побудованого на мікро архітектурі Zen 2. Дослідження проводиться застосовуючи лише одне ядро процесора, в межах одного потоку.

Алгоритми над якими було здійснено дослідження (див. розділ 1.4):

- Naïve MM (Наївний алгоритм)
- Tiled MM (Блочний алгоритм)
- Divide&Conquer (Блочно-рекурсивний алгоритм)
- Winograd-Strassen (Алгоритм Д'Альберто Вінограда-Штрассена)

В таблиці 3.15 представлені дані щодо роботи алгоритмів на центральному процесорі для різних типів даних, де головним параметром є час виконання алгоритму. Кожен експеримент було проведено щонайменше 10 разів після чого визначено середню.

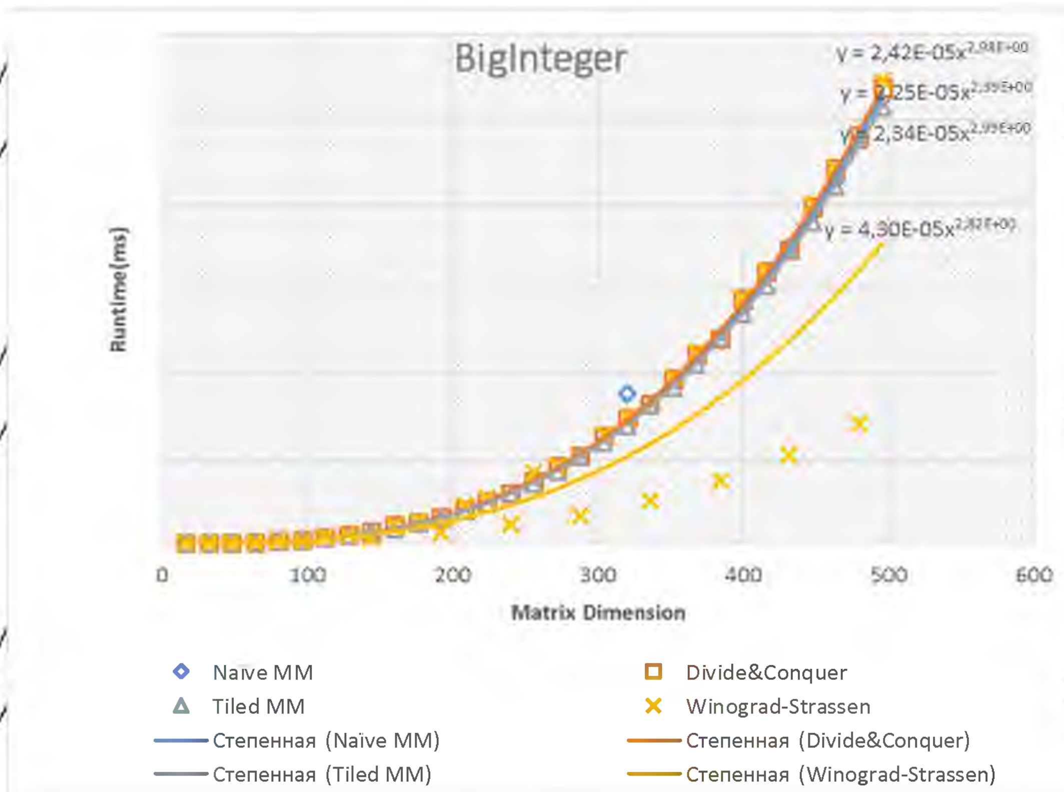
Таблиця 3.1

Тип даних	BigInteger				float				double				int			
Розмір матриці	**Naïve MM**	**Divide&Conquer**	**Tiled MM**	**Winograd-Strassen**	**Naïve MM**	**Divide&Conquer**	**Tiled MM**	**Winograd-Strassen**	**Naïve MM**	**Divide&Conquer**	**Tiled MM**	**Winograd-Strassen**	**Naïve MM**	**Divide&Conquer**	**Tiled MM**	**Winc -Stra**
16	0,096	0,095	0,090	0,104	0,002	0,002	0,002	0,002	0,002	0,002	0,002	0,002	0,003	0,004	0,005	0,0
32	0,75	0,74	0,68	0,69	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,03	0,
48	2,38	2,38	2,42	2,42	0,05	0,05	0,03	0,03	0,05	0,06	0,03	0,04	0,04	0,04	0,08	0,
64	5,66	6,04	5,65	5,51	0,12	0,12	0,08	0,08	0,14	0,14	0,09	0,08	0,09	0,09	0,17	0,
80	11,34	11,82	11,00	12,71	0,25	0,23	0,15	0,15	0,25	0,24	0,20	0,17	0,17	0,20	0,35	0,
96	19,08	19,71	19,47	11,96	0,46	0,43	0,25	0,24	0,46	0,52	0,26	0,26	0,30	0,33	0,55	0,
112	30,64	32,52	31,28	39,21	0,77	0,67	0,40	0,38	0,78	0,68	0,41	0,44	0,47	0,48	0,87	0,
128	48,62	49,07	46,87	52,94	2,08	1,05	0,59	0,56	2,08	1,77	0,64	0,59	2,24	0,86	1,15	1,
144	66,35	67,69	66,20	26,41	1,78	1,25	0,85	0,80	1,82	1,23	0,88	0,84	0,96	1,13	1,83	1,
160	90,83	104,22	86,62	104,66	2,50	1,87	1,19	1,05	2,55	1,91	1,20	1,11	1,36	1,52	2,44	2,
176	118,58	122,83	115,40	133,28	3,40	2,51	1,57	1,38	3,47	2,61	1,62	1,44	1,80	2,11	3,31	3,
192	150,32	152,76	152,45	59,99	4,53	3,54	2,10	1,78	7,20	3,50	2,11	2,12	3,80	2,56	4,19	4,
208	192,50	198,32	188,67	225,63	5,87	4,27	2,53	2,20	5,97	4,54	2,71	2,45	3,32	3,19	5,26	5,
224	241,05	248,29	236,77	270,08	7,35	5,36	3,21	2,78	7,48	5,79	3,43	3,02	4,15	3,93	6,56	6,
240	291,65	294,59	296,33	114,55	9,14	6,66	3,92	3,39	9,52	7,09	4,37	3,71	4,98	5,41	8,69	10
256	358,52	367,81	352,19	417,96	16,41	13,80	4,82	4,03	34,12	30,96	4,91	4,13	17,77	19,70	9,78	9,
272	432,24	453,05	424,11	453,61	13,33	9,23	5,78	5,19	13,41	9,17	5,97	5,18	9,42	8,45	11,99	11
288	505,86	511,87	512,07	160,68	16,04	10,13	6,73	5,64	16,28	9,90	6,95	5,73	12,74	9,31	14,41	13
304	605,22	632,23	592,17	644,52	19,08	13,12	7,98	6,92	19,00	13,08	8,15	7,01	12,00	11,97	22,06	16
320	880,84	735,61	690,07	746,44	22,36	14,98	9,34	7,54	32,11	14,67	9,90	7,84	17,77	12,14	19,01	18
336	812,21	819,85	816,66	252,69	25,66	18,00	10,72	9,23	26,24	17,84	11,05	9,05	15,44	14,73	21,74	20
352	925,89	971,08	916,90	979,62	29,92	20,47	12,25	9,87	29,81	19,87	12,93	10,25	23,85	16,11	25,09	23
368	1066,2	1115,88	1054,15	1126,22	34,40	24,00	14,00	11,66	34,28	23,81	14,40	11,79	19,58	17,92	27,47	26
384	1222,8	1207,74	1213,24	367,91	54,77	26,87	16,06	12,63	67,51	34,50	16,27	13,00	57,22	20,30	31,22	29
400	1391,9	1437,84	1352,24	1442,96	44,63	30,92	18,10	14,91	45,00	31,58	19,49	15,07	26,77	23,13	36,42	32
416	1593,9	1598,67	1517,40	1609,90	50,60	34,63	20,43	15,88	50,89	34,45	20,99	16,24	38,46	26,67	44,88	37
432	1745,2	1730,08	1732,80	518,21	56,47	39,11	22,74	18,33	56,93	39,57	23,54	18,54	34,54	29,23	50,06	39
448	1942,4	1983,80	1888,65	2002,54	63,39	43,14	25,52	19,73	87,80	43,98	26,85	20,47	47,44	31,27	51,62	44
464	2137,5	2207,85	2102,74	2209,46	70,68	48,29	28,40	22,41	71,26	49,63	29,40	23,51	42,85	34,98	57,52	49
480	2387,7	2403,64	2377,39	709,50	78,04	52,91	31,40	24,12	78,59	53,62	32,59	24,76	59,80	38,65	64,36	55
496	2679,5	2684,47	2573,95	2724,56	86,77	59,32	34,58	27,44	87,14	60,23	35,60	28,32	54,47	42,24	67,72	58
512					372,14	249,26	37,94	29,23	761,00	485,34	39,98	30,29	223,00	223,60	74,13	64
528					105,24	67,78	41,78	33,97	105,44	65,73	43,13	34,33	66,87	52,03	83,45	76
544					115,01	74,42	45,92	36,77	115,03	75,21	48,39	38,46	85,54	60,90	91,84	82
560					126,27	81,50	49,56	39,89	126,39	81,02	51,81	40,85	90,58	62,26	100,22	92
576					136,28	82,11	53,76	41,47	191,54	79,65	56,11	42,28	100,37	72,65	109,67	96
592					149,91	97,36	59,05	46,22	150,02	97,37	60,72	47,50	112,52	72,88	118,76	106
608					162,31	108,22	63,69	49,60	161,70	106,76	65,53	51,06	119,19	84,07	128,06	112
624					178,01	114,75	68,76	54,36	175,97	113,96	70,61	55,33	132,81	83,75	135,18	120

640	272,70	123,64	77,99	58,09	316,74	133,16	75,44	56,39	280,55	99,87	146,49	13[illegible]

На рис 3.14 зображено час виконання для кожного алгоритму відносно певного розміру матриці. Також, за методом найменших квадратів було обраховано коефіцієнти ступеневої функції. Отриману функцію розмістили поруч з алгоритмом, до якого вона відноситься. Використання різних алгоритмів майже не впливає на час виконання на протестованому проміжку. Це пояснюється складністю внутрішньої структури класу (тип даних реалізован програмно), та додатковим затратам на роботу з таким типом для різних алгоритмів.

Рис. 3.1 Визначення асимптотичної складності на основі емпіричних даних



множення матриць використовуючи різні алгоритми на AMD Ryzen 5 3500 для BigInteger

Для простих типів даних різниця в роботі різних алгоритмів більш очевидна (див. табл. 3.15). Так, на рис. 3.15 для чисел з одинарною точністю можна побачити, що найбільш ефективною є блочна реалізація, де пришвидшення відносно наївного алгоритма для матриці розміром 640 досягає 272/78 ~ 348%, а для алгоритму Вінограда-Штрассена – 468%. Блочно-рекурсивний алгоритм, не зважаючи на однакову асимптотичну складність з наївним алгоритмом, за рахунок збереження проміжних результатів в швидкої пам'яті дає пришвидшення порядка 50-150%.

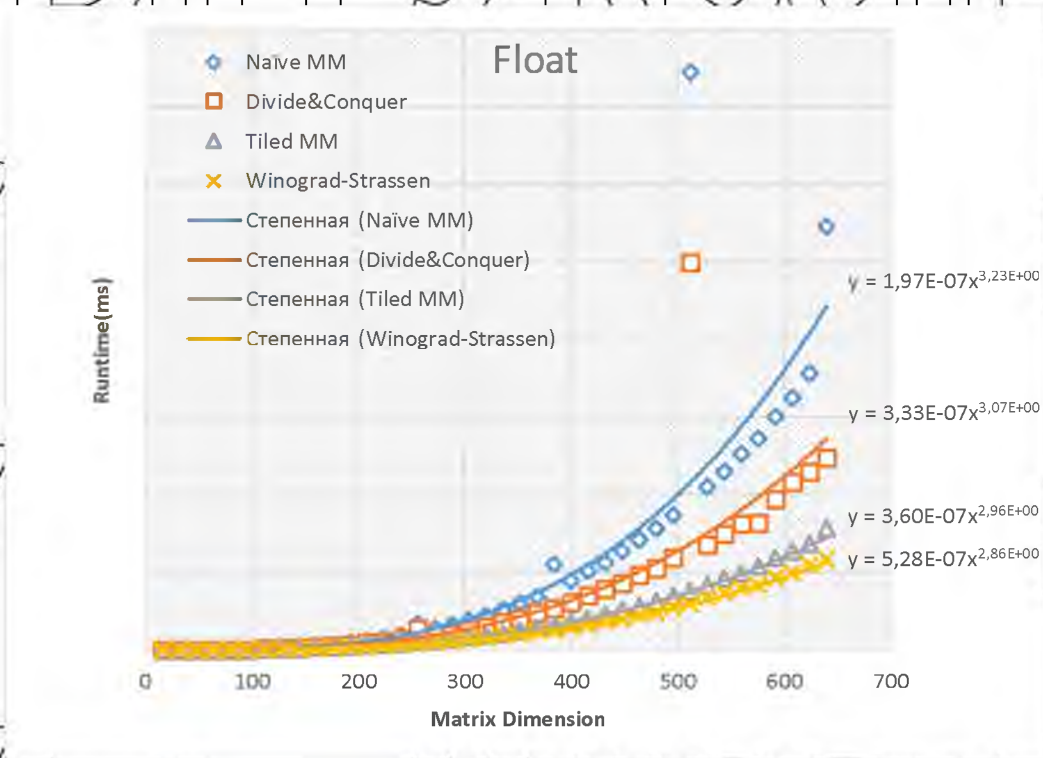


Рис. 3.2 Визначення асимптотичної складності на основі емпіричних даних множення матриць використовуючи різні алгоритми на AMD Ryzen 5 3500 для float

Схожі результати ми отримуємо і для чисел з подвійною точністю (табл. 3.15). Пришвидшення на матриці розміром 640 для блочно-рекурсивного алгоритму досягає 316/133=237%, для блочного алгоритму – 421%, і для алгоритму Вінограда-Штрассена – 564%. Треба зазначити, що обчислення даних типу double зазвичай потребує більшого часу у порівнянні з даними типу float. Проте, не зважаючи на архітектурні особливості обробки таких даних, час на обрахунок матричного множення даних типу double залишився на одному рівні з float на досліжуємої машині.

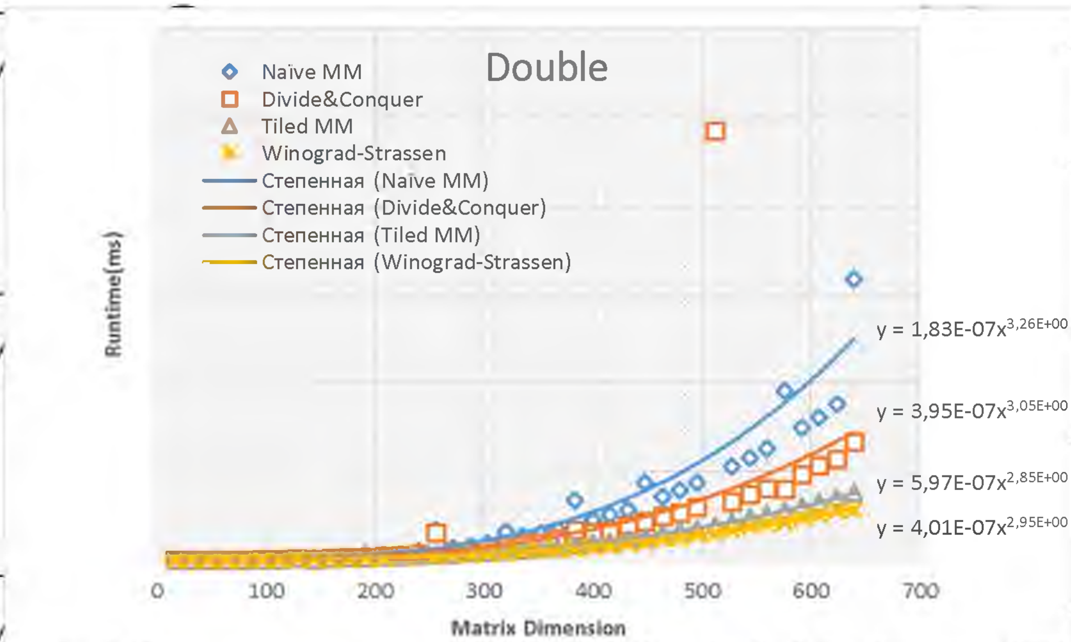


Рис. 3.3 Визначення асимптотичної складності на основі емпіричних даних множення матриць використовуючи різні алгоритми на AMD Ryzen 5 3500 для double

З рис 3.17, можна сказати, що найбільш ефективним є блочно-рекурсивний алгоритм для даних типу int. Так, для матриці розміром 640 ми отримуємо пришвидшення 280/100 = 280%.

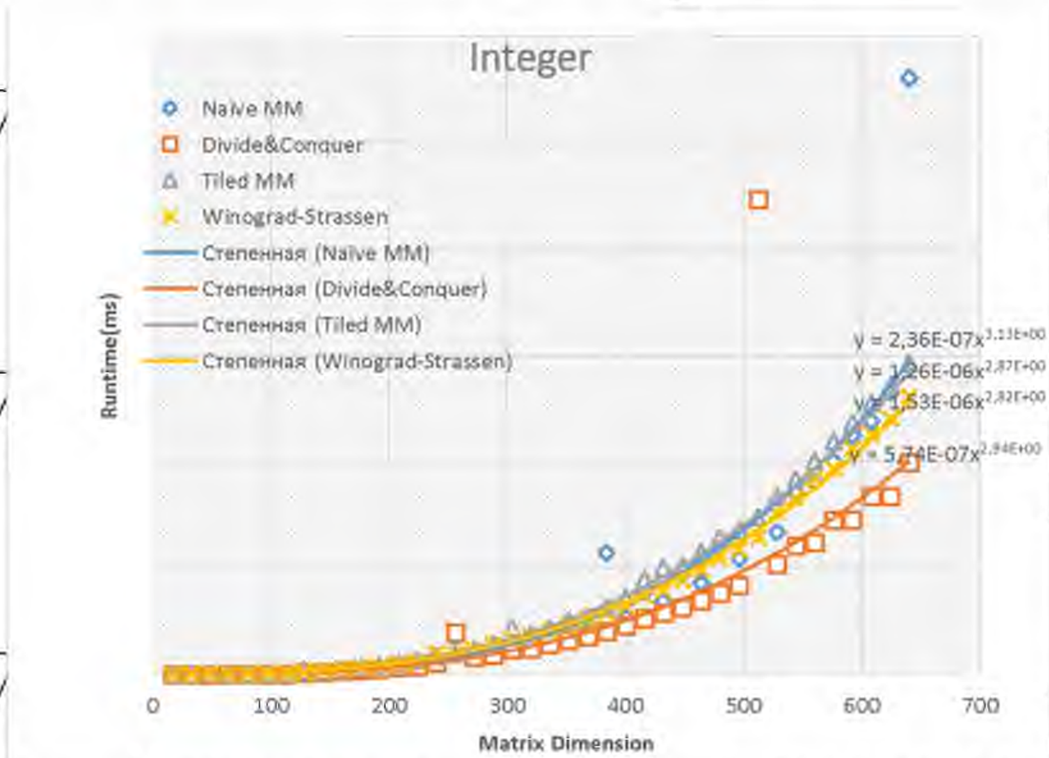


Рис. 3.4 Визначення асимптотичної складності на основі емпіричних даних множення матриць використовуючи різні алгоритми на AMD Ryzen 5 3500 для integer

Висновок. Для простих даних алгоритм Вінограда-Штрассена у більшості випадків показує свою ефективність порівняно з іншими алгоритмами за рахунок використання меншої кількості операцій на обчислення добутку двох матриць. Блочний алгоритм, що має таку ж саме асимптотичну складність що і наївний, і який ефективніше використовує швидку пам'ять, дещо поступає алгоритму Вінограда-Штрассена. Наступним за часом виконання йде блочно-рекурсивний алгоритм (теоретична складність якого за майстер-теоремою дорівнює складності блочного та наївного алгоритмів), практична складність якого в загалом краще ніж теоретично зазначеного. Не зважаючи на наявні рекурсивні виклики, розбиття матриці на підматриці дозволяє процесору краще використовувати доступну кеш пам'ять. За необхідністю цей алгоритм можна пристосувати під багатопоточну систему, отримавши приріст в швидкості обчислення пропорційний задіяним ядрам процесора.

3.3.2 **Аналіз складності алгоритмів для GPU**

В таблиці 3.16 представлені дані щодо роботи алгоритмів на графічному процесорі GeForce GTX 1660 побудованого на архітектурі TU116 для різних типів даних. Основним параметром є швидкодія, тому обчислення похибки для чисел з одинарною чи подвійною точністю не досліджується в цій роботі. Кожен експеримент було проведено щонайменше 10 разів після чого визначено середню.

Алгоритми, які використовуються для аналізу (див. розділ 1.4):

- Naïve MM (Наївний алгоритм)
- Optimized Naive MM (Оптимізований наївний алгоритм)
- Divide&Conquer (Блочно-рекурсивний алгоритм)
- Winograd-Strassen (Алгоритм Д'Альберто Вінограда-Штрассена)

- Cublas MM (Реалізований компанією NVIDIA, але тільки для чисел з одинарною і подвійною точністю)

Таблиця 3.1

Тип	BigInteger 128				float					double					int			
Розмір матриці	**Naïve MM**	**Divide&Conquer**	**Winograd-Strassen**	**Optimized Naïve MM**	**Naïve MM**	**Divide&Conquer**	**Cublas**	**Winograd-Strassen**	**Optimized Naïve MM**	**Naïve MM**	**Divide&Conquer**	**Cublas**	**Winograd-Strassen**	**Optimized Naïve MM**	**Naïve MM**	**Divide&Conquer**	**Winograd-Strassen**	**Optimi ed Naïv MM**
128	4,91	1,81	1,70	1,85	0,05	0,05	0,05	0,04	0,04	0,08	0,07	0,08	0,07	0,07	0,05	0,04	0,04	0,05
192	15,91	4,42	4,65	4,55	0,08	0,06	0,05	0,06	0,06	0,20	0,14	0,14	0,14	0,14	0,07	0,05	0,05	0,05
256	45,46	13,22	12,89	13,13	0,11	0,07	0,07	0,07	0,07	0,25	0,24	0,33	0,24	0,24	0,09	0,06	0,06	0,06
320	90,01	25,01	24,68	25,01	0,19	0,12	0,10	0,13	0,10	0,47	0,46	0,74	0,46	0,57	0,15	0,13	0,08	0,08
384	138,14	33,05	33,09	33,32	0,30	0,13	0,13	0,13	0,13	0,78	0,85	0,87	0,75	0,74	0,23	0,11	0,13	0,15
448	251,01	67,75	67,13	67,45	0,43	0,18	0,12	0,18	0,18	1,14	1,10	1,52	1,09	1,26	0,35	0,14	0,14	0,14
512	375,52	100,19	100,20	100,22	0,87	0,25	0,14	0,25	0,25	1,94	1,80	1,66	1,84	1,65	0,61	0,20	0,19	0,20
576	469,43	110,89	111,04	110,68	1,10	0,34	0,45	0,34	0,33	2,51	2,34	2,66	2,44	2,46	0,68	0,28	0,26	0,26
640	738,60	195,11	194,89	195,00	1,25	0,67	0,26	0,45	0,44	3,49	3,36	3,53	3,37	3,40	1,00	0,41	0,35	0,34
704	984,48	259,33	259,47	258,97	1,82	0,75	0,29	0,57	0,56	4,37	4,13	4,69	4,38	4,24	1,44	0,43	0,43	0,43
768	1117,2	260,80	261,32	260,71	2,10	0,73	0,69	0,72	0,93	5,98	5,60	6,72	5,60	5,49	1,71	0,63	0,58	0,57
832	1658,2	426,41	427,23	428,86	3,03	0,95	0,62	0,89	1,23	7,48	7,05	9,02	6,91	7,02	2,26	0,69	0,69	0,74
896	2032,4	535,78	535,37	535,59	4,25	1,48	0,64	0,97	1,32	9,09	8,74	8,68	8,59	8,69	2,77	0,86	1,03	0,86
960	2183,2	511,28	511,44	512,25	3,35	1,10	0,67	1,05	1,05	11,26	10,45	12,28	10,57	10,53	3,34	1,04	1,16	1,04
1024	3037,6	799,44	797,84	798,17	4,06	1,82	0,72	1,26	1,25	13,29	12,78	14,16	12,81	12,99	4,08	1,26	1,30	1,26
1088	-	-	-	-	4,80	2,24	0,87	2,16	1,49	15,73	16,95	16,22	15,42	15,32	4,78	2,17	2,10	1,67
1152	-	-	-	-	5,74	2,22	1,04	2,33	1,78	18,50	19,35	19,50	17,52	17,85	5,75	2,52	2,37	2,07
1216	-	-	-	-	6,80	2,75	1,30	2,83	2,09	21,98	22,61	21,98	20,95	21,03	6,50	2,64	2,76	2,07
1280	-	-	-	-	7,69	3,14	1,30	3,30	2,59	25,52	26,64	27,20	24,09	24,84	7,72	3,09	3,21	2,59
1344	-	-	-	-	9,16	3,56	1,39	3,55	2,78	29,84	30,68	31,04	27,98	28,86	8,91	3,39	3,57	2,85
1408	-	-	-	-	9,94	4,08	1,76	4,19	3,21	34,03	33,59	32,74	30,51	32,60	10,39	4,02	3,89	3,35
1472	-	-	-	-	11,59	4,65	2,17	4,65	3,96	39,21	39,83	40,39	35,98	38,04	11,76	4,81	4,60	3,81
1536	-	-	-	-	12,85	5,12	2,22	5,21	4,33	44,02	45,10	45,43	40,68	42,72	13,05	5,27	5,37	4,45
1600	-	-	-	-	14,73	5,71	2,42	5,73	4,96	49,80	50,25	50,53	45,03	48,32	14,54	5,68	5,73	4,80
1664	-	-	-	-	16,47	6,27	2,75	6,27	5,43	55,75	55,49	53,97	49,92	54,16	16,80	6,30	6,20	5,61
1728	-	-	-	-	18,30	6,95	2,15	6,74	6,08	62,22	63,28	61,81	56,61	60,82	18,50	7,13	6,89	6,06
1792	-	-	-	-	20,84	7,65	2,31	7,38	6,79	69,59	69,34	67,49	62,33	67,91	20,69	8,06	7,66	6,72
1856	-	-	-	-	22,62	8,60	2,82	8,14	7,26	77,56	77,45	75,89	69,28	74,74	23,01	8,42	8,43	7,65
1920	-	-	-	-	25,40	9,07	2,86	8,91	8,27	85,71	84,44	84,53	75,38	82,93	25,41	9,18	9,08	8,22
1984	-	-	-	-	27,80	9,87	4,71	9,69	8,89	94,46	94,97	97,28	85,07	91,81	27,85	10,13	10,17	9,30
2048	-	-	-	-	30,34	10,70	4,69	10,39	9,93	106,87	102,66	101,73	92,61	103,18	30,61	10,84	10,91	10,07
2176	-	-	-	-	36,74	16,99	6,67	16,52	11,61	126,51	137,28	121,20	112,16	121,10	36,87	16,96	16,39	11,77
2304	-	-	-	-	43,44	18,82	5,03	18,63	13,71	150,72	157,81	146,41	128,54	146,48	43,73	19,01	18,68	13,91
2432	-	-	-	-	51,39	21,75	5,85	21,05	16,17	176,88	184,69	168,99	148,81	167,73	51,46	21,99	21,57	16,34
2560	-	-	-	-	59,01	24,58	7,09	24,12	18,79	202,42	215,01	195,58	174,13	195,53	59,69	25,21	24,14	19,69
2688	-	-	-	-	68,39	27,87	8,40	26,21	22,12	234,21	248,01	227,90	199,90	226,85	68,94	28,42	26,43	22,45
2816	-	-	-	-	78,72	32,11	9,15	30,04	25,31	269,31	271,91	259,47	219,24	259,75	79,13	33,35	30,83	25,70
2944	-	-	-	-	90,13	37,57	10,83	34,76	28,68	307,78	321,71	298,98	258,36	297,43	90,90	38,04	35,43	29,27
3072	-	-	-	-	102,00	41,35	11,78	38,52	32,56	350,23	361,75	337,76	289,97	337,57	103,00	42,09	39,30	33,50
3200	-	-	-	-	115,87	45,86	13,19	42,18	36,88	396,11	403,51	381,60	322,82	381,68	116,28	46,77	42,80	37,41
3328	-	-	-	-	130,10	50,79	14,55	46,03	41,25	444,16	447,54	428,62	357,57	430,15	131,13	51,49	47,02	42,14
3456	-	-	-	-	145,65	56,68	74,22	51,94	46,29	497,49	508,80	485,55	405,48	481,58	146,53	58,18	52,31	47,01
3584	-	-	-	-	162,59	61,50	18,30	56,91	51,73	555,05	558,27	536,70	444,61	537,18	163,62	63,02	57,29	53,05

3712	-	-	-	-	180,65	68,67	26,62	61,75	57,19	617,20	624,12	598,07	496,91	595,58	181,81	69,39	62,63	58,64
3840	-	-	-	-	200,55	74,69	22,37	67,19	63,38	684,56	678,80	658,60	538,03	660,67	200,98	75,32	67,97	64,97
3968	-	-	-	-	221,39	82,38	108,12	73,22	70,01	755,54	756,41	730,12	610,94	743,78	222,01	83,02	73,78	70,92
4096	-	-	-	-	243,56	89,48	26,06	79,16	77,17	843,00	828,74	807,33	655,79	802,66	244,02	89,71	80,03	78,27

На рис 3.18 зображено час виконання для кожного алгоритму відносно певного розміру матриці. Складність збереження даних для BigInteger впливає на час виконання обчислення. Різкий стрибок в часі виконання у порівнянні з наївним методом пояснюється лише ефективним розміщенням даних. В межах цього дослідження середній розмір такої структури в пам’яті дорівнює ~ 24 байти. Враховуючі, що кожний блок отримує підматрицю розміром 32 на 32, в 1 строчці розміщується ~ 24*32=768 байтів. А так, як середній розмір строки кеша дорівнює 64 байта, в 1 рядку підматриці розміщується 768/64 = 12 ліній кешу. Зчитування даних проходить швидше і з меншою затримкою, за рахунок чого і досягається таке пришвидшення. Складність типу впливає і на час виконання алгоритмів, ефективно сповільнюючи його на порядок, а то і на 2 порядкі у порівнянні з звичайними float, double чи int. Треба також врахувати, що для блочно-рекурсивного і Вінограда-Штрассена алгоритмів границя, при досягнені якої починається поділ матриці, находиться на розмірі 1024 (таке значення обґрунтовано архітектурною особливостю графічної карти і наявною кількістю об’єднанної памяті і було прийнято для всіх типів даних).

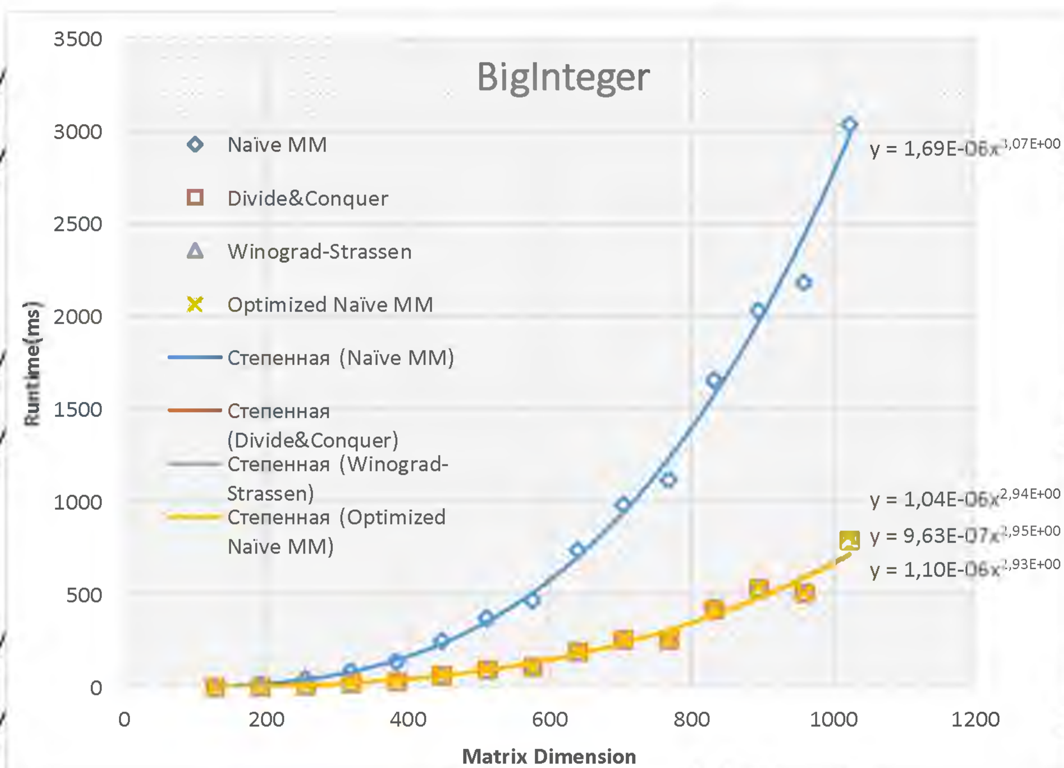


Рис. 3.1 Визначення асимптотичної складності на основі емпіричних даних множення матриць використовуючи різні алгоритми на GeForce GTX 1660 для BigInteger

Для простих типів даних, як для числа з одинарною точністю (рис 3.19) ми отримуємо більш очевидні результати. Для блочно-рекурсивного алгоритму на розмірі матриці 4096 на 4096 ми отримуємо пришвидшення 243/89 = 2.73 разів (273%). Більш ефективним показує себе алгоритм Вінограда-Штрассена – 243/79 = 307%.

Бібліотека Cublas також предоставляє API для виклику функцій обрахунку матричного множення, тому для порівняння ми включили їх для аналізу ефективності рішення. Для обчислення матричного множення з даними типу float була використана функція **cublasSgemm.** За замовчуванням, бібліотека використовує всі апаратні можливості графічної карти. Так, при обрахунку, крім звичайних ядер, використовуються додаткові ядра (тензорні ядра). Це потрібно враховувати при здійснені порівняльного аналізу. В цій роботі не використовуються тензорні ядра, т.я. це спеціалізована версія ядра, архітектура якої націлена переважно на дані типу float (хоча тензорні ядра також вміють працювати і з даними типа char, або так званими int8).

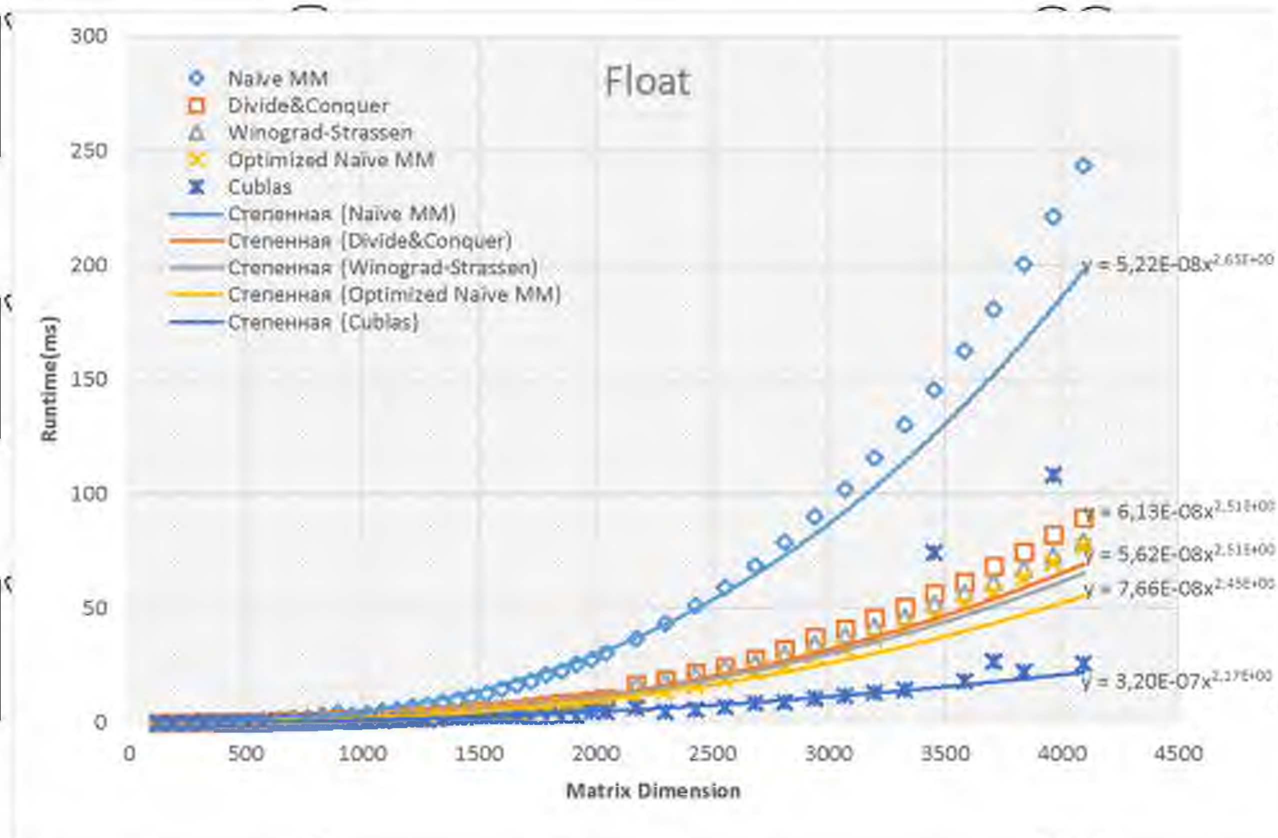


Рис. 3.2 Визначення асимптотичної складності на основі емпіричних даних множення матриць використовуючи різні алгоритми на GeForce GTX 1660 для float

Обчислення матричного множення для з подвійною точністю виявилось найбільш неефективним. В табл 3.16 помітно, що на виконання матричного множення матриць розміром 4096 на 4096 з данимми типу double за наївним алгоритмом потрібно в 843/243 = 3.46 разів більше часу в порівнянні з даними типу float. Такий результат пов’язаний з тим, що внутрішньо такі дані емулюються компілятором використовуючі float. Так, коли компілятор бачить операцію здійснену над double, він розгортає і перетворює її в інструкції, які працюють з float. Оптимізація таких типів ускладнюється наявністю додаткового шару перетворень, що також знижує і ефективність роботи з такими типами в цілому.

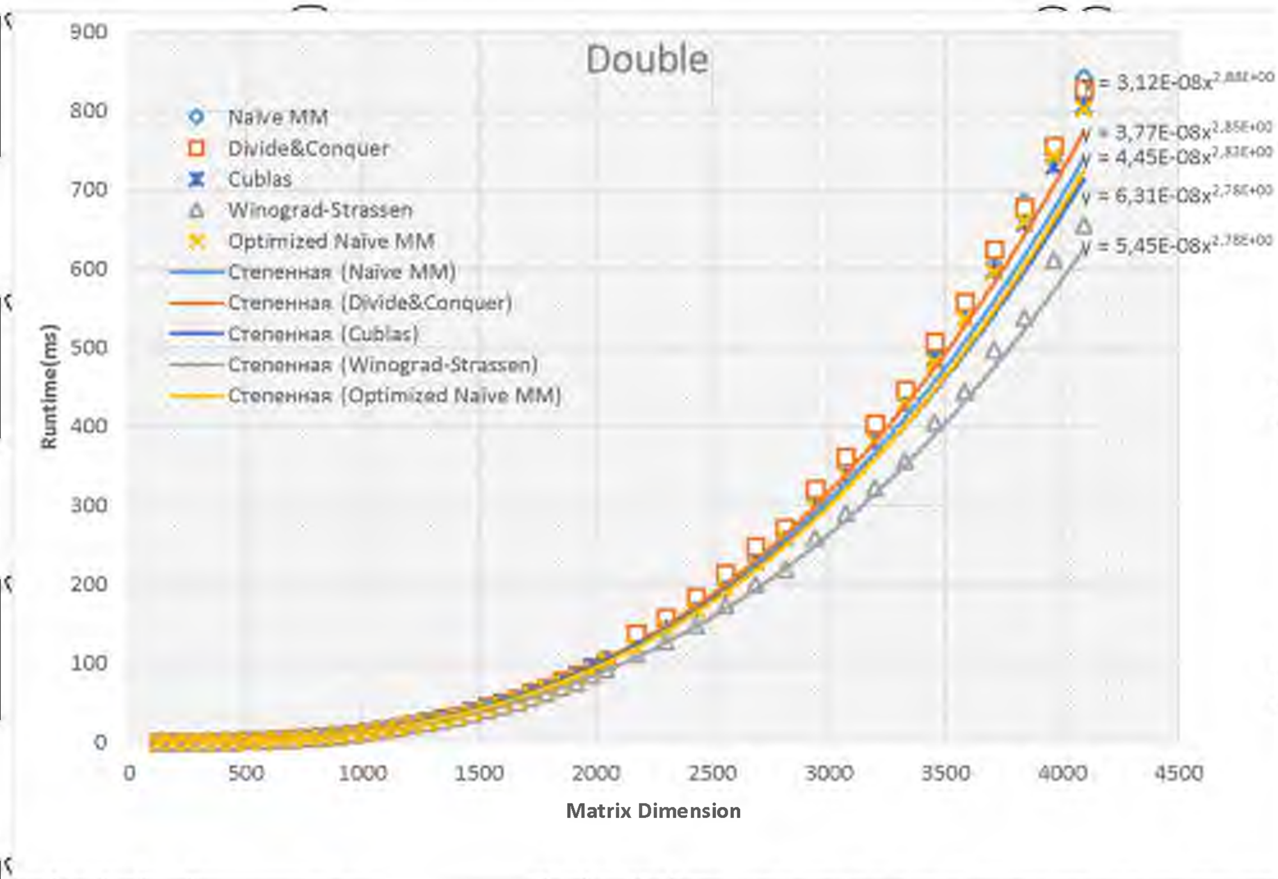


Рис. 3.3 Визначення асимптотичної складності на основі емпіричних даних множення матриць використовуючи різні алгоритми на GeForce GTX 1660 для double

З рис. 3.20 можна побачити, що найбільш ефективним виявився алгоритм Вінограда-Штрассена. Для матриці розміром 4096 ми отримали пришвидшення на 843/655 = 128% у порівнянні з наївним алгоритмом в той час, як реалізація Cublas (для роботи з double використовується функція cublasDgemm) майже не відрізняється від наївного алгоритму (коефіцієнт пришвидшення дорівнює лише 843/807 = 1,04).

Для графічної карти GeForce GTX 1660 ефективність обчислення цілочисельних даних не поступає даним типу float (табл. 3.16). Потрібно також зазначити, що обчислення матричного множення за алгоритмом Вінограда-Штрассена для даного типу усуває один з недоліків такого алгоритму – більша похибка у порівнянні з класичним алгоритмом, тому використання такого алгоритму особливо ефективно для цілочисельних даних.

Найбільш ефективним алгоритмом (рис 3.21) показав себе оптимізована версія наївного алгоритму, так, для матриці 4096 на 4096 досягається пришвидшення 244/78 = 312%. Наступним йде алгоритм Вінограда-Штрассена, де ми отримуємо 244/80 = 305% відносно наївного алгоритму. Останнім йде блочно-рекурсивний алгоритм з 244/89 = 274% пришвидшенням.

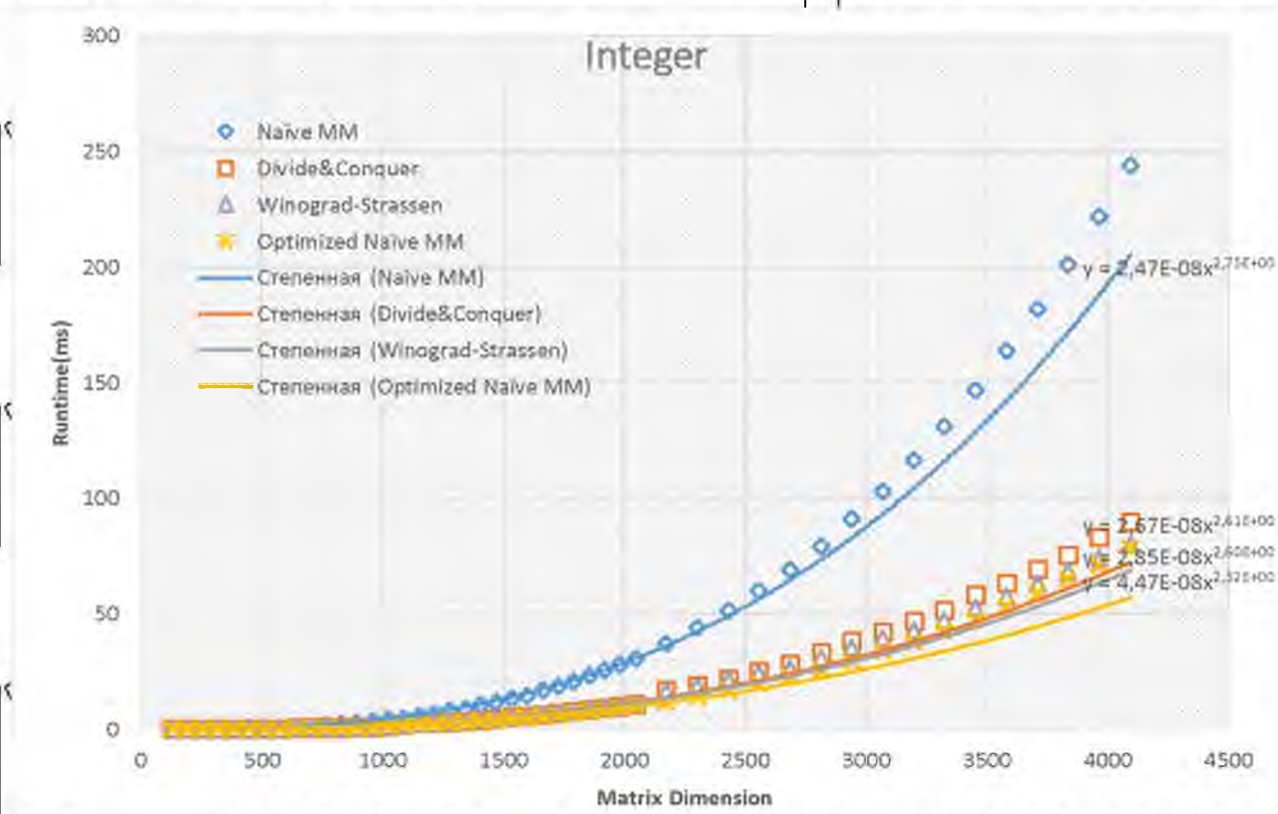


Рис. 3.4 Визначення асимптотичної складності на основі емпіричних даних множення матриць використовуючи різні алгоритми на GeForce GTX 1660 для double

Висновок. Архітектура CUDA найбільш оптимізована під роботу з числами одинарної точності. Для роботи з такими числами компанією було впроваджено використання додаткових спеціалізованих тензорних ядер для подальшої фасилітації таких обчислень, які використовуються при використанні бібліотеки

CUBLAS і які впливають на час виконання обчислення порівняно з іншими типами даних.

На рис. 3.18-3.21 показані результати обчислення паралельних алгоритмів, реалізованих на базі послідовних алгоритмів і які були досліджені в попередній секції . Можна помітити, що складність таких алгоритмів (результати були отримані експериментально) суттєво відрізняється від теоретичних. Такий результат частково пояснюється наявним резервом у вигляді додаткових ядер які «під'єднуються» до обрахунків із збільшенням розміру вхідних даних. Також, на рівні мультипроцесору виконується планування потоків. Це дозволяє зменшити кількість холостих тактів за рахунок виконання тільки тих інструкцій, операнди яких були заздалегідь завантажені і готові до обчислення.

Висновки

В цій роботі були розроблені паралельні алгоритми матричного множення для цілочисельних і поліноміальних матриць на відеокарті з використанням архітектури NVIDIA CUDA. Для здійснення більш повного аналізу, а також порівняння роботи з цілочисельними типами даних, ці алгоритми були також адаптовані під числа з одинарною і подвійною точністю. Для досягнення мети дослідження ми провели огляд архітектури графічного процесора і моделі програмування на ньому, дослідили послідовні версії алгоритмів на центральному процесорі, розробили і експериментально дослідили паралельні алгоритми на CUDA.

В ході виконання дослідження були вирішені наступні проблеми:

- обчислення матриць надвеликих розмірів;
- оптимізація збереження матриць в пам’яті та алгоритмів множення
- використання апаратних особливостей архітектури для підвищення швидкодії алгоритму і більш ефективного використання наявних ресурсів
- обробка матриць нестандартних розмірів

Розроблені алгоритми працюють для будь-яких розмірів матриць (деякі оптимізації потребують використання технік описаних в цій роботі для пришвидшення обчислення). Основним обмеженням для алгоритмів є лише апаратні обмеження графічного чіпу в пам’яті та обчислювальної потужності. Алгоритми автоматично визначають оптимальну конфігурацію, базуючись на апаратних характеристиках відеокарти.

Алгоритми були протестований на різних розмірах матриць, з різними параметрами для оцінки таких показників, як:

- швидкодії алгоритму в залежності від розміру матриці
- швидкодії алгоритму в залежності від різних типів даних
- швидкодії алгоритму в залежності від типу пристрою

- швидкодії алгоритму в порівнянні з іншими алгоритмами.

НУБіП України
НУБіП України
НУБіП України
НУБіП України
НУБіП України
НУБіП України

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

[1] Santos, Eunice E., «Parallel Complexity of Matrix Multiplication,» *The Journal of Supercomputing,* т. 25, № 2, pp. 155-175, 2003.

[2] Saule, Erik; Kaya, Kamer; Çatalyürek, Ümit V., «Performance Evaluation of Sparse Matrix Multiplication Kernels on Intel Xeon Phi,» *arXiv: Performance,* 2013.

[3] M. . Maggioni та T. Y. Berger-Wolf, «An architecture-aware technique for optimizing sparse matrix-vector multiplication on GPUs,» *Procedia Computer Science,* т. 18, №, pp. 329-338, 2013.

[4] «Nvidia CUDA Home Page,» , . [Онлайновий]. Available: https://developer.nvidia.com/cuda-zone. [Дата звернення: 24 11 2021].

[5] Даан-Дальмедико А., Пейффер Ж., Пути и лабиринты. Очерки по истории математики, Мир, 1986.

[6] Strassen, V., «Gaussian elimination is not optimal,» т. 13, № 4, p. 354–356, 1969.

[7] Winograd, S., A new algorithm for inner product, Томи %1 з %2C-17:7, IEEE Trans. Computers, 1968, p. 693–694.

[8] V.Ya. Pan, Fast matrix multiplication without APA-algorithms, т. 8:5, Comput. Math. Appl., 1982, p. 343–366.

[9] V.Ya. Pan, «Strassen's algorithm is not optimal trilinear technique of aggregating, uniting and canceling for constructing fast algorithms for matrix operations,» в *19th Annual Symposium on Foundations of Computer Science (Ann Arbor, Mich., 1978)*, Long Beach, Calif., 1978.

[10] Пан В. Я., «О схемах вычисления произведений матриц и обратной матрицы,» *УМН,* т. 27, № 7, p. 249–250, 1972.

[11] V.Ya. Pan, «Better late than never: filling a void in the history of fast matrix multiplication and tensor decompositions,» arXiv: 1411.1972.

[12] V.Ya. Pan, «Trilinear aggregating with implicit canceling for a new acceleration of matrix multiplication,» *Comput. Math. Appl.*, т. 8, № 1, p. 23–34, 1982.

[13] Roger A. Horn, Charles R. Johnson, Matrix Analysis, Cambridge University Press, 1985.

[14] Strang, Gilbert, Linear Algebra and Its Applications, т. IV Edition, Thomson Brooks/Cole, 2006.

[15] Клепко В.Ю., Голець В.Л, Вища математика в прикладах і задачах: Навчальний посібник, 2-ге видання ред., Київ: Центр учбової літератури, 2009.

[16] Kaare Brandt Petersen, Michael Syskind Pedersen, The Matrix Cookbook, 2012.

[17] Sharma, A.K., Text Book of Matrix, Discovery Publishing House, 2004.

[18] N. . Kartha, «Review of the algorithm design manual, second edition by Steven S. Skiena,» *Sigact News,* т. 42, № 4, pp. 29-31, 2011.

[19] Jean-Noël Quintin, Khalid Hasanov, Alexey Lastovetsky, «Hierarchical Parallel Matrix Multiplication on Large-Scale Distributed Memory Platforms,» в *42nd International Conference on Parallel Processing*, October 1-4 2013.

[20] Rugina, Radu; Rinard, Martin, «Recursion Unrolling for Divide and Conquer Programs,» *Lecture Notes in Computer Science,* pp. 34-48, 2000.

[21] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, Introduction to Algorithms, Third Edition, MIT Press.

[22] Hedtke, Ivo, «Strassen's Matrix Multiplication Algorithm for Matrices of Arbitrary Order,» *Bulletin of Mathematical Analysis and Applications,* т. 3, № 2, pp. 269-277, 2011.

[23] S, Winograd, «On Multiplication of 2 × 2 Matrices. Linear Algebra and Application,» *Linear Algebra and its Applications,* т. 4, p. 381–388, 1971.

[24] D'Alberto, P, «(2014) The Better Accuracy of Strassen-Winograd Algorithms (FastMMW),» *Advances in Linear Algebra & Matrix Theory,* т. 4, № 9-39.

[25] C. Douglas, M. Heroux, G. Slishman, and R. M. Smith, «GEMMW: A portable level 3 BLAS Winograd variant of Strassen's matrix-matrix multiply algorithm,» *Journal of Computational Physics,* т. 110, № 1-10, 1994.

[26] S. Huss-Lederman, E. M. Jacobson, J. R. Johnson, A. Tsao, and T. Turnbull, Implementation of Strassen's Algorithm for Matrix Multiplication, Proceedings of Supercomputing '96, 1996.

[27] Khorasani, Elham S., «Algorithms Sequential & Parallel: A Unified Approach,» *Scalable Computing: Practice and Experience,* т. 8, № 1, 2007.

[28] Michael J. Flynn, «Very High-Speed Computing Systems,» *Proceedings of the IEEE,* т. 54, № 12, p. 1901–1909, December 1966.

[29] Aart, Evert van; Sepasian, Neda; Jalba, Andrei C.; Vilanova, Anna, «CUDA-accelerated geodesic ray-tracing for fiber tracking,» *International Journal of Biomedical Imaging,* т. 2011, pp. 698908-698908, 2011.

[30] Thurley, Matthew; Danell, V., «Fast Morphological Image Processing Open-Source Extensions for GPU Processing With CUDA,» *IEEE Journal of Selected Topics in Signal Processing,* т. 6, № 7, pp. 849-855, 2012.

[31] Brædstrup, C. F.; Egholm, D. L., «CUDA GPU based full-Stokes finite difference modelling of glaciers,» 2012. [Онлайновий]. Available: https://ui.adsabs.harvard.edu/abs/2012eguga..14.2932b/abstract. [Дата звернення: 7 11 2021].

[32] Guide, CUDA C/C++ Programming, 2020. [Онлайновий]. Available: https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf. [Дата звернення: 05. 10. 2021].

[33] CUDA C++ Best Practices Guide, October 2021. [Онлайновий]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf. [Дата звернення: 02 November 2021].

[34] Винберг Э.Б, Алгебра многочленов, Москва: Просвещение, 1980.

[35] E. Mahdi, «A Survey of R Software for Parallel Computing,» *American Journal of Applied Mathematics and Statistics,* т. 2, № 4, pp. 224-230, 2014.

[36] S. Lipschutz та M. Lipson, Linear Algebra, McGraw Hill (USA), p. 30–31.

[37] Firstauthor, A. R.; Secondauthor, I. B.; Thirdauthor, G. V.; Fourthauthor, P. B.; Fifthauthor, M. K., «Biomedical image processing with GPGPU using CUDA,» 2011. [Онлайновий]. Available: http://yadda.icm.edu.pl/yadda/element/bwmeta1.element.ieee-000005967067. [Дата звернення: 7 11 2021].

[38] Березкина, Э.И., Математика древнего Китая, Москва: Наука, 1980, pp. 173-206.

[39] Khan, Rafiqul Zaman; Ali, Firoj, «Current Trends in Parallel Computing,» *International Journal of Computer Applications,* т. 59, № 2, pp. 19-25, 2012.

Додаток А


```
/* ADD_INT KERNEL */
__global__ void add_int(int *d_a, int *d_b, int *d_c, unsigned int N) {
    const int tid = 4 * threadIdx.x + blockIdx.x * (4 * blockDim.x);
    if (tid < N) {

        int a1 = d_a[tid];
        int b1 = d_b[tid];

        int a2 = d_a[tid+1];
        int b2 = d_b[tid+1];

        int a3 = d_a[tid+2];
        int b3 = d_b[tid+2];

        int a4 = d_a[tid+3];
        int b4 = d_b[tid+3];

        int c1 = a1 + b1;
        int c2 = a2 + b2;
        int c3 = a3 + b3;
        int c4 = a4 + b4;

        d_c[tid] = c1;
        d_c[tid+1] = c2;
        d_c[tid+2] = c3;
        d_c[tid+3] = c4;

    }
}

/* ADD_INT1 KERNEL */
__global__ void add_int2(int2 *d_a, int2 *d_b, int2 *d_c, unsigned int N) {

    const int tid = 2 * threadIdx.x + blockIdx.x * (2 * blockDim.x);
```

```
    if (tid < N) {

        int2 a1 = d_a[tid];
        int2 b1 = d_b[tid];

        int2 a2 = d_a[tid+1];
        int2 b2 = d_b[tid+1];

        int2 c1;
        c1.x = a1.x + b1.x;
        c1.y = a1.y + b1.y;

        int2 c2;
        c2.x = a2.x + b2.x;
        c2.y = a2.y + b2.y;

        d_c[tid] = c1;
        d_c[tid+1] = c2;

    }

}

/* ADD_INT4 KERNEL */
__global__ void add_int4(int4 *d_a, int4 *d_b, int4 *d_c, unsigned int N) {

    const int tid = 1 * threadIdx.x + blockIdx.x * (1 * blockDim.x);

    if (tid < N/4) {

        int4 a1 = d_a[tid];
        int4 b1 = d_b[tid];

        int4 c1;
        c1.x = a1.x + b1.x;
        c1.y = a1.y + b1.y;
        c1.z = a1.z + b1.z;
        c1.w = a1.w + b1.w;
```

```
        d_c[tid] = c1;
    }
}
```