Міністерство освіти і науки України

Львівський національний університет імені Івана Франка

Факультет прикладної математики та інформатики

Кафедра програмування

Звіт

до лабораторної роботи №8

**“CUDA”**

Підготував:

студент групи ПМІ-31

 Процьків Назарій

Львів 2023

***Завдання***

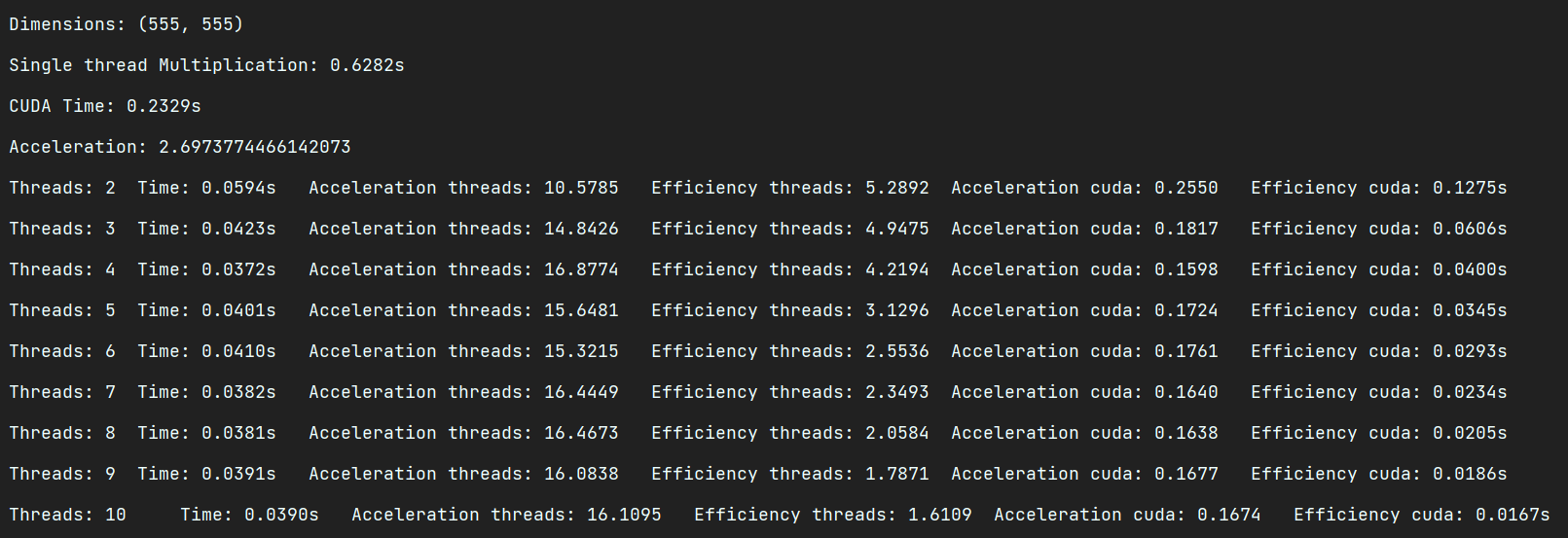
На основі програмно-апаратної архітектури паралельних обчислень CUDA реалізувати **множення матриць**. Продемонструвати результати матриць різної розмірності та порівняти з результатами відповідної попередньої лабораторної роботи.

***Хід роботи***

Виконав цю лабораторну мовою програмування Python.

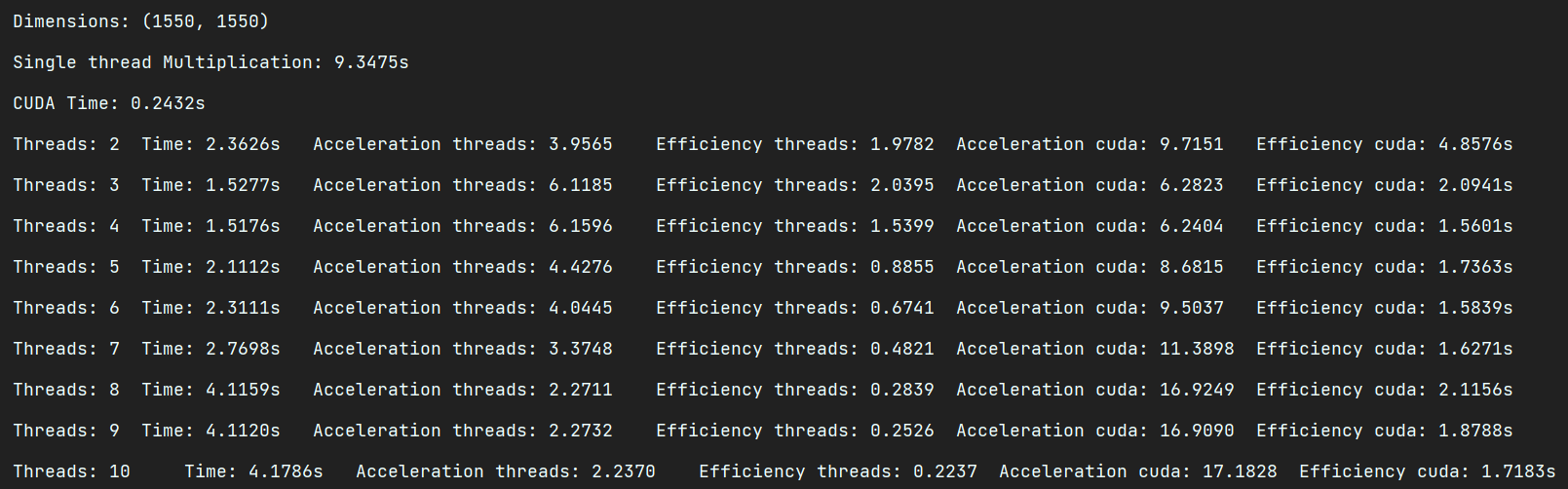
В Python є бібліотека numba, яка дозволяє працювати з CUDA. Для того, щоб вона працювала потрібно встановити CUDA Toolkit від NVIDIA і в змінну глобального середовища покласти шлях до папки, в яку було її встановлено. З допомогою цієї бібліотеки я реалізував множення матриць.

Розглянемо перший результат:



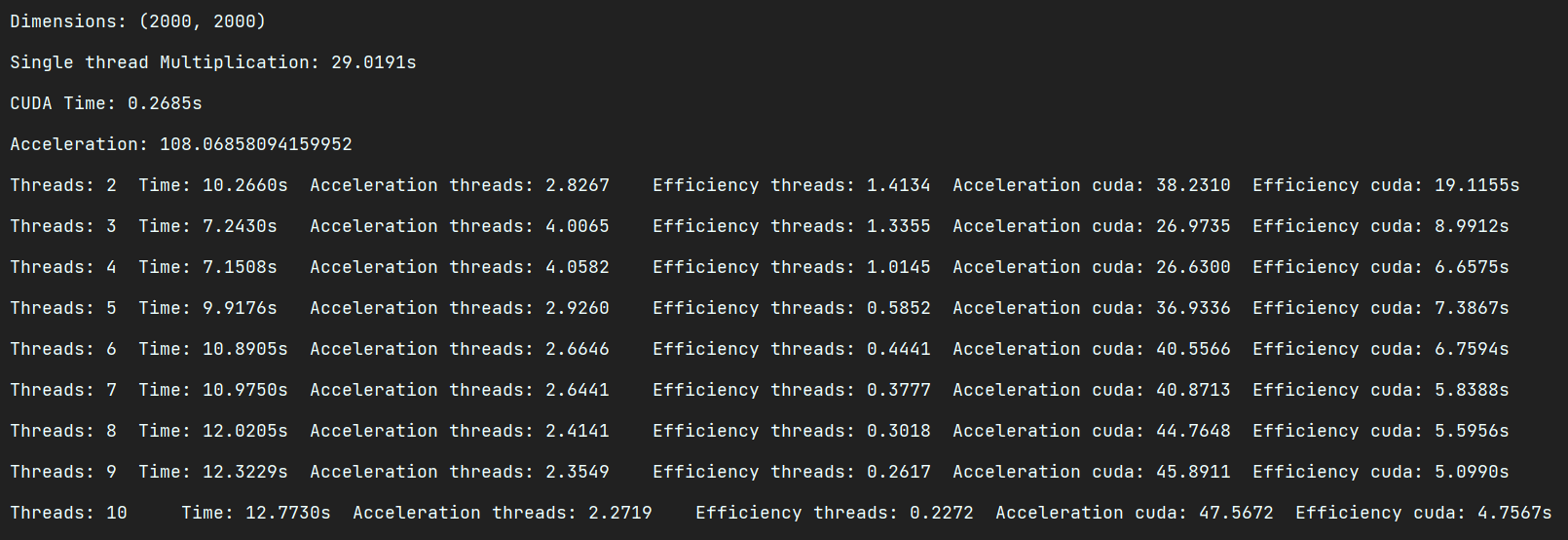
Множимо дві матриці розмірностей (555х555). Час множення одним потоком - 0.62с. Час множення CUDA - 0.232с. Далі показано прискорення CUDA до послідовного множення. Тоді в циклі для кількості потоків від 2 до 10 визначається час множення і два прискорення та дві ефективності. Перші прискорення та ефективність для відношення часу паралельного виконання і послідовного виконання, а друге для відношення CUDA і паралельного виконання, для того, щоб також порівняти наскільки CUDA випереджає ще й паралельний спосіб.

Другий результат:



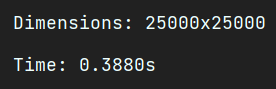
Тут бачимо, що розмір матриці збільшився приблизно на 1000, а час виконання CUDA майже не змінився. Так відбувається тому, що ця технологія динамічно виділяє стільки потоків так, щоб виконувати операції було максимально ефективно по часу. Тому звідси можна зробити висновок, що чим більша буде задача, тим більше буде перевага CUDA над будь-чим іншим. Якщо станеться так, що задача буде настільки складною, що буде здаватись що навіть CUDA виконує її занадто повільно, то завжди можна збільшити розмір блоку на які ми розбиваємо матрицю. В цій програмі в мене розмір блоку має розміри 16x16.

Третій результат:



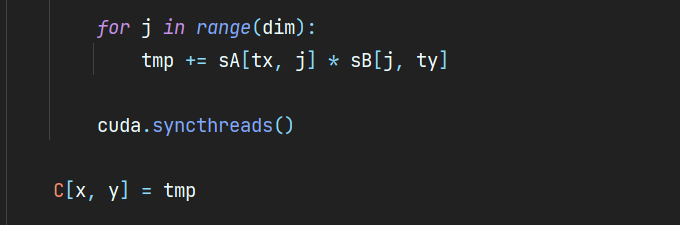
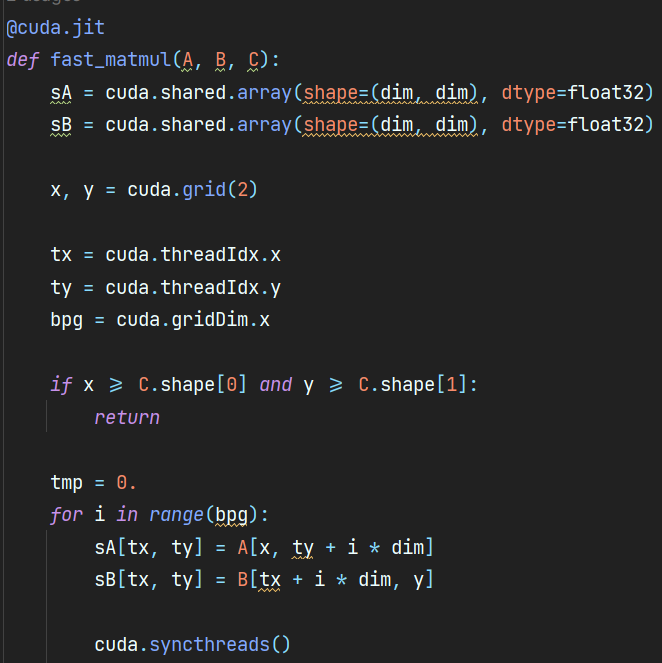
Прискорення та ефективності набагато зросли, бо час виконання послідовно і паралельно збільшується, а час виконання CUDA зовсім не змінюється.

Якщо провести цей експеримент для матриць розмірностей (25000х25000), то CUDA покаже приблизно той же результат (можливо більший на 0.1с):



А множення послідовно цих матриць покаже результат в кілька годин (від 3 до 5).

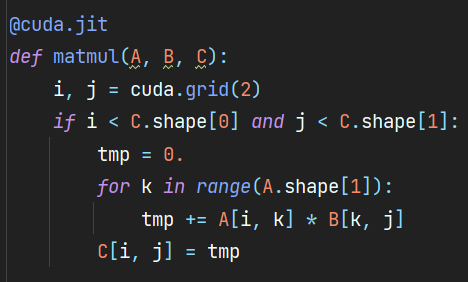
Реалізація CUDA:



Ця функція приймає як аргументи три матриці. Дві для множення і одна для запису результату. Тут проводиться розбиття на блоки та налаштування спільного доступу до пам’яті. Проходить перевірка на валідність розмірів. І тоді основний цикл для множення. cuda.syncthreads() означає, що потоки будуть синхронізовувати дані між собою кожен раз.

Декоратор @cuda.jit імпортується з бібліотеки numba. .jit - означає, що ця функція буде компілюватись, а тому і швидше виконуватись. Це спеціальний компілятор для Python, який має абревіатуру - Just In Time. Тобто в коді буде один шматок, який скомпілюється, а інший інтерпретується, як простий Python.

Також є другий код, він менш оптимізований, але на ньому треба дещо пояснити.



Тут теж відбувається множення двох матриць A і B, а запис в C. Але додавання у змінну tmp може проводитись так як показано на скріншоті, а може теоретично з допомогою функції numpy.dot - для множення рядка на стовпець. Але коли я це писав, то зіткнувся з проблемою: з декоратором @cuda.jit потрібно писати максимально прості операції, для того, щоб вони могли бути скомпільованими. Якщо написати функцію трохи вищого рівня, то вона не буде скомпільована.

Також під час виконання лабораторної роботи зіткнувся ще з однією помилкою - якщо задати занадто малі значення розмірності матриці, то numba.CUDA покаже помилку про те, що кількість роботи є занадто малою і використання CUDA не окупиться, бо створення потоків буде займати більше часу, ніж буде з того користі.

***Висновок***: під час виконання лабораторної роботи №8, написав максимально оптимізовану програму для множення матриць з використанням технології CUDA. Визначив прискорення та ефективність для часу з потоками порівняно з одним потоком та з часом на CUDA. Порівняв результати з відповідною попередньою лабораторною роботою. Дізнався, що існує пайтон компілятор.