DOI: 10.24427/978-83-66391-84-0_1

Rozdział 1
Algorytm k-średnich na wielordzeniowych procesorach CPU oraz akceleratorach GPU

Tomasz Kuczyński

Streszczenie: Celem procesu grupowania jest podział zbioru danych na klasy o podobnych cechach. Klasy te nazywane są klastrami (skupieniami). Grupowanie dużych zbiorów danych jest procesem czasochłonnym. Jednym z najczęściej wykorzystywanych algorytmów w procesie grupowania danych jest algorytm k-średnich początkowo zaproponowany przez Stuarta Lloyda. Algorytm Lloyda nie jest algorytmem najszybszym. Istnieją algorytmy dające taki sam wynik, po takiej samej liczbie iteracji, ale w dużo krótszym czasie. Główną zaletą algorytmu k-średnich jest dość prosta implementacja. W celu przyspieszenia procesu grupowania danych warto jest wykorzystywać akceleratory GPU, gdyż ich moc obliczeniowa jest większa niż moc obliczeniowa procesorów CPU. W przypadku wykorzystania nowoczesnego akceleratora GPU czas obliczeń może być znacznie krótszy w porównaniu do obliczeń z wykorzystaniem współczesnego procesora CPU (nawet kilkanaście-kilkadziesiąt razy szybciej). Jednym z najpopularniejszych rozwiązań jeżeli chodzi o programowanie układów graficznych jest CUDA (ang. *Compute Unified Device Architecture*). Architektura CUDA opracowana przez firmę NVIDIA i zaprezentowana po raz pierwszy w 2006 roku umożliwia wykorzystanie akceleratorów GPU do obliczeń ogólnego przeznaczenia. W rozdziale zaprezentowane zostaną wyniki przeprowadzonych eksperymentów. Porównane zostaną ze sobą zarówno wyniki obliczeń z wykorzystaniem procesora CPU (algorytm Lloyda oraz algorytmy przyspieszone nierównością trójkąta) jak i wyniki uzyskane przy użyciu akceleratora GPU (algorytm Lloyda).

Słowa kluczowe: algorytm k-średnich, grupowanie, CUDA, nierówności trójkąta

Wprowadzenie

Grupowanie [1] jest jedną z technik uczenia bez nadzoru. Problem grupowania polega na podziale danych na grupy (klastry), które zawierają elementy podobne do siebie. Najpierw dokonywany jest wstępny podział danych. Następnie uzyskany podział jest modyfikowany w taki sposób, że część elementów jest przenoszona do innych grup, tak, aby uzyskać minimalną wariancję wewnątrz każdej grupy. Dąży się do tego, aby podobieństwo elementów w obrębie każdej grupy było jak największe, ale przy jednoczesnym zapewnieniu możliwie maksymalnej różnicy pomiędzy

grupami. Metody grupowania danych można podzielić na dwie kategorie: metody hierarchiczne oraz metody niehierarchiczne. Jedną z metod niehierarchicznych jest metoda k-średnich (ang. *k-means*). Zasadniczą różnicą pomiędzy metodami hierarchicznymi i niehierarchicznymi jest to, iż w przypadku metod niehierarchicznych konieczne jest wcześniejsze podanie liczby grup. Grupowanie danych znajduje szerokie zastosowanie w wielu różnych obszarach (dziedzinach) takich, jak biologia, medycyna, ekonomia, przetwarzanie obrazów.

Za jednorodność elementów w każdej grupie odpowiedzialna jest odpowiednia funkcja oceny. Istnieje wiele różnych funkcji oceny, które mogą być miarą jakości grupowania. Najczęściej stosowana jest funkcja nazywana sumą błędów kwadratowych SSE (ang. *sum of squared error*), którą można zdefiniować jako sumę kwadratów odległości pomiędzy elementami w zbiorze danych a centroidami, do których te elementy są przypisane. Najczęściej stosowaną funkcją odległości jest odległość euklidesowa.

Algorytm k-średnich [2, 3] jest bardzo popularny i powszechnie używany. Dla danego zbioru danych oraz rozwiązań początkowych (centroidów) algorytm w kolejnych iteracjach generuje coraz lepsze rozwiązania, w których wartość SSE jest coraz mniejsza. Iterację algorytmu można podzielić na dwie główne fazy: fazę przypisania oraz fazę obliczeń centroidów. W fazie przypisania każdy element zbioru danych zostaje przypisany do najbliższego centroida. Następnie, w fazie obliczeń centroidów, są one przeliczane jako wartość średnia z przypisanych wektorów uczących. Poszczególne fazy algorytmu są powtarzane, zanim nie zostanie osiągnięte odpowiednie kryterium stopu. Najczęstszym kryterium zbieżności jest z góry przyjęta liczba iteracji lub krok, w którym nie zmieniła się już przynależność elementów zbioru danych do centroidów.

Głównym elementem niniejszej pracy jest porównanie kilku wybranych algorytmów (punkt 1.3). Wykorzystując wielordzeniowy procesor CPU [4, 5] obliczenia przeprowadzono dla następujących algorytmów: Lloyda [2], pierścienia (annulus) [6], Yinyang [7]. Obliczenia z wykorzystaniem akceleratora GPU [8, 9] wykonano dla algorytmu Lloyda. Pozostała część niniejszej pracy wygląda nastepująco. W punkcie 1.1 opisano algorytm k-średnich (algorytm Lloyd'a) oraz omówiono techniki przyspieszania algorytmów k-średnich. Opis platformy testowej wykorzystywanej podczas przeprowadzania eksperymentów zawarto w punkcie 1.2. Pracę zakończono podsumowaniem.

1.1. Algorytm k-średnich

Algorytm k-średnich jest klasycznym algorytmem grupowania danych wynalezionym przez Stuarta Lloyda w 1957 roku i opublikowanym przez niego w 1982 roku [2]. Algorytm Lloyda jest algorytmem siłowym (ang. *brute force*). W każdej iteracji musi zostać obliczona odległość pomiędzy każdym elementem danych (wektorem cech) a każdym centroidem (klastrem). Istnieją algorytmy, które wykorzystują nierówności trójkąta w celu pominięcia obliczeń części odległości. Pozwala to na znaczne skrócenie

czasu obliczeń. Algorytmami, które wykorzystują nierówności trójkąta, są np. algorytm pierścienia (annulus), Yinyang, Elkan, Drake. Należą one do metod, które wprowadzają górną granicę odległości do aktualnie przypisanego centroida oraz dolną granicę odległości do innych centroidów. Nierówności trójkąta umożliwiają aktualizację tych granic, co w rezultacie powoduje, że algorytm potrafi pominąć wiele niepotrzebnych obliczeń odległości, co znacznie wpływa na szybkość działania.

Wartość SSE obliczana jest wg następującego wzoru:

$$SSE = \sum_{i=1}^{N} d^2\left(x(i), c\left(a(i)\right)\right) \tag{1}$$

gdzie:
N – liczba wektorów cech,
d – odległość,
x – wektor cech,
c – centroid,
a – przypisanie wektora cech do klastra.

1.1.1. Algorytm Lloyda

W algorytmie Lloyda [2] (algorytm 1) możemy wyróżnić dwie fazy: fazę przypisania oraz fazę obliczeń centroidów. Fazy te są naprzemiennie wykonywane w poszczególnych iteracjach algorytmu. W fazie przypisania każdy wektor cech zostaje przypisany do najbliższego centroida. Natomiast w kolejnej fazie obliczane są współrzędne centroidów jako suma cech wektorów przypisanych do tego samego klastra (y) podzielona przez liczbę tych wektorów (z). Suma błędów kwadratowych (SSE) zmniejsza się wraz z każdą kolejną iteracją i algorytm dąży do minimum lokalnego. W przypadku, gdy nie zmieniała się już przynależność wektorów cech do centroidów, algorytm kończy swoje działanie. Możliwe jest również zakończenie pracy algorytmu w momencie osiągnięcia określonej liczby iteracji lub z wykorzystaniem kryterium minimalnego względnego ulepszenia SSE.

ALGORYTM 1. Pseudokod algorytmu Lloyda [10]

Dane: Centroidy początkowe $c(1), c(2), ..., c(K)$
i wektory cech $x(1), x(2), ..., x(N)$

```
1  repeat
2      [ faza przypisania ]
3      for i ← 1 to N do
4          a(i) ← arg_j min d(x(i)), c(j))
5      [ faza obliczeń centroidów ]
6      for j ← 1 to K do y(j) ← 0, z(j) ← 0
7      for i ← 1 to N do
```

8 $y(a(i)) \leftarrow y(a(i)) + x(i)$

9 $z(a(i)) \leftarrow z(a(i)) + 1$

10 **for** $j \leftarrow 1$ **to** K **do** $c(j) \leftarrow y(j) / z(j)$

11 **until KryteriumStopu**

Wynik: Centroidy końcowe $c(1)$, $c(2)$, ..., $c(K)$
i przypisania do klastrów $a(1)$, $a(2)$, ..., $a(N)$

Złożoność algorytmu Lloyda:
a) obliczeniowa: $O(iNKD)$ – wartość stała
b) pamięciowa: $O(ND)$
gdzie:
i – liczba iteracji,
N – liczba wektorów cech,
K – liczba centroidów,
D – wymiar

1.1.2. Algorytm pierścienia (annulus)

W algorytmie pierścienia [6], (algorytm 2) górna granica odległości $u(i)$ pomiędzy wektorem cech a przypisanym do niego centroidem musi spełniać następującą nierówność:

$$u(i) \geq \mathrm{d}(x(i), c(a(i))) \tag{2}$$

Natomiast dolna granica odległości $l(i)$ do drugiego najbliższego centroida jest dana nierównością:

$$l(i) \leq \min_{k \neq a(i)} \mathrm{d}(x(k), x(i)) \tag{3}$$

Zmienna (b) jest indeksem drugiego najbliższego centroida przypisanego do danego wektora cech. Jeśli $u(i) \leq l(i)$ to wtedy przypisania do klastrów się nie zmieniają i algorytm pomija najbardziej wewnętrzną pętlę iterującą po centroidach. W celu pominięcia najbardziej wewnętrznej pętli wykonywane są obliczenia wstępne (s) połowy odległości najbliższego centroida do innego najbliższego centroida – wykorzystywany jest tutaj lemat Philipsa [11]. Pętli nie da się pominąć, jeśli granice nakładają się na siebie i sprawdzenie lematu Philipsa kończy się niepowodzeniem. W takim przypadku brane pod uwagę są tylko centroidy, które znajdują się w obszarze pierścieniowym (J), a środkiem tego obszaru jest początek układu współrzędnych.

W fazie przypisania sprawdza się jednocześnie, czy obie granice nie nakładają się na siebie i lemat Philipsa. Jeśli sprawdzenie zakończy się niepowodzeniem, to wtedy zawęża się górną granicę odległości i sprawdza ponownie powyższe warunki. Jeśli sprawdzenie po raz kolejny się nie powiedzie, to przechodzi się do najbardziej wewnętrznej pętli przeszukującej pierścień.

Po zakończeniu fazy przypisania algorytm przechodzi do fazy aktualizacji (obliczeń) centroidów, a następnie przeliczane są granice. Zmienna (δ) oznacza wartość przesunięcia centroida w wyniku wykonania fazy aktualizacji centroidów.

ALGORYTM 2. Pseudokod algorytmu pierścienia (annulus), [10]

Dane: Centroidy początkowe $c(1), c(2), ..., c(K)$
i wektory cech $x(1), x(2), ..., x(N)$

for $i \leftarrow 1$ **to** N **do**
  $a(i) \leftarrow 1, b(i) \leftarrow 1$
  $u(i) \leftarrow \infty, l(i) \leftarrow 0$
repeat
  for $k \leftarrow 1$ **to** K **do** $s(k) \leftarrow \min_{j \neq k} d(c(j)), c(k))/2$
  for $i \leftarrow 1$ **to** N **do**
    $z \leftarrow \max(l(i), s(a(i)))$
    if $u(i) \leq z$ **then continue**
    $u(i) = \mathrm{d}(c(a(i)), x(i))$
    if $u(i) \leq z$ **then continue**
    $l(i) \leftarrow \mathrm{d}(x(i), c(b(i)))$
    $r \leftarrow \max(l(i), u(i))$
    $J \leftarrow \{k:|\mathrm{d}(x(i), 0) - \mathrm{d}(c(k), 0)| < \mathrm{r}\}$
    forall $k \in J$ **do**
      if $\mathrm{d}(x(i), c(k)) < u(i)$ **then**
        [nowy najbliższy centroid]
        $l(i) \leftarrow u(i), u(i) \leftarrow \mathrm{d}(x(i), c(k))$
        $b(i) \leftarrow a(i), a(i) \leftarrow j$
      else if $\mathrm{d}(x(i), c(k)) < l(i)$ **then**
        [nowy drugi najbliższy centroid]
        $b(i) \leftarrow j, l(i) \leftarrow \mathrm{d}(x(i), c(k))$
  $c' \leftarrow c, c \leftarrow$ **FazaObliczeńCentroidów**
  for $k \leftarrow 1$ **to** K **do** $\delta(k) = \mathrm{d}(c(k), c'(k))$
  $\delta' = \max \delta(k)$
for $i \leftarrow 1$ **to** N **do**
  $u(i) \leftarrow u(i) + \delta(a(i))$
  $l(i) \leftarrow l(i) + \delta'$
until KryteriumStopu

Wynik: Centroidy końcowe $c(1), c(2), ..., c(K)$
i przypisania do klastrów $a(1), a(2), ..., a(N)$

Złożoność algorytmu pierścienia (annulus):
a) obliczeniowa: $O(iNKD)$ – wartość pesymistyczna
b) pamięciowa: $O(ND + N)$

1.1.3. Algorytm Yinyang

W algorytmie Yinyang [7], (algorytm 3) górna granica odległości $u(i)$ pomiędzy wektorem cech a przypisanym do niego centroidem musi spełniać następującą nierówność:

$$u(i) \geq \mathrm{d}(x(i), c(a(i))) \tag{4}$$

Natomiast dolna granica odległości $l(i, k)$ pomiędzy wektorem cech a najbliższym centroidem z danej grupy, wyłączając z tego centroidy aktualnie już przypisane, jest dana nierównością:

$$l(i, k) \leq \min_{j \in \pi(k) \wedge j \neq a(i)} \mathrm{d}(c(j), x(i)) \tag{5}$$

ALGORYTM 3. Pseudokod algorytmu Yinyang [10]

Dane: Liczba grup B,
centroidy początkowe $c(1), c(2), ..., c(K)$
i wektory cech $x(1), x(2), ..., x(N)$

$\Pi \leftarrow$ **Partycja** $((c(1), ..., c(K)), B)$
for $i \leftarrow 1$ **to** N **do**
  $a(i) \leftarrow \mathrm{arg}_k \min \mathrm{d}(x(i)), c(k))$
  $u(i) \leftarrow \mathrm{d}(x(i), c(a(i)))$
  for $k = 1$ **to** B **do** $l(i, k) \leftarrow \min_{\{j: j \in \pi(k) \setminus a(i)\}} \mathrm{d}(x(i), c(j))$
repeat
  $c' \leftarrow c$, $c \leftarrow$ **FazaObliczeńCentroidów**
  for $j \leftarrow 1$ **to** K **do** $\delta(j) \leftarrow \mathrm{d}(c(j), c'(j))$
  for $k \leftarrow 1$ **to** B **do** $\Delta(k) \leftarrow \max_{j \in \pi(k)} \delta(j)$
  for $i \leftarrow 1$ **to** N **do**
    $u(i) \leftarrow u(i) + \delta(a(i))$
    $a' \leftarrow a(i)$
    for $k \leftarrow 1$ **to** B **do**

14 $l'(k) \leftarrow l(i, k)$

15 $l(i, k) \leftarrow l(i, k) - \Delta(k)$

16 $L \leftarrow \min_k l(i, k)$

17 **if** $u(i) \leq L$ **then continue**

18 $u(i) \leftarrow \mathrm{d}(x(i), c(a(i)))$

19 **if** $u(i) \leq L$ **then continue**

20 $\Pi \leftarrow \{k : l(i, k) < u(i)\}$

21 **FiltrowanieLokalne**

22 **until KryteriumStopu**

Wynik: Centroidy końcowe $c(1), c(2), ..., c(K)$
i przypisania do klastrów $a(1), a(2), ..., a(N)$

Na samym początku działania algorytmu centroidy zostają podzielone na grupy. Twórcy tego algorytmu zalecają, aby liczba takich grup (B) była nie większa niż $K/10$. Podział rozwiązań początkowych (centroidów) na założoną z góry liczbę grup uzyskuje się przez uruchomienie algorytmu k-średnich tylko na tych początkowych grupach przez pięć iteracji. Pozwala to stworzyć w miarę rozsądne grupy przy poniesieniu jednocześnie niewielkiego narzutu pracy algorytmu.

Następnie, w pierwszej iteracji algorytm dla każdego wektora cech przyporządkowuje górną granicę odległości $u(i)$ oraz dolną granicę odległości $l(i,k)$.

W kolejnym kroku algorytm wykonuje petlę główną, w której: jest dokonywana aktualizacja (obliczenia) centroidów, obliczane jest przesunięcie centroidów $\delta(j)$ oraz wykonywane są obliczenia maksymalnego przesunięcia dla każdej grupy $\Delta(k)$. W pętli iterującej po wszystkich wektorach cech jest aktualizowana górna oraz dolna granica odległości, a następnie, jeśli zajdzie taka potrzeba, to wykonywane są kolejne obliczenia oraz przyporządkowania wektorów cech do centroidów.

W przypadku gdy $u(i) \leq l(i, k)$ obliczenia odległości do wszystkich centroidów z danej grupy nie są konieczne, gdyż wtedy żaden z centroidów nie może znajdować się bliżej danego wektora cech niż centroid aktualnie przypisany. Jeśli $u(i) > l(i, k)$ wtedy algorytm wykonuje obliczenia górnej granicy odległości i jeszcze raz sprawdza nierówność $u(i) \leq l(i, k)$. Jeśli ponownie ta nierówność nie zostanie spełniona, wówczas należy wykonać kolejne obliczenia (filtrowanie lokalne). Algorytm wykonuje lokalne filtrowanie tylko w grupach, które nie spełniły nierówności $u(i) \leq l(i, k)$.

Złożoność algorytmu Yinyang:

a) obliczeniowa: $O(iNKD)$ – wartość pesymistyczna
b) pamięciowa: $O(ND + NB)$

gdzie:
B – liczba grup (najczęściej przyjmuje się $B = K/10$)

1.1.4. Rozwiązania początkowe

Wybór rozwiązań (centroidów) początkowych ma zasadniczy wpływ na efekt końcowy działania algorytmu k-średnich: liczbę iteracji oraz jakość rozwiązania końcowego (wartość SSE). Rozwiązania początkowe najczęściej generowane są w sposób losowy. Można również wykorzystać powszechnie znaną metodę inicjalizacji nazywaną K-means++ [12], która jest inteligentną techniką inicjalizacji centroidów.

Poszczególne kroki metody K-means++ wyglądają następująco:

1) Najpierw wybiera się pierwszego centroida jako losowo wybrany wektor cech ze zbioru danych.
2) Oblicza się kwadraty odległości wszystkich wektorów cech w zbiorze danych od najbliższego wcześniej wybranego centroida.
3) Wybiera się następnego centroida spośród wektorów cech w taki sposób, aby prawdopodobieństwo wyboru wektora cech jako centroida było wprost proporcjonalne do jego kwadratu odległości od najbliższego wcześniej wybranego centroida, tzn. wektor cech mający maksymalną odległość od najbliższego centroida najprawdopodobniej zostanie wybrany jako następny centroid.
4) Powtarza się kroki 2–3, aż znajdziemy wymaganą liczbę centroidów.

1.2. Platforma testowa

Teoretyczna moc obliczeniowa współczesnych akceleratorów GPU jest wielokrotnie większa niż moc obliczeniowa współczesnych procesorów CPU. W niektórych przypadkach obliczeń istnieje możliwość ich przyspieszenia, wykorzystując GPU nawet 100-krotnie względem CPU.

Wydajność dzisiejszego sprzętu (dla pojedynczej precyzji obliczeń), [13]:

a) CPU:
   - Intel i7-8700K CofeeLake – 170 Gflops
   - AMD Ryzen2 2700X Pinnacle Ridge – 198 Gflops
   - Intel i9-9900K CofeeLake-R – 236 Gflops
   - Intel i9-7900X SkyLake-X – 262 Gflops
b) GPU:
   - AMD Radeon RX 480 – 5.8 Tflops
   - AMD Radeon RX 5700 XT – 9.8 Tflops
   - NVIDIA Geforce GTX 1080 Ti – 11 Tflops
   - NVIDIA Geforce RTX 2080 Ti – 13.5 Tflops

Wniosek: Moc obliczeniowa GPU jest większa od mocy obliczeniowej CPU nawet kilkudziesięciokrotnie.

Wszystkie eksperymenty opisane w kolejnym punkcie niniejszej pracy zostały przeprowadzone na jednym węźle znajdującym się w klastrze należącym do Politechniki Białostockiej. Węzeł ten jest komputerem zbudowanym m.in. z procesora AMD Ryzen Threadripper 1950X (16 rdzeni, zegar 3,4 GHz) oraz akceleratora GPU GeForce RTX 2080 Ti (architektura Turing).

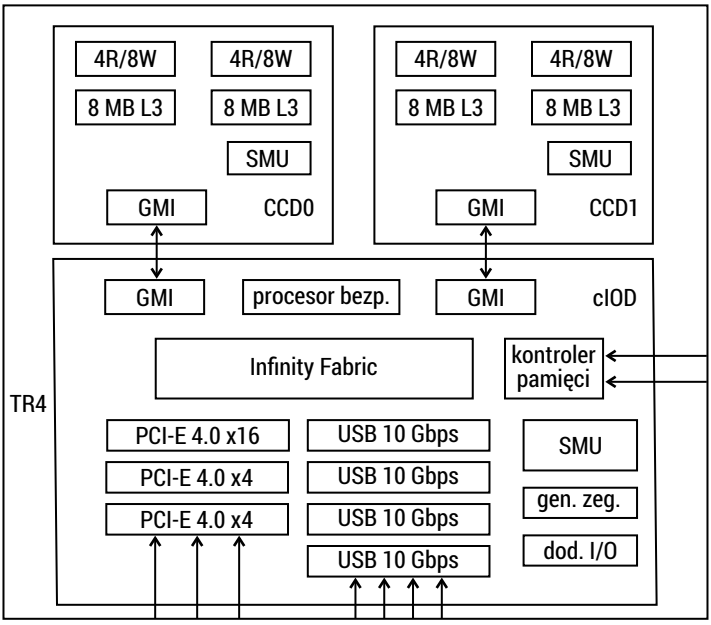
W przypadku obliczeń na procesorze CPU wykorzystano gotową implementację algorytmów autorstwa W. Kwedlo [10, 14]. Trzy algorytmy (Lloyda, pierścienia, Yinyang) zostały zaimplementowane w języku programowania C++ (zrównoleglenie w technologii OpenMP). Kompilacji dokonano przy użyciu kompilatora gcc w wersji 5.4.0.

Wykonując obliczenia z wykorzystaniem akceleratora GPU, bazowano na kodzie zawierającym różne algorytmy grupowania danych (projekt CAMPAIGN, rok 2010) [15]. Wyniki obliczeń oparte o kod projektu CAMPAIGN opublikowano w [16]. Projekt dotyczy generacji akceleratorów pre-Fermi. Algorytmy k-średnich wchodzące w skład projektu zaimplementowano w języku wysokiego poziomu CUDA C. Język ten jest rozszerzeniem języka C/C++ i jednym z najpopularniejszych rozwiązań, jeżeli chodzi o programowanie układów graficznych. Architektura sprzętowo-programistyczna CUDA opracowana przez firmę NVIDIA i zaprezentowana po raz pierwszy w 2006 roku umożliwia wykorzystanie akceleratorów GPU do obliczeń ogólnego przeznaczenia. CUDA jest wieloplatformowym (Windows, Linux, MacOS) i bezpłatnym środowiskiem programowania procesorów firmy Nvidia (kompilator, debuger, profiler, biblioteki) opartym na języku CUDA C. Spośród algorytmów k-średnich dostępnych w projekcie CAMPAIGN do dalszej pracy wybrano jeden algorytm (Lloyda). W kodzie wprowadzono szereg różnorodnych poprawek (np. redukcja warp-synchroniczna wymaga na architekturach >= Volta użycia funkcji syncwarp; usunięto ewidentne błędy typu: nie wszystkie wątki wołają syncthreads) mających na celu dostosowanie go do własnych potrzeb oraz architektur współczesnych akceleratorów GPU. Po wprowadzeniu niezbędnych poprawek dokonano kompilacji przy użyciu kompilatora nvcc w wersji 10.1 (CUDA v10.1).

1.2.1. Architektura procesorów wielordzeniowych na przykładzie procesora AMD Ryzen Threadripper 1950X

Procesor AMD Ryzen Threadripper 1950X jest zbudowany w architekturze Zen [17], (rys. 1.1). Funkcje procesora zostały podzielone pomiędzy dwa lub więcej jąder, które są wykonane w różnych technologiach produkcji. Najwięcej tranzystorów potrzebują rdzenie procesora oraz pamięć podręczna – umieszczono je w jądrach CCD (ang. *Core Chiplet Die*). Natomiast do osobnego jądra produkowanego w innym procesie technologicznym wydzielono części procesora powiązane z interfejsami wejścia/wyjścia, które zawierają dość dużo elementów pasywnych, znoszących wysokie napięcia, których powierzchni nie da się zmniejszyć przy wykorzystaniu najnowszych technik. W jednej obudowie procesora znajdują się dwa lub trzy jądra krzemowe. Procesor umieszczony jest w podstawce TR4. Wewnątrz CCD znajdują się dwa bloki CCX, a w każdym

z nich 4 rdzenie Zen oraz wspólna pamięć podręczna L3. Oprócz rdzeni i pamięci podręcznej znajduje się tam też mikrokontroler SMU (ang. *System Management Unit*) zarządzający zasilaniem, taktowaniem i danymi pochodzącymi z sensorów. Przesył danych do drugiego jądra umożliwia szerokie łącze GMI.



RYSUNEK 1.1. Architektura procesora AMD Ryzen Threadripper 1950X [17]

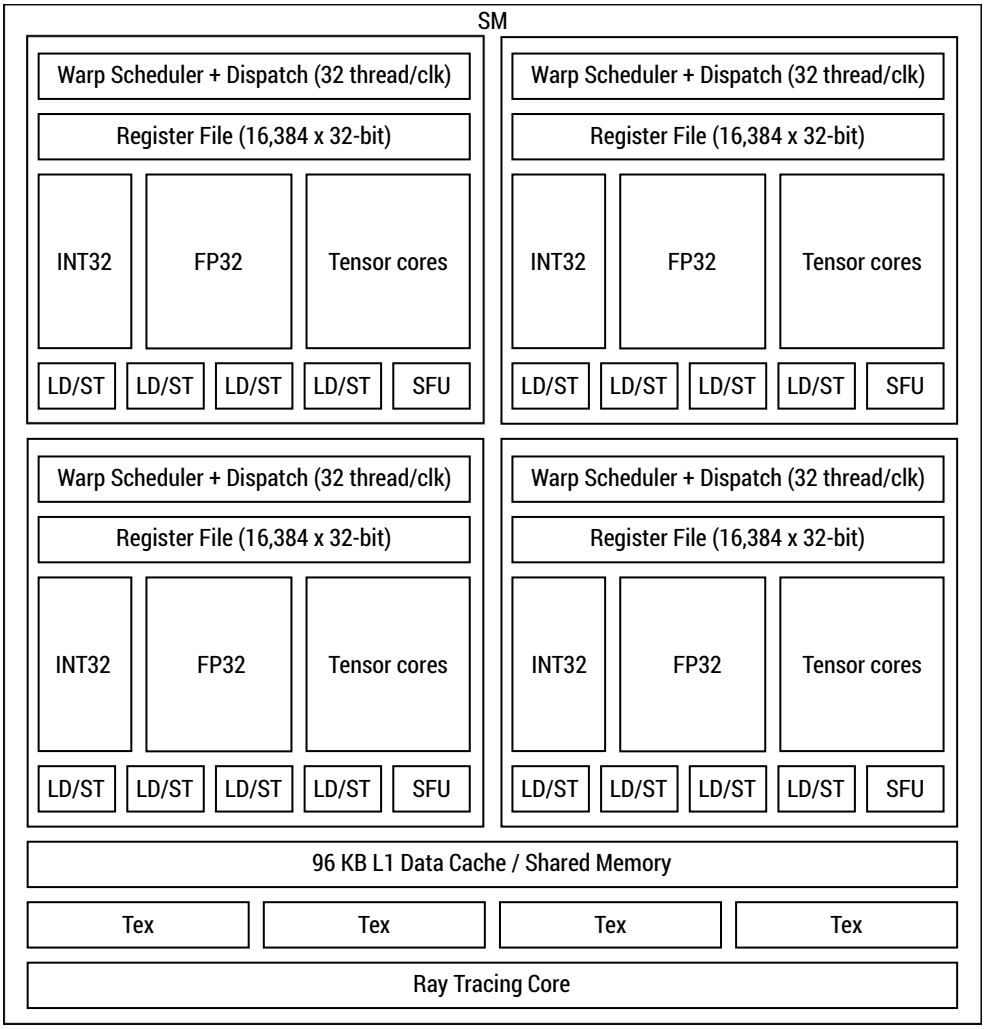
W jądrze cIOD (ang. *coherent I/O Die*) znajduje się kontroler pamięci i kontrolery interfejsów zewnętrznych: 2 x GMI (elektryczna implementacja protokołu Infinity Fabric) do komunikacji z jednym lub dwoma CCD, procesor bezpieczeństwa, 2-kanałowy kontroler pamięci DDR4, kontrolery PCI-E 4.0, kontroler USB 3.2 Gen2x1, SMU (kontroler zasilania i taktowania), generator sygnałów zegarowych, kontrolery innych interfejsów wejścia/wyjścia np. SMBus, eMMC.

1.2.2. Architektura akceleratorów GPU na przykładzie architektury Turing

Firma NVIDIA cały czas opracowuje nowe architektury GPU [18]. Przegląd architektur GPU opracowanych przez firmę NVIDIA oraz daty ich prezentacji przedstawiono poniżej:

- Tesla – listopad 2006
- Fermi – kwiecień 2010
- Kepler – kwiecień 2012
- Maxwell – luty 2014
- Pascal – kwiecień 2016

- Volta – grudzień 2017
- **Turing – wrzesień 2018**
- Ampere – maj 2020.



RYSUNEK 1.2. Schemat budowy bloku SM w architekturze Turing [18]

GPU posiada wiele multiprocesorów, a z kolei każdy multiprocesor ma 64 lub 128 rdzeni CUDA (liczba tych rdzeni jest zależna od architektury GPU np. Turing – 64, Ampere – 128), jednostki przetwarzające specjalnego przeznaczenia i pamięć współdzieloną. Każdy z rdzeni znajdujących się w multiprocesorze jest procesorem wielowątkowym. Poniżej umieszczono opisy poszczególnych jednostek wchodzących w skład bloku SM w architekturze Turing, znajdujących się na rysunku 1.2:

- INT32 – jednostka matematyczna wykonująca operacje na liczbach całkowitych,

- FP32 – jednostka matematyczna przeprowadzająca operacje na liczbach podwójnej precyzji,
- LD/ST – jednostki do odczytu/zapisu danych dla poszczególnych wątków,
- Tex – jednostka tekstur,
- SFU – jednostki przetwarzające specjalnego przeznaczenia, które służą do obliczania funkcji takich jak np. sin, cos, pierwiastek kwadratowy, odwrotność, logarytm.

Multiprocesory wyposażone są w planistów (ang. *Warp Scheduler*).

Architekturę Turing wprowadzono na rynek w 2018 roku i uważana jest za największy skok architektoniczny od ponad dziesięciu lat. W konstrukcji bloków SM (ang. *Streaming Multiprocessor*) wprowadzono dużo zmian. W pojedynczym bloku SM możemy zauważyć, iż poza standardowymi blokami, które wykonują operacje INT32 oraz FP32, dodano dwa dodatkowe bloki wykonawcze (rys. 1.2).

W jednym z tych dodatkowych bloków znajdują się rdzenie Tensor, które odpowiadają za funkcję głębokiego uczenia się (ang. *deep learning*). W drugim dodatkowym bloku umieszczono rdzenie Ray Tracing, które sprzętowo przyspieszają technikę śledzenia promieni (ang. *raytracing*). Wszelkie operacje na liczbach całkowitych oraz zmiennoprzecinkowych dotychczas były wykonywane naprzemiennie. W architekturze Turing te operacje przeprowadzane są równolegle w blokach SM, co powoduje wzrost wydajności. Zmieniono również organizację pamięci podręcznej pierwszego poziomu (L1), pamięci współdzielonej oraz pamięci drugiego poziomu (L2). W architekturze Turing połączono w jeden blok pamięć podręczną pierwszego poziomu (L1) oraz pamięć współdzieloną, co przełożyło się na osiągnięcie dwukrotnie większej przepustowości pomiędzy rdzeniami w bloku SM a pamięcią podręczną. W wyniku takiego połączenia pamięć podręczna pierwszego poziomu oraz współdzielona mogą wysyłać dużo większe porcje danych do pamięci podręcznej drugiego poziomu tzn. odpowiednio o rozmiarze 32 kB i 64 kB. Poza tym, zwiększono dwukrotnie rozmiar pamięci podręcznej drugiego poziomu (do 6 MB). Pamięć GDDR5X zastąpiono nowym systemem pamięci GDDR6, który jest wspierany przez 16 kontrolerów. Prędkość przesyłania danych wzrosła z 11 Gb/s do 14 Gb/s. W architekturze Turing zastosowano również po raz pierwszy (w architekturze zorientowanej na użytkownika) niezależne programowanie wątków, tzn. wątki mogą być programowane w warp przez SM bez potrzeby oczekiwania na ich możliwie najszybszą zbieżność.

1.3. Eksperymentalne porównanie algorytmów

W niniejszym punkcie zostaną przedstawione wyniki przeprowadzonych eksperymentów dla czterech wybranych algorytmów. Do eksperymentów wybrano trzy zbiory danych 10-wymiarowe o rosnącej liczbie wektorów (mix10_1.5mln, mix10_15mln, mix10_150mln) oraz trzy zbiory 300-wymiarowe (mix300_50tys, mix300_500tys, mix300_5mln). Pierwszy człon w nazwie zbioru to wymiar, a drugi człon to liczba

wektorów. Liczbę wektorów dobrano tak, aby całkowita liczba elementów macierzy danych była taka sama dla zbioru najmniejszego, średniego i największego. Dane w tych zbiorach tworzą dobrze odseparowane klastry z wielowymiarowego rozkładu normalnego (Gaussa) – jest to tak zwana mieszanina gaussowska (ang. *Gaussian Mixture Model*). Obrano następujące liczby centroidów: K = { 4, 16, 64, 256, 1024, 4096}. Reprezentacja centroidów jest pojedynczej precyzji (float). Dla każdego zbioru danych w powiązaniu z konkretną liczbą centroidów wygenerowano jedno rozwiązanie początkowe przy użyciu algorytmu k-means|| [19].

Dla każdej kombinacji zbioru danych oraz liczby centroidów uruchomiono trzy algorytmy k-średnich na procesorze CPU (Lloyda, pierścienia, Yinyang) oraz jeden algorytm na akceleratorze GPU (Lloyda). Ze względu na błędy zaokrągleń wynikiem jest średni czas iteracji. Zmierzone czasy mogą jednak różnić się nawet dość znacznie, gdy uruchomimy ten sam algorytm kilkukrotnie na tym samym sprzęcie. Związane jest to z występowaniem szumu systemowego. W związku z tym każdy eksperyment powtórzono trzy razy. Ostatecznym wynikiem jest mediana z trzech średnich czasów iteracji. Wszystkie wyniki zestawiono w tabeli 1.1 oraz 1.2. Jako kryterium stopu dla każdego z algorytmów przyjęto moment, w którym względna poprawa wartości SSE między dwiema kolejnymi iteracjami była mniejsza niż 10^{-6}. Oprócz średnich czasów iteracji wyznaczono również przyspieszenie algorytmiczne danego algorytmu A względem algorytmu Lloyda zgodnie z wzorem (6):

$$S_A = \frac{t_{Lloyd}}{t_A} \tag{6}$$

Pozioma linia znajdująca się przy 10^0 na rysunkach oznacza przyspieszenie algorytmiczne nad algorytmem Lloyda (CPU) wynoszące 1.

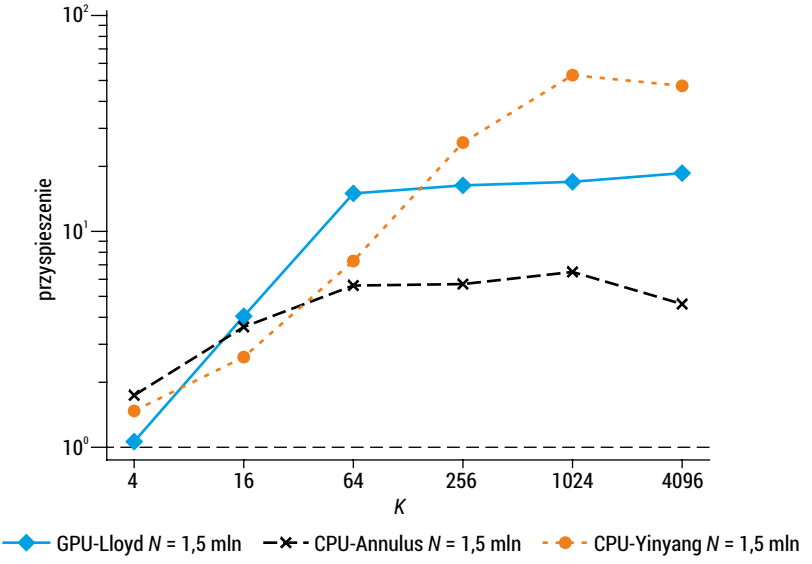
TABELA 1.1. Średnie czasy iteracji (w sekundach) dla różnych algorytmów k-średnich w zależności od liczby klastrów K oraz liczby wektorów uczących N (zbiory mix10)

K	N	Lloyd CPU	Lloyd GPU	Annulus CPU	Yinyang CPU
4	1,5 mln	0,01519	0,01431	0,008734	0,01031
	15 mln	0,1201	0,1241	0,06525	0,06631
	150 mln	1,140	1,237	0,6994	0,7231
16	1,5 mln	0,04528	0,01119	0,01254	0,01730
	15 mln	0,4149	0,1029	0,1369	0,1733
	150 mln	4,096	1,229	1,366	1,738
64	1,5 mln	0,1604	0,01070	0,02859	0,022
	15 mln	1,564	0,1012	0,2615	0,2167
	150 mln	15,57	1,032	2,712	2,297
256	1,5 mln	0,6148	0,03765	0,1078	0,02383
	15 mln	6,118	0,3765	0,9502	0,2092
	150 mln	61,17	3,755	8,188	1,989

K	N	Lloyd CPU	Lloyd GPU	Annulus CPU	Yinyang CPU
1024	1,5 mln	2,434	0,1434	0,3752	0,04599
	15 mln	24,30	1,441	3,186	0,4157
	150 mln	243	14,52	25	3,328
4096	1,5 mln	9,701	0,5212	2,106	0,2054
	15 mln	96,99	5,185	12,11	1,34
	150 mln	969,8	52,07	114,3	9,968

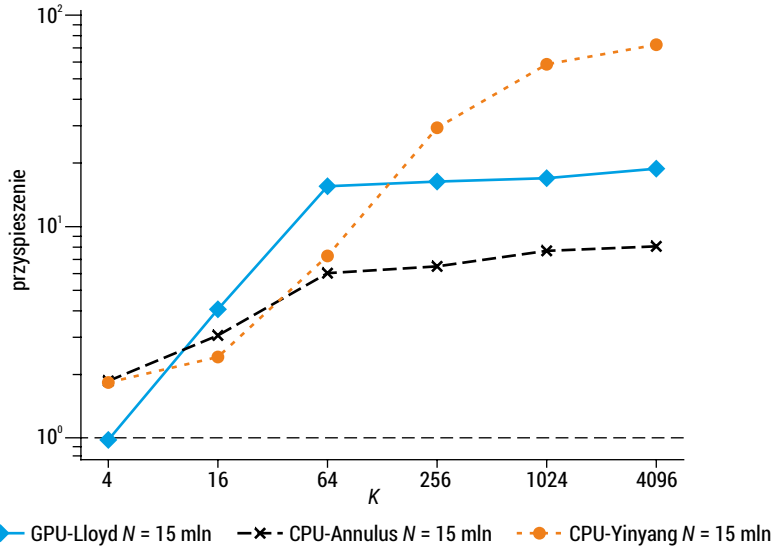
Na rysunkach 1.3, 1.4 i 1.5 przedstawiono wyniki obliczeń z wykorzystaniem zbiorów danych 10-wymiarowych. Każdy z tych trzech rysunków dotyczy innej wartości N, czyli liczby wektorów cech (1.5 mln, 15 mln, 150 mln). Przedstawiono przyspieszenie trzech algorytmów k-średnich nad algorytmem Lloyd'a w wersji na CPU dla różnych wartości K.

Na rysunku 1.3, czyli w przypadku najmniejszego spośród zbiorów 10-wymiarowych, widać, że dla K = 4 najszybszy jest algorytm pierścienia, dla K = 16 i 64 najszybszy jest algorytm Lloyd'a na GPU, a dla największych wartości K = 256, 1024 oraz 4096 algorytm Yinyang jest bezkonkurencyjny. Generalnie przewaga wszystkich trzech algorytmów nad algorytmem Lloyda w wersji na CPU rośnie wraz ze wzrostem liczby klastrów K. Jedynie w przypadku największej liczby klastrów przewaga ta już nie jest rosnąca dla algorytmów pierścienia oraz Yinyang. Największe przyspieszenie osiąga algorytm Yinyang dla wartości K = 1024 oraz 4096 i wynosi ono odpowiednio 52,93 i 47,23.



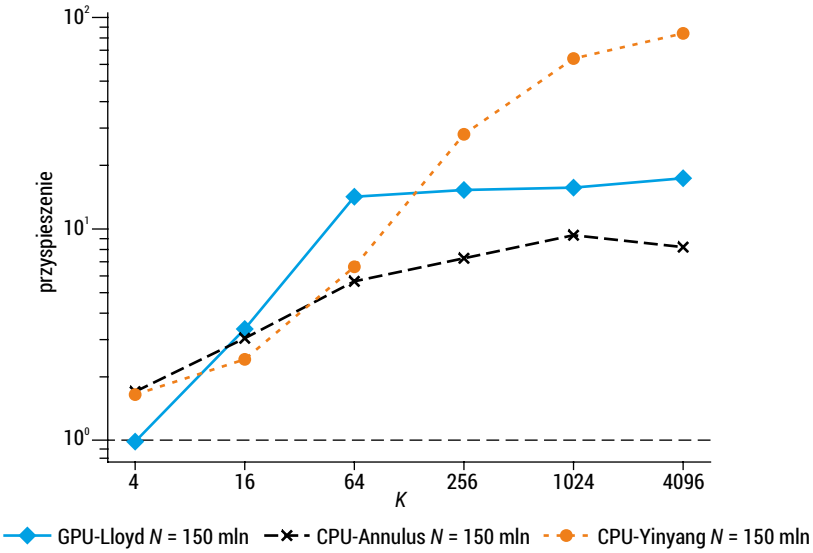
RYSUNEK 1.3. Przyspieszenie trzech algorytmów k-średnich nad algorytmem Lloyda (CPU) dla różnych wartości K oraz N = 1,5 mln

Rysunek 1.4 przedstawia wyniki obliczeń dla średniego zbioru danych pod względem wielkości spośród trzech zbiorów 10-wymiarowych. Szybkość algorytmów dla poszczególnych wartości K, jest identyczna jak w przypadku najmniejszego zbioru danych, tzn. dla K = 256, 1024 oraz 4096 najszybszy jest algorytm Yinyang, a dla mniejszych wartości K najszybszy jest algorytm Lloyda na GPU lub pierścienia. Tutaj również można zaobserwować wzrost przewagi trzech algorytmów nad algorytmem Lloyda na CPU wraz ze wzrostem liczby klastrów K. Algorytm Yinyang osiąga największe przyspieszenie wynoszące 58,46 oraz 72,39 odpowiednio dla K = 1024 i 4096 – są to wartości większe niż w przypadku zbioru najmniejszego.



RYSUNEK 1.4. Przyspieszenie trzech algorytmów k-średnich nad algorytmem Lloyda (CPU) dla różnych wartości K oraz N = 15 mln

Na rysunku 1.5 przedstawiono wyniki obliczeń dla największego zbioru danych spośród zbiorów 10-wymiarowych. Identycznie jak w przypadku dwóch mniejszych zbiorów danych, algorytm Yinyang jest najszybszy dla K = 256, 1024 oraz 4096, a dla mniejszych wartości K najszybszy jest algorytm Lloyda na GPU lub pierścienia. Wraz ze wzrostem liczby klastrów K tutaj również zauważalny jest wzrost przewagi trzech algorytmów nad algorytmem Lloyda na CPU. Największe przyspieszenie wynoszące 73,02 oraz 97,29 osiąga algorytm Yinyang odpowiednio dla K = 1024 i 4096.



RYSUNEK 1.5. Przyspieszenie trzech algorytmów k-średnich nad algorytmem Lloyda (CPU) dla różnych wartości K oraz N = 150 mln

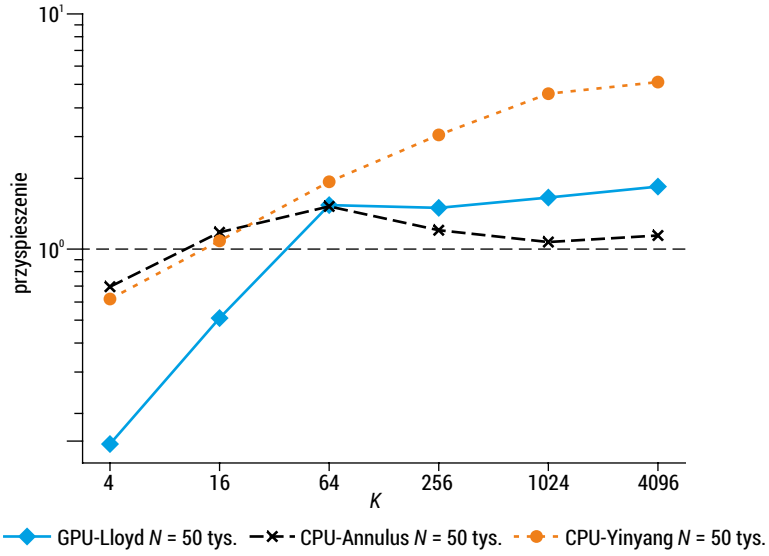
Kolejne trzy rysunki (rys. 1.6, 1.7 i 1.8) przedstawiają wyniki obliczeń dla zbiorów danych 300-wymiarowych. Każdy rysunek jest związany z inną wartością N (50 tys., 500 tys., 5 mln). Dla różnych wartości K przedstawiono przyspieszenie trzech algorytmów k-średnich nad algorytmem Lloyda na CPU.

TABELA 1.2. Średnie czasy iteracji (w sekundach) dla różnych algorytmów k-średnich w zależności od liczby klastrów K oraz liczby wektorów uczących N (zbiory mix300)

K	N	Lloyd CPU	Lloyd GPU	Annulus CPU	Yinyang CPU
4	50 tys 500 tys 5 mln	0,001619 0,01836 0,1411	0,01101 0,09544 0,9203	0,002361 0,02303 0,2057	0,002662 0,02567 0,2286
16	50 tys 500 tys 5 mln	0,005197 0,04866 0,4399	0,01029 0,09497 0,8708	0,004446 0,03811 0,3931	0,004834 0,04192 0,4358
64	50 tys 500 tys 5 mln	0,02141 0,1699 1,644	0,01403 0,1289 1,352	0,01421 0,09906 0,9351	0,01117 0,08932 0,7790
256	50 tys 500 tys 5 mln	0,07159 0,6509 6,467	0,04821 0,4690 4,911	0,06001 0,4835 4,831	0,02360 0,1794 1,742

K	N	Lloyd CPU	Lloyd GPU	Annulus CPU	Yinyang CPU
1024	50 tys 500 tys 5 mln	0,2670 2,59 25,95	0,1625 1,63 17,98	0,2513 2,496 21,53	0,05881 0,61 4,479
4096	50 tys 500 tys 5 mln	1,116 10,38 103,8	0,6102 6,047 69,38	0,9851 10,24 103,1	0,2193 2,27 18,43

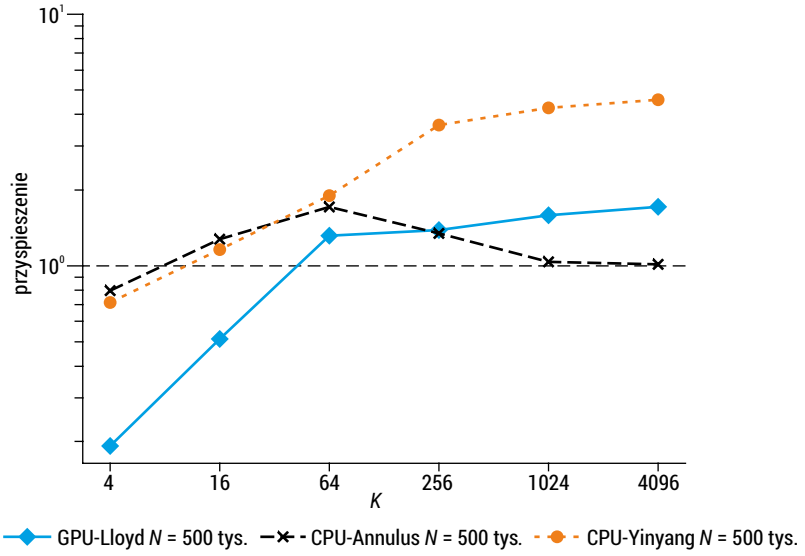
Wyniki obliczeń dla najmniejszego spośród zbiorów 300-wymiarowych przedstawiono na rysunku 1.6. Dla K = 4 oraz 16 najszybszy jest algorytm pierścienia (przewaga nad algorytmem Yinyang jest niewielka), a dla pozostałych wyższych wartości K (64, 256, 1024 oraz 4096) algorytm Yinyang jest najszybszy. Przewaga algorytmów Yinyang oraz Lloyda (GPU) nad algorytmem Lloyda w wersji na CPU zwiększa się wraz ze wzrostem liczby klastrów K. Natomiast w przypadku algorytmu pierścienia przyspieszenie osiąga wartość maksymalną dla K = 64, po czym wartość przyspieszenia zaczyna zmniejszać się wraz ze wzrostem K. Największe przyspieszenie osiąga algorytm Yinyang dla wartości K = 1024 oraz 4096 i wynosi ono odpowiednio 4,54 i 5,09.



RYSUNEK 1.6. Przyspieszenie trzech algorytmów k-średnich nad algorytmem Lloyda (CPU) dla różnych wartości K oraz N = 50 tys

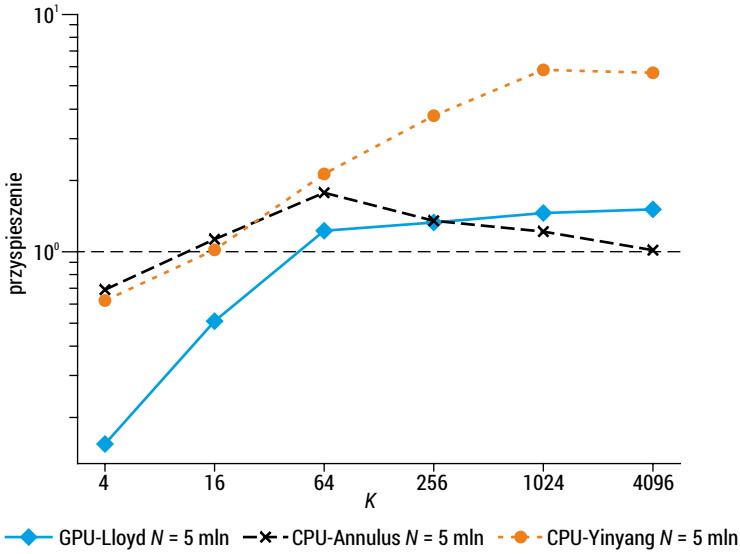
Na rysunku 1.7 przedstawiono wyniki obliczeń dla średniego zbioru danych spośród zbiorów 300-wymiarowych. Identycznie jak w przypadku najmniejszego zbioru danych, algorytm pierścienia jest najszybszy dla K = 4 i 16, a dla większej liczby klastrów najszybszym algorytmem jest algorytm Yinyang. Wraz ze wzrostem liczby klastrów K zwiększa się również przewaga algorytmów Yinyang oraz Lloyda

(GPU) nad algorytmem Lloyda (CPU). Podobnie jak w przypadku najmniejszego zbioru danych, algorytm pierścienia największe przyspieszenie osiąga dla K = 64, a potem dla większych wartości K przyspieszenie już tylko zmniejsza się i dla K = 1024 oraz 4096 przewaga tego algorytmu nad algorytmem Lloyda (CPU) jest znikoma. Największe przyspieszenie wynoszące 4,25 oraz 4,57 osiąga algorytm Yinyang odpowiednio dla K = 1024 i 4096.



RYSUNEK 1.7. Przyspieszenie trzech algorytmów k-średnich nad algorytmem Lloyda (CPU) dla różnych wartości K oraz N = 500 tys

Rysunek 1.8 przedstawia wyniki obliczeń dla największego zbioru danych pod względem wielkości spośród trzech zbiorów 300-wymiarowych. Szybkości algorytmów dla poszczególnych wartości K, jest identyczna jak w przypadku dwóch mniejszych zbiorów danych, tzn. dla K = 4 oraz 16 najszybszy jest algorytm pierścienia, a dla większych wartości K najszybszy jest algorytm Yinyang. Przewaga algorytmów Yinyang oraz Lloyda (GPU) nad algorytmem Lloyda (CPU) zwiększa się wraz ze wzrostem liczby klastrów K. Identycznie jak w przypadku dwóch mniejszych zbiorów danych, algorytm pierścienia największe przyspieszenie osiąga dla K = 64, a dla większych wartości K przyspieszenie zmniejsza się i dla K = 4096 przewaga tego algorytmu nad algorytmem Lloyda (CPU) jest już znikoma. Algorytm Yinyang osiąga największe przyspieszenie wynoszące 5,79 oraz 5,63 odpowiednio dla K = 1024 i 4096 – są to wartości większe niż w przypadku dwóch mniejszych zbiorów.



RYSUNEK 1.8. Przyspieszenie trzech algorytmów k-średnich nad algorytmem Lloyda (CPU) dla różnych wartości K oraz N = 5 mln

Podsumowanie

Podstawowym celem niniejszej pracy było porównanie czterech algorytmów. Na mocy przeprowadzonych eksperymentów można stwierdzić, że najszybszym algorytmem spośród testowanych okazał się algorytm Yinyang, którego przewaga nad innymi jest szczególnie widoczna dla K ≥ 256. W przypadku mniejszych wartości K czasami szybszy okazywał się algorytm Lloyda (GPU) lub pierścienia, ale ta przewaga nad algorytmem Yinyang była niewielka. Bardzo duży wpływ na przyspieszenie ma też wymiar przestrzeni cech oraz centroidów. Przy wektorach cech i centroidach 10-wymiarowych przyspieszenie może wynieść nawet około 100 (algorytm Yinyang, N = 150 mln, K = 4096). Natomiast w przypadku wektorów cech i centroidów 300-wymiarowych maksymalne przyspieszenie wynosi zaledwie około 6 (algorytm Yinyang, N = 5 mln, K = 1024). Wyraźnie widać, że testowane algorytmy niezbyt dobrze radzą sobie z dużymi wymiarami. Wielkość zbioru danych (liczba wektorów cech N) również może mieć zauważalny wpływ na przyspieszenie przy większych wartościach K, np. 1024 lub 4096. W szczególności jest to widoczne w przypadku zbiorów 10-wymiarowych. Wtedy wraz ze wzrostem liczby wektorów cech w zbiorze danych wzrasta również zauważalnie wartość przyspieszenia. W przypadku zbiorów danych 300-wymiarowych jest to bardzo mało widoczne. Algorytm pierścienia wyjątkowo słabo radzi sobie ze zbiorami danych 300-wymiarowymi dla K = 1024 i 4096 – wtedy jego przewaga nad algorytmem Lloyda (CPU) jest bardzo mała bądź nawet znikoma.

W celu poprawy rezultatów warto rozważyć dalsze prace nad algorytmem Lloyda na GPU, gdyż w kodzie bazującym na projekcie CAMPAIGN zbyt wolno działa faza przypisania i zajmuje ona > 95% czasu pracy algorytmu. Innym podejściem mogłoby być opracowanie w wersji na akcelerator GPU któregoś z algorytmów wykorzystujących nierówności trójkąta. Z dwóch przetestowanych algorytmów opierających się na tych zależnościach lepszym wyborem byłby algorytm Yinyang.

Badania zostały zrealizowane w ramach pracy nr WZ/WI-IIT/3/2020, sfinansowanej ze środków Ministerstwa Nauki i Szkolnictwa Wyższego w Polsce.

Bibliografia

[1] Jain A. K., *Data clustering: 50 years beyond K-means*, Pattern Recognition Letters, 2010, t. 31, nr 8, pp. 651–666

[2] Lloyd S., *Least squares quantization in PCM*, IEEE Transactions on Information Theory, 1982, t. IT 28, nr 2, pp. 129–137

[3] MacQueen J., *Some methods for classification and analysis of multivariate observations*, Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, 1967

[4] Dhillon I.S. i Modha D.S., *A data-clustering algorithm on distributed memory multiprocessors*, Large-Scale Parallel Data Mining, Berlin, Germany, Springer, 2002, pp. 245–260

[5] Rodrigues L.M., Zárate L.E., Nobre C.N. i Freitas H.C., *Parallel and distributed kmeans to identify the translation initiation site of proteins*, Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (SMC), 2012

[6] Hamerly G. i Drake J., *Accelerating Lloyd's algorithm for k-means clustering*, Partitional Clustering Algorithms, Cham, Switzerland, Springer, 2015, pp. 41–78

[7] Ding Y., Zhao Y., Shen X., Musuvathi M. i Mytkowicz T., *Yinyang k-means: A drop-in replacement of the classic k-means with consistent speedup*, Proceedings of the 32nd International Conference on Machine Learning (ICML), 2015

[8] Li Y., Zhao K., Chu X. i Liu J., *Speeding up k-means algorithm by GPUs*, Journal of Computer and System Sciences, 2013, t. 79, nr 2, pp. 216–229

[9] Cuomo S., De Angelis V., Farina G., Marcellino L. i Toraldo G., *A GPU-accelerated parallel K-means algorithm*, Computers and Electrical Engineering, 2019, t. 75, pp. 262–274

[10] Kwedlo W. i Czochanski P., *A Hybrid MPI/OpenMP Parallelization of K-Means Algorithms Accelerated Using the Triangle Inequality*, IEEE Access, 2019, t. 7, pp. 42280–42297

[11] Phillips S., *Acceleration of k-means and related clustering algorithms*, Algorithm Engineering and Experiments: 4th International Workshop, Springer, 2002, t. 2409, pp. 166–177

[12] Arthur D. i Vassilvitskii S., *K-means++: The advantages of careful seeding*, Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2007

[13] TechPowerUp, *TechPowerUp*, 1 Grudzień 2020. [Online]. Available: www.techpowerup.com

[14] Kwedlo W., *Two modifications of Yinyang K-means algorithm*, Lecture Notes in Computer Science, t. 10246, Cham, Switzerland, Springer, 2017, pp. 94–103

[15] SimTK, *Clustering Algorithms for Massively Parallel Architectures Including GPU Nodes*, 1 Grudzień 2020. [Online]. Available: simtk.org/projects/campaign

[16] Kohlhoff K.J., Pande V.S. i Altman R.B., *K-Means for Parallel Architectures Using All-Prefix-Sum Sorting and Updating Steps*, IEEE Transactions on Parallel and Distributed Systems, 2013, t. 24, nr 8, pp. 1602–1612

[17] PCLab, *PCLab*, 19 Styczeń 2021. [Online]. Available: pclab.pl

[18] ITIGIC, *ITIGIC*, 19 Styczeń 2021. [Online]. Available: itigic.com/pl/

[19] Bahmani B., Moseley B., Vattani A., Kumar R. i Vassilvitskii S., *Scalable k-means++*, Proceedings of the VLDB Endowment, 2012, t. 5, nr 7, pp. 622–633