PROJEKT K-means Clustering z wykorzystaniem CUDA

K-means Clustering służy do grupowania podobnych do siebie punktów w zbiorze danych. Wykorzystuje w tym celu prosty algorytm:

1. Losowo wybieramy k punktów - “znaczniki”. Następnie przypisujemy każdemu punktowi jego grupę (1,2…,k). Punkt należy do grupy l, jeśli jego odległość do znacznika l jest najmniejsza spośród k znaczników.
2. Obliczamy centroid każdej grupy (środek masy punktów)
3. Ponownie przypisujemy każdemu punktowi ze zbioru jego grupę, wg. zasady w punkcie 1.
4. Powtarzamy krok 2,3 aż obliczone centroidy w punkcie 2. nie będą się zmieniać.

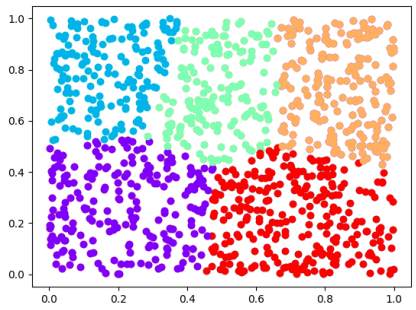
1. Wizualizację i generację punktów zrobiłem przy użyciu skryptu Python.
2. Na początku napisałem K-means w cpp. Zaimplementowałem powyżej opisany algorytm. Być może małym usprawnieniem, jest liczenie średniej podczas przypisywania punktowi grupy. Tzn. trzymamy specjalną tablicę w której sumujemy wartości poszczególnych współrzędnych i utrzymujemy liczbę punktów w danej grupie. Następnie w prosty sposób obliczamy średnią arytmetyczną.
3. Kolejnym etapem było zaimplementowanie K-means z wykorzystanie CUDA. Na GPU przeniosłem klasyfikowanie punktów, oraz uaktualnianie nowych centroidów.
    a. Klasyfikowanie punktów - był to najprostszy i najbardziej naturalny element to zrównoleglenia. Każdy thread obsługuje jeden punkt i uaktualnia jego grupę poprzez porównanie odległości do innych centroidów. Musiałem stworzyć tyle bloków, aby mieć pewność, że każdy punkt otrzyma swój thread (num_of_blocks = (points_num + num_of_threads - 1) / num_of_threads - jest to po prostu sufit ([points_num / num_of_threads]) ((n-1)/m + 1) = sufit(n/m)
    b. aby nie czytać z pamięci globalnej współrzędnych centroidów, utrzymuję w lokalnej pamięci bloku współrzędne K centroidów
    c. Następnie uaktualnia globalne przypisanie grupy dla danego punktu
    d. Kolejnym etapem jest obliczenie centroidów. W tym celu użyłem redukcji. Jednak nie jest to klasyczny problem sumowania tablicy na GPU, ponieważ chcemy zsumować tylko współrzędne w obrębie danego klastra.
    e. Moje podejście polegało na tym, aby w każdym bloku zsumować współrzędne wszystkich punktów w obrębie tego bloku, z zachowaniem podziału na klastry. Pod koniec otrzymamy tablicę o rozmiarze CLUSTERS_NUM * BLOCKS_NUM, gdzie każdym elementem jest krotka (x,y,count). W ten sposób każdy blok zapisze swoją sumę współrzędnych dla danej grupy, oraz liczbę punktów w tej grupie. Na koniec wystarczy dodać odpowiadające wartości z różnch bloków i otrzymamy środek masy dla klastra. Na obrazku poniżej widać efekt końcowy - każdy blok (jest ich 4) ma zapisaną krotkę (x,y,count) dla każdej grupy.

    f. Aby uzyskać taki efekt, każdy blok sumuje wartości punktów poprzez redukcję. Ponieważ chcemy sumować wartości w obrębie klastru,

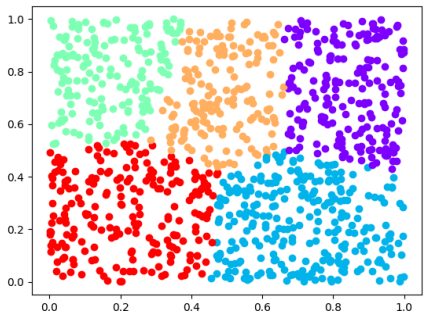
przeprowadzamy redukcję K razy (gdzie K to liczba klastrów). Wystarczy jedynie uaktualnić lokalną pamięć bloku dla każdej grupy:

```
temp[3 * local_tid] = (assigned_cluster == c) ? x : 0;
temp[3 * local_tid + 1] = (assigned_cluster == c) ? y : 0;
temp[3 * local_tid + 2] = (assigned_cluster == c) ? 1 : 0;
__syncthreads();
```

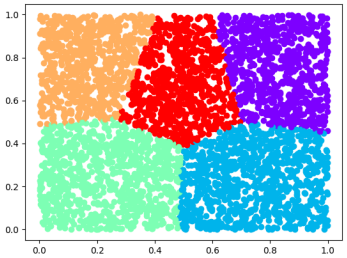
   g. (Używam spłaszczonej tablicy, x,y to współrzędne, a 3 wartość to 1,0 czyli count). Na koniec thread o lokalnym id 0 zapisuje sumy z każdego klastra w odpowiedniej globalnej tablicy (new_markers o wielkości NUM_CLASTERS*NUM_BLOCKS)
   h. Kolejnym elementem jest zsumowanie wartości z każdego bloku. Wykorzystuję w tym celu osobny kernel, który ma 1 blok i wątków tyle ile grup (K). Każdy wątek sumuje wartości grupy odpowiadającej jego indeksowi. Następnie uaktualnia globalne centroidy (markers).
4. Wszystkie te kroki wykonujemy max-iter liczbę razy.
5. Stworzyłem również makefile - poleceniem make wykonujemy cały eksperyment, wg wartości parametrów zdefiniowanych w Makefile. Aby zwizualizować punkty, należy wykonać polecenie make plot_data
6. Poniżej zamieszczam przykładowe wyniki (z wizualizacją dla małej ilości punktów - żeby miało to sens)



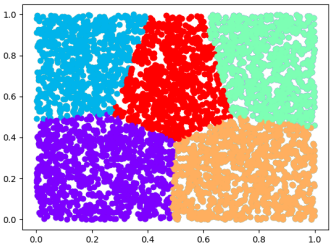
Pogrupowanie 1k punktów z CPU



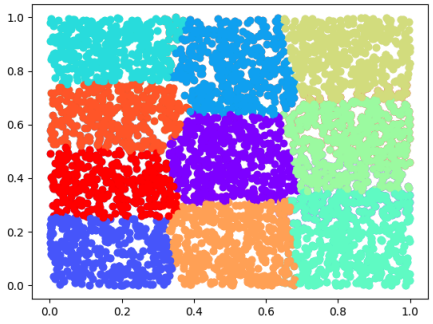
Pogrupowanie 1k punktów z GPU



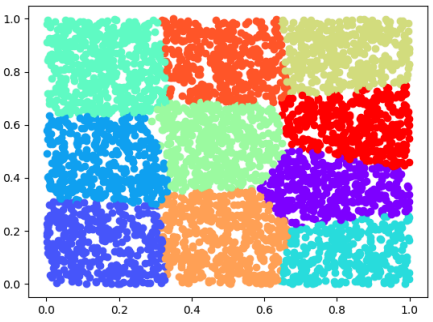
Pogrupowanie 4k punktów z CPU



Pogrupowanie 4k punktów z GPU



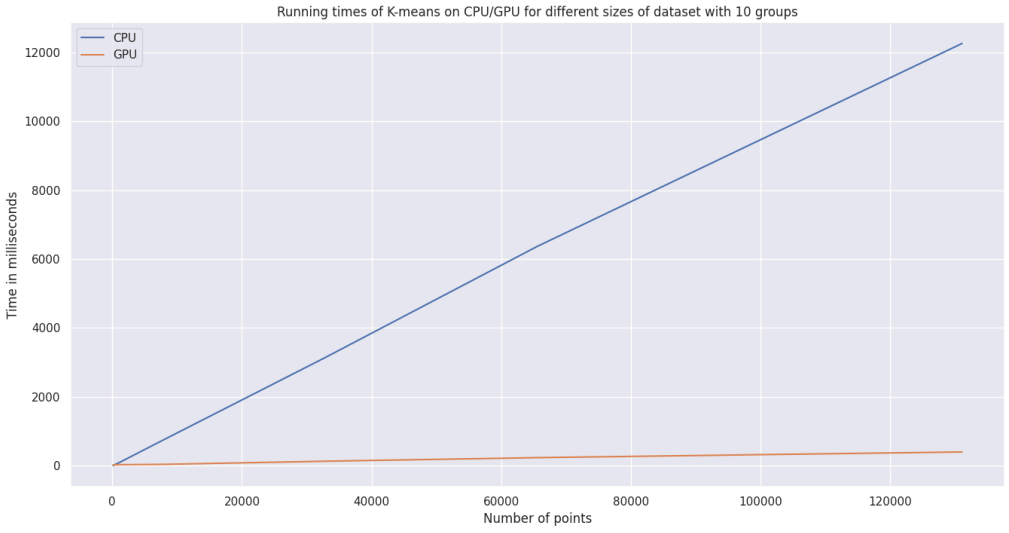
CPU z 10 grupami i z 4k punktów



GPU z 10 grupami i z 4k punktów

Wyniki eksperymentów:
Liczba grup: 5

Liczba grup: 10



Liczba grup: 100

Wnioski:
Dla niewielkiej liczby punktów (~100) różnice w czasie są niezauważalne, a nawet przemawiają na korzyść k-means na CPU. Spowodowane jest prawdopodobnie uruchamianiem kerneli i transferem pamięci. Jednak przy większej ilości punktów i grup, GPU zdecydowanie dominuje.

Przy dużej liczbie punktów (~100 000) punktów działa 30 krotnie szybciej. Przy trudnym problemie (100 grup - bardzo duża złożoność 100^liczbapunktów) program na CPU działał ponad 20 sekund, podczas gdy GPU poniżej sekundy. Warto zauważyć, że czas działania na GPU rośnie zdecydowanie wolniej, niż czas działania na CPU.