Współczesne Środowiska Programowania

Laboratorium 2 – Całkowanie numeryczne

1. Celem ćwiczenia było zapoznanie się z programowaniem kart graficznych firmy Nvidia i napisanie programu obliczającego całkę z losowo utworzonych punktów odpowiednio na CPU oraz na GPU.

2. Testy zostały wykonane na karcie graficznej GTX 1050 Ti, program został napisany w języku python, do wykorzystania środowiska CUDA posłużyła biblioteka numba.

3. Operacje wykonywane przez program:

- generacja losowych punktów na CPU (wartości do 100 000, zmiennoprzecinkowe)

- sortowanie bąbelkowe na CPU otrzymanych punktów względem osi X

- interpolacja liniowa posortowanych punktów na CPU

- całkowanie punktów otrzymanych z interpolacji na CPU metodą trapezów

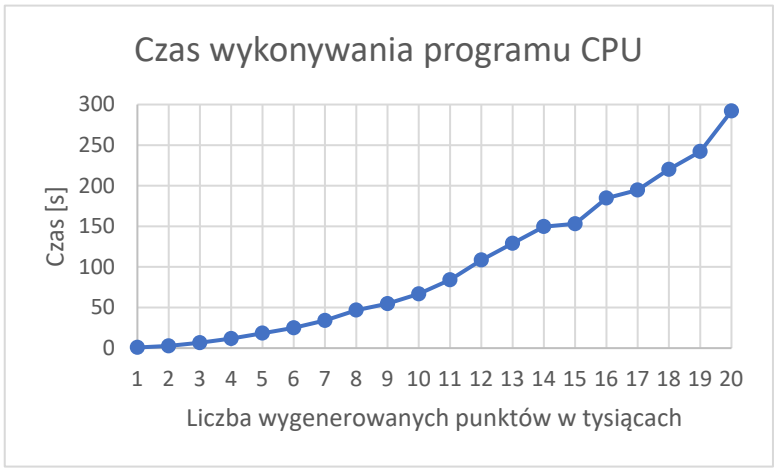
- sortowanie ‘odd-even’ na GPU (jest najbliższym algorytmem możliwym do zrównolegnienia, podobnym do sortowania bąbelkowego)

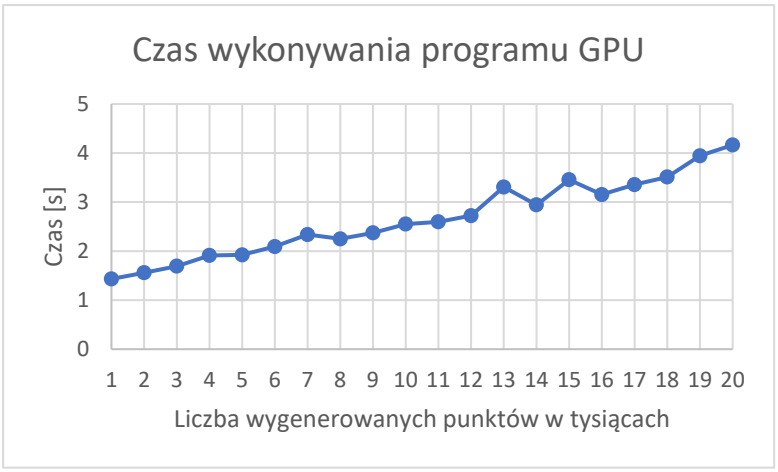
- interpolacja liniowa posortowanych punktów na GPU

- całkowanie metodą trapezów na GPU

3.5. Biblioteka numba tłumaczy funkcje Pythona na kod PTX, który wykonuje się na sprzęcie CUDA. Dekorator jit jest stosowany do funkcji Pythona napisanych w naszym kodzie Pythona dla CUDA. Numba współdziała z CUDA Driver API w celu załadowania PTX na urządzenie CUDA i wykonania. Semantyka wykorzystywana przez bibliotekę numba przypomina tą na C++, jesteśmy w stanie pisać kernele oraz przekazywać informacje z hosta na device i na odwrót. Nie musimy natomiast alokować pamięci poszczególnych zmiennych, ponieważ odbywa się to automatycznie (pamięć na GPU jest alokowana tak długo jak jest potrzebna).

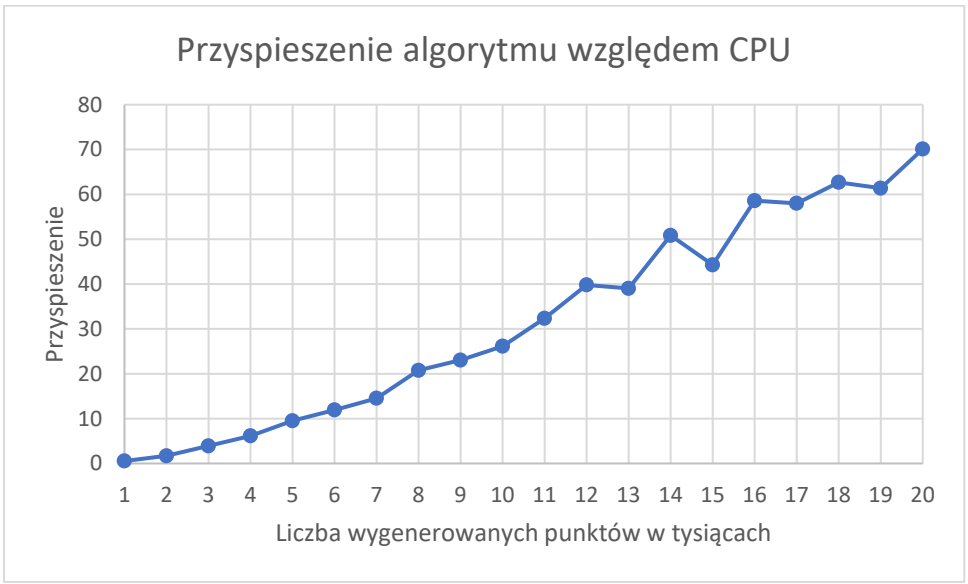
4. Wstęp - porównanie szybkości obliczania całki (wszystkie kroki) pomiędzy CPU a GPU.





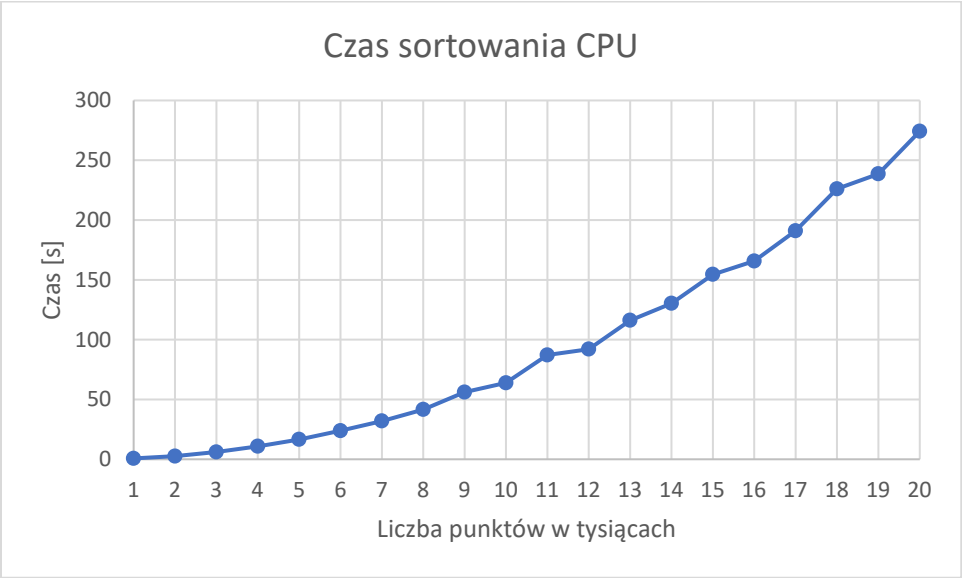
Na podstawie wykresów widać znaczne przyspieszenie dzięki wykonywaniu obliczeń na GPU, w tym ćwiczeniu eksperymentalnie wybrałem najprostsze algorytmy w celu sprawdzenia jak bardzo karta graficzna potrafi pomóc nawet przy najłatwiejszych rozwiązaniach. Punkty, na których została policzona całka zostały wygenerowane losowo i zawierały się w przedziale (0,100 000). Z powodu prostych algorytmów – na CPU liczenie sekwencyjnie wykonuje się dość długo, więc testy zostały wykonane na liczbie punktów od 1 do 20 tysięcy.

Zastosowałem również 10-krotną interpolację liniową, stąd po interpolacji liczba punktów wzrasta do 10*(n-1), gdzie n jest ich początkową ilością.

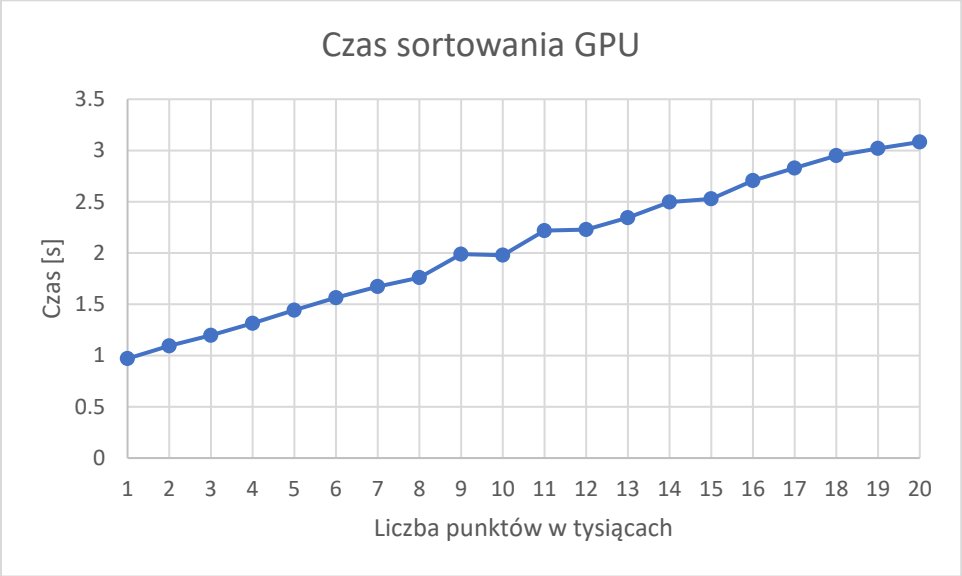


Po obejrzeniu wykresów widać oczywiście astronomiczne przyspieszenie. Przy 20 tysiącach punktów jest aż 80-krotne. Tak jak na innych ćwiczeniach, w tym wypadku również początkowe przyspieszenie jest znikome, GPU najbardziej przydaje się do operacji na dużej ilości danych. Dalsze wnioski rozwiną temat znacznego przyspieszenia na karcie graficznej.

5. Porównanie szybkości sortowania bąbelkowego

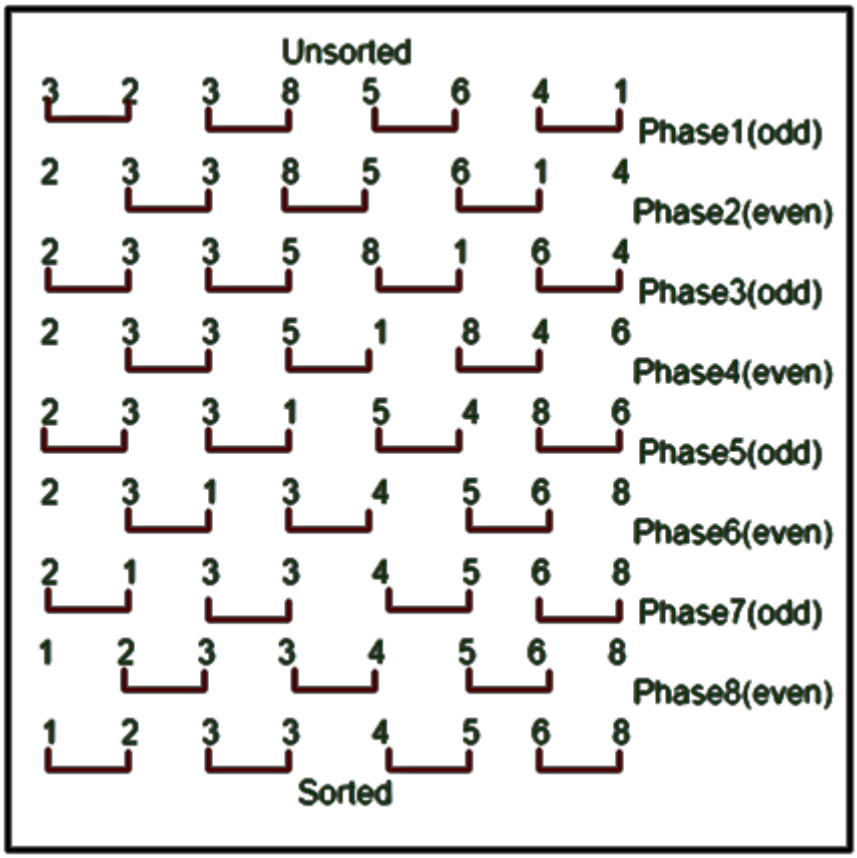


Sortowanie bąbelkowe jest podstawowym algorytmem do sortowania danych, jego złożoność czasowa to typowym, średnim przypadku $O(n^2)$, a w najlepszym $O(n)$. Nie są to zadowalające wyniki i na wykresie możemy zaobserwować tę zależność.



Sortowanie bąbelkowe jest algorytmem, który ciężko zrównolegnić; jednak, żeby sprawozdanie było rzetelne wybrałem odd-even sort – algorytm, który posiada działanie bardzo podobne do sortowania bąbelkowego. Działanie sortowania odd-even polega na porównywaniu na przemian stojących obok siebie wartości w pętli, w fazach parzystych i nieparzystych. W ten sposób w najgorszym przypadku otrzymujemy złożoność czasową $O(n^2)$, a w najlepszym $O(n)$. Z tego powodu uznałem, że oba algorytmy możemy ze sobą porównywać.

Schemat działania odd-even sort:



Zrównolegnienie powyższego rysunku mogłoby wyglądać w następujący sposób w pseudokodzie:

For k = 0 to n-2

If k is even then

for i = 0 to (n/2)-1 do in parallel

If A[2i] > A[2i+1] then

Exchange A[2i] ↔ A[2i+1]

Else

for i = 0 to (n/2)-2 do in parallel

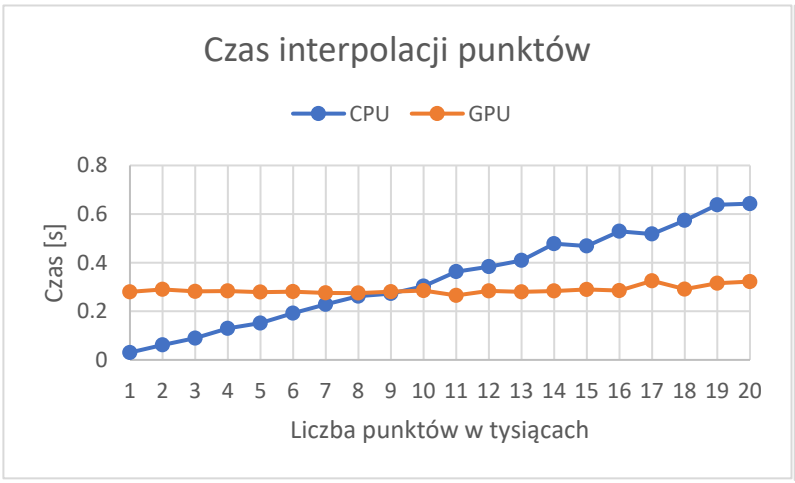
If A[2i+1] > A[2i+2] then

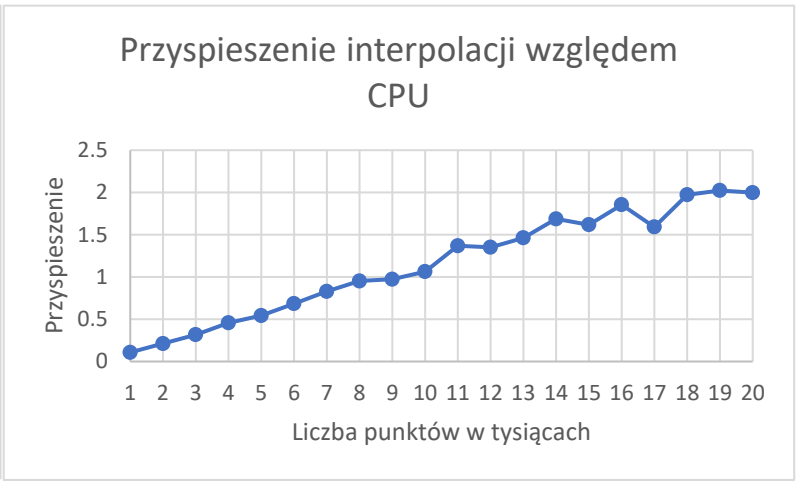
Exchange A[2i+1] ↔ A[2i+2]

Next k

Po odpowiednim zakodowaniu powyższego pseudokodu otrzymane wyniki pokazały jak bardzo karta graficzna potrafi przyspieszyć sortowanie. Sortowanie punktów było najbardziej znaczącą operacją w tym ćwiczeniu i to właśnie ona zajmowała większość czasu.

6. Porównanie szybkości interpolacji liniowej punktów

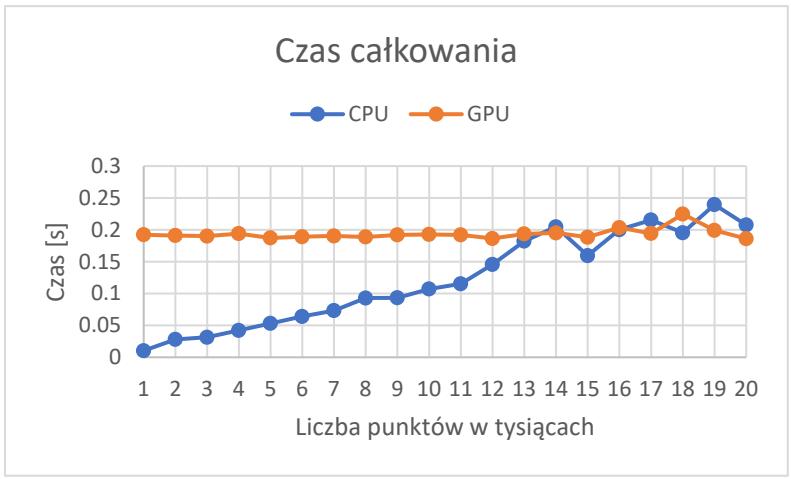


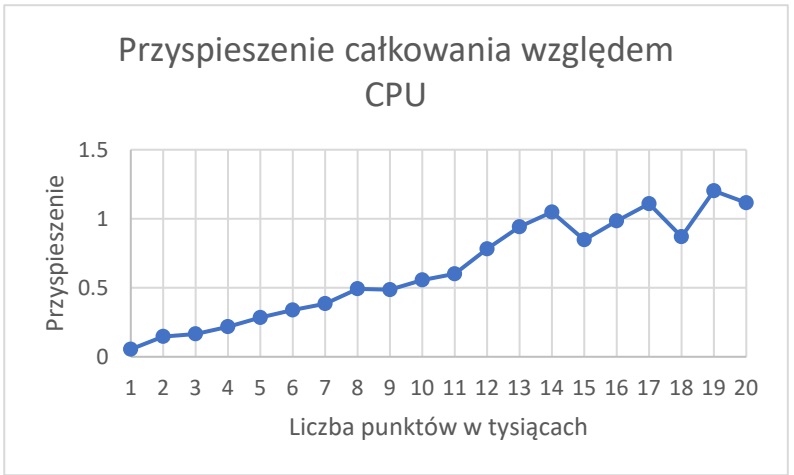


Interpolacja punktów na karcie graficznej wykonuje się właściwie zawsze w tym samym czasie dla takiej ilości punktów – wyliczanie nowych punktów odbywa się równocześnie na każdym wątku w ilości 10*(n-1), dlatego dla każdego punktu jest dostępne miejsce na obliczenia.

Podobnie sytuacja wygląda podczas całkowania.

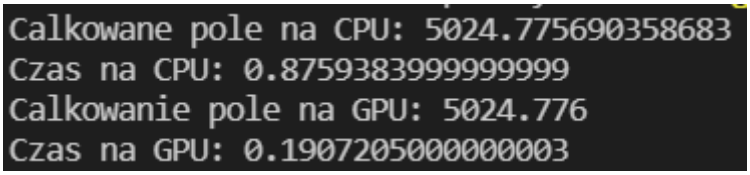
7. Porównanie szybkości całkowania – metoda trapezów





Z powodu alokacji odpowiedniej liczby bloków i wątków na GPU równoczesne obliczanie, a następnie dodawanie do siebie obliczonej listy całek jest szybkie, dla przedstawionych liczb wykonuje się praktycznie w tym samym czasie 0.2 sekundy. Dodawanie odbywa się za pomocą atomicAdd. Podczas pisania programów warto zastanowić się, czy planujemy operować na odpowiednio dużej ilości danych. Na powyższych wykresach opłacalność wykonywania całkowania na karcie graficznej pojawia się dopiero przy 14 tysiącach punktów.

Ewentualne błędy pomiędzy wartością obliczonej całki na GPU a CPU są niewielkie:



Przykład błędu wartości całkowanego pola. Błędy wynikają z zaokrąglania, podczas operacji całkowania na GPU liczby zostają przekonwertowane z float64 na float32, a następnie dodane. Nie ma to większego wpływu na czas obliczeń, jednak dzięki temu możemy wywnioskować, że zawsze lepiej przetestować i porównać nasz program z jego odpowiednikiem na CPU w celu sprawdzenia precyzji oraz jej korekcji w zależności od potrzeb.

8. Wnioski

Wszystkie wykresy podkreślają jak GPU sprawdza się do obliczania większej ilości danych. Biblioteka numba do obsługi CUDA w języku Python jest zdecydowanie dobrym rozwiązaniem oraz bardzo wygodnym dla osób, które dopiero zaczynają programowanie na kartach graficznych. Spodziewałem się o wiele gorszych wyników czasowych spowodowanych połączeniem prostych algorytmów i programowania w Pythonie.

9. Przemyślenia - pomysły na poprawę szybkości na CPU/GPU, napotkane trudności.

Programowanie równoległe wciąż wydaje mi się trudnym konceptem i ciężko mi ‘przestawić się’ z sekwencyjnego podejścia. Wydaje mi się jednak, że dobrze poradziłem sobie z postawionymi założeniami.

Po napisaniu programu myślę, że w celu poprawy szybkości zdecydowanie zmieniłbym algorytmy sortujące na takie, które w pełni wykorzystują zjawisko zrównoleglenia (QuickSort, BitonicSort).