Urszula Walińska 127216
Julia Będziechowska 127306

Sprawozdanie

Programowanie CUDA na NVIDIA GPU

Temat 1: Mnożenie macierzy, badanie prędkości obliczeń w zależności od typu wykorzystywanej pamięci (wersje 2, 3, i 4)

2. grid wieloblokowy, jeden wątek oblicza jeden element macierzy wynikowej, obliczenia przy wykorzystaniu pamięci globalnej,
3. grid wieloblokowy, jeden wątek oblicza jeden element macierzy wynikowej, obliczenia przy wykorzystaniu pamięci współdzielonej bloku wątków,
4. grid wieloblokowy, jeden wątek oblicza jeden element macierzy wynikowej, obliczenia danych przy wykorzystaniu pamięci współdzielonej bloku wątków ze zrównolegleniem obliczeń i pobierania danych z pamięci globalnej w ramach każdego bloku wątków,

Prowadzący: dr hab. Inż. Rafał Walkowiak

1. **Opis wykorzystywanej karty graficznej**

Obliczenia mnożenia macierzy były wykonywane na karcie graficznej GeForce GTX 850M. Dla uzyskania informacji o karcie graficznej skorzystałyśmy z programu deviceQuery.

- Compute Capability: 5.0
- Pamięć globalna: 2048 megabajtów
- 5 multiprocesorów
- 128 rdzeni na multiprocesor
- Pamięć współdzielona dla bloku: 49152 bajtów
- Liczba rejestrów na blok: 65536
- Rozmiar wiązki: 32 wątki
- Maksymalna liczba wątków na multiprocesor: 2048
- Maksymalna liczba wątków na blok: 1024
- Maksymalny rozmiar bloku: 1024 x 1024 x 64
- Częstotliwość taktowania zegara rdzenia: 902 MHz
- Częstotliwość taktowania zegara pamięci: 1001 MHz
- Rozmiar pamięci L2 Cache: 2097152 bajtów
- Architektura: Maxwell (pierwsza generacja)

Ograniczenia dla Compute Capability 5.0:

- Maksymalna liczba bloków na multiprocesor: 32
- Pamięć lokalna dla wątku: 512 KB

2. Wersje programu mnożenia macierzy:

a. grid wieloblokowy, jeden wątek oblicza jeden element macierzy wynikowej, obliczenia przy wykorzystaniu pamięci globalnej

```
template <int BLOCK_SIZE>
__global__ void matrixMulGLOBAL(float *C, float *A, float *B, int WIDTH)
{
	int row = blockIdx.y * blockDim.y + threadIdx.y;
	int col = blockIdx.x * blockDim.x + threadIdx.x;
	float C_local = 0;

	for (int k = 0; k < WIDTH; k++)
	{
		C_local += A[row * WIDTH + k] * B[k * WIDTH + col];
	}

	C[row * WIDTH + col] = C_local;
}
```

Każdy wątek w danym momencie wylicza jeden element macierzy wynikowej. Wybór elementu zależy od numeru bloku (blockDim.x i blockDim.y) oraz położenia wątku wewnątrz niego (threadIdx.x i threadIdx.y). W pętli wątek mnoży odpowiednie wartości elementów z macierzy A oraz B zajmujących miejsce w pamięci globalnej (przekazanych jako parametr funkcji) i dodaje wynik do lokalnej zmiennej pomocniczej C_local, żeby po zakończeniu wykonywania pętli umieścić obliczoną wartość w odpowiednim miejscu w macierzy wynikowej, również przechowywanej w pamięci globalnej. Ostatecznie uzyskujemy odwołania do kolejnych elementów macierzy w pamięci globalnej co iterację pętli.

b. grid wieloblokowy, jeden wątek oblicza jeden element macierzy wynikowej, obliczenia przy wykorzystaniu pamięci współdzielonej bloku wątków

```
template <int BLOCK_SIZE>
__global__ void matrixMulSHARED(float *C, float *A, float *B, int wA)
{
	int bx = blockIdx.x;
	int by = blockIdx.y;
	int tx = threadIdx.x;
	int ty = threadIdx.y;
	int aBegin = wA*BLOCK_SIZE*by;
	int aEnd = aBegin + wA-1;
	int aStep = BLOCK_SIZE;
	int bBegin = BLOCK_SIZE*bx;
	int bStep = BLOCK_SIZE*wA;
	float Csub = 0;

	for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep)
	{
		__shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
		__shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
		As[ty][tx] = A[a + wA*ty + tx];
		Bs[ty][tx] = B[b + wA*ty + tx];
		__syncthreads();
```

```
#pragma unroll
              for (int k = 0; k<BLOCK_SIZE; ++k)
              {
                     Csub += As[ty][k] * Bs[k][tx];
              }
              __syncthreads();
       }

       int c = wA*BLOCK_SIZE*by + BLOCK_SIZE*bx;
       C[c + wA*ty + tx] = Csub;
}
```

W tym wariancie algorytmu podobnie jak wyżej mamy wiele bloków składających się z wątków przetwarzających poszczególne elementy z macierzy. Jednak w celu poprawy wydajności dostępu do pamięci macierze podzielone są na 'kafelki' o rozmiarze `BLOCK_SIZE`x `BLOCK_SIZE`, zadeklarowane jako dodatkowe zmienne tablicowe w pamięci współdzielonej, do której dostęp jest znacznie szybszy niż do globalnej. Umożliwia to przydział poszczególnym blokom wątków różnych fragmentów macierzy do wykonywania obliczeń na tym fragmencie.

Pierwszym etapem algorytmu jest przepisanie wartości elementów macierzy A i B z pamięci globalnej do pamięci współdzielonej, przy czym w obrębie jednego bloku każdy wątek przepisuje po jednym elemencie z obu macierzy. Następnie występuje bariera synchronizacyjna dla upewnienia się, że do pamięci współdzielonej wpisane zostały wszystkie elementy macierzy A i B.

Na tak przygotowanych danych w pamięci współdzielonej wątki dokonują obliczeń elementu wynikającego z threadIdx.x i threadIdx.y danego wątku. Pojawia się druga bariera synchronizacyjna, ponieważ w kolejnym etapie wątki znowu przystąpią do pobierania danych z pamięci globalnej do współdzielonej, jednak będą one dotyczyły innego (następnego) fragmentu macierzy.

Na koniec w odpowiednie miejsce w oryginalnej macierzy wynikowej zapisywany jest wynik częściowych obliczeń wątku.

c. grid wieloblokowy, jeden wątek oblicza jeden element macierzy wynikowej, obliczenia danych przy wykorzystaniu pamięci współdzielonej bloku wątków ze zrównolegleniem obliczeń i pobierania danych z pamięci globalnej w ramach każdego bloku wątków

```
template <int BLOCK_SIZE>
__global__ void mat(float *C, float *A, float *B, int wA)
{
       int bx = blockIdx.x;
       int by = blockIdx.y;
       int tx = threadIdx.x;
       int ty = threadIdx.y;
       int aBegin = wA*BLOCK_SIZE*by;
       int aEnd = aBegin + wA-1;
       int aStep = BLOCK_SIZE;
       int bBegin = BLOCK_SIZE*bx;
       int bStep = BLOCK_WIDTH*wA;
```

```
       float Csub = 0;

    // macierze na których wykonujemy obliczenia
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    // macierze do których równolegle z obliczeniami wpisujemy dane
kolejnych bloków
    __shared__ float AsNext[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float BsNext[BLOCK_SIZE][BLOCK_SIZE];

    // wskaźniki do SWAP'owania (wskaźnik na pierwszy wiersz macierzy)
    float (*AsPtr)[BLOCK_SIZE] = As;
    float (*BsPtr)[BLOCK_SIZE] = Bs;

    float (*AsNextPtr)[BLOCK_SIZE] = AsNext;
    float (*BsNextPtr)[BLOCK_SIZE] = BsNext;

    float (*temp)[BLOCK_SIZE] = NULL;

    // wypełniamy pierwszy raz
    As[ty][tx] = A[aBegin + wA * ty + tx];
    Bs[ty][tx] = B[bBegin + wA * ty + tx];

    __syncthreads();

    for (int a = aBegin, b = bBegin;
         a <= aEnd;
         a += aStep, b += bStep)
    {

        if (a+aStep <= aEnd) {
            // jeśli jest kolejny blok, zapisujemy go do AsNext i BsNext
            AsNext[ty][tx] = A[a+aStep + wA * ty + tx];
            BsNext[ty][tx] = B[b+bStep + wA * ty + tx];
        }

#pragma unroll

        // liczymy na bieżących As i Bs
        for (int k = 0; k < BLOCK_SIZE; ++k)
        {
            Csub += As[ty][k] * Bs[k][tx];
        }


        // wymieniamy wskaźniki na macierze, tak że to co wpisaliśmy do
AsNext i BsNext posłuży teraz do obliczeń

        temp = AsPtr;
        AsPtr = AsNextPtr;
        AsNextPtr = temp;

        temp = BsPtr;
        BsPtr = BsNextPtr;
        BsPtr = temp;

    }
```

```
        int c = wA*BLOCK_SIZE*by + BLOCK_SIZE*bx;
        C[c + wA*ty + tx] = Csub;
}
```

Ten algorytm jest modyfikacją poprzedniego, ponieważ także dotyczy dostępu do pamięci współdzielonej, macierz jest dzielona na fragmenty które przydzielamy różnym bloków wątków żeby prowadziły na nich obliczenia. Dodatkową funkcjonalnością algorytmu jest zrównoleglenie obliczeń. Oznacza to, że wątki zamiast sekwencyjnie pobierać dane z pamięci globalnej, następnie zsynchronizować się w obrębie bloku i prowadzić obliczenia na danych pobranych w ramach tej samej iteracji, dokonują pobrania danych z pamięci globalnej dotyczących następnego fragmentu.

Aby to było możliwe, w pamięci współdzielonej przechowujemy teraz 4 macierze zamiast 2 - macierze As i Bs to macierze na których prowadzone są obliczenia, natomiat AsNext i BsNext to macierze do których wpisujemy dane które będą potrzebne do obliczeń w następnej iteracji (to znaczy przejściu na kolejny fragment macierzy).

Oczywiście na początku konieczne jest wypełnienie macierzy As i Bs, zastosowałyśmy barierę synchronizacyjną dla pewności, że wszystkie dane zostały pobrane (choć i tak ta bariera wystąpi tylko raz, w przeciwieństwie do poprzedniego algorytmu). Wówczas, w pętli, każdy wątek jest odpowiedzialny za wpisanie odpowiednich wartości w AsNext i BsNext, a następnie wykonanie ‘swoich’ obliczeń w As i Bs (determinowane indeksami wątku threadIdx, jak w pozostałych przypadkach). Aby uniknąć niepotrzebnego przepisywania wartości między macierzami As i AsNext (tak samo B), zastosowałyśmy lokalne dla każdego wskaźniki. Wystarczy po każdej iteracji zamienić miejsca, w które one wskazują (AsPtr, AsNextPtr, tak samo B…).

3. **Wielkości instancji:**
    - Wielkości bloku: 8x8, 16x16, 32x32
    - Wielkości macierzy (kwadratowa NxN) 32x32, 80x80, 160x160, 320x320, 480x480, 640x640

4. **Analiza wydajności algorytmów na podstawie zebranych miar**

    a. prędkość obliczeń w GFlop / s - liczba operacji operacji zmiennoprzecinkowych na sekundę, w zależności od zastosowanej wersji algorytmu, rozmiaru bloku oraz wymiarów macierzy

	Prędkość obliczeń w GFLop / s			
BLOCK	N	GLOBAL	SHARED	PARALLEL
8	80	41,23	76,65	98,8
	160	42,19	119,62	219,62
	320	47,66	134,27	276,7
	480	50,13	135,56	224,32
	640	54,16	144,89	230,89
16	160	67,76	109,28	246,44
	320	71,11	154,43	316,81
	480	70,28	156,92	333,08
	640	76,58	163,42	340,73
32	32	6,83	8,15	8,26
	320	66,26	157,28	328,72
	640	73,21	165,75	375,84

Tabela 1. Prędkość obliczeń GFlop / s dla różnych wersji algorytmu

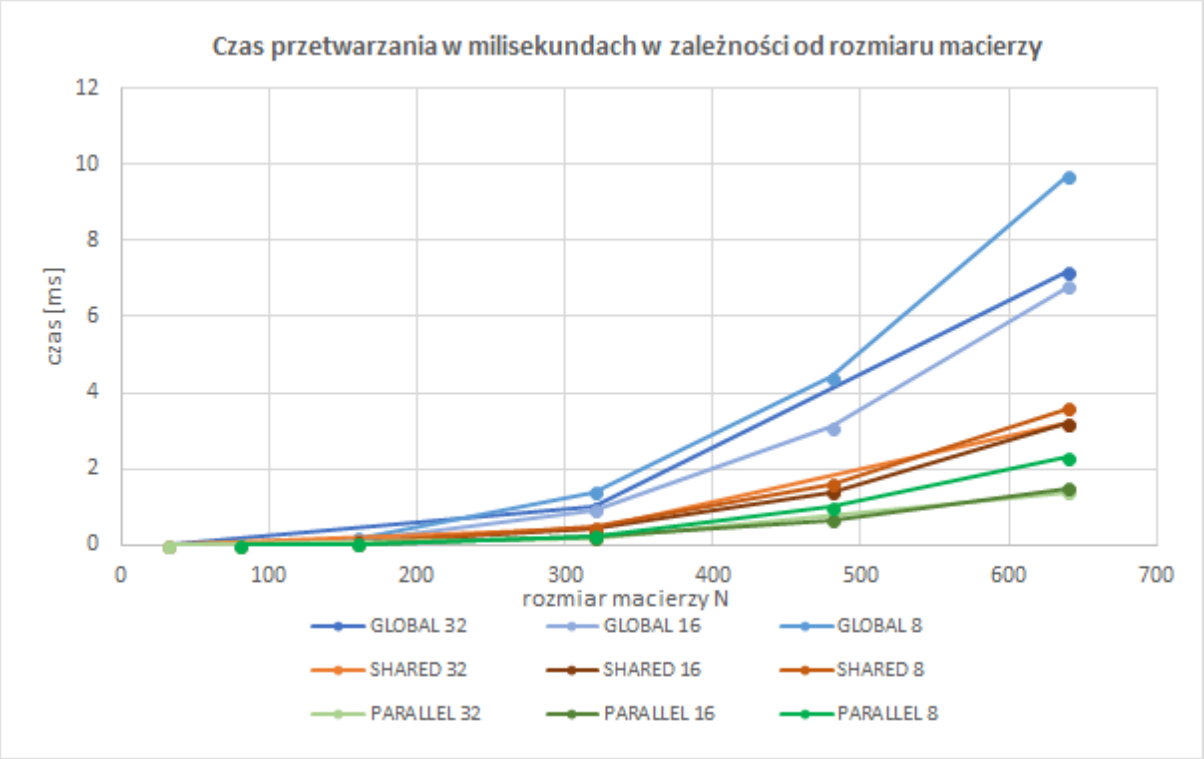
Najmniejsze prędkości obliczeń zaobserwowałyśmy dla pierwszej wersji algorytmu z dostępem do pamięci globalnej. Należało się tego spodziewać, ponieważ ciągły dostęp do globalnej pamięci jest dużo bardziej kosztowny czasowo od przechowywania tych danych tymczasowo w pamięci współdzielonej. Ponadto w każdym z badanych przypadków obserwujemy gwałtowny wzrost prędkości obliczeń w ramach danego algorytmu i danego rozmiaru bloku wątków wraz z zwiększeniem rozmiaru macierzy - dotyczy najmniejszych rozmiarów instancji, ponieważ dobór tak małego N naraża także pomiary na koszty związane z przygotowaniem karty graficznej, inicjalizacją i tak dalej. W kolejnych przypadkach zwiększanie rozmiarów macierzy nie ma już tak znaczącego wpływu na przyspieszenie prędkości obliczeń.

Ponadto wersja 3 algorytmu charakteryzuje się większą prędkością obliczeń od wersji 2, ponieważ nie występuje w niej dodatkowy koszt czasowy synchronizacji wątków, która w wersji 2 pojawia się co iterację (przejście na kolejny fragment macierzy). W większości przypadków wzrost prędkości jest tutaj około dwukrotny.
Zwiększanie rozmiaru bloku w obrębie tego samego algorytmu i tego samego rozmiaru macierzy również pozwala zaobserwować wzrost prędkości obliczeń, ponieważ przypadek rozmiaru bloku 32 oznacza lepsze wykorzystanie możliwości karty graficznej (zmniejszając rozmiar bloku na pewno nie zyskamy, ponieważ pracę obliczania macierzy wynikowej niepotrzebnie ‘rozdrabniamy’).

b. przyspieszenie (i zmiana przyspieszenia w funkcji wielkości instancji) w stosunku do obliczeń sekwencyjnych na komputerze ogólnego przeznaczenia najlepszą dostępną metodą - pętle ikj

		GLOBAL	SHARED	PARALLEL	GLOBAL	SHARED	PARALLEL
BLOCK	N	T [msec]			Przyspieszenie		
8	80	0,025	0,013	0,01	0	0	0
	160	0,19	0,068	0,037	62	180	320
	320	1,4	0,49	0,24	11	31	63
	480	4,4	1,6	0,99	7,9	21	35
	640	9,7	3,6	2,3	7,9	21	33
16	160	0,12	0,075	0,033	99	160	360
	320	0,92	0,42	0,21	16	35	72
	480	3,1	1,4	0,66	11	25	53
	640	6,8	3,2	1,5	11	24	49
32	32	0,01	0,008	0,008	0	0	0
	320	0,99	0,42	0,2	15	36	75
	640	7,2	3,2	1,4	11	24	54

Tabela 2. Czas przetwarzania i przyspieszenie względem algorytmu sekwencyjnego dla trzech wersji algorytmów, różnych rozmiarów macierzy i różnych rozmiarów bloku.



Wykres 1. Zależność czasu przetwarzania w milisekundach od rozmiaru macierzy, dla poszczególnych algorytmów i różnych rozmiarów bloków (np. GLOBAL 8 - dostęp do pamięci globalnej, blok 8x8)

Rzeczywisty czas przetwarzania dla algorytmu z dostępem do pamięci globalnej jest najgorszy, jak się należało spodziewać podobnie jak w przypadku pomiaru prędkości obliczeń GFlop / s. W obrębie tej wersji najlepszy okazał się blok 16x16, co jest widoczne przy większych macierzach, jednak nie na tyle aby dorównywać pozostałym wersjom. Algorytmy z dostępem do pamięci współdzielonej działają szybciej, a dodatkowo wersja ze zrównolegleniem obliczeń (PARALLEL) jest jeszcze około dwa razy szybsza od SHARED. W tych przypadkach również wpływ rozmiaru bloku nie jest na tyle duży, aby zakłócić przewagę wersji PARALLEL nad pozostałymi. Obserwując wzrost czasu przetwarzania na wykresie dla coraz większych rozmiarów macierzy, widzimy że dla wersji z pamięcią globalną czas ten rośnie najszybciej - to najmniej efektywna wersja algorytmu.

Ponadto dokonałyśmy pomiarów algorytmu sekwencyjnego IKJ dla badanych przez nas rozmiarów macierzy, celem wyznaczenia przyspieszenia jakie oferuje przeniesienie zadania mnożenia macierzy na kartę graficzną. W każdym przypadku czas był kilka do kilkudziesięciu razy mniejszy niż dla obliczeń sekwencyjnych, aż do przyspieszenia rzędu kilkuset raz w przypadku algorytmu ze zrównolegleniem obliczeń. W tabeli pojawiły się także wartości zerowe, jednak uzasadniamy je niedokładnością pomiaru po stronie obliczeń sekwencyjnych (niewystarczająca rozdzielczość), jak i samym rozmiarem macierzy - bardzo mały.

c. **CGMA** (compute to global memory access ratio) czyli liczba (niezbędnych) operacji na danych dzielona przez liczbę umożliwiających te operacje dostępów do pamięci globalnej. Współczynnik wyznaczany w zależności wersji kodu, rozmiaru bloku i instancji.

- obliczane teoretycznie na podstawie kodu:

Algorytm 1:

W tej wersji algorytmu podczas obliczania elementu macierzy wynikowej w pętli odwołujemy się do odpowiednich elementów macierzy A i B, które ulokowane są w pamięci globalnej. W związku z tym musimy wykonać 2 pobrania z pamięci, następnie wykonujemy jedną operację mnożenia i jedną dodawania, tak więc współczynnik CGMA będzie równy 1.

Algorytm 2:

W tym algorytmie wątek korzysta z tablic pomocniczych zaalokowanych w pamięci współdzielonej, każda o wielkości BLOCK_SIZE x BLOCK_SIZE. W tym celu pobiera do nich dane z pamięci globalnej dla macierzy A i B, w związku z czym mamy 2 pobrania. Następnie na uzyskanych danych w wewnętrznej pętli dokonuje obliczeń - jedno dodawanie i jedno mnożenie. W pętli mamy BLOCK_SIZE iteracji, a więc łącznie na 2 pobrania z pamięci globalnej przypada 2 * BLOCK_SIZE operacji czyli CGMA = BLOCK_SIZE

Algorytm 3:

Tak samo jak w algorytmie powyżej korzystamy z tablic pomocniczych w pamięci współdzielonej ale mamy ich 2 razy więcej - po 2 dla macierzy A i B. Tak więc poza pierwszą iteracją zewnętrznej pętli, w której musimy załadować dane z pamięci globalnej do wszystkich 4 tablic, mamy 2 pobrania z pamięci globalnej (tylko do 2 tablic, na których w następnej iteracji będą wykonywane obliczenia). Równocześnie wykonujemy obliczenia w wewnętrznej pętli tak samo jak powyżej, a więc w ogólności współczynnik CGMA będzie w tym algorytmie taki sam, czyli CGMA = BLOCK_SIZE.

- obliczane na podstawie wyników z Nvidia profiler

W tym przypadku posłużyłyśmy się wzorem:

$$CGMA = \frac{FPInst}{G_{LoadT} + G_{StoreT}}$$

gdzie FPInst - liczba operacji zmiennoprzecinkowych, GLoadT - liczba transakcji pobrania danych z pamięci globalnej, GStoreT - liczba transakcji wysłania danych do pamięci globalnej

		CGMA		
BLOCK	N	GLOBAL	SHARED	PARALLEL
8	80	2,6	15	17
	160	2,7	16	16
16	160	2,7	31	34
	320	2,7	32	33
32	320	2,7	62	69
	640		63	66

Tabela 3. Współczynnik CGMA dla różnych wersji algorytmu

Jak widać nasze teoretyczne rozważania co do wielkości współczynnika CGMA dla poszczególnych algorytmów okazały się być pesymistyczne. W rzeczywistości są one znacznie większe, co prawdopodobnie jest spowodowane optymalizacją dostępu do pamięci zrealizowaną niezależnie przez środowisko wykonujące program. Można również zauważyć, że dla algorytmów SHARED i PARALLEL wraz ze wzrostem wielkości rozmiaru bloku zwiększa się też CGMA, co jest pożądanym efektem - oznacza to, że dłużej wykorzystujemy raz pobrane dane. Natomiast dla wersji GLOBAL wartość ta jest stała, w związku z czym wykorzystywanie pobranych danych jest nieefektywne. Można było się spodziewać braku wpływu rozmiaru bloku na ten algorytm, ponieważ w kodzie kernela nie jest on w ogóle uwzględniany, w przeciwieństwie do wersji z pamięcią współdzieloną - w przypadku których jej wielkość była wyznaczana właśnie przez BLOCK_SIZE. W Tabeli 3. widzimy, że dwukrotne zwiększenie rozmiaru bloku (w obrębie tego samego rozmiaru macierzy) skutkuje dwukrotnym zwiększeniem współczynnika CGMA i takiej zależności oczekiwałyśmy także na podstawie analizy teoretycznej.

Algorytm z dostępem do pamięci globalnej okazał się być na tyle nieefektywny, że podczas wykonywania eksperymentu natknęłyśmy się na trudność z pomiarem wartości wyznaczanych miar dla tej wersji algorytmu i dla największych dobranych rozmiarów instancji (tzn. rozmiar bloku 32x32, wymiary macierzy 640x640).

d. Miara stopnia łączenia dostępu do pamięci globalnej

		GLOBAL	SHARED	PARALLEL	GLOBAL	SHARED	PARALLEL
BLOCK	N	Global Memory Load Efficiency(%)			Global Memory Store Efficiency (%)		
8	80	30	100	100	50	50	50
8	160	30	100	100	50	50	50
16	160	56,25	100	100	100	100	100
16	320	56,25	100	100	100	100	100
32	320	82,5	100	100	100	100	100
32	640		100	100		100	100

Tabela 4. Efektywność pobierania danych z pamięci globalnej (Load) i efektywność zapisywania danych (Store) dla poszczególnych algorytmów, rozmiarów macierzy i bloków.

W Tabeli 4. zamieszczono efektywności odczytu/zapisu do pamięci, co dokładniej rozumiemy jako:

- Global Memory Load Efficiency: stosunek prób odczytu z pamięci globalnej do zrealizowanych odczytów
- Global Memory Store Efficiency: stosunek prób zapisu do pamięci globalnej do zrealizowanych zapisów

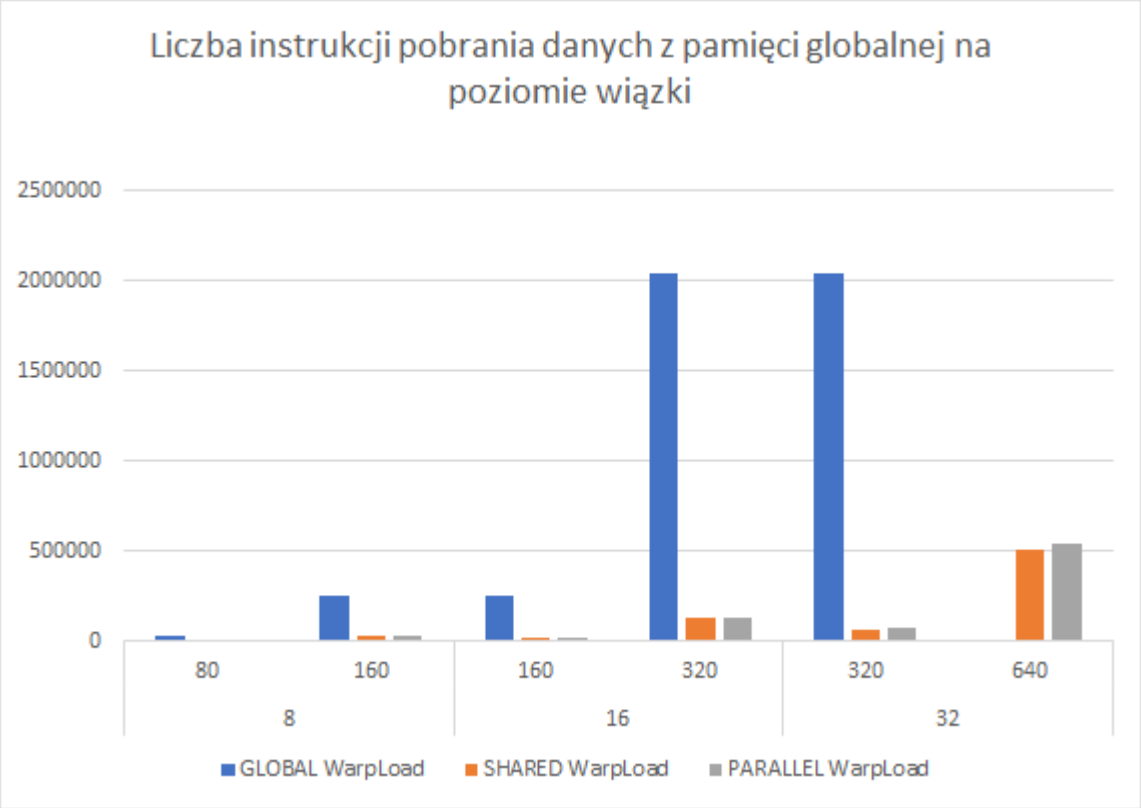
i są to stosunki wyznaczone bezpośrednio przez Profiler.

Analizując miarę stopnia łączenia dostępu do pamięci globalnej należy przede wszystkim zwrócić uwagę na wpływ rozmiaru bloku. Dla najmniejszego z rozpatrywanych rozmiarów bloku 8 spodziewamy się najmniej efektywnego łączenia dostępu przede wszystkim dla wersji algorytmu z dostępem do pamięci globalnej za każdym pobraniem elementu macierzy. Dla kolumny GLOBAL widzimy, że im większy rozmiar bloku, tym efektywność pobierania danych wzrasta.

Algorytmy z dostępem do pamięci współdzielonej charakteryzują się większą efektywnością pobierania danych z pamięci globalnej. W każdym przypadku dla tych dwóch wersji zaobserwowałyśmy stuprocentową efektywność. Z perspektywy pamięci zasadnicza różnica między algorytmem z pamięcią globalną a współdzieloną polega na dostępie do danych macierzy - w pierwszym, nieefektywnym przypadku nie występuje podział na bloki, który sprzyja ‘uporządkowaniu’ tych pobrań. Jak widzimy na Wykresie 2. poniżej, algorytm operujący na pamięci globalnej charakteryzuje się największą liczbą odwołań do pamięci celem pobrania danych z poziomu wiązki.

		GLOBAL	SHARED	PARALLEL	GLOBAL	SHARED	PARALLEL
BLOCK	N	Global Load Transactions Per Request			Global Store Transactions Per Request		
8	80	6	8	7,27	8	8	8
	160	6	8	7,62	8	8	8
16	160	6	8	7,27	4	4	4
	320	6	8	7,62	4	4	4
32	320	6	8	7,27	4	4	4
	640		8	7,62		4	4

Tabela 5. Średnia liczba transakcji z pamięcią globalną przypadającą na żądanie dostępu do pamięci



Wykres 2. GLOBAL - algorytm z dostępem do pamięci globalnej, pozostałe - z wykorzystaniem pamięci współdzielonej

e. Aktywność multiprocesora - stosunek czasu w którym multiprocesor jest aktywny (czyli przynajmniej jedna wiązka wątków jest aktywna) do całego czasu przetwarzania (wartość procentowa)

		Multiprocessor Activity(%)		
BLOCK	N	GLOBAL	SHARED	PARALLEL
8	80	85	65	58
	160	97	93	89
16	160	97	92	88
	320	99	99	98
32	320	97	99	97
	640		100	99

Tabela 5. Aktywność multiprocesora dla różnych wersji algorytmu

Jak widać na podstawie załączonych wyników doświadczenia, multiprocesor jest prawie zawsze aktywny. Wyjątkiem jest przetwarzanie przez multiprocesor instancji o rozmiarze 80x80, wtedy aktywność jest mniejsza, zwłaszcza dla metod z efektywnym dostępem do pamięci.

f. Osiągnięta zajętość multiprocesora - stosunek aktywnych wiązek wątków w ramach cyklu do maksymalnej liczby wiązek

Zajętość multiprocesora zmierzyłyśmy w Profilerze a także posłużyłyśmy się arkuszem CUDA Occupancy Calculator dla wyznaczenia wartości teoretycznych. Kompilując kod programu z parametrem --ptxas-options możliwe było określenie, ile rejestrów używanych jest przez wątek w naszej funkcji kernela, i było to: 32 rejestry dla wersji z pamięcią globalną, 27 rejestrów dla pamięci współdzielonej, a z dodatkowym zrównolegleniem obliczeń - 48 rejestrów.

		GLOBAL		SHARED		PARALLEL	
BLOCK	N	Profiler	Teoretyczne	Profiler	Teoretyczne	Profiler	Teoretyczne
8	80	49%	100%	57%	100%	56%	63%
	160	86%	100%	88%	100%	86%	63%
16	160	86%	100%	90%	100%	84%	63%
	320	93%	100%	97%	100%	93%	63%
32	320	87%	100%	98%	100%	46%	50%
	640			99%	100%	47%	50%

Tabela 6. Osiągnięta zajętość multiprocesora dla różnych wersji algorytmu

Wnioski

Po przeanalizowaniu wyników naszego eksperymentu można łatwo zauważyć, że znacznie bardziej efektywne są algorytmy wykorzystujące w przetwarzaniu szybszą pamięć współdzieloną. W wariancie pierwszym kodu odwołujemy się do pamięci globalnej znacznie częściej co spowalnia działanie programu. Tę sytuację bardzo dobrze obrazują pomiary wartości czasu przetwarzania i prędkości obliczeń.

Możemy również zauważyć, że najszybciej działa algorytm wykorzystujący współdzieloną pamięć bloku wątków ze zrównolegleniem obliczeń i pobieraniem danych z pamięci globalnej w ramach każdego bloku wątków. Genezy mniejszej efektywności algorytmu pierwszego doszukujemy się w niskim i niezależnym od wielkości bloku wątków współczynniku CGMA, który mówi nam jak długo wykorzystujemy dane pobrane z pamięci globalnej do obliczeń. Zaobserwowałyśmy również zgodnie z oczekiwaniami, że algorytmy wykorzystujące do przetwarzania pamięć współdzieloną charakteryzują się wyższą efektywnością pobierania i zapisywania danych do pamięci globalnej.

Natomiast zwracając uwagę na zależność jakości przetwarzania od rozmiarów instancji możemy śmiało powiedzieć, że dla większych instancji znacznie wzrasta prędkość obliczeń, lepiej wykorzystywany jest również multiprocesor, który osiąga zajętość bliską 100%.