

# Implementacja algorytmu K-Means na GPU

## CUDA implementation of the K-Means algorithm

Mateusz Markiewicz

Styczeń 2021

## 1 Wstęp

Celem projektu jest efektywnie zaimplementowanie algorytmu K-Means na karcie graficznej, oraz porównanie czasu działania tej wersji do standardowej implementacji na CPU.

## 2 Algorytmy grupowania danych

Grupowanie danych to klasyczny problem w eksploracji danych, ma on wiele zastosowań, stąd powstało wiele algorytmów, które próbują go rozwiązywać. Problem polega na przypisaniu każdego punktu do pewnej grupy w taki sposób, by punkty wewnętrz jednej grupy były do siebie jak najbardziej podobne, a pomiędzy różnymi grupami było od siebie jak najbardziej różne.

Jednym z podstawowych i najprostszych algorytmów rozwiązujących ten problem jest K-Means, który jest jednocześnie jednym z najpopularniejszych algorytmów do grupowania danych.

## 3 K-Means

Mając  $n$  obserwacji chcemy wyznaczyć dokładnie  $k$  grup, a następnie przypisać każdą obserwację do dokładnie jednej grupy. Grupa jest reprezentowana przez swój środek, a dokładniej centroid. Algorytm stara się minimalizować średnią odległość między punktami, a centroidami grup, do których punkty te należą.

Algorytm ma postać iteracyjną i zazwyczaj jest inicjowany w sposób losowy.

- Wyznacz losowo początkowe centroidy
- Dopóki przydział obserwacji do grup się zmienia powtarzaj:
  - wyznacz odległości między każdą z obserwacji, a każdym z centroidów
  - przypisz każdą obserwację do grupy, której centroid jest najbliższy
  - wyznacz centroid każdej z grup

## 4 Początkowe założenia

Postanowilem wykorzystać wyżej opisany algorytm grupowania do kompresji obrazów. Mając obraz składający się z  $m \times n$  pikseli, gdzie każdy piksel to trójką liczb całkowitych z zakresu 0 – 255 staramy się przyporządkować każdy piksel do jednej grupy oraz wyznaczyć kolor najlepiej opisujący tę grupę (wspólny dla wszystkich jej członków) w ten sposób, by końcowy obraz był jak najbardziej podobny do oryginału. Dzięki temu każdy piksel może być reprezentowany za pomocą jednej wartości całkowitej z zakresu 0 –  $K$  oraz każdy z  $K$  centroidów reprezentowany jest jako trójką liczb całkowitych z zakresu 0 – 255. Jeśli  $K$  nie jest duże możemy w ten sposób uzyskać skompresowany obraz (zakładając naturalną reprezentację jako pełna macierz, w rzeczywistości wiele sąsiednich pikseli będzie miała tę samą wartość, stąd możliwe jest zastosowanie bardziej zaawansowanej reprezentacji takiej macierzy, która pozwoli jeszcze mocniej skompresować dane).

Na tej podstawie założyłem, że danej wejściowej mają rozmiar  $m \times n \times 3$ , a każda z tych wartości to uint8. Dodatkowo założyłem, że  $K$  nie może być większe od 255, by mógł zapisać ją również jako uint8.

## 5 Implementacja CPU

Do wczytania oraz wyświetlenia obrazu wykorzystuję bibliotekę openCV.

Wersja na CPU jest prostą implementacją opisanego wyżej algorytmu. Przebiega ona następująco:

- Dla każdej z  $K$  grup losuję 3 wartości całkowite z zakresu 0 – 255 jako początkowe centroidy. Co ważne początkowe centroidy są takie same dla wersji na CPU oraz GPU.
- Do momentu ustabilizowania grup lub do czasu przekroczenia maksymalnej liczby iteracji (ustawionej domyślnie na 100) powtarzamy następujące czynności:
  - Dla każdego punktu wyznaczamy grupę, której centroid znajduje się najbliżej. Jako miarę odległości wykorzystujemy kwadrat odległości euklidesowej dwóch 3-wymiarowych punktów. Jeśli dla dowolnego punktu wyznaczona grupa różni się od poprzedniej ustawiamy flagę mówiącą, że grupy nie są jeszcze ustabilizowane.
  - Wartość każdego piksela dodaję do sumy pikseli w grupie, do której ten piksel przynależy. Dodatkowo zwiększam licznik ilości członków tej grupy
  - Dla każdej grupy dzielę sumę wartości pikseli do niej należących przez ich ilość. Jeśli grupa jest pusta na nowo losuję wartość jej centroidu.

## 6 Implementacja GPU

Ogólny schemat implementacji na GPU nie różni się od tej dla CPU. 3 główne części powtarzane w pętli realizowane są za pomocą 3 osobnych kerneli.

### 6.1 Przyporządkowanie do grup

Stworzyłem 2 wersje tego kernela. Pierwsza z nich nie korzysta z shared memory. Każdy thread zajmuje się pojedynczym pikselem, wyznacza dla niego najbliższą grupę poprzez obliczenie K odległości i wybraniu najmniejszej z nich. Jednocześnie jeśli znaleźliśmy mniejszą odległość, niż ta początkowa oznacza to, że zmieniamy przyporządkowanie tego piksela przez co ustawiamy odpowiednią flagę. Odległość obliczania jest w taki sam sposób, jak na CPU.

Druga wersja wykorzystuje shared memory. Kernel zakłada, że każdy blok ma co najmniej K threadów. W pierwszym etapie thready o indeksach mniejszych, niż K zapisują informacje o centroidzie K-tej grupy do shared memory. Dodatkowo dla każdej grupy zapisujemy sumę kwadratów wartości RGB centroidu. Następnie wykonujemy takie same obliczenia jak w poprzedniej wersji, korzystamy jednak z innej metody obliczania dystansu. Poprzednia polegała na obliczaniu:

$$(x_1 - y_1)^2 + (x_2 - y_2)^2 + (x_3 - y_3)^2 = \\ x_1^2 + x_2^2 + x_3^2 + y_1^2 + y_2^2 + y_3^2 - 2x_1y_1 - 2x_2y_2 - 2x_3y_3$$

Teraz możemy zaważyć, że wartość  $x_1^2 + x_2^2 + x_3^2$  nie zależy od centroidu (reprezentowanego przez y), stąd może zostać pominięta. Wartość  $y_1^2 + y_2^2 + y_3^2$  została wcześniej obliczona i zapisana do shared memory. Dzięki temu możemy ograniczyć się do wyznaczania wartości  $2x_1y_1 - 2x_2y_2 - 2x_3y_3 + \text{suma kwadratów } y$ .

### 6.2 Sumowanie pikseli wewnętrz grup

Ta część algorytmu była najczęściej do zrównoleglenia. Pierwszym pomysłem było wykonanie sumy za pomocą redukcji. Nie chcemy jednak sumować wszystkich pikseli do jednej zmiennej, chcemy, by piksele wewnętrz poszczególnych grup były zsumowane do odpowiednich zmiennych (elementów tablicy). Można to zrobić za pomocą K-krotnego wykonania redukcji. Pomimo, że sumowanie za pomocą redukcji jest szybkie, to wykonanie go K-krotnie okazuje się czasochłonne.

Moim drugim pomysłem było wykorzystywanie jedynie 4 threadów z każdego bloku. Pierwsze 3 thready były odpowiedzialne za sumowanie wartości RGB pikseli o indeksach należących do danego bloku. 4-ty thread zliczał elementy poszczególnych grup (w takim samym zakresie). Na końcu za pomocą atomicAdd zliczone wartości są sumowane do tablic wynikowych, wspólnych dla wszystkich bloków. Pomimo, że wiele threadów w takim podejściu staje się bezużyteczne, podejście to okazało się szybsze.

### 6.3 Obliczanie centroidów na podstawie sum

Etap ten był realizowany za pomocą jednego bloku składającego się z K thredów. I-ty z nich dzielił sumę wartości pikseli przez ich ilość dla i-tej grupy. Jeśli suma była zerowa wartości pikseli dla tej grupy ustawiane były losowo. Wykorzystywałem w tym celu bibliotekę curand.

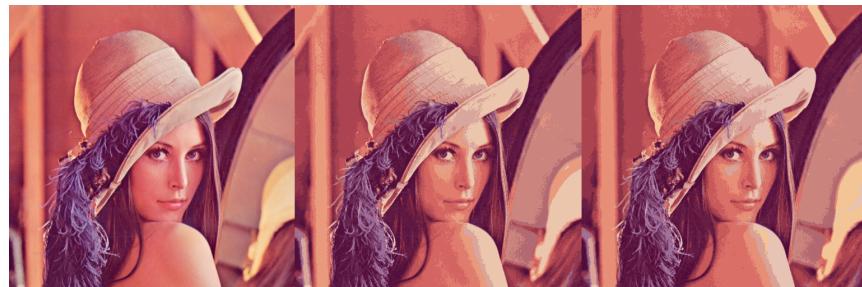
## 7 Rezultaty

Wersja GPU działa zauważalnie szybciej od wersji CPU. Różnica jest tym większa, im większe są rozmiary obrazu oraz wartość parametru K. Poniżej przedstawione są pomiary dla różnych obrazów i wartości K. Pomiary są średnimi z kilku uruchomień algorytmu. Obrazy przedstawiają kolejno wersję oryginalną, wynik algorytmu na CPU oraz wynik algorytmu na GPU. Pomiary wykonane były na procesorze Intel I7-6700HQ oraz karcie graficznej NVIDIA GTX 960M.

### 7.1 Obraz $512 \times 512$ pikseli

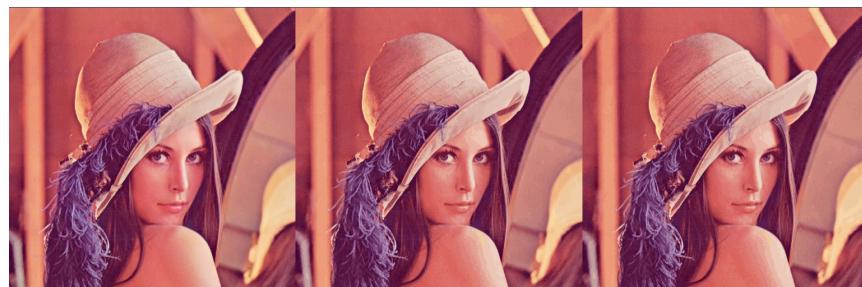
- $K = 16$

- czas dla wersji CPU: 7.25s
- czas dla wersji GPU: 0.8s (9 razy szybciej)

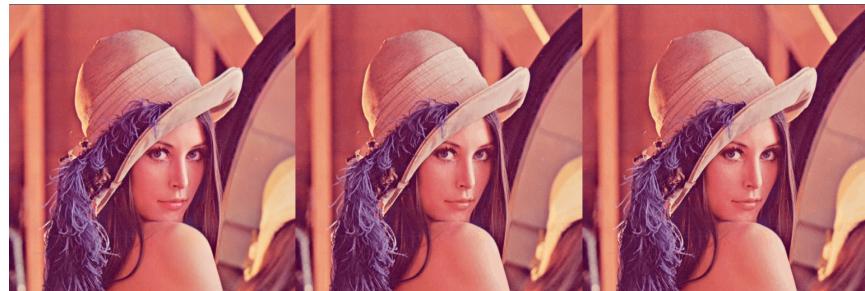


- $K = 128$

- czas dla wersji CPU: 122.7s
- czas dla wersji GPU: 7.7s (16 razy szybciej)



- $K = 255$ 
  - czas dla wersji CPU: 256.7s
  - czas dla wersji GPU: 14.3s (18 razy szybciej)



## 7.2 Obraz $1024 \times 1024$ pikseli

- $K = 16$ 
  - czas dla wersji CPU: 30s
  - czas dla wersji GPU: 2.7s (11 razy szybciej)



- $K = 128$ 
  - czas dla wersji CPU: 510s
  - czas dla wersji GPU: 28.2s (18 razy szybciej)



- $K = 255$ 
  - czas dla wersji CPU: 1011s
  - czas dla wersji GPU: 50.6s (20 razy szybciej)



## 8 Źródła

- K-means Clustering: Algorithm, Applications, Evaluation Methods, and Drawbacks - Imad Dabbura, Towards Data Science
- k-means clustering - Wikipedia