Sprawozdanie
Przetwarzanie równoległe
Projekt 2 PKG

Dominik Doberski, Artur Olejnik

25 czerwca 2019

1 Wstęp

1.1 Temat

Mnożenie macierzy - ukrycie kosztów transferu danych w czasie obliczeń (dla 5. wersji kodu), badania należy wykonać dla przetwarzania równoległego dla zbioru tablic jednocześnie dostarczanych, obliczanych i pobieranych do/z systemu PKG, porównanie z prędkością przetwarzania dla wersji 3.

- 5. wersja kodu – grid wieloblokowy, obliczenia przy wykorzystaniu pamięci współdzielonej bloku wątków, zrównoleglenie obliczeń i transferu danych między pamięciami: operacyjną procesora, a globalną karty,
- 3. wersja kodu – grid wieloblokowy, obliczenia przy wykorzystaniu pamięci współdzielonej bloku wątków.

1.2 Autorzy

Dominik Doberski	132207	dominik.doberski@student.put.poznan.pl
Artur Olejnik	122402	artur.olejnik@student.put.poznan.pl

Grupa dziekańska I3
Termin zajęć: poniedziałek 16:50

1.3 Opis wykorzystywanej karty graficznej

Do wykonania pomiarów wykorzystano program CodeXL. Testy zrealizowano na jednostce obliczeniowej o poniższej specyfikacji:

- Nazwa: Nvidia GeForce GTX 1060

- Pamięć: 3072MB
- Typ: dedykowana
- CC – możliwości obliczeniowe: 6.1
- Liczba SM: 9
- Liczba rdzeni: 1152
- Maksymalna liczba wątków na multiprocesor: 2048
- Maksymalna liczba wątków na blok: 1024
- Rozmiar osnowy: 32
- Maksymalny rozmiar bloku: 1024x1024x64

2 Analiza algorytmu

2.1 Kluczowe fragmenty kodu

2.1.1 Algorytm dla 3. wersji kodu

Grid wieloblokowy, obliczenia przy wykorzystaniu pamięci współdzielonej bloku wątków:

```
1  template <int BLOCK_SIZE> __global__ void
2  matrixMul3(float *C, float *A, float *B, int width) {
3      int bx = blockIdx.x;
4      int by = blockIdx.y;
5
6      int tx = threadIdx.x;
7      int ty = threadIdx.y;
8
9      int aBegin = width * BLOCK_SIZE * by;
10     int aEnd   = aBegin + width - 1;
11     int aStep  = BLOCK_SIZE;
12     int bBegin = BLOCK_SIZE * bx;
13     int bStep  = BLOCK_SIZE * width;
14
15     float Csub = 0;
16
17     __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
18     __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
19
20     for (int a = aBegin, b = bBegin;
21          a <= aEnd;
22          a += aStep, b += bStep) {
```

```
23          As[ty][tx] = A[a + width * ty + tx];
24          Bs[ty][tx] = B[b + width * ty + tx];
25
26          __syncthreads();
27
28  #pragma unroll
29          for (int k = 0; k < BLOCK_SIZE; ++k) {
30              Csub += As[ty][k] * Bs[k][tx];
31          }
32
33          __syncthreads();
34      }
35
36      int c = width * BLOCK_SIZE * by + BLOCK_SIZE * bx;
37      C[c + width * ty + tx] = Csub;
38  }
```

Funkcja 1: 3. wersja kodu

Funkcja ma za zadanie wyznaczenie jednej wartości macierzy C przez jeden wątek. Argumentami powyższej funkcji są wskaźniki na tablice A, B i C, oraz wartość *width*, która oznacza wielkość każdego wymiaru każdej macierzy.

Pierwszą czynnością, która wykonuje powyższa funkcja jest wyznaczenie numeru bloku w 3. i 4. linii kodu, następnie w 6. i 7. wyznaczane jest położenie wątku wewnątrz tego bloku. W kolejnych liniach wyznaczane są adresy macierzy A i B potrzebne do obliczeń. W 15. linii tworzymy zmienną lokalną $Csub$, która posłuży nam do obliczenia wartości kolejnych komórek macierzy wynikowej C. W linii 17. i 18. deklarowana jest pamięć współdzielona dla obu macierzy.

W pętli funkcja kopiuje wartości z obu macierzy do pamięci współdzielonej, a zaraz po tym następuje synchronizacja wątków w obrębie bloku (linie 23-26), która zapewnia nas o tym, że do pamięci współdzielonej przypisane zostały wszystkie wymagane elementy. Dyrektywa *#pragma unroll* posłuży do odwinięcia ciała pętli (linie 29-31) w szereg pojedynczych instrukcji, których zadaniem jest wyliczenie częściowego wyniku wykorzystując dane z pamięci współdzielonej. Następnie ponownie występuje bariera synchroniczna, która zapobiegnie nadpisaniu tej pamięci.

Po uzyskaniu wyniku częściowego funkcja zapisze go, już poza pętlą w 37. linii kodu, do pamięci głównej w odpowiedniej komórce macierzy C.

2.1.2 Algorytm dla 5. wersji kodu

Grid wieloblokowy, obliczenia przy wykorzystaniu pamięci współdzielonej bloku wątków, zrównoleglenie obliczeń i transferu danych między pamięciami: operacyjną procesora, a globalną karty:

```
39  template <int BLOCK_SIZE> __global__ void
40  matrixMulCUDA(float *C, float *A, float *B, int width) {
41      int bx = blockIdx.x;
```

```
42      int by = blockIdx.y;
43
44      int tx = threadIdx.x;
45      int ty = threadIdx.y;
46
47      int aBegin = width * BLOCK_SIZE * by;
48      int aEnd = aBegin + width - 1;
49      int aStep = BLOCK_SIZE;
50      int bBegin = BLOCK_SIZE * bx;
51      int bStep = BLOCK_SIZE * width;
52
53      float Csub = 0;
54
55      __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
56      __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
57
58      __shared__ float A_shared[BLOCK_SIZE][BLOCK_SIZE];
59      __shared__ float B_shared[BLOCK_SIZE][BLOCK_SIZE];
60
61      A_shared[ty][tx] = A[aBegin + width * ty + tx];
62      B_shared[ty][tx] = B[bBegin + width * ty + tx];
63
64      for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) {
65          As[ty][tx] = A_shared[ty][tx];
66          Bs[ty][tx] = B_shared[ty][tx];
67
68          __syncthreads();
69
70          A_shared[ty][tx] = A[a + width * ty + tx];
71          B_shared[ty][tx] = B[b + width * ty + tx];
72
73  #pragma unroll
74          for (int k = 0; k < BLOCK_SIZE; ++k) {
75              Csub += As[ty][k] * Bs[k][tx];
76          }
77
78          __syncthreads();
79      }
80
81      int c = width * BLOCK_SIZE * by + BLOCK_SIZE * bx;
82      C[c + width * ty + tx] = Csub;
83  }
```

Funkcja 2: 5. wersja kodu

Powyższa funkcja jest zmodyfikowaną wersją poprzedniej. Jej zadaniem jest również wykorzystanie pamięci współdzielonej bloku wątków do wygenerowania

częściowego wyniku macierzy C. Różnica polega na dodatkowej funkcjonalności, którą jest zrównoleglenie obliczeń i transfer danych pomiędzy CPU i GPU.

W linii 58. i 59. deklarujemy dodatkowe pamięci współdzielone dla macierzy A i B, do których równolegle z obliczeniami wpisujemy dane kolejnych bloków. Przed wykonaniem każdej iteracji pętli do tych pamięci zapisywane są dane z macierzy (w 70. i 71. wierszu). Tą czynność należy wykonać również przed pierwszą iteracją (61. i 62. linia). Oznacza to, że wątki po pobraniu danych do pamięci współdzielonej nie muszą się synchronizować, ponieważ w danej iteracji nie pobierają danych wykorzystywanych od razu do obliczeń, tak jak to było w poprzednim algorytmie. W tym przypadku dane pobierane do pamięci współdzielonej z macierzy będą wykorzystywane dopiero w kolejnej iteracji, w następnym fragmencie.

2.2 Analiza dostępu do pamięci

2.2.1 Dostęp do pamięci dla 3. wersji kodu

Dostęp do danych odbywa się za pomocą pamięci współdzielonej co sprawia, że jeden wątek nie musi sam pobierać wszystkich danych z macierzy, a może do tego wykorzystać dane pobrane przez inne wątki. Takie podejście jest zdecydowanie szybsze niż wykorzystywanie tylko i wyłącznie pamięci globalnej, jednak sprawia, że wymagana jest synchronizacja po pobieraniu danych z macierzy. Wymagana jest również synchronizacja na końcu każdej iteracji, aby zapobiec podmienieniu zawartości pamięci współdzielonej, kiedy inny wątek nie zdążył wykonać jeszcze wszystkich obliczeń.

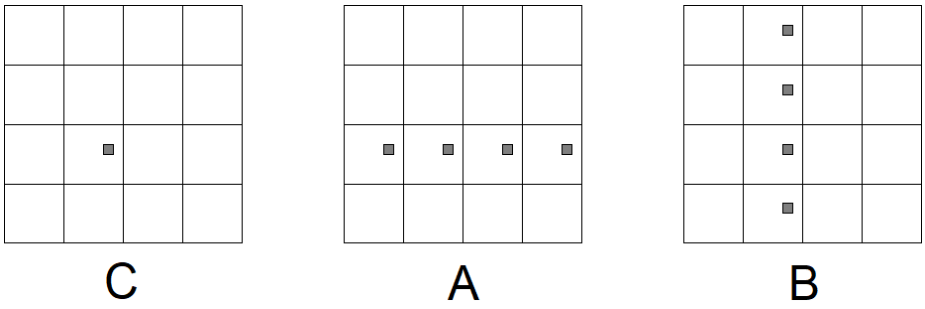
2.2.2 Dostęp do pamięci dla 5. wersji kodu

W tym przypadku dostęp odbywa się w trochę inny sposób. Dane z pemięci globalnej są pobierane przed iteracją, w której te dane zostaną użyte i zostają one zapisane do innego obszaru pamięci współdzielonej. Po tej operacji nie jest wymagana synchronizacja, gdyż ta pamięć nie jest wykorzystywana do obliczeń. We właściwej iteracji, która te dane potrzebuje, dane są przepisywane do nowego obszaru pamięci już bez bezpośredniego dostępu do pamięci globalnej. Ten obszar będzie już użyty do obliczeń, dokładnie tak jak w 1. funkcji, zatem synchronizacja po przepisaniu tych danych jest wymagana, tak jak ta na koniec każdej iteracji pętli zewnętrznej.

2.3 Analiza dostępu do danych

2.3.1 Dostęp do danych dla 3. wersji kodu

W celu uzyskania wyniku dla częściowego każdy wątek musi w każdej iteracji pętli zewnętrznej pobrać po 1 elemencie z macierzy A i B. To, którym elementem macierzy C zajmuje się konkretny wątek, zależne jest od położenia bloku i wątku wewnątrz tego bloku.



Rysunek 1: Bezpośredni dostęp do danych w jednym wątku

W każdej iteracji pętli zewnętrznej wątek musi pobrać dane z innego fragmentu macierzy. Wewnątrz każdego z tych fragmentów położenie pobieranego elementu jest stałe i odpowiada położeniu wątku wewnątrz bloku wątków. Powyższy rysunek przedstawia, które elementy musi pobrać wątek z macierzy A i B dla wyznaczenia przykładowego elementu macierzy C.

Oczywiście pobrane wartości nie są wykorzystywane w tej funkcji bezpośrednio. Nie jest ich też wystarczająco tyle, aby móc od razu wyznaczyć wynik cząstkowy. Należy się posłużyć elementami wyznaczonymi w pamięci współdzielonej bloku wątków. Cały blok wątków będzie w stanie odczytać elementy wszystkich fragmentów w w wierszu fragmentów macierzy A i kolumnie fragmentów macierzy B. W każdej iteracji pamięć współdzielona przechowuje informacje o wartościach z całego jednego fragmentu dla macierzy A i B.

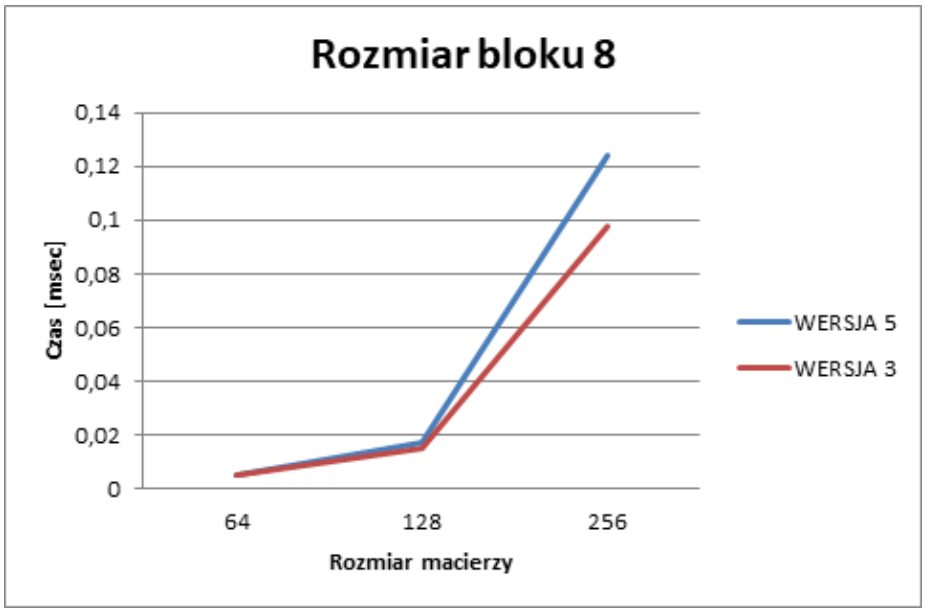
Wiemy, że do wyznaczenia jednego elementu macierzy C potrzebujemy sumy iloczynów odpowiadających elementów w tym samym wierszu macierzy A i kolumnie macierzy B, zatem otrzymane wartości po przebiegu całej pętli zewnętrznej pozwolą na wyznaczenie wartości dla całego fragmentu macierzy C, którym blok wątków się zajmuje.

2.3.2 Dostęp do danych dla 5. wersji kodu

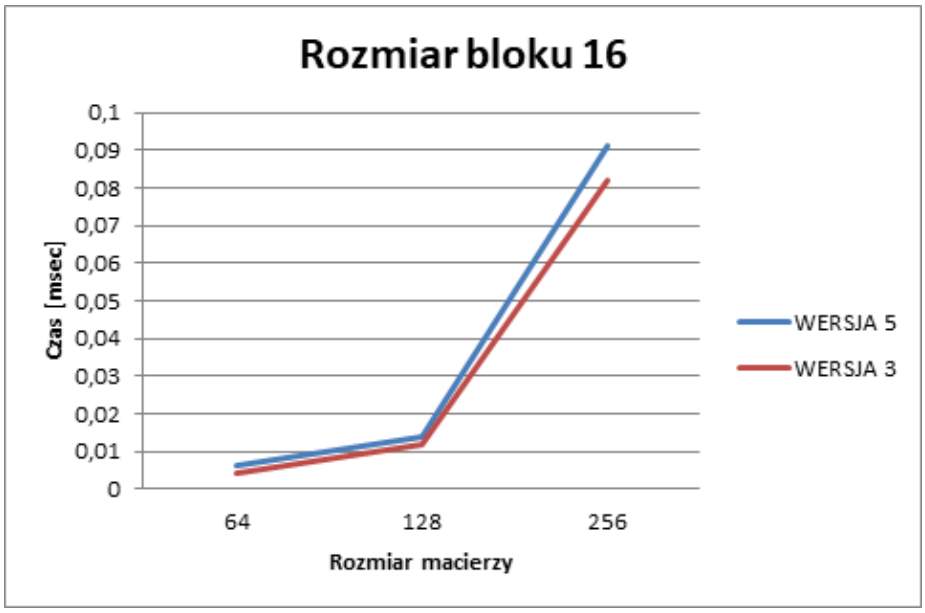
Dla 2. funkcji dostęp do danych odbywa się w przybliżony sposób. Zmienia się miejsce, gdzie te dane są przetrzymywane w pamięci współdzielonej, oraz moment w którym te dane są pobierane (przed iteracją pętli zewnętrznej).

3 Wyniki przeprowadzonych eksperymentów

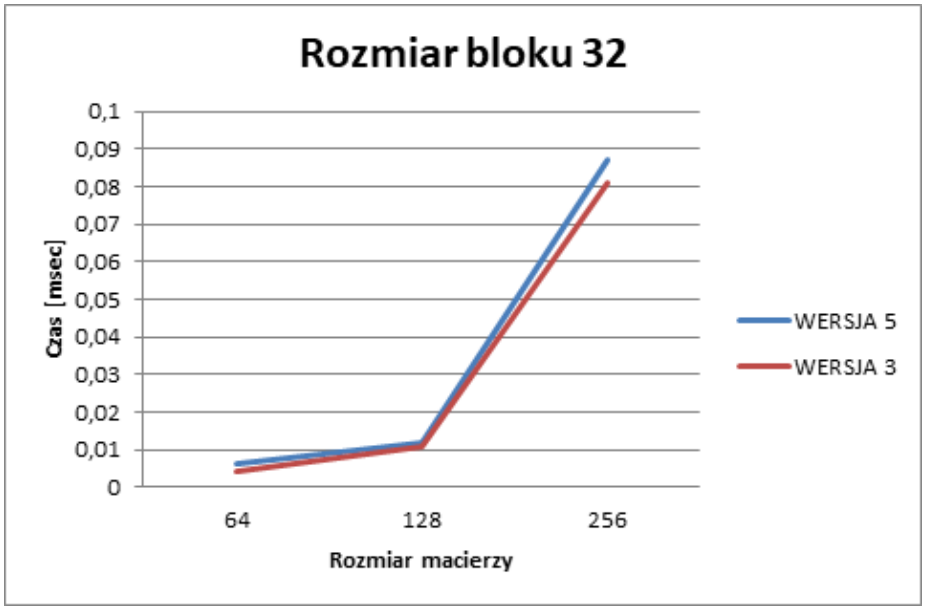
3.1 Czas przetwarzania



Rysunek 2: Czas przetwarzania dla bloku o rozmiarze 8x8



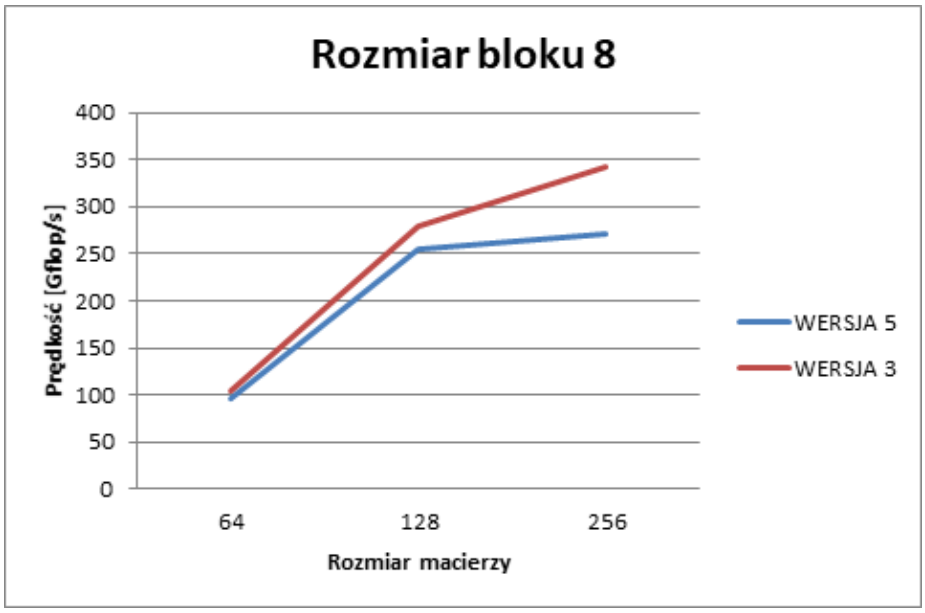
Rysunek 3: Czas przetwarzania dla bloku o rozmiarze 16x16



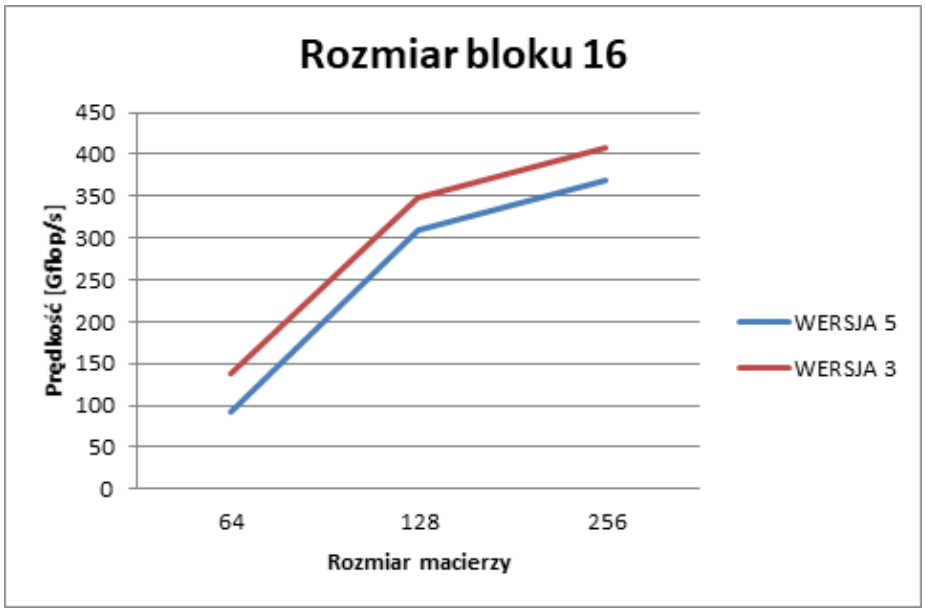
Rysunek 4: Czas przetwarzania dla bloku o rozmiarze 32x32

Jak widać na powyższych wykresach czas przetwarzania dla obydwu metod jest podobny. Różnice są niemal niezauważalne z niewielką przewagą dla algorytmu w wersji 3. Zwiększenie rozmiaru bolku przyspiesza przetwarzanie, jednak jest to dużo bardziej zauważalne między blokiem o rozmiarze 8 i 16, niż 16 i 32 gdzie zmiana jest nieznaczna. Zgodnie z oczekiwaniami macierz o większym rozmiarze jest liczona dłużej. Różnica między macierzą o wymiarach 64x64 i 128x128 jest niewielka, co pokazuje niepełne wykorzystanie możliwości multiprocesora przy zbyt małej instancji. Dopiero przy przetwarzaniu macierzy o rozmiarze 256x256 można zauważyć znaczące zwiększenie czasu przetwarzania.

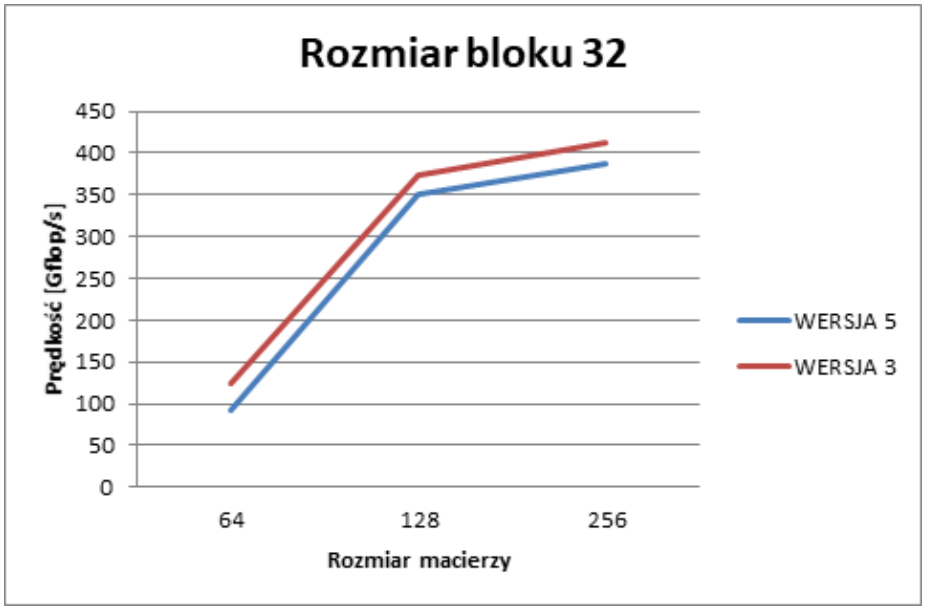
3.2 Prędkość przetwarzania



Rysunek 5: Czas przetwarzania dla bloku o rozmiarze 8x8



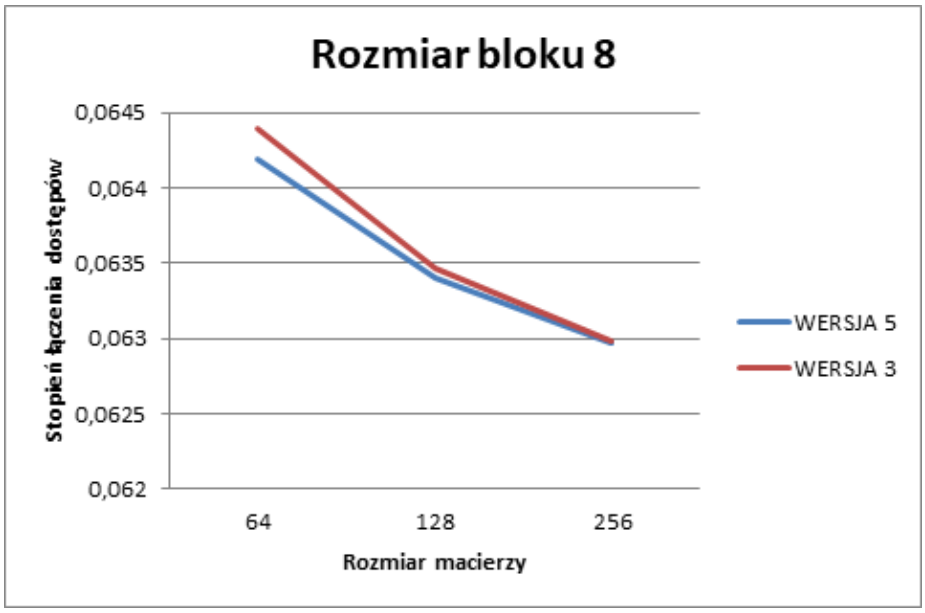
Rysunek 6: Czas przetwarzania dla bloku o rozmiarze 16x16



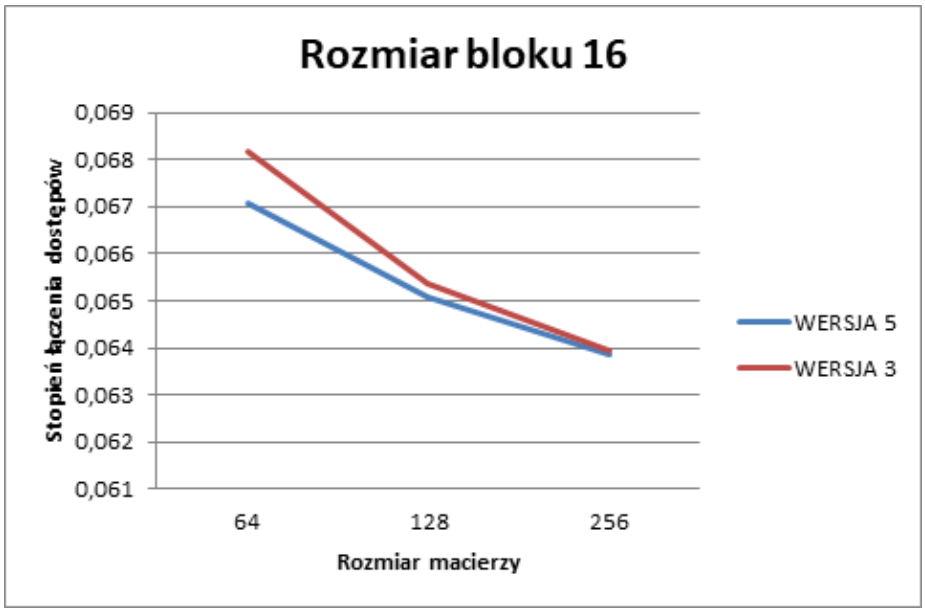
Rysunek 7: Czas przetwarzania dla bloku o rozmiarze 32x32

Prędkość przetwarzania dla bloków o rozmiarach 32 i 16 jest bardzo podobna, co oznacza, że w obydwu przypadkach multiprocesor był wykorzystywany najlepiej jak można. W wersji z blokiem o rozmiarze 8, prędkość przetwarzania jest znacząco niższa, prawdopodobnie blok był zbyt mały by zająć cały multiprocesor ze względu na ogrniczenie na ilość bloków. Najniższe prędkośći przetwarzania można zaobserwować w przypadku najmniejszej instancji problemu. Dla zbyt małej ilości danych przetwarzanie na karcie graficznej działa gorzej.

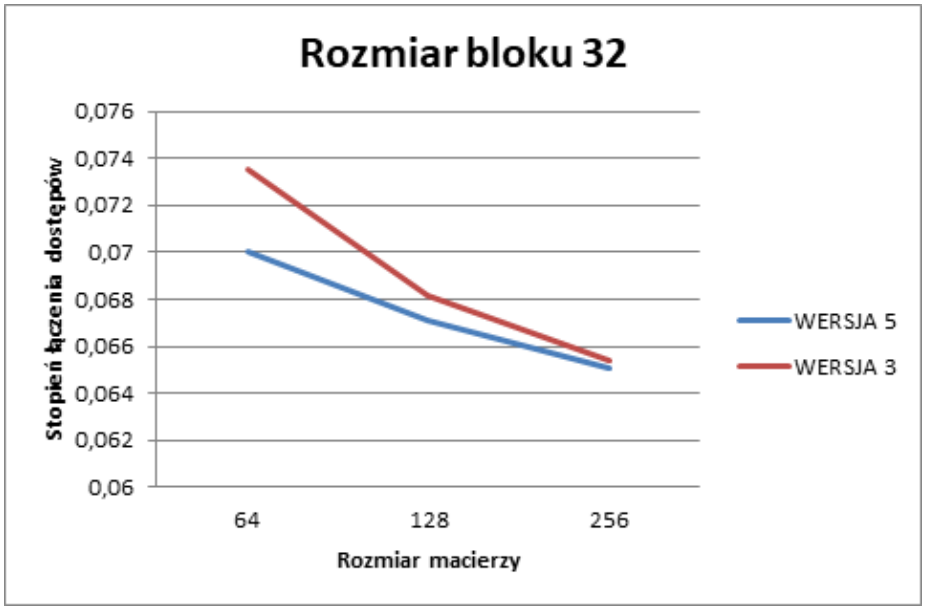
3.2.1 Stopień łączenia dostępów



Rysunek 8: Stopień łączenia dostępów do pamięci dla bloku o rozmiarze 8x8



Rysunek 9: Stopień łączenia dostępów do pamięci dla bloku o rozmiarze 16x16



Rysunek 10: Stopień łączenia dostępów do pamięci dla bloku o rozmiarze 32x32

Wersja	Rozmiar bloku	Rozmiar macierzy	Global Load Transactions	Globall Store Transactions	global load	global store
5	8	64	36866	1024	2304	128
5	8	128	278530	4096	17408	512
5	8	256	2162690	16384	135168	2048
5	16	64	20482	512	1280	128
5	16	128	147458	2048	9216	512
5	16	256	1114114	8192	69632	2048
5	32	64	12290	512	768	128
5	32	128	81922	2048	5120	512
5	32	256	589826	8192	36864	2048
3	8	64	32770	1024	2048	128
3	8	128	262146	4096	16384	512
3	8	256	2097154	16384	131072	2048
3	16	64	16386	512	1024	128
3	16	128	131074	2048	8192	512
3	16	256	1048578	8192	65536	2048
3	32	64	8194	512	512	128
3	32	128	65538	2048	4096	512
3	32	256	524290	8192	32768	2048

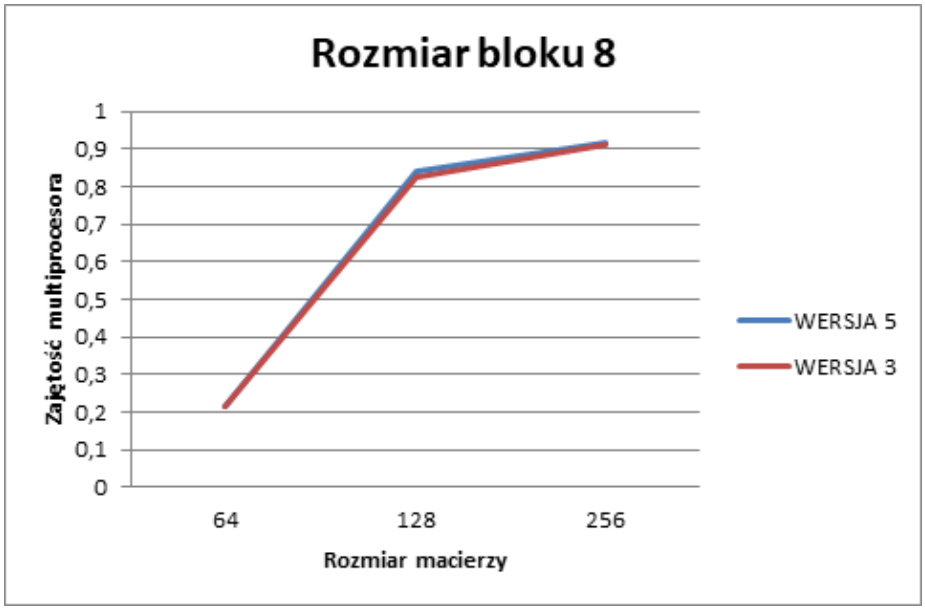
Rysunek 11: Wyniki obliczeń stopienia łączenia dostępów do pamięci

Stopień łączenia dostępów do pamięci wynosi:

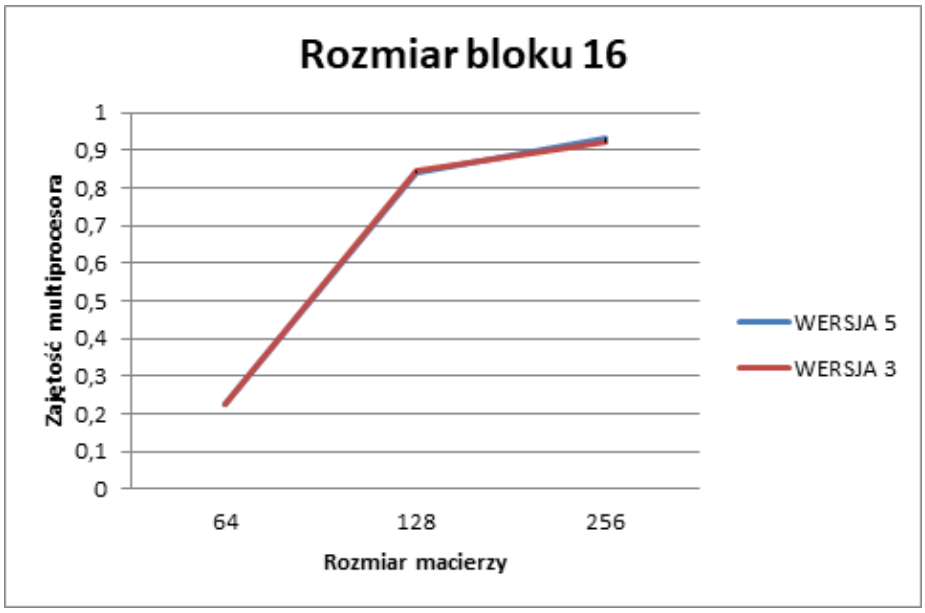
$$\frac{global\ load + global\ store}{global\ load\ transactions + global\ store\ transactions}$$

Stopień łączenia dostępów do pamięci oznacza jak często potrzebujemy jednocześnie danych zawartych w jednej linii. Takie zapytania do pamięci można łączyć w transakcję i dzięki temu ograniczyć liczbę odwołań do wolnej pamięci. Wraz ze wzrostem rozmiaru macierzy liczba łączonych dostępów maleje. Wzrasta co prawda liczba wykonanych transakcji, jednak nieproporcjonalnie do wzrastającej liczby wszystkich odwołań. Zwiększenie rozmiaru bloku nieznacznie poprawia sytuację, ze względu na miejszą liczbę wątków.

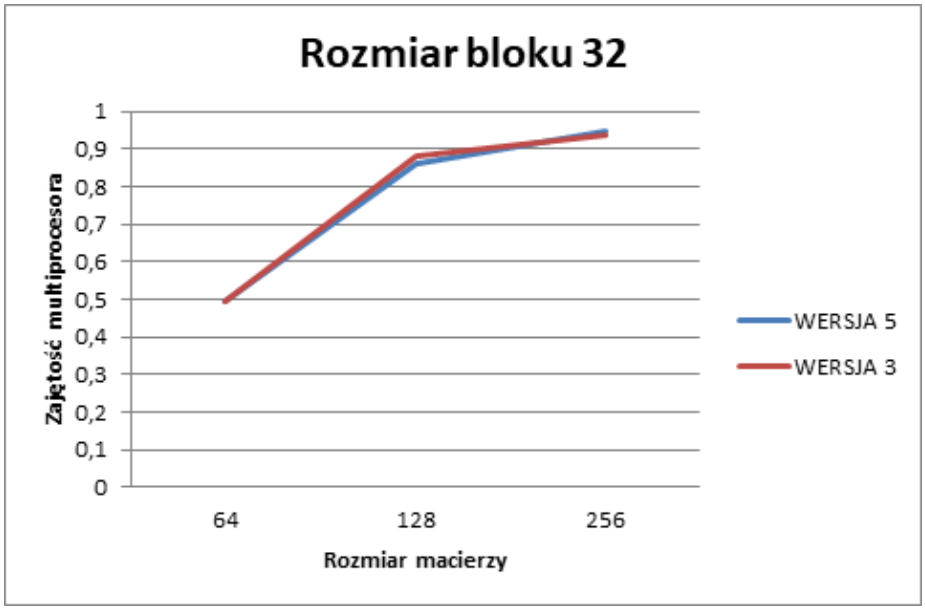
3.3 Zajętość multiprocesora



Rysunek 12: Zajętość multiprocesora dla bloku o rozmiarze 8x8



Rysunek 13: Zajętość multiprocesora dla bloku o rozmiarze 16x16



Rysunek 14: Zajętość multiprocesora dla bloku o rozmiarze 32x32

Wersja	Rozmiar bloku	Rozmiar macierzy	Achieved Occupancy
5	8	64	0.216
5	8	128	0.841
5	8	256	0.915
5	16	64	0.224
5	16	128	0.839
5	16	256	0.93
5	32	64	0.494
5	32	128	0.862
5	32	256	0.947
3	8	64	0.215
3	8	128	0.826
3	8	256	0.913
3	16	64	0.224
3	16	128	0.847
3	16	256	0.924
3	32	64	0.494
3	32	128	0.883
3	32	256	0.937

Rysunek 15: Wyniki odczytu zajętości multiprocesora

Zajętość multiprocesora odczytujemy z wartości Achieved Occupancy z programu Nvidia visual profiler. Oznacza ona stosunek aktywnych wiązek do maksymalnej liczby wiązek w SM. Jak widać na powyższych wykresach zgodnie z przewidywaniami dla macierzy o rozmiarze 64x64 mamy zbyt mało wiązek by móc w pełni wykorzystać możliwości multiprocesora. Dopiero rozmiar 128x128 i 256x256 daje wartości bliskie 1, czyli maksymalnej możliwej zajętości. Zwiększenie rozmiaru bloku również nieznacznie poprawia tę wartość.

3.4 Współczynnik poziomu rozbieżności przetwarzania

Wersja	Rozmiar bloku	Rozmiar macierzy	branch	divergent branch
5	8	64	1280	0
5	8	128	9216	0
5	8	256	69632	0
5	16	64	768	0
5	16	128	5120	0
5	16	256	36864	0
5	32	64	512	0
5	32	128	3072	0
5	32	256	20480	0
3	8	64	1280	0
3	8	128	9216	0
3	8	256	69632	0
3	16	64	768	0
3	16	128	5120	0
3	16	256	36864	0
3	32	64	512	0
3	32	128	3072	0
3	32	256	20480	0

Rysunek 16: Wyniki odczytu współczynnika rozbieżności przetwarzania

Współczynnik poziomu rozbieżności przetwarzania wynosi:

$$\frac{branch - divergent\ branch}{branch}$$

Współczynnik poziomu rozbieżności przetwarzania wiązek w kodzie to liczba gałęzi rozbierznych które powstają w kodzie przy instrukcjach warunkowych. Wówczas wiązka musi przetwarzać różne wersje kodu co bardzo źle wpływa na efektywność przetwarzania. Jeżeli wszystkie wątki wybiorą tę samą wersję rozwiazania instrukcji warunkowej, to nie będzie gałęzi rozbierznych. Obie wersje kodu które testujemy nie posiadają instrukcji warunkowych, dlatego zgodnie z oczekiwaniami wartość ta, jest zawsze równa 1.

4 Wnioski końcowe

Zrównoleglenie obliczeń nie wpłynęło w pozytywnie na działanie algorytmu.