Programowanie równoległe i rozproszone

Politechnika Krakowska

Laboratorium 3

Paweł Suchanicz,
Rafał Niemczyk

11 listopada 2019

Spis treści

1 Wstęp

1.1 Opis laboratorium

Celem laboratorium było wykorzystanie CUDA do zrównoleglenia kodu C++. CUDA (Compute Unified Device Architecture) to architektura kart graficznych opracowana przez firmę Nvidia. Pozwala wykorzystanie mocy obliczeniowej kart graficznych w programach opartych o język C/C++.

Algorytmy, które są implementowane a następnie zrównoleglane w ramach laboratorium to normalizacja min-max, standaryzacja rozkładem normalnym i klasyfikacja KNN (k-najbliższych sąsiadów). Zaimplementowany KNN uwzględnia jednego sąsiada i używa metryki euklidesowej.

Szybkość działania każdego algorytmu została zmierzona dla implementacji sekwencyjnej w C++, implementacji równoległej w C++ z wykorzystaniem CUDA dla różnej ilości wątków na blok oraz implementacji w Python (ze skorzystaniem z funkcji z pakietu scikit-learn).

1.2 Specyfikacja sprzętowa

Przy pomiarach szybkości wykonywania algorytmów wykorzystany był sprzęt w konfiguracji
- do implementacji z wykorzystaniem CUDA:
Serwer obliczeniowy cuda.iti.pk.edu

- Procesor: Intel Core i7-950 4 x 3.06GHz
- Ram: 23GB DDR3
- GPU: GeForce RTX 2080 Ti
- System: Ubuntu 18.04.3 LTS

- do implementacji w Python:

- Procesor: Intel Core i7-4712MQ 4 x 2.30GHz
- Ram: 8GB DDR3
- System: Fedora 22

1.3 Zbiór danych

Wykorzystany został zbiór obrazów ręcznie pisanych cyfr MNIST. Zbiór danych ma format .csv i zawiera 60000 rekordów, gdzie każdy rekord odpowiada za jeden obrazek 28x28 pikseli w skali szarości. Pierwsza wartość w rekordzie jest cyfrą która widnieje na obrazku, a kolejne to wartości pikseli obrazka.

Dla zadań postawionych w laboratorium zbiór danych jest dość duży, więc został on obcięty do pierwszych 6000 rekordów, z czego 4500 przeznaczono do trenowania, a pozostałe 1500 do testowania.

2 Wyniki

2.1 Normalizacja min-max

Wzór:

$x^* = \frac{x - min(x)}{max(x) - min(x)}$

2.1.1 Implementacja

W C++ normalizacja została samodzielnie zgodnie z podanym powyżej wzorem. W pętli przechodzącej tablicy (po kolumnach) wyszukiwane są wartości minimum i maxium dla każdej kolumny a następnie wyliczana nowa wartość dla każdego z elementów tablicy. W implementacji CUDA ustalono ilość bloków równą 16 i mierzono czasu dla różnych ilości wątków na blok.

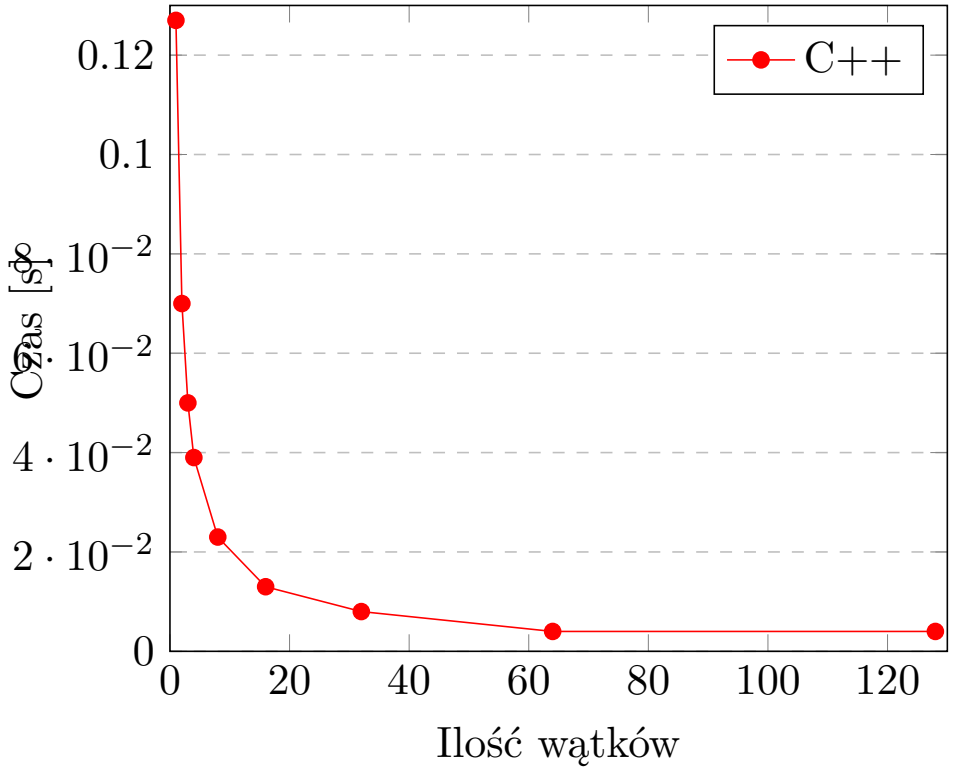
W Pythonie użyta została funkcja MinMaxScaler z pakietu sklearn .

2.1.2 Porównanie wyników

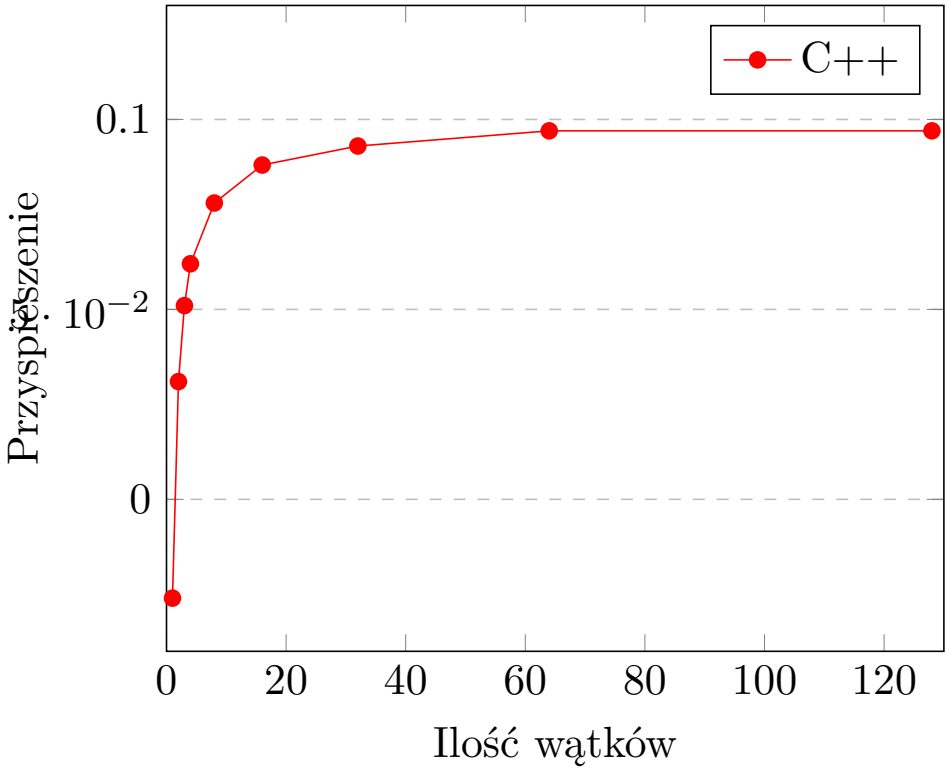
Parametry	Czas [s]
C++	0.101
C++ CUDA 1 wątek	0.127
C++ CUDA 2 wątki	0.070
C++ CUDA 3 wątki	0.050
C++ CUDA 4 wątki	0.039
C++ CUDA 8 wątków	0.023
C++ CUDA 16 wątków	0.013
C++ CUDA 32 wątków	0.008
C++ CUDA 64 wątków	0.004
C++ CUDA 128 wątków	0.004
Pyhon sklearn	0.037

Zastosowanie CUDA i zwiększanie ilości wątków powodowało spadek czasów wykonania. Czasy spadają gdy liczba procesów do pewnego momentu (około 60 wątków). Ograniczeniem jest tu wielkość tablicy z danymi która jest zrównoleglana - ze względu na sposób implementacji zwiększanie liczby wątków na blok ma efekt tylko jeżeli całkowita liczba wątków nie jest większa niż wielkość tablicy. Udało się uzyskać czas wykonania mniejszy niż w implementacji sklearn o ponad 1 rząd wielkości.

Zależność czasu od ilości wątków - normalizacja



Zależność przyspieszenia od ilości wątków - normalizacja



2.2 Standaryzacja rozkładem normalnym

Wzór:

$x^* = \frac{x-\mu}{\sigma}$

2.2.1 Implementacja

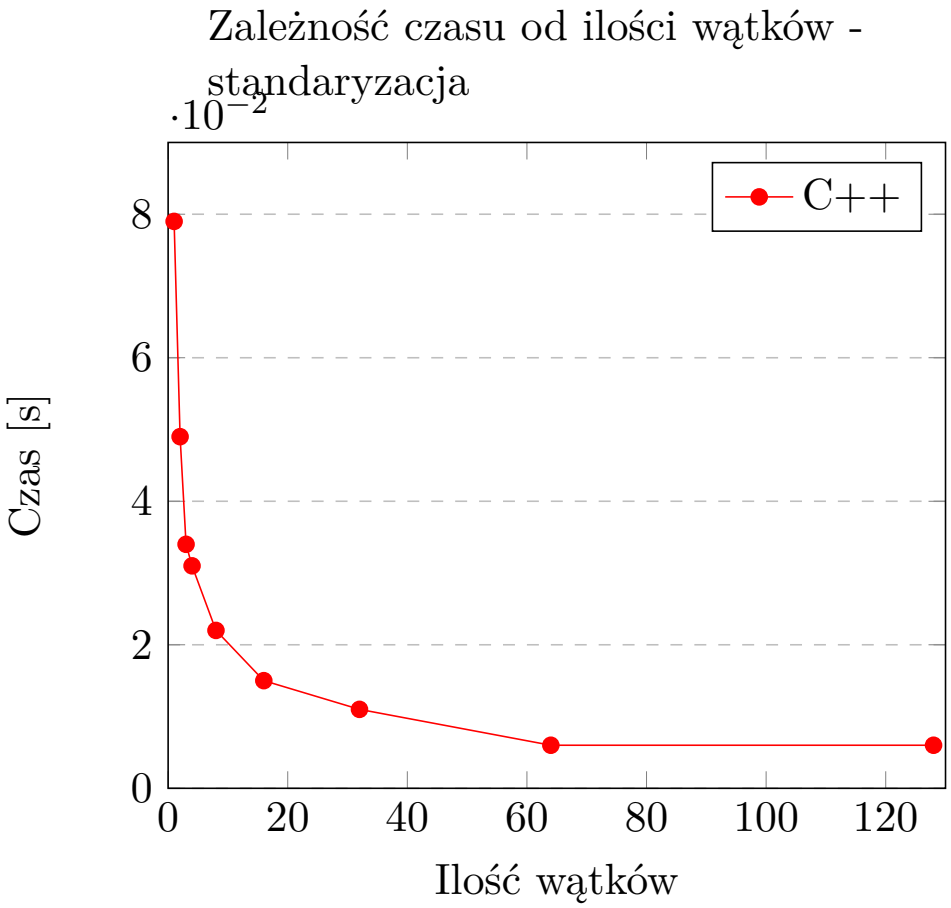
W C++ standaryzacja została zaimplementowana samodzielnie zgodnie z podanym powyżej wzorem. Przechodzimy w pętli po kolumnach i dla każdej kolumny szukamy wartości średniej i wariancji, a następnie wyliczamy nowe wartości dla każdego elementu tablicy. W implementacji CUDA ustalono ilość bloków równą 16 i mierzono czasu dla różnych ilości wątków na blok.

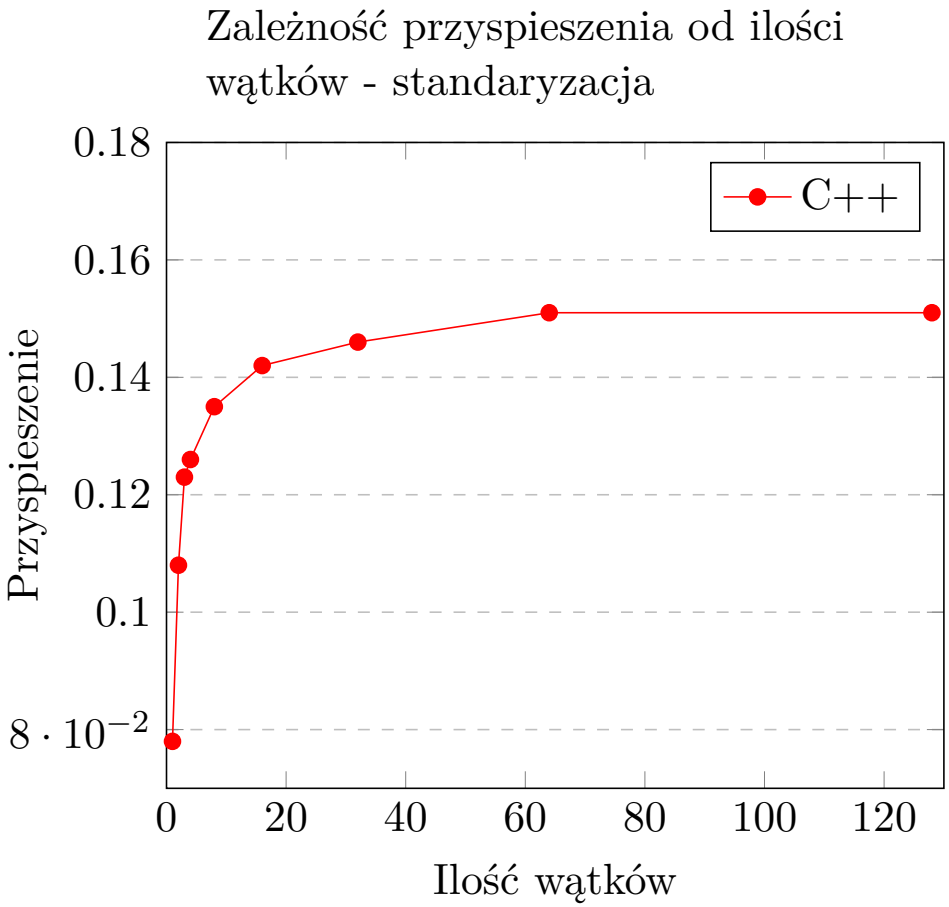
W Pythonie użyta została funkcja StandardScaler z pakietu sklearn.

2.2.2 Porównanie wyników

Parametry	Czas [s]
C++	0.157
C++ CUDA 1 wątek	0.079
C++ CUDA 2 wątki	0.049
C++ CUDA 3 wątki	0.034
C++ CUDA 4 wątki	0.031
C++ CUDA 8 wątków	0.022
C++ CUDA 16 wątków	0.015
C++ CUDA 32 wątków	0.011
C++ CUDA 64 wątków	0.006
C++ CUDA 128 wątków	0.006
Pyhon sklearn	0.086

W przypadku standaryzacji samo użycie CUDA spowodowało już przyspieszenie o 49% w stosunku do implementacji sekwencyjnej. Zwiększanie ilości wątków do pewnego momentu granicznego (około 60) powodowało spadek czasu wykonania. Udało się uzyskać czas wykonania mniejszy niż w implementacji sklearn o ponad 1 rząd wielkości.


Zależność czasu od ilości wątków - standaryzacja
$\cdot 10^{-2}$
8
6
4
2
0
Czas [s]
0
20
40
60
80
100
120
Ilość wątków
C++


Zależność przyspieszenia od ilości wątków - standaryzacja
0.18
0.16
0.14
0.12
0.1
$8 \cdot 10^{-2}$
Przyspieszenie
C++
0
20
40
60
80
100
120
Ilość wątków

2.3 Klasyfikacja KNN

2.3.1 Implementacja

W C++ algorytm k najbliższych sąsiadów zaimplementowany samodzielnie. Algorytm uwzględnia tylko najbliższego sąsiada i korzysta z metryki euklidesowej. W implementacji CUDA ustalono ilość bloków równą 16 i mierzono czasu dla różnych ilości wątków na blok.

W Pythonie użyta została funkcja KNeighborsClassifier z pakietu sklearn z parametrami:

```
KNeighborsClassifier(n_neighbors=1, algorithm='brute', p=2, metric='minkowski',
n_jobs=app_conf['jobs_number'])
```

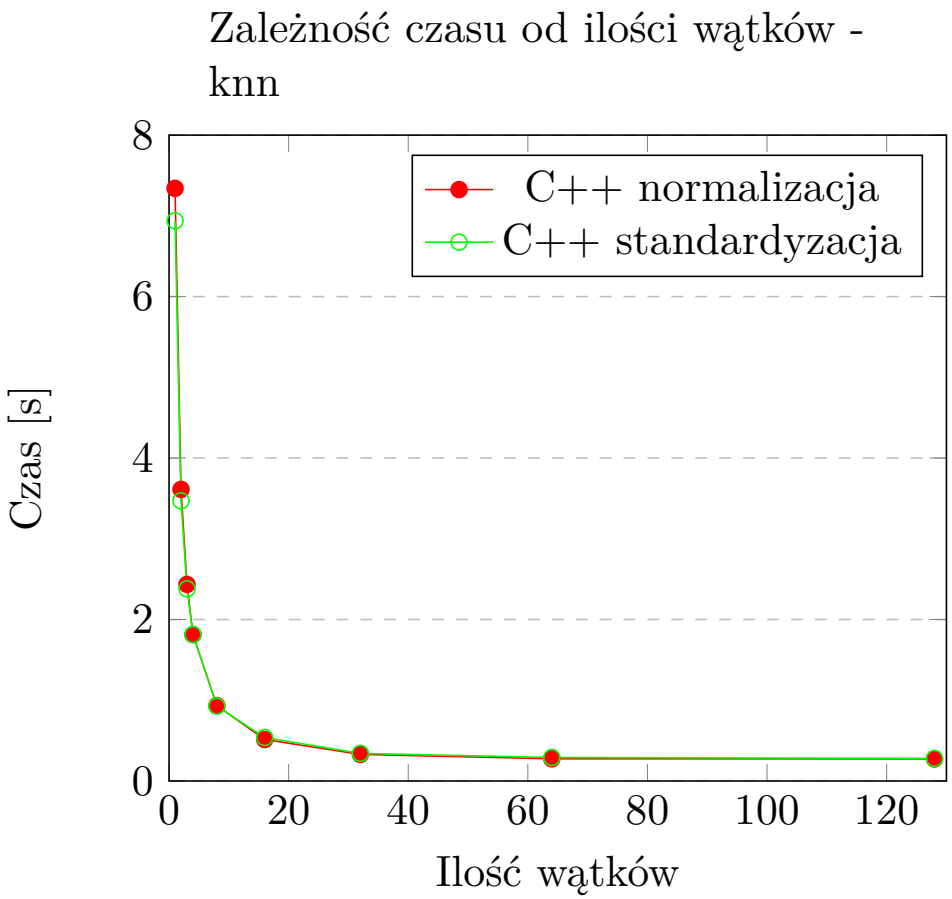
Czasy były mierzone dla wartości njobs od 1 do 4.
Dokładność accuracy wyniosła 71% dla danych po standard scalerze oraz 66% dla danych po min-max scalarze. W przypadku normalizacji w c++ otrzymano dokładność 93.67%. W przypadku standaryzacji dokładność wyniosła 90%. Użycie równoległości oczywiście nie miało wpływu na dokładność działania Knn.

2.3.2 Porównanie wyników

Parametry	Czas [s]
C++ CUDA 1 wątek normalizacja	7.341
C++ CUDA 2 wątki normalizacja	3.612
C++ CUDA 3 wątki normalizacja	2.435
C++ CUDA 4 wątki normalizacja	1.813
C++ CUDA 8 wątków normalizacja	0.934
C++ CUDA 16 wątków normalizacja	0.514
C++ CUDA 32 wątki normalizacja	0.329
C++ CUDA 64 wątki normalizacja	0.275
C++ CUDA 128 wątków normalizacja	0.273
Pyhon sklearn njobs=1 normalizacja	0.215
Pyhon sklearn njobs=2 normalizacja	0.323
Pyhon sklearn njobs=3 normalizacja	0.455
Pyhon sklearn njobs=4 normalizacja	0.386
C++ CUDA 1 wątek standaryzacja	6.939
C++ CUDA 2 wątki standaryzacja	3.470
C++ CUDA 3 wątki standaryzacja	2.380
C++ CUDA 4 wątki standaryzacja	1.814
C++ CUDA 8 wątków standaryzacja	0.927
C++ CUDA 16 wątków standaryzacja	0.538
C++ CUDA 32 wątki standaryzacja	0.342
C++ CUDA 64 wątki standaryzacja	0.291
C++ CUDA 128 wątków standaryzacja	0.280
Pyhon sklearn 1 wątek standaryzacja	0.208
Pyhon sklearn 2 wątki standaryzacja	0.326
Pyhon sklearn 3 wątki standaryzacja	0.329
Pyhon sklearn 4 wątki standaryzacja	0.328

Użycie CUDA w c++ przyniosło pozytywny skutek. Wyniki były niemal identyczne dla danych po normalizacji jak i standaryzacji. Już przy użyciu dwóch wątków czas zmniejszył się

około dwukrotnie, przy użyciu 4 wątków około czterokrotnie. Pomiędzy czasem dla 64 i 128 wątków na blok nie widać już dużej różnicy w czasie wykonania. Program był wykonywany dla 16 bloków, a zrównoleglaniu podlegała tablica o 1500 rekordach więc już 94 wątki dawały maksymalne przyspieszenie. Udało się osiągnąć czas zbliżony do implementacji w Pythonie. Podczas testów można było zauważyć, że lepsze przyspieszenie daje zwiększanie liczby bloków niż wątków. W przypadku Python zwiększanie parametru njobs algorytmu KNN przynosiło odwrotny skutek do oczekiwanego - czas wykonania wydłużał się.



Zależność czasu od parametru njobs - knn