# Optymalizacja obliczeń wartości minimalnej oraz maksymalnej na procesorze graficznym GPU

Filip Rynkiewicz & Marcin Daszuta

29 sierpnia 2018

## 1 Wstęp

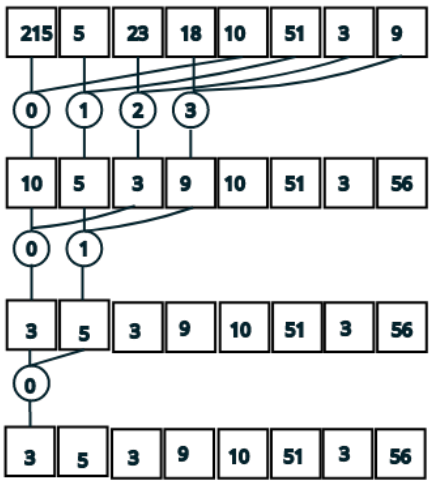
Zadania proste na procesorze CPU nie zawsze można w trywialny sposób zaimplementować na procesorze graficznym GPU. Zadania takie jak dodawanie elementów w tablicy, szukanie najmniejszej/największej liczby ze zbioru można tutaj dać jako przykład. Karta graficzna z powodu swojej architektury wykonuje wiele zadań na raz. Z tego powodu tak trywialne zadanie jak wyszukiwanie najmniejszej wartości w tablicy można potraktować jako optymalizacyjne wyzwanie. Wprawdzie karty firmy Nvidia udostępnia tzw. *operacje atomowe*, w których widnieje funkcja *atomicMin()*, lecz jej zachowanie często jest zbyt powolne dla potrzeba algorytmów. Jest to uwarunkowane blokadą dostępu wszystkich możliwych wątków karty graficznej do pamięci w której są dane. Jednocześnie funkcje atomowe nie są zaimplementowane dla liczb zmiennoprzecinkowych. Oczywiśćie można przed obliczeniami przekształcić liczbę zmiennoprzecinkową do postaci liczby całkowitej, badź skorzystać z funkcji *atomicCAS*.

### 1.1 Adresowanie sekwencyjne

Jednym z pomysłów na przyspieszenie takich obliczeń są operacje *redukcji*. Polegają one na iteracyjnym zmniejszaniu ilości wątków na których obliczenia zostały już wykonane.

Przykład z Rysunku 1 pokazuje algorytm poszukiwania najmniejszego elementu w tablicy $N$ 8-elementowej.Na początku algorytmu kernel jest uruchamiany z $n = 4$ wątkami. Na każdym z nich wykonywana jest operacja porównania elementu $e_1$ o numerze wątku $i$ oraz elementu $e_2$ odsuniętego od niego o $w$=$N/2$. Tak więc dla wątku $i = 0$ element $e_1 = 215$ jest porównywany z elementem $e_2 = 10$ przesuniętym o $w$=$4$. Wynikiem porównania jest zapisanie elementu $e_1$ jako elementu mniejszego do tablicy dla następnej iteracji algorytmu. Kolejna iteracja wykorzystuję już 2 wątki i porównuję ze sobą 4 elementy tablicy, tylko te które zostały uznane za mniejsze w poprzedniej iteracji algorytmu. Po ukończeniu wszystkich iteracji element najmniejszy w tablicy będzie zapisany na pierwszym miejscu takiej tablicy. Największy element tablicy wyszukiwany

jest w identyczny sposób, niejako przy okazji sprawdzania liczby najmniejszej kosztem jednego dodatkowego porównania i przypisania.




Rysunek 1: Przykład poszukiwania elementu najmniejszego w tablicy operacją redukcji

## 1.2 Implementacja

Implementacja powyżej wymienionego pomysłu została stworzona za pomocą 2 kerneli. Pierwszy z nich $seq\_minmaxKernel$ wykonuję się jako pierwszy. Przyjmując tablice $max$,$min$ oraz $a$. . Tablica $a$ jest tablicą w której algorytm ma wyszukać liczbę najmniejszą oraz największą. Tablica max oraz min to tablice w których przechowywany jest wynik porównań, tablice te posiadają rozmiar dwa razy mniejszy niż tablica $a$.

Ważną częścią tego kernela jest stworzenie dwóch tablic $s\_min$ oraz $s\_max$ w pamięci współdzielonej dla bloku wątków. Każdy wątek w bloku przepisuję wartości z tablicy $a$ do tablic w pamięci współdzielonej. Następuje pierwsza synchronizacją wątków. Kolejnym krokiem jest iterowanie przez połowę tablicy $a$, wykonanie porównań oraz odpowiednie podmienienie wartości w tablicach współdzielonych. Po każdej takiej iteracji następuje synchronizacja wątków. Ostatnim krokiem tego kernela dla wątku numer 0 jest przypisanie wartości minimalnej z każdego bloku wątków do tablic przekazanych przy uruchomieniu kernela $min$ oraz $max$.

Kolejny kernel różni się od poprzedniego parametrami wejściowymi oraz sposobem uruchamiania. Do tego kernela przekazywane są tablice uzyskane w kernelu poprzednim. Następnie wartości z nich przepisywane są do tablic współdzielonych i następuje pierwsza synchronizacja. Kolejnym krokiem jest wykonanie porównań, w taki sam sposób jak było to robione w poprzednim kernelu. Na koniec każdy wątek zerowy w bloku przepisuję wartość najmniejszą i największą

do tablicy wynikowych.

Różnica pomiędzy kernelem $seq\_finalminmaxKernel$ a $seq\_minmaxKernel$ jest jeszcze sposób uruchamiania. Drugi z nich uruchamiany jest tylko z jednym blokiem w którym jest dimBlock wątków, natomiast pierwszy uruchamiany jest z dimGrid na dimBlock.

```
1 seq_minmaxKernel <<< dimGrid, dimBlock>>>(dev_max, dev_min, dev_a);
2 seq_finalminmaxKernel <<< 1, dimBlock>>>(dev_max, dev_min);
```

Pierwszy z nich musi też posiadać taką samą ilość wątków w bloków co ilość bloków, np (1024 bloki i w każdym 1024). Jedynym zastrzeżeniem jest aby ilość bloków opierała się na funkcji wykładniczej o podstawie dwa gdzie wykładnik należy do liczb naturalnych.

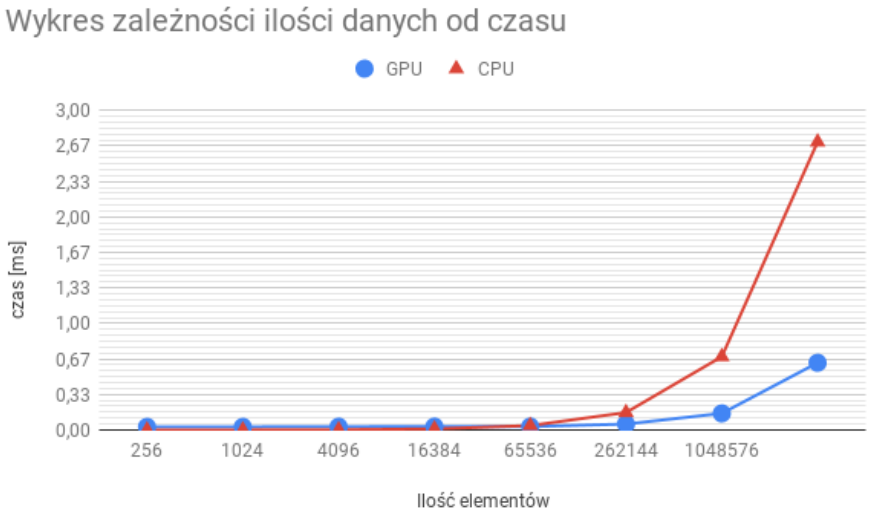
```
1  __global__ void seq_minmaxKernel(double* max, double* min, const double* a)
2  {
3      __shared__ double s_max[BLOCKSIZE];
4      __shared__ double s_min[BLOCKSIZE];
5      unsigned int tid = threadIdx.x;
6      unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
7      s_max[tid] = s_min[tid] = a[i];
8      __syncthreads();
9      for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1)
10     {
11         if (tid < s)
12         {
13             if (s_max[tid + s] > s_max[tid])
14             {
15                 s_max[tid] = s_max[tid + s];
16             }
17
18             if (s_min[tid + s] < s_min[tid])
19             {
20                 s_min[tid] = s_min[tid + s];
21             }
22         }
23         __syncthreads();
24     }
25     if (tid == 0)
26     {
27         max[blockIdx.x] = s_max[0];
28         min[blockIdx.x] = s_min[0];
29     }
30
31 }
32 __global__ void seq_finalminmaxKernel(double* max, double* min)
33 {
```

```
34      __shared__ double s_max[BLOCKSIZE];
35      __shared__ double s_min[BLOCKSIZE];
36      unsigned int tid = threadIdx.x;
37      unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
38      s_max[tid] = max[i];
39      s_min[tid] = min[i];
40      __syncthreads();
41      for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1)
42      {
43          if (tid < s)
44          {
45              if (s_max[tid + s] > s_max[tid])
46              {
47                  s_max[tid] = s_max[tid + s];
48              }
49
50              if (s_min[tid + s] < s_min[tid])
51              {
52                  s_min[tid] = s_min[tid + s];
53              }
54          }
55
56          __syncthreads();
57      }
58
59      if (tid == 0)
60      {
61          max[blockIdx.x] = s_max[0];
62          min[blockIdx.x] = s_min[0];
63      }
64  }
```

### 1.3 Wyniki

Wyniki przedstawione na Rysunku 2 zostały uzyskane na Procesorze Intel Core i7-5700HQ CPU @ 2.70GHz oraz na NVIDIA GeForce GTX960M. Dane losowe w tablicach zostały wygenerowane losowo z zakresu od -1 do 1, dla rozmiaru tablic w zakresie od 16 do 1045876 elementów. GPU zostało uruchomione na systemie Ubuntu 14 LTS(GPU) oraz Windows 8.1 (CPU).

Tablice o małych rozmiarach, to znaczy poniżej 65536 elementów przeszukują się szybciej dla CPU. Znaczny spadek szybkości wykonywania się algorytmu następuje dopiero po przekroczeniu tej liczby i dopiero wtedy można zauważyć przewagę optymalizacyjną obliczeń na GPU dla tego zagadnienia.




Rysunek 2: Wyniki porównawcze dla GPU i CPU szukania najmniejszej i największej wartości w zbiorze liczb zmiennoprzecinkowych.

## 2 Bibliografia


- `https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf`