Jakub Pawleniak 248897
Błażej Ułanowicz 249452

22.11.2021
Prowadzący: mgr inż. Szymon Datko

Akceleracja obliczeń w przetwarzaniu danych

Projekt

Politechnika Wrocławska

1. Problem obliczeniowy

Zdecydowaliśmy się na wybór zagadnienia przyspieszenia algorytmu sortującego typu *Bubble Sort*, wykorzystując do tego akcelerację *GPU*. Sam w sobie algorytm jest prosty i jednocześnie dosyć wolny (średnia złożoność obliczeniowa rzędu $O(n^2)$). Chcemy w tym przypadku pokazać, że z wykorzystaniem przetwarzania równoległego, można znacząco przyspieszyć działanie takiego algorytmu.

Zadanie w tym problemie polega na znalezieniu możliwości zrównoleglenia operacji sortowania, bez znaczącej modyfikacji podstawowego algorytmu.

Największym wyzwaniem jest przygotowanie obu wersji programów w taki sposób, aby bazowy algorytm się nie zmienił i było widać różnice wynikające tylko i wyłącznie z wykorzystania akceleracji obliczeń.

2. Planowane przyspieszenie

Algorytm operuje na zasadzie dwóch pętli - zewnętrznej oraz wewnętrznej. To właśnie w tym miejscu skupiona będzie nasza uwaga podczas skracania czasu wykonania. *Bubble Sort* polega na ciągłej podmianie elementów sąsiednich do momentu, w którym wszystkie obiekty znajdują się na odpowiedniej pozycji (po każdej iteracji ostatni element jest poprawnie umiejscowiony). Przyspieszenie zatem polegało będzie na zrównolegleniu sortowania, poprzez uruchomienie wielu iteracji algorytmu w tym samym czasie ale na różnych danych (co iterację mniej o jedną, ostatnią pozycję).

Spodziewany wzrost szybkości jest głównie zależny od liczby wykorzystanych wątków, a szacowana złożoność obliczeniowa będzie wynosić około $O(n^2/p)$, gdzie p jest liczbą wątków.

3. Środowisko oraz narzędzia

Do realizacji problemu postanowiliśmy wykorzystać język **Python** oraz open-source'ową bibliotekę **PyCUDA** dostarczającej interfejs umożliwiający korzystanie z platformy obliczeń równoległych opracowany przez firmę **NVIDIA** służącą do ogólnych obliczeń na procesorach graficznych (GPU).

(a) Język programowania. (b) Platforma obliczeń równoległych

Rysunek 1: Wykorzystane technologie.

Planujemy wykorzystać dwa środowiska testowe - w różnej konfiguracji i wydajności - jednak będące w obu przypadkach jednostaki mobilnymi (laptopy).

Środowisko 1:

- CPU - Intel Core i7-7700K
- GPU - NVidia GeForce GTX 1050 Ti

Środowisko 2:

- CPU - Intel Core i5-1035G1
- GPU - NVidia GeForce MX350

4. Realizacja projektu

Realizacja prostego algorytmu sortowania nie należała do wielkich wyzwań. Pierwszym krokiem było przygotowanie **implementacji naiwnej** mającej na celu ustanowienie punktu odniesienia dla pomiarów już po implementacji wersji zrównoleglonej algorytmu. Przygotowana funkcja sortująca została przedstawiona w listingu nr 1.

Listing 1: Implementacja naiwna algorytmu Bubble Sort.

```
def sort(self) -> List:
        tmp = self.original_array.copy()
        self._timer.start()
        for last_index in range(len(tmp) - 1, -1, -1):
            for curr_index in range(last_index):
                if tmp[curr_index] > tmp[curr_index + 1]:
                    tmp[curr_index], tmp[curr_index +
                                         1] = tmp[curr_index +
                                                  1], tmp[curr_index]
        self._timer.stop()
        self._sorting_time = self._timer.elapsed()
        return tmp
```

Oprócz samej funkcji sortującej na tym etapie przygotowane zostały pomocnicze klasy oraz funkcje wykorzystywane do generowania danych, pomiarów czasu, eksportu wyników do pliku csv itp.

4.1 Planowane użycie algorytmu

Zgodnie ze wcześniejszymi planami, realizację zrównoleglenia sortowania oparliśmy na algorytmie opisanym m.in przez Ali Yazici oraz Hakan Gokahmetoglu w artykule dotyczącym implementacji algorytmów sortowania przy wykorzystaniu architektury CUDA[1]. W oparciu o ten i wiele podobnych artykułów, udało się przygotować algorytm przedstawiony w listingu nr 2.

Listing 2: Kod kernela dla klasycznego algorytmu.

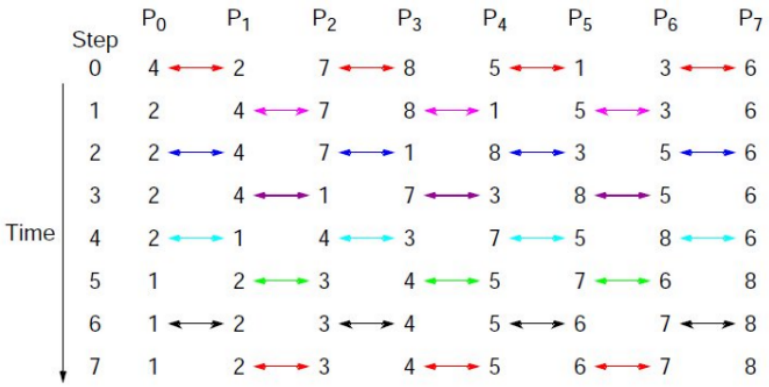
```
__global__ void bubbleSort(float *inputArray, int size) {
             int idx = threadIdx.x;
             int N = size-1;

             for(int i = idx; i <= N; i++) {
                 for(int j = 0; j <= N-1-i; j++) {
                     if(inputArray[j] > inputArray[j+1]) {
                         float tmp = inputArray[j];
                         inputArray[j] = inputArray[j+1];
                         inputArray[j+1] = tmp;
                     }
                 }
             }
         }
```

Niestety podczas pierwszych testów, już przy niewielkiej ilości danych wejściowych okazało się, że algorytm ten działa niepoprawnie. Sortował on prawidłowo wprowadzoną tablicę danych, jednak dane wyjściowe ulegały zniekształceniu. Wynikało to ze zjawiska wyścigów [2], występującego podczas równoległej pracy algorytmu.

Po wielu próbach pozbycia się tego zjawiska, znacznie pogarszających wyniki czasowe oraz wciąż powodujące zakłamanie danych wyjściowych, postanowiliśmy zmienić sposób wykonywania algorytmu na taki, w którym problem ten nie będzie występował.

4.2 Algorytm odd-even

Algorytmem, dzięki któremu udało się nam osiągnąć cel projektu, okazało się podejście **odd-even**. Jest to jeden z wariantów sortowania typu **Bubble Sort** [3], który opiera się na przemiennym sortowaniu elementów o parzystych i nieparzystych indeksach. Dzięki temu, unikamy problemu wyścigów i jednocześnie przyspieszamy, nie wykorzystując żadnych rozwiązań synchronizacyjnych (jak np. *mutex*). Poniższa grafika przedstawia zarys działania algorytmu.



Rysunek 2: Działanie algorytmu odd-even.

Korzystając ze zwykłej implementacji sekwencyjnej, algorytm ten nie posiada wyraźnej przewagi nad klasycznym algorytmem, jednak jest on o wiele szybszy i bezpieczniejszy (brak wyścigów) przy wykorzystaniu programowania równoległego. Jego szacowana złożoność obliczeniowa wynosi $O(n)$. Zaimplementowany kod kernela przedstawiony został w listingu nr 3.

Listing 3: Kod kernela dla algorytmu odd-even.

```
__global__ void bubbleSortOdd(int *inputArray, int size) {

            int i = blockIdx.x * blockDim.x + threadIdx.x;
            if(i % 2 == 0 && i < size-1){
                if(inputArray[i+1] < inputArray[i]){
                    int temp = inputArray[i];
                    inputArray[i] = inputArray[i+1];
                    inputArray[i+1] = temp;
                }
            }
        }

__global__ void bubbleSortEven(int *inputArray, int size) {

            int i = blockIdx.x * blockDim.x + threadIdx.x;
            if(i % 2 != 0 && i < size-1){
                if(inputArray[i+1] < inputArray[i]){
                    int temp = inputArray[i];
                    inputArray[i] = inputArray[i+1];
                    inputArray[i+1] = temp;
                }
            }
        }
```

Pełny kod źródłowy dostępny jest na publicznym repozytorium pod adresem:
https://github.com/Buglik/AO-BubbleSort

5. Analiza wyników

5.1 Przebieg eksperymentu

Ze względu na problemy z odpowiednim przygotowaniem środowiska na systemie Windows, zdecydowaliśmy się na przeprowadzenie eksperymentów na urządzeniu z systemem Linux.

Środowisko:

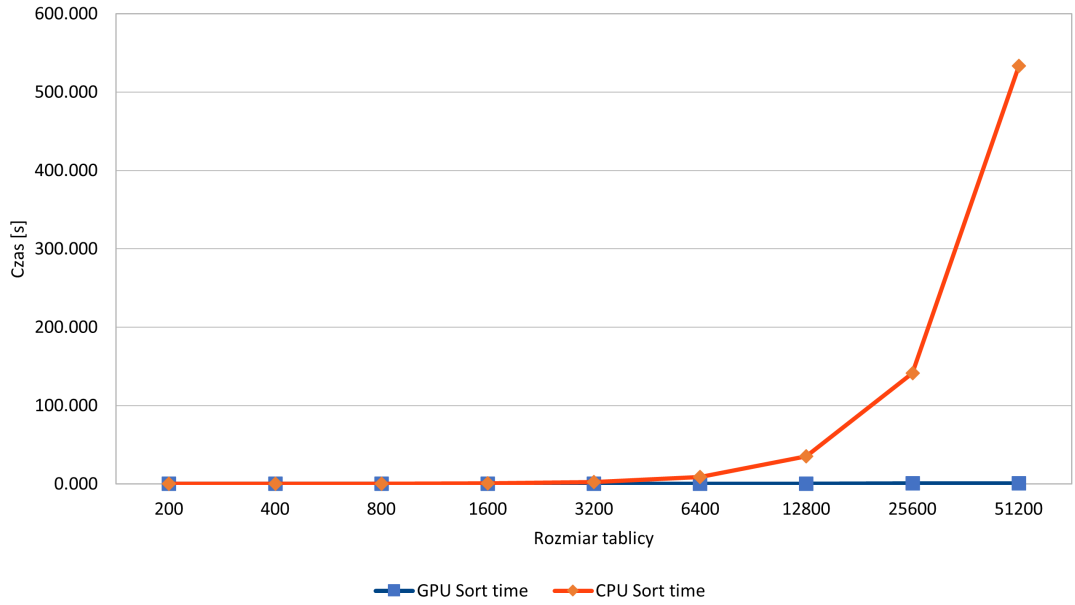
- CPU - Intel Core i7-7700K
- GPU - NVidia GeForce GTX 1050 Ti

Dla każdej wielkości tablicy danych przeprowadzone zostało **10 prób**. Poniższa tabela przedstawia uśrednione wyniki czasu działania w zależności od wielkości tablicy danych

Tablica 1: Średnie wyniki pomiarów.

Data length	GPU sort time [s]	CPU sort time [s]
200	0.004	0.009
400	0.007	0.034
800	0.013	0.143
1600	0.026	0.544
3200	0.050	2.192
6400	0.108	8.720
12800	0.210	35.014
25000	0.434	141.461
51200	0.856	533.413

Zgodnie z przewidywaniami wersja algorytmu wykorzystująca programowanie równoległe w ogromnym stopniu przyśpieszyła pracę sortowania. Różnice w prędkościach przy mniejszej ilości danych nie były rażąco niższe, jednak przy większej tablicy danych wejściowych poprawa wydajności czasowej stała się kolosalna - dochodząca nawet do ponad 500-krotnego przyśpieszenia.



Rysunek 3: Wykres czasu działania algorytmu względem rozmiaru tablicy wejściowej.

Pomiary nie uwzględniają czasów skopiowania danych do pamięci GPU oraz ich późniejszego pobrania - zostały pominięte, choć ze względu na bardzo szybki czas ich wykonania i tak mają niewielki wpływ na końcowy wynik (czasy te są rzędu 1 milisekundy dla tablicy wielkości 50000 elementów).

Literatura

[1] Ali Yazici, Hakan Gokahmetoglu: - *Implementation of Sorting Algorithms with CUDA: An Empirical Study*
https://dergipark.org.tr/en/download/article-file/225714

[2] Wikipedia: - *Race condition*
https://en.wikipedia.org/wiki/Race_condition

[3] Ricardo Rocha, Fernando Silva: - *Parallel Sorting Algorithms*
https://www.dcc.fc.up.pt/ ricroc/aulas/1516/cp/apontamentos/slides_sorting.pdf