Przetwarzanie równoległe

3. Mnożenie macierzy - badanie prędkości obliczeń w funkcji ilości pracy wątków

Metody

1. (Numer 1) Jeden blok wątków przetwarzania, obliczenia przy wykorzystaniu pamięci globalnej, mnożenie dowolnych tablic o rozmiarach będących wielokrotnością rozmiaru bloku wątków.
2. (Numer 2) Grid wieloblokowy, jeden wątek oblicza jeden element macierzy wynikowej, obliczenia przy wykorzystaniu pamięci globalnej.
3. (Numer 6) Grid wieloblokowy, jeden wątek oblicza 2 lub 4 (podział pracy dwuwymiarowy) sąsiednich elementów macierzy wynikowej, obliczenia danych przy wykorzystaniu pamięci współdzielonej bloku wątków ze zrównolegleniem obliczeń i pobierania danych z pamięci globalnej w ramach bloku wątków.

Autorzy
Tomasz Gil, 127295
Piotr Ptak, 127347
Grupa I1, laboratoria środa 13:30

1. Wstęp

Celem ćwiczenia jest zapoznanie się z zasadami programowania procesorów kart graficznych i optymalizacji kodu na przykładzie algorytmu mnożenia macierzy. Następnie dokonanie pomiarów prędkości i efektywności przetwarzania oraz ocena i porównanie przygotowanych wariantów kodu.

2. Architektura GPU

Testy zostały wykonane na platformie wyposażonej w kartę Nvidia GTX 1050, której parametry przedstawiamy poniżej:

- Liczba multiprocesorów : 5
- Liczba rdzeni: 640
- Maksymalna liczba wątków na blok: 1024
- Maksymalna liczba wątków na multiprocesor: 2048
- Compute capability: 6.1
- Częstotliwość taktowania zegara rdzenia: 1493 MHz
- Częstotliwość taktowania zegara pamięci: 3504 MHz
- Warp size: 32
- Maksymalny rozmiar bloku: 1024 x 1024 x 64
- Liczba dostępnych rejestrów na blok: 65536
- Rozmiar pamięci globalnej: 4096 MB
- Rozmiar pamięci współdzielonej na blok: 49152 B

3. Implementacje algorytmów

Część wspólna

```
#define BLOCK_SIZE ?
#define MATRIX_SIZE ?
```

Dla każdej wersji algorytmu mnożenia macierzy będziemy wykonywać pomiary dla rozmiarów bloku wątków kolejno 8x8, 16x16 oraz 32x32 (co daje odpowiednio 64, 256 i 1024 wątki na blok) oraz rozmiaru macierzy kwadratowej 64, 128 i 256.

Poszczególne wersje

1. Jeden blok wątków przetwarzania, obliczenia przy wykorzystaniu pamięci globalnej, mnożenie dowolnych tablic o rozmiarach będących wielokrotnością rozmiaru bloku wątków.

```
#define GRID_SIZE 1

__global__ void matrixMulCUDA(float *C, float *A, float *B) {
	int tx = threadIdx.x;
	int ty = threadIdx.y;

	for (int bx = 0; bx < MATRIX_SIZE / BLOCK_SIZE; ++bx) {
		for (int by = 0; by < MATRIX_SIZE / BLOCK_SIZE; ++by) {
			float C_local = 0;
			int row = BLOCK_SIZE * by + ty;
			int col = BLOCK_SIZE * bx + tx;
			for (int k = 0; k < MATRIX_SIZE; ++k) {
				float A_d = A[ty * MATRIX_SIZE + k];
				float B_d = B[k * MATRIX_SIZE + tx];
				C_local += A_d * B_d;
			}
			C[MATRIX_SIZE * row + col] = C_local;
		}
	}
}

// wywołanie
dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid(GRID_SIZE, GRID_SIZE);
matrixMulCUDA<<< grid, threads >>>(d_C, d_A, d_B);
```

W tym wariancie algorytmu wykonujemy przetwarzanie w gridzie o wielkości 1, a co za tym idzie mamy tylko jeden blok wątków. Każdy wątek w danym momencie wylicza jedną wartość macierzy wynikowej (w zależności od swojego indeksu w bloku - zmienne tx i ty) w najbardziej wewnętrznej pętli. Przez to, że mamy tylko jeden blok, dodajemy dwie dodatkowe pętle, w których przesuwamy się po kolejnych pod-macierzach wielkości równej wielkości bloku wątków. Każdy wątek pobiera wartości mnożonych macierzy z pamięci globalnej karty graficznej.

2. Grid wieloblokowy, jeden wątek oblicza jeden element macierzy wynikowej, obliczenia przy wykorzystaniu pamięci globalnej.

```
#define GRID_SIZE (MATRIX_SIZE / BLOCK_SIZE)
```

```
__global__ void matrixMulCUDA(float *C, float *A, float *B) {
	int bx = blockIdx.x;
	int by = blockIdx.y;
	int tx = threadIdx.x;
	int ty = threadIdx.y;
	int row = BLOCK_SIZE * by + ty;
	int col = BLOCK_SIZE * bx + tx;
	float C_local = 0;

	for (int i = 0; i < GRID_SIZE; i++) {
		for (int k = 0; k < BLOCK_SIZE; ++k) {
			C_local += A[(row + i) * BLOCK_SIZE + k] * B[(k + i) * BLOCK_SIZE +
col];
		}
	}
	C[MATRIX_SIZE * row + col] = C_local;
}

// wywołanie
dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid(GRID_SIZE, GRID_SIZE);
matrixMulCUDA<<< grid, threads >>>(d_C, d_A, d_B);
```

Wariant ten jest bardzo podobny do wersji pierwszej, jednak przez to, że wielkość gridu jest iloczynem wielkości macierzy i wielkości bloku wątków, mamy dostępną w szczególności więcej niż jeden blok wątków. Dany wątek liczy wartość macierzy wynikowej na podstawie indeksów bloku, do którego należy oraz położenia wewnątrz niego (odpowiednio zmienne bx, by oraz tx, ty). Mnożenie odbywa się w wewnętrznej pętli, zewnętrzna służy wątkowi do dostania się do potrzebnych wartości z macierzy mnożonych (położonych poza blokiem do którego sam należy). Również tutaj dane macierzy A i B pobierane są z pamięci globalnej karty graficznej.

3. Grid wieloblokowy, jeden wątek oblicza 2 lub 4 (podział pracy dwuwymiarowy) sąsiednich elementów macierzy wynikowej, obliczenia danych przy wykorzystaniu pamięci współdzielonej bloku wątków ze zrównolegleniem obliczeń i pobierania danych z pamięci globalnej w ramach bloku wątków.

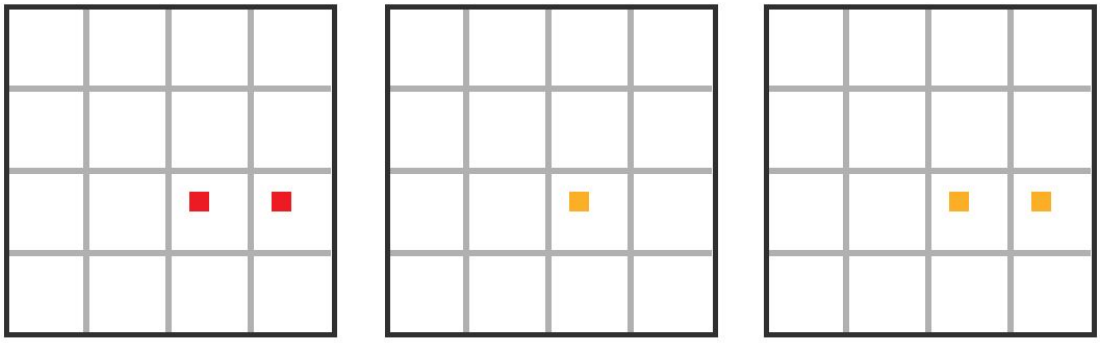
    a. Wątek oblicza 2 sąsiednie elementy

```
#define GRID_HEIGHT (MATRIX_SIZE / BLOCK_SIZE)
#define GRID_WIDTH (GRID_HEIGHT / 2)

__global__ void matrixMulCUDA(float *C, float *A, float *B) {
	int bx = blockIdx.x * 2;
```

```
        int by = blockIdx.y;
        int tx = threadIdx.x;
        int ty = threadIdx.y;

        int aBegin = MATRIX_SIZE * BLOCK_SIZE * by;
        int aEnd = aBegin + MATRIX_SIZE - 1;
        int aStep = BLOCK_SIZE;
        int bBegin = BLOCK_SIZE * bx;
        int bStep = BLOCK_SIZE * MATRIX_SIZE;

        float C_local = 0;
        float C_local2 = 0;
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][2 * BLOCK_SIZE];

        __shared__ float A_shared[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float B_shared[BLOCK_SIZE][2 * BLOCK_SIZE];

        A_shared[ty][tx] = A[aBegin + MATRIX_SIZE * ty + tx];
        B_shared[ty][tx] = B[bBegin + MATRIX_SIZE * ty + tx];
        B_shared[ty][tx + BLOCK_SIZE] = B[bBegin + MATRIX_SIZE * ty + tx + BLOCK_SIZE];

        for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) {
                As[ty][tx] = A_shared[ty][tx];
                Bs[ty][tx] = B_shared[ty][tx];
                Bs[ty][tx + BLOCK_SIZE] = B_shared[ty][tx + BLOCK_SIZE];

                __syncthreads();

                A_shared[ty][tx] = A[a + MATRIX_SIZE * ty + tx];
                B_shared[ty][tx] = B[b + MATRIX_SIZE * ty + tx];
                B_shared[ty][tx + BLOCK_SIZE] = B[b + MATRIX_SIZE * ty + tx + BLOCK_SIZE];


#pragma unroll
                for (int k = 0; k < BLOCK_SIZE; ++k) {
                        C_local += As[ty][k] * Bs[k][tx];
                }
#pragma unroll
                for (int k = 0; k < BLOCK_SIZE; ++k) {
                        C_local2 += As[ty][k] * Bs[k][tx + BLOCK_SIZE];
                }

                __syncthreads();
        }

        int c = MATRIX_SIZE * BLOCK_SIZE * by + BLOCK_SIZE * bx;
        C[c + MATRIX_SIZE * ty + tx] = C_local;
        C[c + MATRIX_SIZE * ty + tx + BLOCK_SIZE] = C_local2;
}
```

```
// wywołanie
dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid(GRID_WIDTH, GRID_HEIGHT);
matrixMulCUDA<<< grid, threads >>>(d_C, d_A, d_B);
```

Dwie wartości wyliczane przez wątek są położone w jednym wierszu macierzy wynikowej. Dodatkowo, chcemy zapewnić jednoczesne sprowadzanie danych z pamięci (także współdzielonej). W momencie gdy w wiązce wykonywana jest ta sama instrukcja, jeśli pobierane wartości nie będą położone obok siebie, tylko na odpowiadających sobie miejscach w sąsiednich pod-macierzach (odpowiadających blokom wątków) faktycznie będziemy odwoływać się do sąsiadujących danych. Przykładowo, w efekcie pojedynczy wątek liczy (pierwsza macierz) oraz pobiera (dwie kolejne macierze) wartości:



Macierz wyjściowa C Macierz wejściowa A Macierz wejściowa B

Jeden wątek pobiera 3 wartości z pamięci globalnej do pamięci współdzielonej w zależności od swojego położenia w bloku. Jeśli taką operację wykonają wszystkie wątki i następnie zsynchronizujemy wszystkie wątki, w pamięci współdzielonej znajdą się wszystkie wartości, których potrzebują wątki do wykonania obliczeń. Dodatkowo wykorzystujemy bufory na macierze wejściowe, aby usunąć zależność pomiędzy fazą wczytania danych a ich wykorzystaniem do obliczeń. Dalej w dwóch pętlach obliczamy wartości końcowe, korzystając ze sprowadzonych przez wszystkie wątki danych współdzielonych. Deklaracja `#pragma unroll` powoduje rozwinięcie ciała pętli w szereg pojedynczych instrukcji, równoważnych przetwarzaniu pętli.

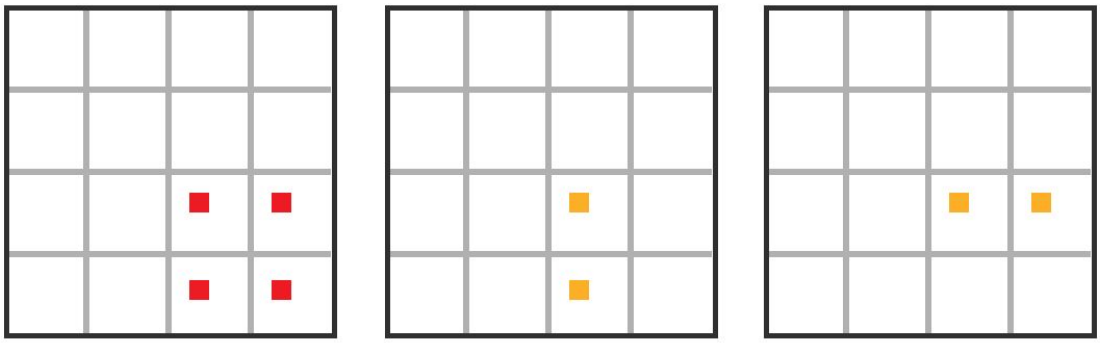
b. Wątek oblicza 4 sąsiednie elementy

```
#define GRID_SIZE ((MATRIX_SIZE / BLOCK_SIZE ) / 2)

__global__ void matrixMulCUDA(float *C, float *A, float *B) {
	int bx = blockIdx.x * 2;
	int by = blockIdx.y * 2;
	int tx = threadIdx.x;
	int ty = threadIdx.y;
```

```
        int aBegin = MATRIX_SIZE * BLOCK_SIZE * by;
        int aEnd = aBegin + MATRIX_SIZE - 1;
        int aStep = BLOCK_SIZE;
        int bBegin = BLOCK_SIZE * bx;
        int bStep = BLOCK_SIZE * MATRIX_SIZE;

        float C_local = 0;
        float C_local2 = 0;
        float C_local3 = 0;
        float C_local4 = 0;
        __shared__ float As[2 * BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][2 * BLOCK_SIZE];

        __shared__ float A_shared[2 * BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float B_shared[BLOCK_SIZE][2 * BLOCK_SIZE];

        A_shared[ty][tx] = A[aBegin + MATRIX_SIZE * ty + tx];
        A_shared[ty + BLOCK_SIZE][tx] = A[aBegin + MATRIX_SIZE * (ty + BLOCK_SIZE) + tx];

        B_shared[ty][tx] = B[bBegin + MATRIX_SIZE * ty + tx];
        B_shared[ty][tx + BLOCK_SIZE] = B[bBegin + MATRIX_SIZE * ty + tx + BLOCK_SIZE];

        for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) {
                As[ty][tx] = A_shared[ty][tx];
                As[ty + BLOCK_SIZE][tx] = A_shared[ty + BLOCK_SIZE][tx];
                Bs[ty][tx] = B_shared[ty][tx];
                Bs[ty][tx + BLOCK_SIZE] = B_shared[ty][tx + BLOCK_SIZE];

                __syncthreads();

                A_shared[ty][tx] = A[a + MATRIX_SIZE * ty + tx];
                A_shared[ty + BLOCK_SIZE][tx] = A[a + MATRIX_SIZE * (ty + BLOCK_SIZE) + tx];
                B_shared[ty][tx] = B[b + MATRIX_SIZE * ty + tx];
                B_shared[ty][tx + BLOCK_SIZE] = B[b + MATRIX_SIZE * ty + tx + BLOCK_SIZE];

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

```
                C_local += As[ty + BLOCK_SIZE][k] * Bs[k][tx + BLOCK_SIZE];
            }

            __syncthreads();
        }

        int c = MATRIX_SIZE * BLOCK_SIZE * by + BLOCK_SIZE * bx;
        C[c + MATRIX_SIZE * ty + tx] = C_local;
        C[c + MATRIX_SIZE * ty + tx + BLOCK_SIZE] = C_local2;
        C[c + MATRIX_SIZE * (ty + BLOCK_SIZE) + tx] = C_local3;
        C[c + MATRIX_SIZE * (ty + BLOCK_SIZE) + tx + BLOCK_SIZE] = C_local4;
}

// wywołanie
dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid(GRID_SIZE, GRID_SIZE);
matrixMulCUDA<<< grid, threads >>>(d_C, d_A, d_B);
```

W tym przypadku powielamy schemat z poprzedniego podpunktu, jednak zamiast 2 wartości jeden wątek oblicza 4 wartości, gdzie dwie dodatkowe znajdują się w miejscach odpowiadających dwóm pierwszym elementom, jednak w blokach poniżej. W tym przypadku, analogiczny przykład liczonych i pobieranych przez pojedynczy wątek danych będzie wyglądać w następujący sposób:



Macierz wyjściowa C Macierz wejściowa A Macierz wejściowa B

4. Wielkości teoretyczne

CGMA

CGMA, czyli Compute to Global Memory Access Ration to parametr określany jako stosunek niezbędnych operacji zmiennoprzecinkowych do dostępów do pamięci umożliwiających te operacje. Praktycznie, dostarcza informacje jak długo wykorzystujemy raz pobrane dane do obliczeń. W trakcie

optymalizacji algorytmu należy dążyć do maksymalizacji tej miary, mając na celu zwiększenie przepustowości pamięci - taniej jest wykonać dodatkowe operacje wykonując możliwości wielu rdzeni procesorów karty graficznej niż odwoływać się do wolnej pamięci.

$$CGMA = \frac{Floating\ Point\ Operations}{Global\ Memory\ Load + Global\ Memory\ Store}$$

Dla naszych metod współczynnik ten przyjmuje następujące wartości:

1. Metoda 1 - para operacji dodawania i mnożenia przypada na dwa odczyty z pamięci (macierze A i B): $CGMA = 1$
2. Metoda 2 - tak samo, nadal każda para dodawanie-mnożenie przypada na dwa odczyty z pamięci: $CGMA = 1$
3. Metoda 6 (2 elementy na wątek) - tutaj jednym wątku w jednej iteracji zewnętrznej pętli wykonujemy zawsze 3 odczyty z pamięci globalnej, a następnie w dwóch pętlach obliczających elementy macierzy C mamy tyle par dodawanie-mnożenie, jaki jest wymiar bloku. Otrzymujemy:
$$CGMA = \frac{2 * BlockSize * 2}{3} = \frac{4}{3} * BlockSize$$
4. Metoda 6 (4 elementy na wątek) - analogicznie do poprzedniego punktu, dokonujemy 4 odczytów oraz mamy 4 pętle obliczające elementy macierzy. Otrzymujemy:
$$CGMA = \frac{4 * BlockSize * 2}{4} = 2 * BlockSize$$

Zajętość multiprocesora

Zajętość multiprocesora jest stosunkiem aktywnych wiązek do maksymalnej wspieranej liczby wiązek. Teoretyczna zajętość została obliczona z użyciem kalkulatora zajętości dostępnego na stronie NVIDIA. W naszym przypadku dla przyjętych rozmiarów bloków wątków i 32 rejestrów na blok wątków teoretyczna zajętość procesora wynosiła 100% w każdym przypadku.

5. Zebrane miary

Pomiarów dokonaliśmy dla wymienionych poniżej wielkości bloków oraz macierzy:

- Wielkości bloku: 8 x 8, 16 x 16, 32 x 32
- Wielkości macierzy: 64 x 64, 128 x 128, 256 x 256

Do wykonania testów korzystaliśmy z programu NVIDIA Visual Profiler. Dokonaliśmy zbierania następujących wielkości:

- Ogólne:
    - Czas przetwarzania [ms]
    - Prędkość przetwarzania [GFlop/s]
- Pamięć:
    - Static Shared Memory - wielkość wykorzystanej statycznej pamięci współdzielonej
    - Global Memory Load Efficiency (GMLE) - stosunek prób odczytu z pamięci globalnej do zrealizowanych odczytów

    - Global Memory Store Efficiency (GMSE) - stosunek prób zapisu do pamięci globalnej do zrealizowanych zapisów
    - Requested Global Load Throughput (RGLT) - zamawiana przepustowość odczytu z pamięci globalnej
    - Requested Global Store Throughput (RGST) - zamawiana przepustowość zapisu do pamięci globalnej
- Multiprocessor:
    - Achieved Occupancy - stosunek średniej liczby aktywnych wiązek na cykl do maksymalnej liczby wiązek wspieranej przez multiprocesor
    - Multiprocessor Activity - stosunek czasu, przez który chociaż jedna wiązka jest aktywna do całkowitego czasu przetwarzania

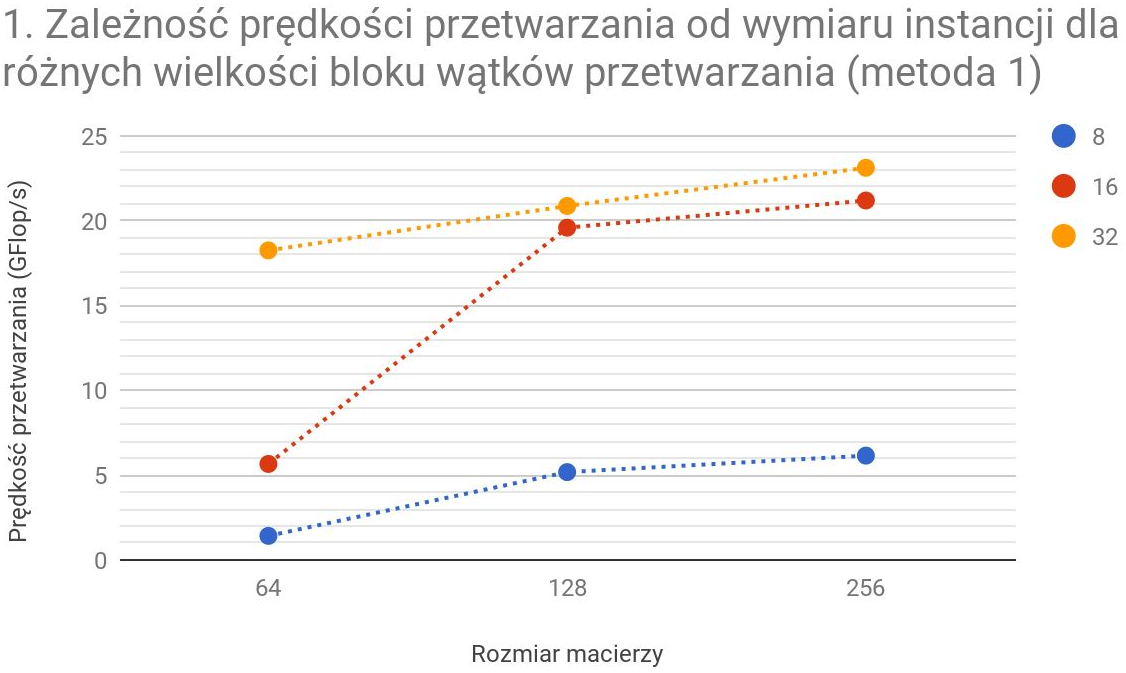
Prędkości przetwarzania są widoczne na wykresach w punkcie 6. sprawozdania. Uzyskane wyniki pozostałych miar przedstawiamy w tabeli poniżej.

Rozmiar bloku	**Rozmiar macierzy**	**Static Shared Memory**	**GMLE (%)**	**GMSE (%)**	**RGLT (GB/s)**	**RGST (GB/s)**	**Multiprocessor Activity(%)**	**Achieved Occupancy**
Metoda 1								
8	64	0	30,00%	50,00%	1,31	0,05	19,90%	31,00%
8	128	0	30,00%	50,00%	4,72	0,10	19,97%	31,00%
8	256	0	30,00%	50,00%	4,67	0,05	20,25%	31,00%
16	64	0	56,25%	100,00%	6,10	0,17	19,41%	12,40%
16	128	0	56,25%	100,00%	26,27	0,36	19,73%	12,50%
16	256	0	56,25%	100,00%	26,49	0,18	20,10%	12,40%
32	64	0	82,50%	100,00%	36,55	0,55	17,39%	49,20%
32	128	0	82,50%	100,00%	49,64	0,38	19,69%	49,80%
32	256	0	82,50%	100,00%	51,72	0,19	20,30%	49,90%
Metoda 2								
8	64	0	23,29%	50,00%	26,96	1,59	66,37%	37,60%
8	128	0	23,29%	50,00%	35,80	1,05	93,12%	83,90%
8	256	0	23,29%	50,00%	33,28	0,49	98,59%	94,00%
16	64	0	56,25%	100,00%	54,58	1,52	63,14%	39,30%
16	128	0	56,25%	100,00%	88,44	1,23	91,68%	85,00%
16	256	0	56,25%	100,00%	108,27	0,75	98,14%	91,00%
32	64	0	73,77%	100,00%	74,39	1,96	55,13%	46,90%
32	128	0	82,50%	100,00%	151,14	1,14	77,29%	85,30%
32	256	0	82,50%	100,00%	229,63	0,87	95,57%	90,60%
Metoda 6 (2 elementy na wątek)								

8	64	1536	100,00%	50,00%	29,74	2,20	61,97%	19,00%
8	128	1536	100,00%	50,00%	55,69	2,18	86,99%	75,70%
8	256	1536	100,00%	50,00%	63,17	1,27	97,79%	92,40%
16	64	6144	100,00%	100,00%	19,15	2,55	57,33%	20,60%
16	128	6144	100,00%	100,00%	37,11	2,75	80,89%	75,60%
16	256	6144	100,00%	100,00%	41,79	1,64	96,59%	92,80%
32	64	24576	100,00%	100,00%	9,27	2,06	28,34%	49,30%
32	128	24576	100,00%	100,00%	21,65	2,89	72,50%	80,30%
32	256	24576	100,00%	100,00%	24,72	1,83	89,60%	95,60%
Metoda 6 (4 elementy na wątek)								
8	64	2048	100,00%	50,00%	19,68	2,19	66,45%	98,00%
8	128	2048	100,00%	50,00%	54,51	3,21	83,81%	37,00%
8	256	2048	100,00%	50,00%	67,78	2,05	95,25%	84,50%
16	64	8192	100,00%	100,00%	13,95	2,79	49,65%	12,40%
16	128	8192	100,00%	100,00%	33,80	3,75	71,98%	39,60%
16	256	8192	100,00%	100,00%	45,26	2,66	94,20%	85,40%
32	64	32768	100,00%	100,00%	4,95	1,65	14,31%	49,20%
32	128	32768	100,00%	100,00%	18,90	3,78	64,67%	49,50%
32	256	32768	100,00%	100,00%	22,85	2,54	81,14%	88,50%

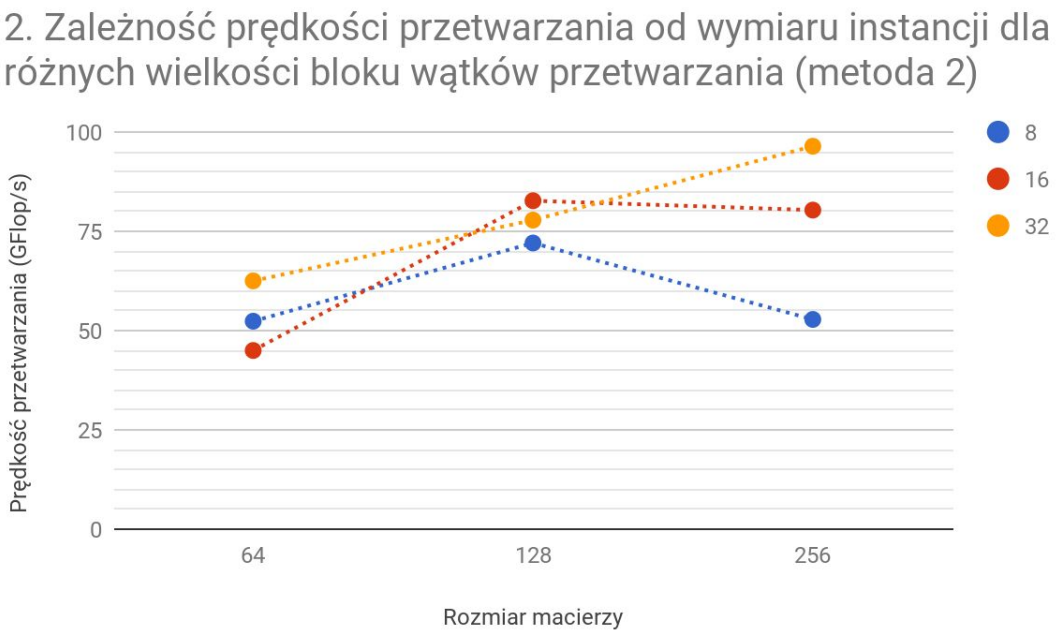
Tabela 1.

6. Analiza pomiarów i wnioski



Na podstawie wyznaczonych prędkości można zauważyć, że prędkość przetwarzania wzrasta wraz ze wzrostem wielkości macierzy dla każdego przypadku wielkości bloku wątków przetwarzania. Pierwsza metoda, gdzie występuje tylko jeden blok wątków, a obliczenia dodatkowo są wykonywane jedynie przy wykorzystaniu pamięci globalnej, charakteryzuje się ogólnie bardzo niskimi wartościami prędkości. Jest to związane z tym, że istnieje tylko jedna grupa wątków, a więcej multiprocesorów, na których mogłyby one pracować. Widzimy to także w parametrze aktywności multiprocesora w tabeli 1. - we wszystkich przypadkach ma ona wartość około 20%, a więc ⅕ (jeden z 5 procesorów). Dodatkowo, efektywność dostępów do pamięci oraz zajętość procesora są zależne przede wszystkim od rozmiaru bloku wątków, co również ma związek z istnieniem tylko jednej grupy wątków. Im większa grupa wątków pracuje, tym większe wartości tych współczynników, co ma przełożenie na poprawienie efektywności oraz prędkości przetwarzania.

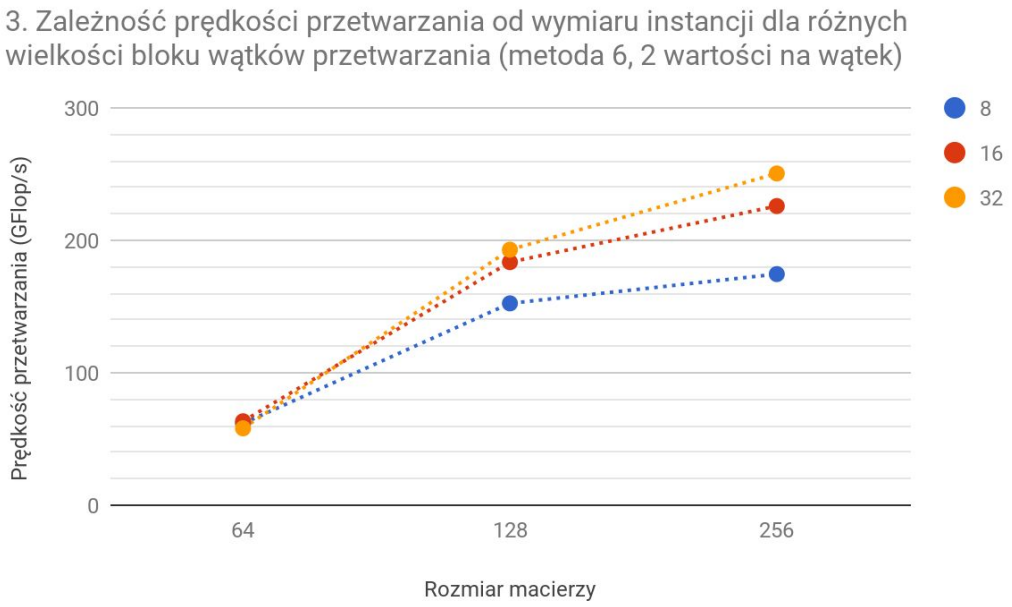
W metodzie tej bardzo nieefektywne wykorzystanie pamięci widać przy porównaniu RGLT oraz RGST z tabeli 1. do wyników innych metod. Już sama wartość CGMA pokazuje, że aby wykonać jakąkolwiek operację, potrzebujemy dostępu do bardzo wolnej pamięci globalnej.



W przypadku metody drugiej, gdzie występuje grid wieloblokowy, na pierwszy rzut oka można zauważyć dużo lepsze wartości prędkości przetwarzania, co wiąże się przede wszystkim z tym, że wykorzystany został potencjał większej liczby multiprocesorów. Różnice w prędkościach dla różnych wielkości bloków wątków są dzięki temu mniejsze niż w metodzie 1.

Obserwując aktywność i zajętość multiprocesora widzimy, że w tej metodzie decydujące znaczenie ma stosunek wielkości macierzy do wielkości bloku wątków - parametr ten określa wielkość macierzy pomocniczej, na której pracuje grupa wątków. Im większa jest ta wartość, tym lepsze uzyskujemy wykorzystanie multiprocesora.

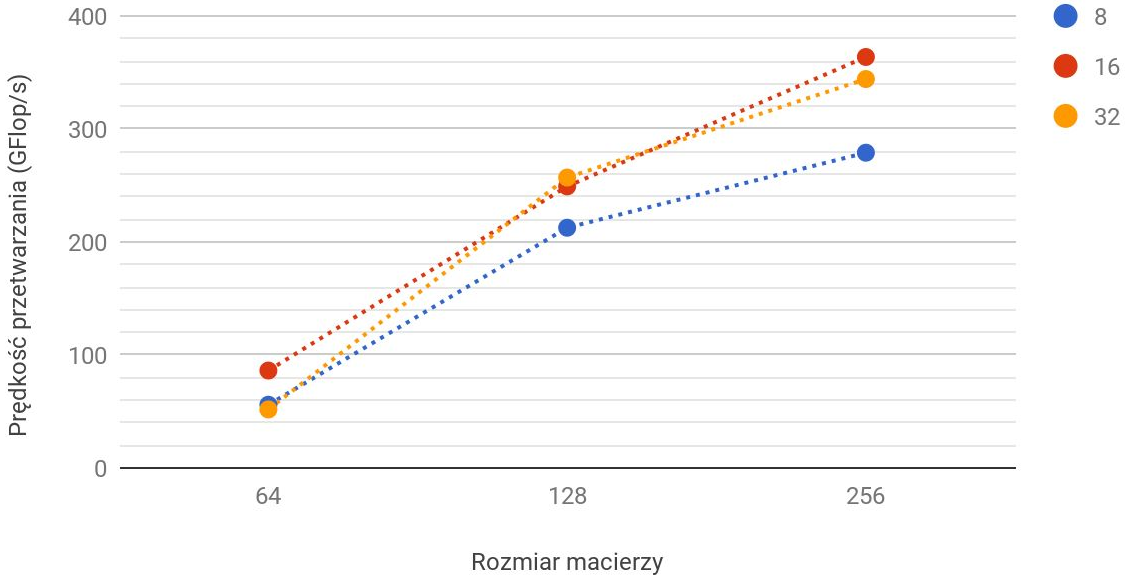
Pamięć nadal wykorzystywana jest mało efektywnie - jak wynika z parametru CGMA każda operacja wymaga odniesienia do pamięci globalnej. Uzyskujemy jednak wartości przepustowości większe niż w metodzie pierwszej, co może wynikać z wykorzystania większej liczby pracujących wątków.



W ostatniej metodzie miejsce ma ogromny wzrost prędkości przetwarzania. Powodem tego jest przede wszystkim zmniejszenie odwołań do pamięci globalnej. Wątki dzielą między siebie nie tylko wykonywanie obliczeń, ale także bardziej czasochłonne pobieranie danych z pamięci. Dodatkowo wykonujemy buforowanie wczytywania danych z pamięci - wykorzystujemy osobne tablice do wczytania danych i do wykorzystania danych podczas obliczeń. W ten sposób unikamy sytuacji, w której wątki czekałyby na pozostałe wątki, które jeszcze wykonują obliczenia, zamiast sprowadzać już następne dane z pamięci. Na koniec, podczas obliczania sąsiednich elementów macierzy wynikowej potrzebna jest mniejsza liczba odczytów z pamięci globalnej, ponieważ wielokrotnie wykorzystamy raz sprowadzone dane - wykonamy 3 zamiast 4 odczytów. Wątek nie oblicza jednak sąsiadujących elementów w tablicy wynikowej, a elementy na odpowiadających sobie pozycjach w sąsiadujących blokach - w ten sposób zapewniamy, że wiązki wątków wykonują jednocześnie te same instrukcje.

Widzimy, że wykorzystujemy współdzieloną pamięć w parametrze Static Shared Memory (tym więcej pamięci jest potrzebne, im więcej wątków w bloku), której nie wykorzystywaliśmy w metodach 1. i 2.. Aktywność i zajętość multiprocesora kształtują się podobnie jak w metodzie 2 - podobnie zależymy tutaj od wielkości pomocniczej macierzy, na której pracuje grupa wątków. Mamy też 100% efektywność odczytów z pamięci dla wszystkich badanych instancji problemu i 100% efektywność zapisów dla prawie wszystkich instancji problemu. Widzimy też spadek wartości parametrów RGLT - nie potrzebujemy już tylu odczytów z pamięci globalnej, wzrasta natomiast RGST - każdy wątek wykonuje więcej zapisów (bo po 2). Zdecydowaną poprawę w wykorzystaniu pamięci widzimy też w postaci parametru CGMA - jest on teraz zależny od rozmiaru bloku wątków, czyli w ogólności wielokrotnie większy niż uzyskana we wcześniejszych metodach wartość 1.

4. Zależność prędkości przetwarzania od wymiaru instancji dla różnych wielkości bloku wątków przetwarzania (metoda 6, 4 wartości na wątek)



Kiedy aż cztery sąsiednie elementy są rozpatrywane w tym samym czasie, prędkość wzrasta jeszcze bardziej. Obliczając w tej metodzie cztery sąsiednie elementy zamiast dwóch, ponownie zmniejsza się potrzeba na ilość odwołań do pamięci globalnej - dane, które były wykorzystane do obliczenia dwóch sąsiednich elementów, przydadzą się do obliczenia dwóch kolejnych, a więc zamiast 8 odwołań do pamięci globalnej mamy jedynie 4. Prowadzi to do blisko dwukrotnego zwiększenia prędkości przetwarzania. Analogicznie, aby zapewnić wykonanie tej samej instrukcji w wiązce, obliczamy dwa kolejne elementy znajdujące się na odpowiadających miejscach w blokach dokładnie poniżej. Lepsze wykorzystanie pamięci widzimy w parametrze CGMA, który jest równy dwukrotności wielkości bloku. Reszta parametrów kształtuje się podobnie do metody 6 z obliczaniem 2 elementów, jedynie mamy większe zapotrzebowanie na przepustowość zapisu do pamięci (wątek liczy 4 a nie dwie wartości).

7. Podsumowanie

Analizując metody o numerach 1, 2 i 6 można bez wątpienia stwierdzić, że ogromne znaczenie na wydajność algorytmu ma wykorzystywanie współdzielonej pamięci. Czym mniejsza liczba odwołań do pamięci globalnej, tym większą prędkość osiągają metody. Najlepszym tego przykładem jest ostatnia metoda (szósta, z czterema liczonymi wartościami na wątek).

Bardzo wpływ na efektywność ma również struktura gridu. Na przykładzie pierwszej metody widać dobrze, że grid jednoblokowy powoduje powolne działanie algorytmu. Jest to spowodowane tym, że nie wykorzystujemy możliwości wszystkich multiprocesorów dostarczanych przez kartę graficzną.

Dla każdej z metod pod uwagę braliśmy bloki o wielkościach równych 8, 16 oraz 32. Uogólniając, można zauważyć, że w tym przedziale wzrost rozmiaru bloku powodował wzrost prędkości

przetwarzania, zwłaszcza w metodzie z wykorzystaniem pamięci współdzielonej, ze zrównolegleniem obliczeń. Dodatkowo, czym większy blok, tym większe wartości GMLE (stosunek prób odczytu z pamięci globalnej do zrealizowanych odczytów) oraz GMSE (stosunek prób zapisu do pamięci globalnej do zrealizowanych zapisów).

Na dzisiejszym rynku wydajność pojedynczego rdzenia nie rośnie w tak znaczącym tempie jak kiedyś. Wzrost efektywności oprogramowania można jednak zapewnić korzystając z równoległości. NVIDIA CUDA oferuje bardzo ciekawy model umożliwiający podążanie tym nurtem.