Wrocław, 22 października 2024

Programowanie **r**ównoległe i **r**ozproszone

Laboratorium 2

Programowanie równoległe z wykorzystaniem `GPU`

prowadzący: *Dr inż. Radosław Idzikowski*

---

1 Wprowadzenie

Celem laboratorium jest zapoznanie się z wykonywaniem obliczeń z wykorzystaniem procesora graficznego (*Graphics Processing Unit*, `GPU`) w technologii `CUDA` z poziomu języka `Python`. Wszystlko obliczenia zostaną wykonane w chmurze przy użyciu narzędzia `Google Colab`. Czas przewidziany na wykonanie zadania to 1 termin wraz z ocenieniem pracy. Praca będzie oceniana na bieżąco w trakcie zajęć wraz z postępem programowania kolejnych przykładów. **Po ukończeniu każdego zadania należy zawołać prowadzącego w celu zaakceptowania etapu i odnotowania postępów.**

2 Zadania

W ramach zajęć należy w zespołach wykonać następujące zadania:

1. Pierwszy program na `GPU` – poznanie narzędzi.
2. Programowanie niskopoziomowe.
3. Analiza i pomiar efektywności.

Za wykonanie zadania nr 1 jest ocena dostateczna, za każde kolejne zadanie jest +1 do oceny. Na ocenę bardzo dobrą (5.0) należy wykonać wszystkie trzy zadania. Po zakończeniu zajęć należy kody źródłowe napisanych programów przesłać do prowadzącego.

3 Opis zadań

3.1 Zadanie 1

Zasadniczym celem zadania jest poznanie narzędzi potrzebnych do zrobienia zadania. W pierwszej kolejność należy utworzyć nowy `notebook` w serwie `Google Colab`. Następnie należy utworzyć nowy `notebook`, w którym musimy zmienić środowisko wykonawcze zawierające `GPU NVIDIA Tesla T4`, będzie niezbędne do wykonania zadania. Poprawnie wykrycie karty graficznej można sprawdzić poprzez użycie polecenia `!nvidia-smi`, które zwróci podstawowe informacje o dostępnym `GPU`, w tym m. in. aktualne użycie pamięci.

Implementacja wysokopoziomowa z wykorzystaniem biblioteki `CuPy` w swoim działaniu jest bardzo podobna do biblioteki `NumPy`. Dla przypomnienia, mnożenie w `NumPy` przedstawiono na listingu 1. W celu stworzenia efektywniej implementacji należy użyć tablic z biblioteki `NumPy`. Linia `np.random.rand(10 , 10)` zwróci nam odpowiednią macierz o rozmiarze 10×10. Do samego mnożenia można użyć funkcji `np.matmul`, która jest dedykowana do mnożenia macierzy lub uniwersalnej funkcji `numpy.dot`, którą można pomnożyć zarówno macierze jak i wektory. Na koniec podano prosty sposób pomiaru czasu.

```python
import numpy as np
import time
N = 1024
A_cpu = np.random.rand(N, N).astype(np.float32)
B_cpu = np.random.rand(N, N).astype(np.float32)
start = time.time()
C_cpu = np.matmul(A_cpu, B_cpu)
end = time.time()
print(f"Mnożenie macierzy na CPU trwało: {end - start:.5f} sekund.")
```

Listing 1: Mnożenie macierzy na CPU

W przypadku mnożenia na GPU musimy dołączyć jeszcze odpowiednią bibliotekę do tego typu obliczeń. W przypadku mnożenia wysokopoziomowego można skorzystać z CuPy, która domyślnie jest zainstalowana w środowisku stworzonym w ramach Google Colab. Samo wywołanie przebiega bardzo podobnie jak z użyciem NumPy, gotową implementację przedstawiono na listingu 2. Do utworzenia samych macierzy używamy tych samych funkcji, następnie macierze należy przesłać na GPU, co należy zrobić funkcją cp.array. Zarówno funkcja np.matmul oraz numpy.dot mają swoje odpowiedniki w bibliotece CuPy. Po zakończeniu mnożenia niezbędne jest wykonanie synchronizacji (cp.cuda.Stream.null.synchronize()).

```python
import cupy as cp
import time
N = 1024
A_cpu = np.random.rand(N, N).astype(np.float32)
B_cpu = np.random.rand(N, N).astype(np.float32)
A_gpu = cp.array(A_cpu)
B_gpu = cp.array(B_cpu)
start = time.time()
C_gpu = cp.matmul(A_gpu, B_gpu)
cp.cuda.Stream.null.synchronize()  # Synchronizacja z GPU
end = time.time()
print(f"Mnożenie macierzy na GPU trwało: {end - start:.5f} sekund.")
```

Listing 2: Mnożenie macierzy wysokopoziomowe na GPU

3.2 Zadanie 2

Celem tego zadania jest napisanie typowego kernela CUDA, który zostanie wykorzystany w naszym mnożeniu. Jednak cała jego zewnętrzna obsługa będzie z poziomu języka Python. Gotowy kod został umieszczony w listingu 3. Kernel działa na poziomie bloków i wątków, co umożliwia wykonywanie obliczeń na wielu elementach macierzy jednocześnie. Każdy wątek odpowiada za obliczenie jednego elementu macierzy wynikowej. Każdy wątek korzysta z identyfikatorów bloków (blockIdx) oraz identyfikatorów wątków (threadIdx), aby ustalić, który element macierzy wynikowej C ma obliczyć. Indeks wiersza row oraz kolumny col jest obliczany na podstawie identyfikatorów wątków wewnątrz bloków (threadIdx.x, threadIdx.y) oraz pozycji bloku w siatce (blockIdx.x, blockIdx.y). Kernel sprawdza, czy indeksy wiersza i kolumny znajdują się w zakresie rozmiaru macierzy N, aby uniknąć odwołania się do elementów poza tablicą. Wewnątrz pętli for, wątek sumuje iloczyny odpowiednich elementów wiersza macierzy A i kolumny macierzy B. Na koniec sumuje wyniki w odpowiednim elemencie macierzy C.

```python
kernel_code = """
__global__ void matrixMul(float *A, float *B, float *C, int N) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0;
    if(row < N && col < N) {
        for (int k = 0; k < N; k++) {
            sum += A[row * N + k] * B[k * N + col];
        }
        C[row * N + col] = sum;
    }
}
"""
```

Listing 3: Mnożenie macierzy wysokopoziomowe na GPU

Przed przystąpieniem do uruchamiania kernela, należy najpierw zainstalować PyCuda. Podczas instalacji pakietu mogą wystąpić problemy z kodowaniem, więc przed można ustawić odpowiednie kodowanie.

```
import locale
locale.getpreferredencoding = lambda: "UTF-8"
!pip install pycuda
```

Po zainstalowaniu biblioteki, można przejść do kolejnych kroków uruchamiana kernela. W przeciwieństwie do biblioteki wysokopoziomowej, teraz musimy większość etapów wymusić ręcznie:

1. Dodanie odpowiednich bibliotek:

   ```
   import pycuda.driver as cuda
   import pycuda.autoinit
   from pycuda.compiler import SourceModule
   ```

2. Kompilacja kernela `SourceModule(kernel_code)` oraz przypisanie mu nazwy `matrixMul = mod.get_function("matrixMul")`.
3. Inicjalizacja pustej macierzy wynikowej `C_cpu = np.empty((N, N), np.float32)`.
4. Alokacja pamięci na `GPU`, np.: `A_gpu = cuda.mem_alloc(A_cpu.nbytes)`, dla każdej macierzy osobno, w tym wynikowej.
5. Przesłanie danych na `GPU`, `cuda.memcpy_htod(A_gpu, A_cpu)`, dla obu macierzy `A` i `B`.
6. Definiowanie rozmiaru bloku `block_size = (32, 32, 1)`, `GPU` są zoptymalizowane do pracy z wątkami w grupach o rozmiarze 32 (to minimalna jednostka równoległego wykonania na `GPU`, zwana `warp`). Wybór bloków 32x32 jest powszechną praktyką, ponieważ dobrze wykorzystuje architekturę `GPU`, zwiększając efektywność obliczeń.
7. Definiowanie rozmiaru siatki `grid_size = (int(N/32), int(N/32), 1)`, siatka będzie miała wymiary 32x32 bloków (łącznie 1024 bloki), gdzie każdy blok przetwarza fragment macierzy o rozmiarze 32x32 elementów.
8. Uruchamianie kernela `CUDA`
   `matrixMul(A_gpu, B_gpu, C_gpu, np.int32(N), block=block_size, grid=grid_size)`
9. Synchronizacja z `GPU`: `cuda.Context.synchronize()`.
10. Pobranie danych z `GPU`: `cuda.memcpy_dtoh(C_cpu, C_gpu)`.

Po wykonaniu wszystkich operacji należy jeszcze wyczyścić pamięć karty graficznej. Jednym z możliwych rozwiązań jest ręczne zwolnienie zmiennych zaalokowanych na `GPU`, np.: `del A_gpu`.

3.3 Zadanie 3

Dla wszystkich napisanych algorytmów należy przeprowadzić analizę efektywności w zależności od rozmiaru macierzy. W tym celu należy narysować wykres (biblioteka `matplotlib.pyplot`) z poziomu `Google Colab`. Na listingu 4 pokazano prosty sposób sposób na narysowanie wykresu.

```
1 import matplotlib.pyplot as plt
2 x = [i for i in range(1,11)]
3 y = [i**2 for i in range(1,11)]
4 plt.plot(x, y, label="Chart")
5 plt.legend()
6 plt.xlabel("x")
7 plt.ylabel("y")
8 plt.show()
```

Listing 4: Podstawy kod to rysowania wykresów