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:

```
Dimensions: (555, 555)
Single thread Multiplication: 0.6282s
CUDA Time: 0.2329s
Acceleration: 2.6973774466142073
Threads: 2  Time: 0.0594s   Acceleration threads: 10.5785   Efficiency threads: 5.2892  Acceleration cuda: 0.2550   Efficiency cuda: 0.1275s
Threads: 3  Time: 0.0423s   Acceleration threads: 14.8426   Efficiency threads: 4.9475  Acceleration cuda: 0.1817   Efficiency cuda: 0.0606s
Threads: 4  Time: 0.0372s   Acceleration threads: 16.8774   Efficiency threads: 4.2194  Acceleration cuda: 0.1598   Efficiency cuda: 0.0400s
Threads: 5  Time: 0.0401s   Acceleration threads: 15.6481   Efficiency threads: 3.1296  Acceleration cuda: 0.1724   Efficiency cuda: 0.0345s
Threads: 6  Time: 0.0410s   Acceleration threads: 15.3215   Efficiency threads: 2.5536  Acceleration cuda: 0.1761   Efficiency cuda: 0.0293s
Threads: 7  Time: 0.0382s   Acceleration threads: 16.4449   Efficiency threads: 2.3493  Acceleration cuda: 0.1640   Efficiency cuda: 0.0234s
Threads: 8  Time: 0.0381s   Acceleration threads: 16.4673   Efficiency threads: 2.0584  Acceleration cuda: 0.1638   Efficiency cuda: 0.0205s
Threads: 9  Time: 0.0391s   Acceleration threads: 16.0838   Efficiency threads: 1.7871  Acceleration cuda: 0.1677   Efficiency cuda: 0.0186s
Threads: 10     Time: 0.0390s   Acceleration threads: 16.1095   Efficiency threads: 1.6109  Acceleration cuda: 0.1674   Efficiency cuda: 0.0167s
```

Множимо дві матриці розмірностей (555x555). 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:

```
Dimensions: (1550, 1550)
Single thread Multiplication: 9.3475s
CUDA Time: 0.2432s
Threads: 2  Time: 2.3626s   Acceleration threads: 3.9565    Efficiency threads: 1.9782  Acceleration cuda: 9.7151   Efficiency cuda: 4.8576s
Threads: 3  Time: 1.5277s   Acceleration threads: 6.1185    Efficiency threads: 2.0395  Acceleration cuda: 6.2823   Efficiency cuda: 2.0941s
Threads: 4  Time: 1.5176s   Acceleration threads: 6.1596    Efficiency threads: 1.5399  Acceleration cuda: 6.2404   Efficiency cuda: 1.5601s
Threads: 5  Time: 2.1112s   Acceleration threads: 4.4276    Efficiency threads: 0.8855  Acceleration cuda: 8.6815   Efficiency cuda: 1.7363s
Threads: 6  Time: 2.3111s   Acceleration threads: 4.0445    Efficiency threads: 0.6741  Acceleration cuda: 9.5037   Efficiency cuda: 1.5839s
Threads: 7  Time: 2.7698s   Acceleration threads: 3.3748    Efficiency threads: 0.4821  Acceleration cuda: 11.3898  Efficiency cuda: 1.6271s
Threads: 8  Time: 4.1159s   Acceleration threads: 2.2711    Efficiency threads: 0.2839  Acceleration cuda: 16.9249  Efficiency cuda: 2.1156s
Threads: 9  Time: 4.1120s   Acceleration threads: 2.2732    Efficiency threads: 0.2526  Acceleration cuda: 16.9090  Efficiency cuda: 1.8788s
Threads: 10     Time: 4.1786s   Acceleration threads: 2.2370    Efficiency threads: 0.2237  Acceleration cuda: 17.1828  Efficiency cuda: 1.7183s
```

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:

```
Dimensions: (2000, 2000)
Single thread Multiplication: 29.0191s
CUDA Time: 0.2685s
Acceleration: 108.06858094159952
Threads: 2  Time: 10.2660s  Acceleration threads: 2.8267    Efficiency threads: 1.4134  Acceleration cuda: 38.2310  Efficiency cuda: 19.1155s
Threads: 3  Time: 7.2430s   Acceleration threads: 4.0065    Efficiency threads: 1.3355  Acceleration cuda: 26.9735  Efficiency cuda: 8.9912s
Threads: 4  Time: 7.1508s   Acceleration threads: 4.0582    Efficiency threads: 1.0145  Acceleration cuda: 26.6300  Efficiency cuda: 6.6575s
Threads: 5  Time: 9.9176s   Acceleration threads: 2.9260    Efficiency threads: 0.5852  Acceleration cuda: 36.9336  Efficiency cuda: 7.3867s
Threads: 6  Time: 10.8905s  Acceleration threads: 2.6646    Efficiency threads: 0.4441  Acceleration cuda: 40.5566  Efficiency cuda: 6.7594s
Threads: 7  Time: 10.9750s  Acceleration threads: 2.6441    Efficiency threads: 0.3777  Acceleration cuda: 40.8713  Efficiency cuda: 5.8388s
Threads: 8  Time: 12.0205s  Acceleration threads: 2.4141    Efficiency threads: 0.3018  Acceleration cuda: 44.7648  Efficiency cuda: 5.5956s
Threads: 9  Time: 12.3229s  Acceleration threads: 2.3549    Efficiency threads: 0.2617  Acceleration cuda: 45.8911  Efficiency cuda: 5.0990s
Threads: 10     Time: 12.7730s  Acceleration threads: 2.2719    Efficiency threads: 0.2272  Acceleration cuda: 47.5672  Efficiency cuda: 4.7567s
```

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 (25000x25000), то CUDA покаже приблизно той же результат (можливо більший на 0.1с):

```
Dimensions: 25000x25000

Time: 0.3880s
```

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

Реалізація CUDA:

```python
@cuda.jit
def fast_matmul(A, B, C):
    sA = cuda.shared.array(shape=(dim, dim), dtype=float32)
    sB = cuda.shared.array(shape=(dim, dim), dtype=float32)

    x, y = cuda.grid(2)

    tx = cuda.threadIdx.x
    ty = cuda.threadIdx.y
    bpg = cuda.gridDim.x

    if x >= C.shape[0] and y >= C.shape[1]:
        return

    tmp = 0.
    for i in range(bpg):
        sA[tx, ty] = A[x, ty + i * dim]
        sB[tx, ty] = B[tx + i * dim, y]

        cuda.syncthreads()
```

```python
    for j in range(dim):
        tmp += sA[tx, j] * sB[j, ty]

    cuda.syncthreads()

C[x, y] = tmp
```

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.

```
@cuda.jit
def matmul(A, B, C):
    i, j = cuda.grid(2)
    if i < C.shape[0] and j < C.shape[1]:
        tmp = 0.
        for k in range(A.shape[1]):
            tmp += A[i, k] * B[k, j]
        C[i, j] = tmp
```

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.