Przetwarzanie równoległe

Programowanie w CUDA na NVIDIA GPU – sumowanie wektora

Temat 8

Michał Kukieła 132265, Patryk Dolata 132209

Grupa I3, poniedziałek 16:50

michal.kukiela@student.put.poznan.pl

patryk.m.dolata@student.put.poznan.pl

1. Wstęp

Celem projektu jest zapoznanie się z zasadami programowania równoległego procesorów kart graficznych. Naszym tematem była analiza porównawcza dwóch metod sumowania wektora. Pierwsza polegała sumowaniu z rozbieżnością wątków, druga zaś na sumowaniu bez rozbieżności wątków. W obu przypadkach wykorzystujemy wyłącznie pamięć globalną oraz łączenie dostępów do pamięci. W celu przeprowadzenia niezbędnych pomiarów wykorzystaliśmy program NVIDIA Visual Profiler z CUDA Toolkit.

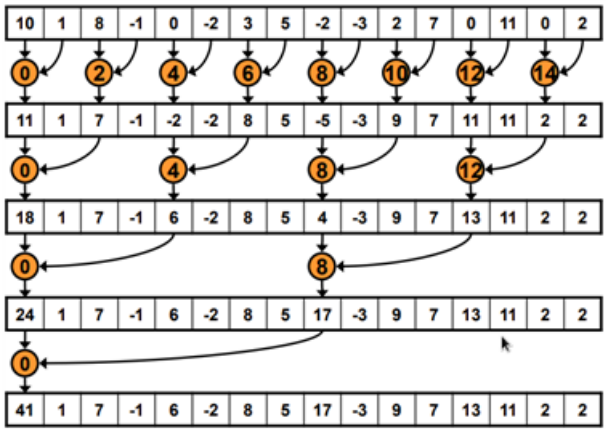
2. Architektura GPU

Testy zostały wykonane na platformie wyposażonej w kartę NVIDIA GeForce GTX 960M. Jej parametry przedstawiają się następująco:

- CUDA wersja 10.1
- Compute Capability: 5.0
- rozmiar pamięci globalnej: 4GB
- 5 multiprocesorów
- 128 rdzeni na SM, łącznie 640 rdzeni
- zegar rdzenia: 1176 MHz (1.18 GHz)
- zegar pamięci: 2505 MHz
- 1024 max wątków na blok
- 2048 max wątków na multiprocesor
- warp size: 32
- 1024 x 1024 x 64 max rozmiar bloku

3. Analiza algorytmów

- **Metoda RW**
  Schemat działania:



Kod:

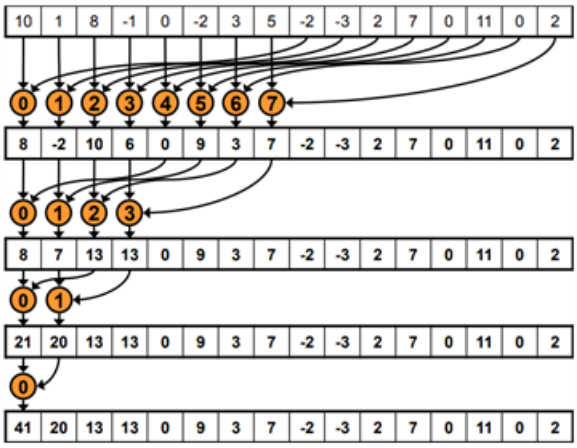
```
82  __global__ void block_sumRW(int *wyniki, int *dane, const size_t rozmiar_danych) {
83
84      unsigned int i = (blockIdx.x * blockDim.x + threadIdx.x);
85
86      for (unsigned int odstep = 1; odstep < blockDim.x; odstep *= 2) {
87          if (threadIdx.x % (2 * odstep) == 0) {
88              if (i + odstep < rozmiar_danych)
89                  dane[i] += dane[i + odstep];
90          }
91          __syncthreads();
92      }
93
94      if (threadIdx.x == 0) {
95          wyniki[blockIdx.x] = dane[blockIdx.x * blockDim.x];
96      }
97  }
```

threadIdx.x – identyfikator wątku w bloku
i – identyfikator zmiennej czytanej i zapisywanej z/do pamięci globalnej w danym bloku
odstep – odstęp pomiędzy wartościami sumowanych elementów

Każdy wątek jeśli jest odpowiedzialny za wygenerowanie częściowego wyniku – dodaje jedną komórkę z tablicy (oddaloną o **odstep**) do "swojej" komórki. Kolejne wątki z wiązki nie uczestniczą w kolejnych etapach obliczeń. Wiązka zostaje uszeregowana do osnowy, lecz (wraz z kolejnymi krokami algorytmu) coraz mniej wątków wykonuje równoległe obliczenia, następuje utrata efektywności przetwarzania. Przez zastosowanie operatora modulo wprowadziliśmy rozbieżność wątków - co drugi wątek wykonuje pracę, co oznacza, że również co drugi (ten, który nie wykonuje pracy) jest bezczynny i w ogóle niepotrzebny. Jak już było wspomniane następuje utrata efektywności również dlatego, że operator modulo jest bardzo wolny.

Ilość wątków, z którą uruchamiamy powinna jest równa rozmiarowi danych (mimo, że na samym początku jedynie połowa z nich pracuje, a z każdym krokiem pracuje o połowę mniej), tzn. **ilość_bloków*ilość_wątków_w_bloku = rozmiar_danych.**

- **Metoda BRW**
  Schemat działania:



Kod:

```
99  __global__ void block_sumBRW(int *wyniki, int *dane, const size_t rozmiar_danych) {
100     unsigned int i = blockIdx.x*blockDim.x * 2 + threadIdx.x;
101     for (unsigned int odstep = blockDim.x; odstep > 0; odstep >>= 1) {
102         if (threadIdx.x < odstep) {
103             dane[i] += dane[i + odstep];
104         }
105
106         __syncthreads();
107     }
108
109     if (threadIdx.x == 0) {
110         wyniki[blockIdx.x] = dane[blockIdx.x*blockDim.x * 2];
111     }
112
113 }
```

threadIdx.x – identyfikator wątku w bloku
i – identyfikator zmiennej czytanej i zapisywanej z/do pamięci globalnej w danym bloku
odstep – odstęp pomiędzy wartościami sumowanych elementów

Kolejne wątki wiązki odwołują się do sąsiednich danych w pamięci – łączenie dostępów pamięci globalnej. Grupa sumujących, sąsiednich wątków zmniejsza się dwukrotnie w każdym kroku, tak samo jak w poprzednim algorytmie, lecz na samym początku pracują wszystkie wątki (jest ich o połowę mniej niż danych w wektorze, czyli o połowę mniej niż w algorytmie RW.
Ilość wątków, z którą uruchamiamy powinna być o połowę mniejsza od rozmiaru danych, tzn. **ilość_bloków*ilość_wątków_w_bloku = rozmiar_danych/2.**

4. Teoria

W rozważanym przez nas zadaniu korzystamy wyłącznie z pamięci globalnej. Dostęp do niej jest zdecydowanie wolniejszy niż dostęp do pamięci współdzielonej. Jest ona jednak dużo większa niż pamięć współdzielona. Zmienne przechowywane w pamięci globalnej są dostępne dla wszystkich wątków, przez cały czas życia aplikacji. Łączenie dostępów w GPU to zupełnie inna kwestia niż łączenie dostępów w przypadku CPU, ponieważ tutaj wątki nie mają pamięci podręcznej, a jedynie rejestry. Łączenie dostępów jest zatem rozumiane jako jednoczesne pobranie przez kolejne wątki kolejnych elementów z pamięci globalnej do rejestru wątków

- **liczba operacji zmiennoprzecinkowych w funkcji rozmiaru problemu**

Dla obu algorytmów (BRW i RW), niezależnie od ilości wielkości bloków, wykonuje się N-1 sumowań (operacji zmiennoprzecinkowych), gdzie N - długość wektora (liczba sumowanych składników). Prowadzi to do uzyskania wyniku w postaci jednej liczby.

- **synchronizacja przetwarzania w kodzie**

Synchronizacja wątków następuje po każdym sumowaniu, ponieważ każdy wątek, po sumowaniu nr k, do sumowania w kroku k+1 potrzebuje częściowej sumy obliczonej przez inny wątek w kroku k. Innymi słowy każdy wątek wykonuje jedno sumowanie, po czy czeka, aż wszystkie inne też wykonają ten krok, aby móc sumować dalej. Dzieje się tak również z wątkami, które przestają uczestniczyć w sumowaniu po każdym kroku, ponieważ dalej przechodzą przez pętlę, lecz nie spełniają warunku if(), wewnątrz którego następuje sumowanie.

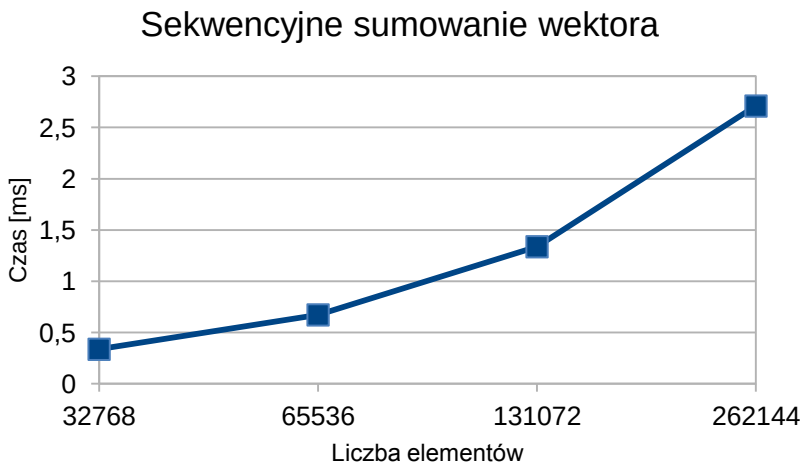
5. Pomiary

Ta część sprawozdania przeznaczona jest na analizę pomiarów otrzymanych podczas badań wykonanych przy pomocy NVIDIA Visual Profiler, a także pomiaru czasu dla przetwarzania sekwencyjnego.
Niestety mieliśmy problemy z programem NVIDIA Visual Profiler, gdyż nie mierzył niektórych metryk\eventów - wyniki niektórych pomiarów zawsze podawał równie "0". Radząc się innych osób mających podobny problem włączaliśmy Profiler oraz wykonywany program .exe ze wszystkimi możliwymi uprawnieniami, jako Administrator, lecz w naszym przypadku nie rozwiązało to problemu. W przypadku niektórych miar pozostały nam jedynie obliczenia teoretyczne.

- ## Czas przetwarzania

  - **Algorytm sekwencyjny**

N	czas [ms]
32768	0,337515
65536	0,673053
131072	1,337796
262144	2,71033



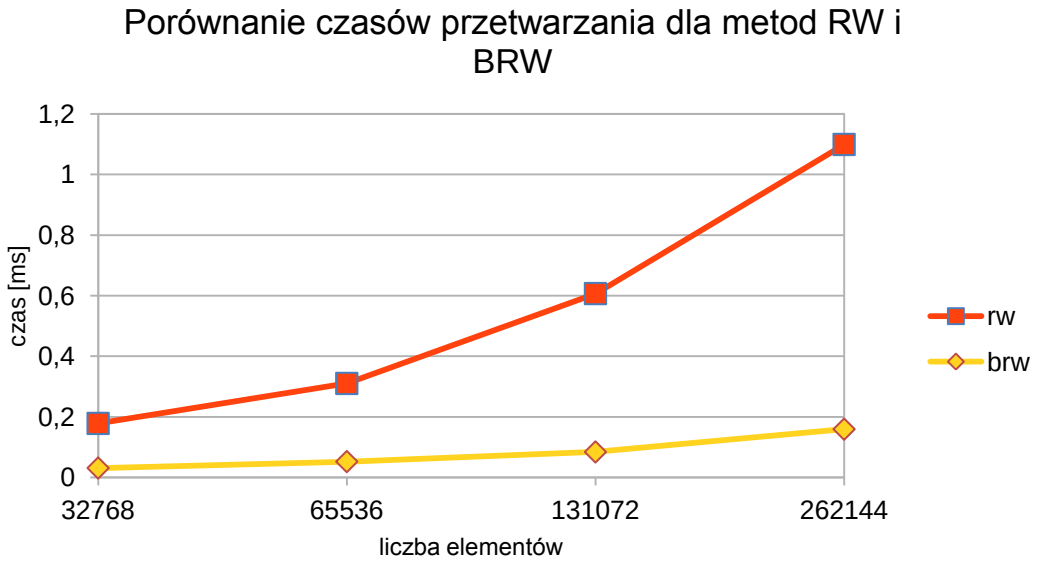
Powyższe dane dla algorytmu sekwencyjnego zostały zaprezentowane po to, żeby być punktem odniesienia dla późniejszych rozważań na temat czasu przetwarzania, prędkości obliczeń oraz w przyspieszenia (przyspieszenie w stosunku do prędkości sumowania sekwencyjnego) dla metod RW i BRW.

- **Porównanie czasów przetwarzania algorytmów RW i BRW z sekwencyjnym**

N	Metoda	Liczba bloków	Rozmiar bloku	Czas [ms]
32768	RW	1024	32	0,144495
32768	RW	256	128	0,154879
32768	RW	128	256	0,171771
32768	RW	32	1024	0,242505
65536	RW	2048	32	0,166182
65536	RW	512	128	0,298887
65536	RW	256	256	0,336022
65536	RW	64	1024	0,440053
131072	RW	4096	32	0,304642
131072	RW	1024	128	0,589871
131072	RW	512	256	0,659726
131072	RW	128	1024	0,873064
262144	RW	8192	32	0,601541
262144	RW	2048	128	0,761356
262144	RW	1024	256	1,303582
262144	RW	256	1024	1,729406
32768	BRW	512	32	0,034159
32768	BRW	128	128	0,028448
32768	BRW	64	256	0,028335
32768	BRW	16	1024	0,033791
65536	BRW	1024	32	0,058174
65536	BRW	256	128	0,046846
65536	BRW	128	256	0,04707
65536	BRW	32	1024	0,05763
131072	BRW	2048	32	0,070654
131072	BRW	512	128	0,082574
131072	BRW	256	256	0,08339
131072	BRW	64	1024	0,100428
262144	BRW	4096	32	0,131356
262144	BRW	1024	128	0,155756
262144	BRW	512	256	0,156364
262144	BRW	128	1024	0,194122

W tabelce poniżej znajdują się średnie czasy dla metod RW i BRW w danej instancji wektora. Dane te są uśrednione wyłącznie dla późniejszych łatwiejszych obliczeń oraz, żeby lepiej móc porównać ogólnie prędkości algorytmów RW i BRW - samych metod, nie biorąc pod uwagę liczby bloków i ich rozmiaru:

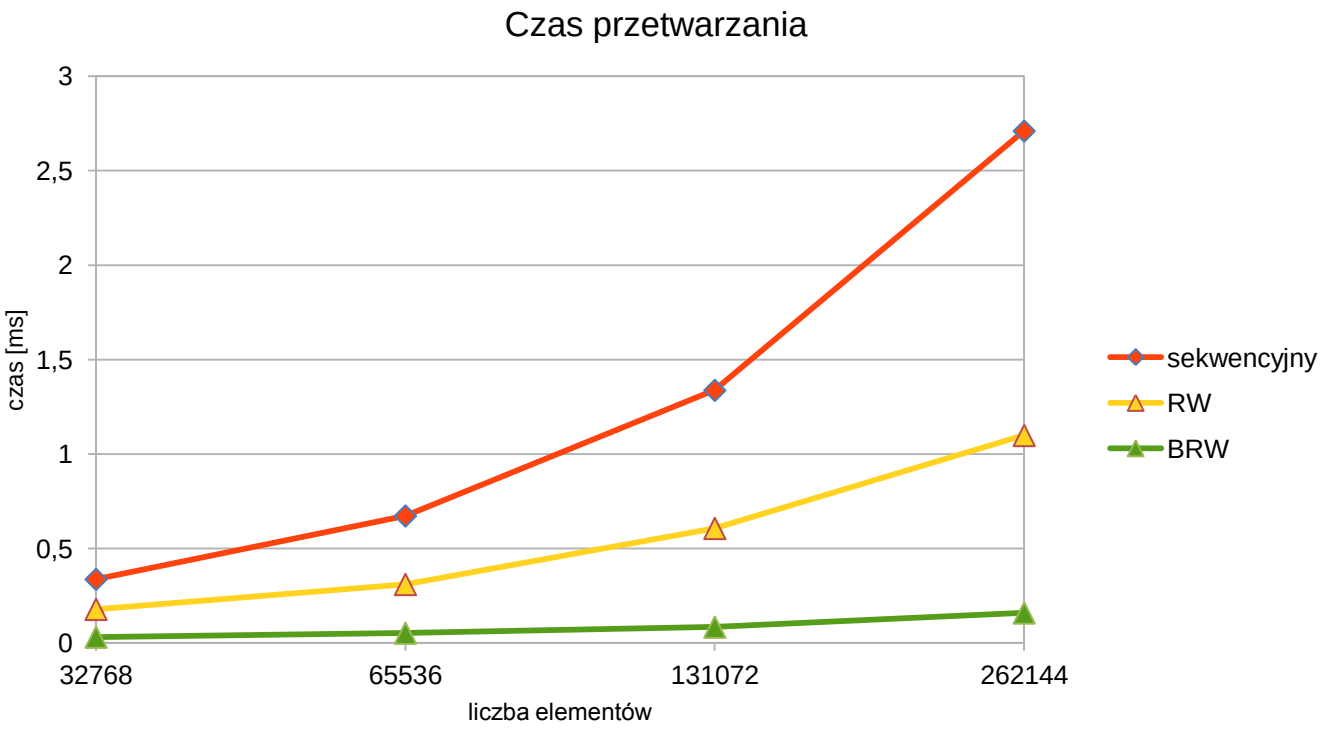
N	czas RW [ms]	czas BRW [ms]
32768	0,1784125	0,03118325
65536	0,310286	0,05243
131072	0,60682575	0,0842615
262144	1,09897125	0,1593995



Na podstawie zaprezentowanych danych tabelarycznych oraz wykresów możemy stwierdzić, że czas przetwarzania dla metody bez rozbieżności wątków jest znacznie lepszy niż czas przetwarzania metody z rozbieżnością.

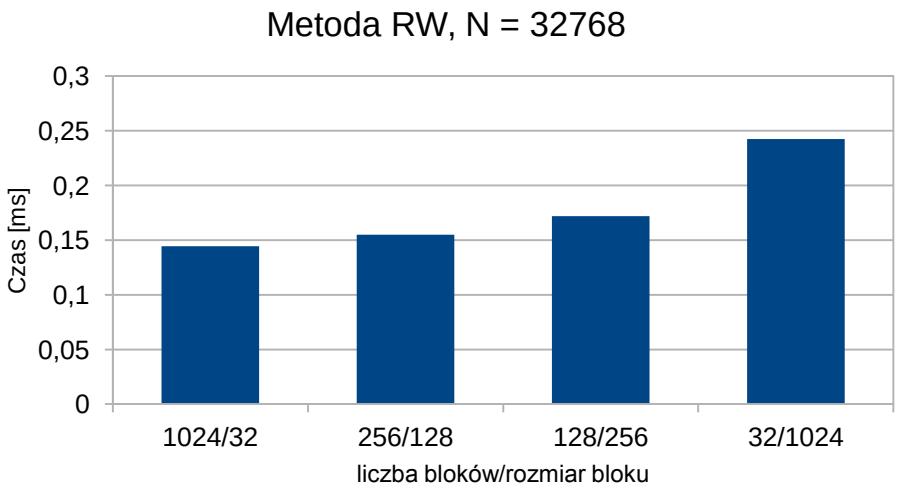
W tabelce poniżej znajduje się porównanie czasów (uśrednionych) dla metod RW i BRW z czasami dla algorytmu sekwencyjnego.

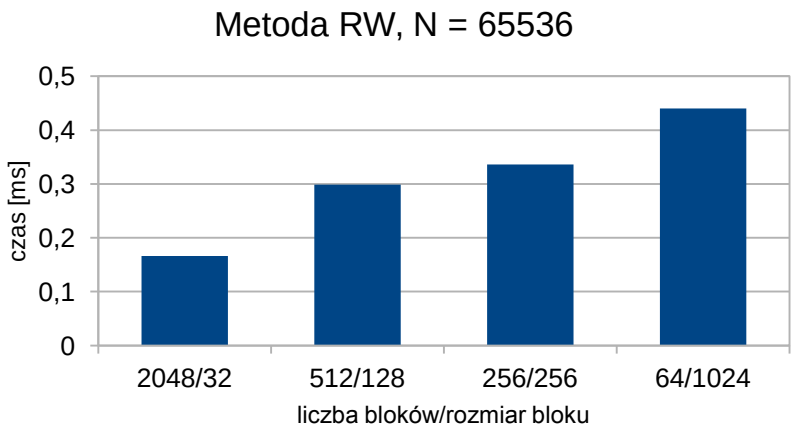
N	sekwencyjny[s]	RW [s]	BRW [s]
32768	0,337515	0,1784125	0,03118325
65536	0,673053	0,310286	0,05243
131072	1,337796	0,60682575	0,0842615
262144	2,71033	1,09897125	0,1593995

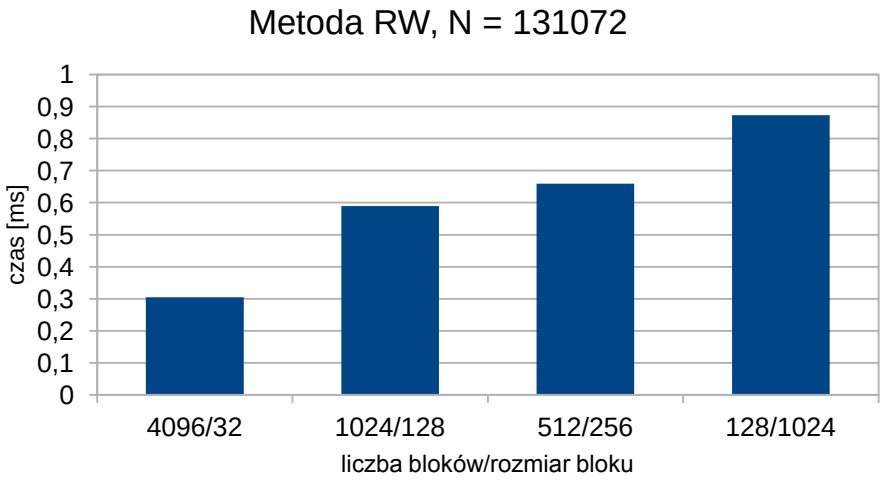


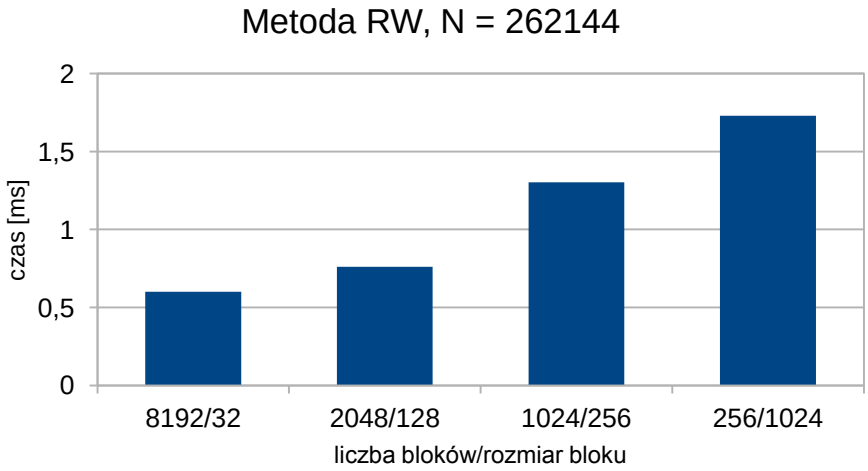
Jak widać na pierwszy rzut oka, metody równoległe, które wykonują obliczenia na GPU prezentują się lepiej pod względem czasu przetwarzania od algorytmu sekwencyjnego wykonywanego na CPU. Nie dziwi nas to, ponieważ algorytm sekwencyjny wykonuje wszystkie sumowania po kolei, zaś algorytmy równoległe wykonują wiele sumowań równocześnie, co pozwala na efektywniejszą pracę, mimo synchronizacji w każdym kroku algorytmów równoległych (nie trwają one długo, każdy wątek ma takie samo zadanie i wykonuje je w podobnym czasie, więc na żaden nie powinno się długo czekać.

- **Metoda RW**



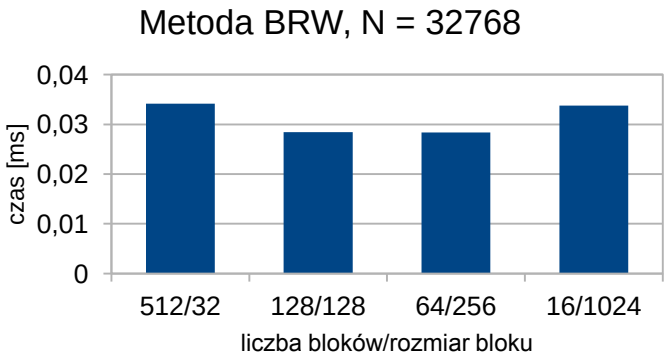

Metoda RW, N = 65536
0,5
0,4
0,3
0,2
0,1
0
czas [ms]
2048/32
512/128
256/256
64/1024
liczba bloków/rozmiar bloku

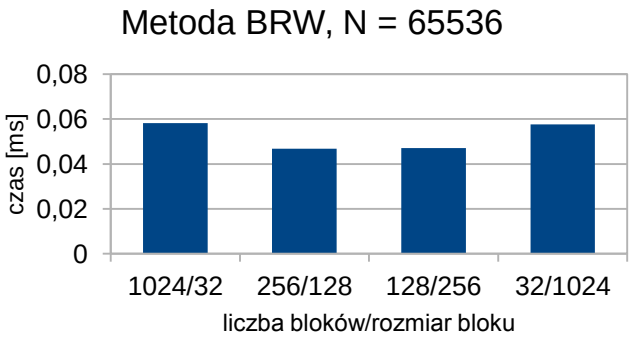

Metoda RW, N = 131072
1
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
czas [ms]
4096/32
1024/128
512/256
128/1024
liczba bloków/rozmiar bloku

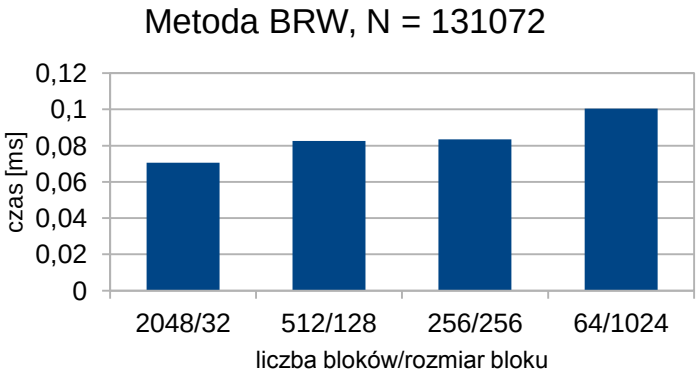

Metoda RW, N = 262144
2
1,5
1
0,5
0
czas [ms]
8192/32
2048/128
1024/256
256/1024
liczba bloków/rozmiar bloku

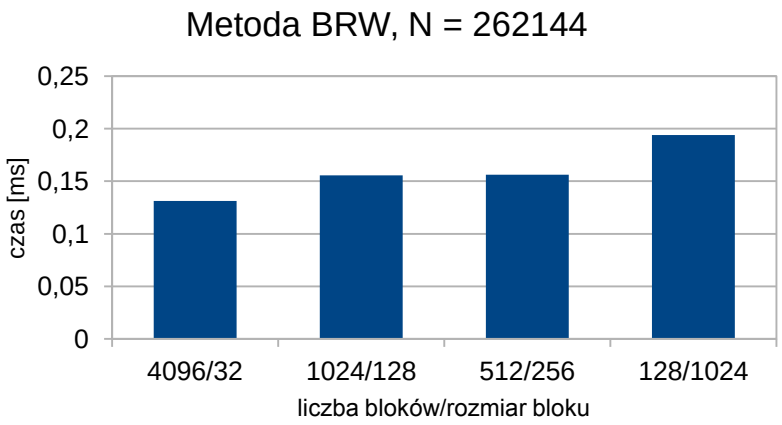
Oczywistym jest, że wraz z wielkością instancji wektora czas przetwarzania zwiększa się, lecz innym interesującym wnioskiem, który się nasuwa po przeanalizowaniu wykresów jest to, że im większa ilość bloków tym mniejszy czas przetwarzania w metodzie RW.
Wpływ na szybkość metody przy rozmiarze bloku równym 32 może mieć wyższa niż w innych przypadkach miara łączenia dostępów do pamięci globalnej, co zostanie opisane później.

- **Metoda BRW**









W przypadku metody BRW tylko w dwóch największych instancjach (N = 131072 i N = 262144) widzimy, zwiększenie instancji wektora zwiększa czas przetwarzania. Dla instancji N = 32768 i N = 65536 szybsze są pomiary, w których liczba bloków i liczb wątków w bloku są do siebie zbliżone. Duży wpływ na niski czas obliczeń metody BRW ma miara łączenia dostępów do pamięci globalnej (która zostanie policzona i opisana później) posiadająca korzystną wartość w tym algorytmie.

- **Prędkość obliczeń**

  Obliczana ze wzoru:

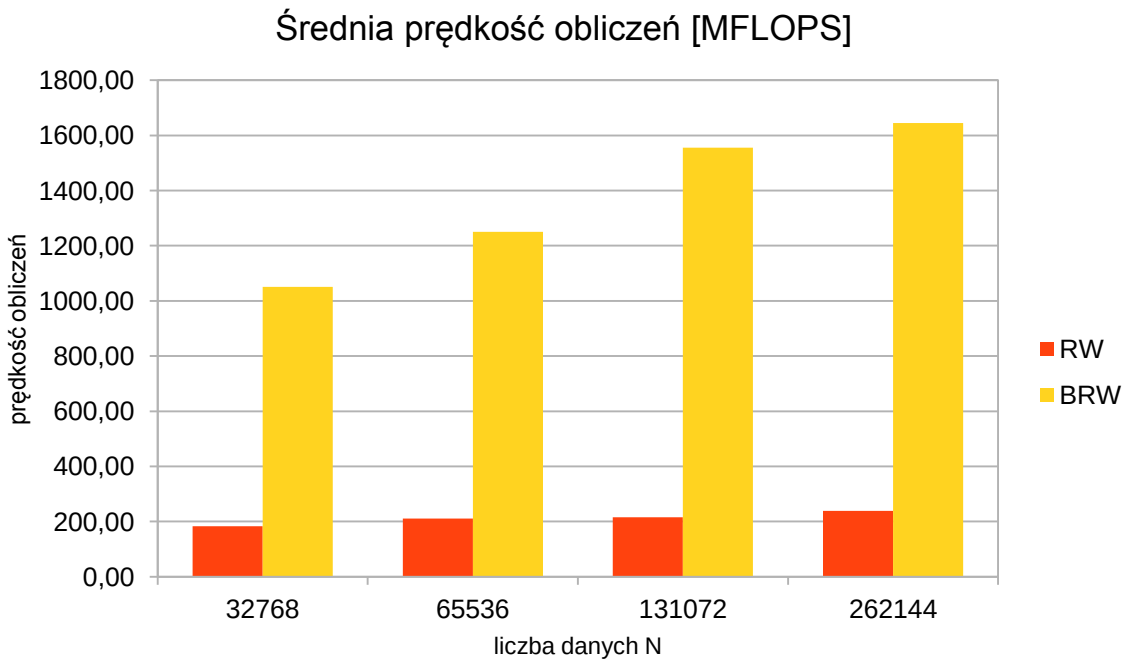
  złożoność_obliczeniowa / czas_obliczeń

  Złożoność obliczeń to w tym przypadku ilość operacji dodawania jaką musi wykonać algorytm, żeby zsumować wszystkie elementy wektora. W tym przypadku wynosi ona **N-1,** gdzie N to rozmiar wektora.

  - Ogólna

    W poniższej tabeli zaprezentowano prędkość obliczeń dla uśrednionych czasów dla metod RW i BRW:

Prędkość obliczeń [MFLOPS]		
N	RW	BRW
32768	183,66	1050,79
65536	211,21	1249,95
131072	215,99	1555,53
262144	238,53	1644,57



- RW

N=32768

bloki/rozmiar	Prędkość obl. [MFLOPS]
1024/32	226,77
256/128	211,57
128/256	190,76
32/1024	135,12

N=65536

bloki/rozmiar	Prędkość obl. [MFLOPS]
2048/32	394,36
512/128	219,26
256/256	195,03
64/1024	148,93

N=131072

bloki/rozmiar	Prędkość obl. [MFLOPS]
4096/32	430,25
1024/128	222,20
512/256	198,67
128/1024	150,13

N=262144

bloki/rozmiar	Prędkość obl. [MFLOPS]
8192/32	435,79
2048/128	344,31
1024/256	201,09
256/1024	151,58

- BRW

N=32768

bloki/rozmiar	Prędkość obl. [MFLOPS]
512/32	959,25
128/128	1151,82
64/256	1156,41
16/1024	969,70

N=65536

bloki/rozmiar	Prędkość obl. [MFLOPS]
1024/32	1126,53
256/128	1398,95
128/256	1392,29
32/1024	1137,17

N=131072

bloki/rozmiar	Prędkość obl. [MFLOPS]
2048/32	1855,11
512/128	1587,32
256/256	1571,78
64/1024	1305,12

N=262144

bloki/rozmiar	Prędkość obl. [MFLOPS]
4096/32	1995,67
1024/128	1683,04
512/256	1676,49
128/1024	1350,40

Jako, że czas przetwarzania lepiej prezentował się dla metody BRW, prędkość obliczeń również jest dużo lepsza dla metody BRW, co widać na wykresie słupkowym. Bezczynność wątków w metodzie RW, jak widać, wpływa na prędkość obliczeń. Analizując dane tabelaryczne dochodzimy do podobnych wniosków co w poprzednim punkcie. W metodzie RW im większa liczba bloków tym lepsza prędkość obliczeń, zaś w metodzie BRW nie można dojść do konkretnego wniosku co do ilości i rozmiarów bloków. Dodatkowo można zauważyć, że prędkości przetwarzania rosną wraz ze wzrostem instancji dla obu algorytmów, lecz dla RW dużo wolniej, niż dla BRW.

- ## Przyspieszenie

  Obliczane ze wzoru:

  czas_przetwarzania_sekwencyjnego / czas_przetwarzania

  Tabela poniżej przedstawia przyspieszenie względem metody sekwencyjnej dla metod RW i BRW. Dla metod RW i BRW zostały wykorzystane uśrednione czasu dla danej instancji wektora.

N	RW	BRW
32768	1,89	10,82
65536	2,17	12,84
131072	2,20	15,88
262144	2,47	17,00

Analizując dane tabelaryczne jak i wykres, można stwierdzić, że przyspieszenie jest oczywiście wartością większa od 1, gdyż metody obliczające na GPU są szybsze od metody sekwencyjnej, która jest wykonywana na CPU. Można również zauważyć, że metoda BRW pokazała się z lepszej strony niż metoda RW, z powodu braku rozbieżności wątków. Innym wnioskiem, który się również nasuwa jest to, że wraz ze wzrostem instancji przyspieszenie również rośnie, co dosadnie widać na wykresie słupkowym.

- **CGMA**

  CGMA (compute to global memory access ratio) jest to parametr, który określa jak długo zajmujemy procesor pobranymi z pamięci globalnej danymi. Liczony jest on jako stosunek liczby operacji zmienno przecinkowych do liczby dostępów do pamięci globalnej. Miara ta jest wykorzystywana do optymalizacji algorytmu. Miarę tą należy maksymalizować w celu zwiększenia przepustowości pamięci. Dlatego warto czasem jest wykonać obliczenia na GPU niż pobierać dane z pamięci globalnej. Jest to spowodowane tym, że GPU dostarcza nam przede wszystkim wiele jednostek arytmetyczno-logicznych (ALU), a nie szybkiej pamięci.

  CGMA obliczamy ze wzoru:

$$\frac{liczba\_operacji}{liczba\_odczytów\_z\_pamięci\_lobalnej + liczba\_zapisów\_do\_pamięci\_lobalnej}$$

  Teoretyczna:
  W każdym kroku następuje jedna operacja sumowania, dwa odczyty i jeden zapis (odczyty i zapis z\do pamięci globalnej, bo tylko takiej używamy), czyli w sumie 2W odczytów i W zapisów.
  Inaczej można to opisać, że w kolejnych krokach następuje odczytów (W - liczba wątków - dla RW = N, dla BRW = N/2): W, W/2, W/4, ... , 2 odczytów - co sumuje się do 2 W, oraz: W/2, W/4, W/8, ... , 1 zapisów - co sumuje się do W.

Liczba operacji to N-1, bo tyle następuje sumowań.
CGMA dla metody RW: N - 1/(2N + N) = 1/3
CGMA dla metody BRW: N - 1/(N + N/2) = 2/3

Praktyczna:
Profiler nie mierzył potrzebnych nam: $liczba\_odczytów\_z\_pamięci\_lobalnej$ oraz $liczba\_zapisów\_do\_pamięci\_lobalnej$. Możemy obliczyć jedynie wartości teoretyczne.

- ## Miara stopnia łączenia dostępów do pamięci globalnej

Kiedy wątek pobiera dane z pamięci globalnej, tworzy transakcje. Oznacza to, że nie pobiera tylko potrzebnej mu aktualnie danej, ale całe słowo. Im mniej transakcji tym mniej czasu wątki czekają na dane.
Miara liczona jest ze wzoru:

$$\frac{liczba\_odczytów\_z\_pamięci\_lobalnej \; + \; liczba\_zapisów\_do\_pamięci\_lobalnej}{liczba\_transakcji\_odczytu + liczba\_transakcji\_zapisu}$$

Znowu miary: : $liczba\_odczytów\_z\_pamięci\_lobalnej$ oraz $liczba\_zapisów\_do\_pamięci\_lobalnej$ przyjmujemy teoretyczne (w poprzednim punkcie je wyliczaliśmy teoretycznie).

W tabeli miarę stopnia łączenia dostępów do pamięci globalnej zapisaliśmy skrótowo jako "miara".

N	metoda	liczba bloków	rozmiar bloku	load transaction	store transactions	load	store	miara
32768	RW	1024	32	24831	8448	65536	32768	**2,95**
32768	RW	256	128	24639	8256	65536	32768	**2,99**
32768	RW	128	256	24607	8224	65536	32768	**2,99**
32768	RW	32	1024	24583	8200	65536	32768	**3,00**
65536	RW	2048	32	33109	11264	131072	65536	**4,43**
65536	RW	512	128	49279	16512	131072	65536	**2,99**
65536	RW	256	256	49215	16448	131072	65536	**2,99**
65536	RW	64	1024	49167	16400	131072	65536	**3,00**
131072	RW	4096	32	66219	22529	262144	131072	**4,43**
131072	RW	1024	128	98559	33024	262144	131072	**2,99**
131072	RW	512	256	98431	32896	262144	131072	**2,99**
131072	RW	128	1024	98335	32800	262144	131072	**3,00**
262144	RW	8192	32	132438	45057	524288	262144	**4,43**
262144	RW	2048	128	131413	44032	524288	262144	**4,48**
262144	RW	1024	256	196863	65792	524288	262144	**2,99**
262144	RW	256	1024	196671	65600	524288	262144	**3,00**
32768	BRW	512	32	9217	2881	32768	16384	**4,06**
32768	BRW	128	128	8449	2257	32768	16384	**4,59**
32768	BRW	64	256	8321	2153	32768	16384	**4,69**
32768	BRW	16	1024	8225	2075	32768	16384	**4,77**
65536	BRW	1024	32	18433	5761	65536	32768	**4,06**
65536	BRW	256	128	16897	4513	65536	32768	**4,59**
65536	BRW	128	256	16641	4305	65536	32768	**4,69**
65536	BRW	32	1024	16449	4149	65536	32768	**4,77**
131072	BRW	2048	32	24579	7683	131072	65536	**6,09**
131072	BRW	512	128	33793	9025	131072	65536	**4,59**
131072	BRW	256	256	33281	8609	131072	65536	**4,69**
131072	BRW	64	1024	32897	8297	131072	65536	**4,77**
262144	BRW	4096	32	49158	15365	262144	131072	**6,09**
262144	BRW	1024	128	67585	18049	262144	131072	**4,59**
262144	BRW	512	256	66561	17217	262144	131072	**4,69**
262144	BRW	128	1024	65793	16593	262144	131072	**4,77**

Jak można było się spodziewać - metoda BRW wypada tutaj korzystniej, cechuje się kilkukrotnie (>2) mniejszą liczbą transakcji (która to liczba oczywiście rośnie wraz z rozmiarem N) przy tylko dwukrotnie mniejszej liczbie odczytów i zapisów do\z pamięci globalnej.
Algorytmy korzystają z pamięci globalnej.
BRW pobiera dane z pamięci z dwóch komórek, które są od siebie znacznie oddalone (widać to na schemacie), lecz kolejny wątek (o identyfikatorze o 1 większym) pobiera dane z komórek, które są tuż obok tych, z których korzystał poprzedni proces. Skoro używamy wspólnej pamięci globalnej, to że pierwszy wątek pobiera całe słowo, w którym znajdują się także dane potrzebne drugiemu wątkowi i kolejnym. Gdy wątki pobiorą dane z całego słowa, kolejny pobiera nowe słowo i te po nim znów mogą korzystać z zawartych w nim danych bez potrzeby pobierania nowego słowa.
W przypadku algorytmu RW sytuacja wygląda gorzej. W pierwszym kroku wątek pobiera dwie sąsiednie komórki, które na pewno będą wchodziły w skład jednego słowa. W kolejnym kroku dwie potrzebne komórki są oddalone od siebie o 2 pozycje, potem o 4, o 8 itd. Szybko przestają mieścić się w jednym słowie i spada miara łączenia dostępów do pamięci globalnej.
Sytuację zawsze poprawia (w oby algorytmach) rozmiar bloku = 32. Jest mało wątków, więc w ramach jednego bloku łączenie dostępów do pamięci będzie częste.

- ## Zajętość multiprocesora

Jest to stosunek aktywnych wiązek do maksymalnej liczby aktywnych wiązek. Pierwszym wykonanym przez nas badaniem w tym punkcie było teoretyczne przeliczenie zajętości multiprocesora w zależności od wielkości bloku. Obliczenia teoretyczne wykonaliśmy przy pomocy CUDA Occupancy Calculator. Kolejnym etapem było zbadanie zajętości multiprocesora przy pomocy Profilera.

N	metoda	liczba bloków	rozmiar bloku	zajętość teoretyczna	zajętość zmierzona
32768	RW	1024	32	50%	57,49%
32768	RW	256	128	100%	56,86%
32768	RW	128	256	100%	55,19%
32768	RW	32	1024	100%	53,27%
65536	RW	2048	32	50%	48,79%
65536	RW	512	128	100%	57,82%
65536	RW	256	256	100%	57,61%
65536	RW	64	1024	100%	56,97%
131072	RW	4096	32	50%	61,97%
131072	RW	1024	128	100%	58,83%
131072	RW	512	256	100%	58,80%
131072	RW	128	1024	100%	57,59%
262144	RW	8192	32	50%	64,26%
262144	RW	2048	128	100%	49,11%
262144	RW	1024	256	100%	59,65%
262144	RW	256	1024	100%	58,10%
32768	BRW	512	32	50%	52,67%
32768	BRW	128	128	100%	50,67%
32768	BRW	64	256	100%	50,80%
32768	BRW	16	1024	100%	45,44%
65536	BRW	1024	32	50%	55,41%
65536	BRW	256	128	100%	54,49%
65536	BRW	128	256	100%	53,82%
65536	BRW	32	1024	100%	49,65%
131072	BRW	2048	32	50%	47,52%
131072	BRW	512	128	100%	55,63%
131072	BRW	256	256	100%	55,62%
131072	BRW	64	1024	100%	54,70%
262144	BRW	4096	32	50%	60,79%
262144	BRW	1024	128	100%	57,36%
262144	BRW	512	256	100%	57,15%
262144	BRW	128	1024	100%	56,24%

Jak widać z danych tabelarycznych, zajętość multiprocesora i dla RW i BRW plasuje się w okolicach 50-60%. Różnica pomiędzy teoretyczną wartością, a praktyczną może wynikać z opóźnień związanych z dostępem do pamięci globalnej. Jako, że w obu algorytmów korzystamy tylko z takowej pamięci to ta różnica jest bardzo widoczna.

- **współczynnik poziomu rozbieżności przetwarzania wiązek w kodzie**

Duży współczynnik rozbieżności wiązek w kodzie ma spory wpływ na efektywność przetwarzania. Obliczane:

(liczba_gałęzi – liczba_gałęzi_rozbieżnych) / liczba_gałęzi

Czyli im mniej gałęzi rozbieżnych (im bliżej 100%) tym lepsze przetwarzanie. Rozbieżność wątków powstaje w kodzie przy każdej instrukcje warunkowej z identyfikatorem wątku w roli głównej, kiedy wiązka musi przetwarzać kilka wersji kodu. W naszych wersjach kodu, dla metody BRW współczynnik powinien być bliski 100%, zaś dla metody RW wręcz przeciwnie. Niestety, podczas robienia pomiarów Profiler pokazywał nam dla obu metod za każdym razem wartość współczynnika w okolicach 99-100%. Dla BRW jest to dobrze, lecz dla naszej metody RW nie powinno się tak dziać. Gdy wczytaliśmy się w dokumentacje Profilera wyszło, że parametr divergent_branches (mówiący o ilości rozbieżnych gałęzi) jest liczony w specyficzny sposób. Otóż parametr ten jest inkrementowany jeśli co najmniej jeden wątek w bloku się rozbiega, czyli na każdy blok wartość ta jest inkrementowana tylko raz. Jest to bardzo niemiarodajne, gdyż przez to nie wiemy jaka jest faktyczna liczba tych rozbieżnych gałęzi.
Na poparcie tego, możemy powiedzieć, że widać, że nasza metoda RW ma rozbieżność wątków, gdyż było to wytłumaczone w opisie algorytmu. Od metody BRW różni się zaledwie niewiele. Wprowadzony jest dodatkowy operator modulo, dzięki któremu tylko co drugi wątek wykonuje pracę. Innym argumentem na poparcie tego, że nasza metoda RW posiada rozbieżność wątków jest to, że jest znaczny spadek efektywności przetwarzania w porównaniu do BRW, który wykazaliśmy poprzednimi pomiarami.

6. Wnioski

Tematem naszego zadania było porównanie metod równoległego sumowania wektora z rozbieżnością wątków i bez rozbieżności wątków. Najważniejszym wnioskiem uzyskanym na podstawie tego zadania, jest to, że metoda BRW okazuje się lepsza na wszystkich omawianych przez nas parametrach. Innym wnioskiem jest również to, że równoległe sumowanie wektora na GPU jest dużo bardziej efektywniejsze niż obliczanie sekwencyjne na CPU. Różnicę między metodami BRW i RW wynikają właśnie z rozbieżności wątków w metodzie RW, która psuje efektywność. Ostatnim wnioskiem wyciągniętym przez nas jest to, że w większości przypadków, dla takiej samej instancji wektora ale innej ilości bloków w wywołaniu kernela, bardziej efektywne były te wywołania z większą ilością bloków.